The Journal of Systems & Software 202 (2023) 111693

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Do names echo semantics? A large-scale study of identifiers used in C++'s named casts[☆]

Constantin Cezar Petrescu[a,*], Sam Smith[a], Rafail Giavrimis[a], Santanu Kumar Dash[b]

[a] University of Surrey, Guildford, Surrey, United Kingdom
[b] Royal Holloway, University of London, Egham, Surrey, United Kingdom

ARTICLE INFO

Article history:
Received 1 September 2021
Received in revised form 30 September 2022
Accepted 29 March 2023
Available online 6 April 2023

Keywords:
C++ type conversions
Languages
Programme analysis

ABSTRACT

Developers relax restrictions on a type to reuse methods with other types. While type casts are prevalent, in weakly typed languages such as C++, they are also extremely permissive. Assignments where a source expression is cast into a new type and assigned to a target variable of the new type, can lead to software bugs if performed without care. In this paper, we propose an information-theoretic approach to identify poor implementations of explicit cast operations. Our approach measures accord between the source expression and the target variable using conditional entropy. We collect casts from 34 components of the Chromium project, which collectively account for 27MLOC and random-uniformly sample this dataset to create a manually labelled dataset of 271 casts. Information-theoretic vetting of these 271 casts achieves a peak precision of 81% and a recall of 90%. We additionally present the findings of an in-depth investigation of notable explicit casts, two of which were fixed in recent releases of the Chromium project.

1. Introduction

Developers like flexibility while using programming language features during software development. Type casts allow developers to work around the restrictions imposed on a specific type and use methods written for other types. While casting offers flexibility, it can lead to undefined behaviour in weakly typed languages like C/C++. For example, considering the cast operation `a=(T)b`, the outcome of this statement is unclear unless we know what `T` stands for and what are the types of `a` and `b`. If `a` and `b` are scalars, this could be a value conversion. If they are objects, this could be a downcast from `b` to create `a`, if `a`'s class is derived from `b`'s class. `a` and `b` could be unrelated pointer types, in which case, the set of permissible operations is so vast that compilers might struggle to identify semantic errors.

Winther (2011) studied the safety of type casts and found that a quarter of them were guarded with type checks to ensure their validity of type casts against run-time errors. This was corroborated in a later study by Mastrangelo et al. (2019) on the classification of patterns for type casting. A study of implicit casting in JavaScript (Pradel and Sen, 2015) found most implicit casts to be harmless and useful, implying that developers use them judiciously. Mastrangelo et al. (2019) performed a study of how developers use type casts in Java and found 26 usage patterns for type casts. Importantly, they discovered that half of the casts inspected by them were not guarded locally which could potentially cause run-time errors. Thus, there is a need to vet type casts to understand if they are being used carefully.

Type casts come in two forms: *implicit* and *explicit*. Implicit casts or coercions are conversions from one type to another without explicitly specifying the new type, and they are usually limited to numeric types. Explicit casts are type conversions where syntactic constructs are used to specify the target type. Compilers have multiple checks to vet implicit casts on numerics. Even so, it is not possible to categorically enforce checks on casts for several mainstream languages with user-defined types. Therefore, for languages like C++, that are permissive in how memory is used at a low-level, several primitives for explicit type conversion have been introduced. These primitives, which are called *named casts*, come with a unique set of checks on the cast operation. They are the recommended technique for explicitly changing one type to another in C++ and have two placeholders in the primitive: a *source* expression that needs to be cast and the *destination* type for the cast.

In this paper, we propose a lightweight approach to check if casts are used judiciously. Casalnuovo et al. (2020) presented source code as being dual channel. One channel is represented by the algorithmic channel comprised of instructions understood and executed by computers. The second channel is the natural language channel which consists of identifiers and comments to

[☆] Editor: Christoph Treude.
[*] Corresponding author.
E-mail addresses: c.petrescu@surrey.ac.uk (C.C. Petrescu), ss02680@surrey.ac.uk (S. Smith), r.giavrimis@surrey.ac.uk (R. Giavrimis), santanu.dash@rhul.ac.uk (S.K. Dash).

https://doi.org/10.1016/j.jss.2023.111693

provide semantics for the instructions. In line with the recent work that uses meaning in identifiers in programmes (Dash et al., 2018; Pârundefinedachi et al., 2020), we propose a dual channel approach to analyse *named casts*. Our assumption is that developers leave hints about their intent in the identifiers that they choose and that this information can be used to check fidelity of an explicit type conversion. In particular, we are interested in knowing if the *source* expression that is being cast is related to the *destination* variable to which the result of the cast is being assigned. Our main contributions are as follows:

1. We propose an information-theoretic approach and a tool to detect poor programming and naming practices in named casts. Our tool's code, the dataset we evaluated it on and the results of the information-theoretic analysis are available online.[1]
2. We extract named casts from the Chromium project (Google, 2019a), which is an aggregation of over 34 components with nearly 27 million lines of C++ code. We sample this dataset random-uniformly at 90% confidence (5% margin of error) and create a dataset of 271 named casts vetted by human developers to create a ground truth.
3. We propose a taxonomy of how a user can build a named cast operation along with a set of broad categories of cast usage. We show how the taxonomy fits the ground truth set to provide insights on how casts are used.
4. We evaluate our tool on the vetted dataset and achieve a peak accuracy of 92%. Additionally, we perform an in-depth investigation of the cases flagged by the tool and provide insights into both poor programming practices and poor naming choices when using named casts.
5. We demonstrate impact through two instances of named casts flagged by the tool that have been patched in a recent release of the software. In addition, we discover another two cases where the named casts were part of code with a high complexity that eventually led to bugs. After the bugs were fixed, the named casts were completely removed.

We discuss an overview of casting in C++, along with an example of imprecise *named cast* usage and the motivation for our research in Section 2. We describe our methodology in Section 3 and the results of our evaluation in Section 4. Section 5 discusses some threats to validity. Section 6 presents the related work and Section 7 concludes this work.

2. Cast operations, their use and the motivation of the work

C++ provides several ways in which a type conversion can be effected. We first provide an overview of these ways. Then, we show through an example how, despite clear guidelines on how casts should be used, type casts can be used imprecisely.

2.1. Implicit and explicit casts

Type conversions are operations where the type of an expression is changed from one type to another. There are two types of conversions: implicit and explicit casts. In implicit casts, the conversion is done without the developers explicitly specifying the type to which a value needs to be converted. Implicit casts are performed automatically by the compiler if there is a viable conversion. For example, in C/C++, it is possible to pass a `float` as an argument to a method which expects a `double` (Cppreference, 2019e). Implicit conversions, also known as standard conversions (Cppreference, 2019a), are generally applied on built-in numerical data types, booleans and some pointer conversions (Cppreference, 2019h). The implicit conversions between numerical types are called promotions (Cppreference, 2019a) and are allowed from smaller size types to larger size types.

```cpp
double x = 10.3;
int y;
y = int (x); // functional notation
y = (int) x; // c-like cast notation
```

Listing 1: Functional and C-style syntax for implicit type conversion.

```cpp
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

Listing 2: Example of `static_cast` .

C/C++ also allows explicit conversion using syntactic constructs. The syntactic constructs tell the compiler to perform a type conversion where the new type is specified in comparison to implicit conversions. There are two ways to perform explicit casts, which are presented in Listing 1. Here, a variable `x` of type `double` is converted to an `int` type. The first is the functional style, where the target type is treated as a method and the variable that will be converted is passed as an argument. The other is commonly referred to as the C-style syntax where the use of the variable is qualified by the target type within parenthesis.

The function and C-style explicit casts can handle conversion of built-in types in the compiler such as numeric types. However, using those operators on user-defined types, and in particular, class hierarchies, requires additional language constructs, and named cast operators were introduced in C++. There are four named cast operators in C++: `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`. Out of the four, `static_cast`, `dynamic_cast` and `const_cast` perform additional checks either statically or at runtime to avoid undefined behaviour, because of incorrect usage of type casts (Cppreference, 2019d). `reinterpret_cast` is the most permissive with no checks on the validity of the type conversion. It merely reinterprets the memory holding an object as another type.

The `static_cast` operator.`static_cast` vets the casts by statically checking the validity of the conversions against the class hierarchies (Cppreference, 2019g). As shown in Listing 2, a downcast of an object `a` typed as base class `Base` to a derived class `Derived` is allowed, but the developer needs to be confident that `a` will never be an object of another derived class of `Base`. If the latter happens, accessing a field of the `Derived` class through `b` would lead to undefined behaviour. This is because `static_cast` does not apply runtime checks to validate if `a` is an object of type `Derived` or another derived class `Derived2` of `Base`. Therefore, the correctness of a `static_cast` is reliant on the developer. `static_cast` operations are also used for converting `enum` and `void` types where the developer is sure of the type of the data pointed to by a `void` pointer.

The `dynamic_cast` operator.`dynamic_cast` is an operator used for casting pointers and class reference conversions. Unlike `static_cast`, `dynamic_cast` checks whether the *named cast* is permissible at runtime. If not, it returns a null pointer (Cppreference, 2019c). This operation guarantees that the result points to a valid object of the new type at the end of the type conversion. Listing 3 presents an example of `dynamic_cast` for a pointer `pBDerived`. The

[1] https://github.com/Constantin-Petrescu/CastChecker/

```
 1 class Base { virtual void vf(){} };
 2
 3 class Derived : public Base { };
 4 int main()
 5 {
 6         Base *pBDerived = new Derived;
 7         Derived *pd;
 8         pd = dynamic_cast<Derived*>(pBDerived);
 9         return 0;
10 }
```

Listing 3: Example of dynamic_cast.

```
1 class A { /* ... */ };
2 class B { /* ... */ };
3
4 A * a = new A;
5 B * b = reinterpret_cast<B*>(a);
```

Listing 4: Example of reinterpret_cast.

pointer has the initial type {Base*} and it points to a Derived object. Through the cast on Line 8, the pBDerived pointer becomes an object of class Derived. dynamic_cast operations perform validity checks using the Run-Time Type Identification (RTTI) which is a feature in C++ to inspect types of objects at runtime. Naturally, the runtime checks introduce overheads and dynamic_cast is an expensive operation for performance-sensitive applications.

The reinterpret_cast operator. This operator's role is to reinterpret the memory holding an object of one type as another type to convert it from one type to another. The pointer to the memory is recast into a new pointer type without any checks if the content can be of the new type. In general, this cast is used on low-level conversions based on a reinterpretation of the binary values of the variables (Cppreference, 2019f). In Listing 4, a reinterpret_cast example is shown on Line 5. The variable a of class A is reinterpreted to type B and assigned to pointer b even though A and B are unrelated in the class hierarchy. The reinterpret_cast has a lower overhead than the other operators since it does not perform validity checks. Like the static_cast, though, the correctness for this conversion relies entirely on the developer.

The const_cast operator. This operator makes it possible to modify variables that have the type qualifier const, which directs the compiler not to allow any modification for a variable, and volatile , which prevents the compiler from applying any optimisations on the variable. An example is presented in Listing 5. The variable c of type const char* is passed as an argument to a method print which only supports char*. This forces the use of const_cast in Line 9 as mandatory to match the actual type to the formal parameter type. The C++ standard states that the const_cast operator can introduce undefined behaviour in programmes. This situation can appear if the constness is removed from a variable and after the variable is modified (Cppreference, 2019b).

2.2. An example of imprecise named cast usage

Named casts were proposed initially to provide semantic clarity. However, developers sometimes use them to bypass type system restrictions at the cost of increased code complexity. Consider Listing 6 as an example. The code is a snippet taken from the implementation of QUIC protocol (Google, 2020). QUIC is a general-purpose transport layer network protocol open sourced as a part of the Chromium project. There are two uses of the

```
 1 void print (char * str)
 2 {
 3   cout << str << '\n';
 4 }
 5
 6 int main ()
 7 {
 8   const char * c = "sampletext";
 9   print ( const_cast<char *> (c) );
10   return 0;
11 }
```

Listing 5: Example of const_cast.

```
 1 // Add information on the relationship between QUIC error
       codes
 2 // and their symbolic names.
 3 std::unique_ptr<base::DictionaryValue> dict(new
       base::DictionaryValue());
 4
 5 for (QuicErrorCode error = QUIC_NO_ERROR;
 6      error < QUIC_LAST_ERROR;
 7      error = static_cast<QuicErrorCode>(error + 1)) {
 8   dict->SetInteger(QuicErrorCodeToString(error),
 9      static_cast<int>(error));
10 }
```

Listing 6: An example where two static_cast operators are used to iterate over an enumeration and store integer values in a dictionary. The snippet is from the file *net_log_util.cc* of component *Net* taken from an open source implementation of the QUIC protocol in the Chromium project

static_cast operator in this snippet, which populates a dictionary dict with key–value pairs, which are strings representing an error description and an integer representing the error code. It is important to note here that error itself is neither an integer nor a string but an *unscoped* enum type QuicErrorCode.

The type enum or enumeration is a user-defined type which consists of a set of named integral constants (Microsoft, 2019). Enumerations are generally used in three situations: a single choice where the developer filters through the choices with a switch statement, a multiple choice through C-style bitsets, or as a type definition for integral types. In Listing 6, the type enum is not used for any of the three situations, but it is used to iterate over the enumeration values and populate dict. By design, C++ does not encourage the iteration over objects of type enum since it does not provide an iterator. In the example, the iteration is achieved by implicitly casting the loop control variable error into an integer, incrementing it and casting it back to QuicErrorCode using a static_cast in Line 7. In the loop expression, QUIC_NO_ERROR and QUIC_LAST_ERROR are the first and last elements of the enumeration. The second static_cast in Line 9 converts the variable error of type QuicErrorCode to an int. It is used as a parameter for the function SetInteger, which populates the dictionary dict with key–value pairs. This is the second time that the developers chose to cross the boundaries between an enum type to an int to be able to use operators of the type int.

The iteration on enum objects can be pernicious, as enum types are not guaranteed to be contiguous. The Clang++ compiler would replace QUIC_NO_ERROR and QUIC_LAST_ERROR to their actual values in the loop from the snippet. This means that error would take all the values in the corresponding range. The enumeration QuicErrorCode is not contiguous and the values for each entry are defined by the developers. This means that the dict could contain error codes that were not described originally in QuicErrorCode. However, the developers handle those cases explicitly in the

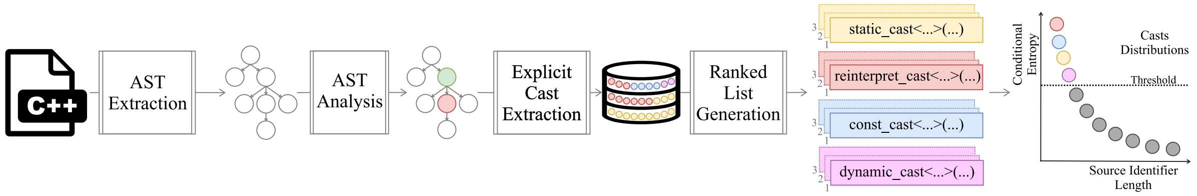


Fig. 1. Software architecture diagram of our tool which extracts named casts from a C++ codebase and analyses them using information theory.

function `QuicErrorCodeToString`, which contains a `switch` over all the values from `QuicErrorCode`. This function returns the string of the `error` or an invalid error code for any other values. This implementation is not erroneous; however, it is suboptimal.

One may wonder at this stage, what could be a better solution and what should the solution aim to achieve? Type systems came about to ensure type safety and casts typically should be avoided wherever possible. The aim of a better solution should be to keep the `enum` and `int` types separate and implement all operators essential to iterate or operate in the `enum` space. The developers used an enumeration to generate a dictionary object type which will be used later by the rest of the application. The enumeration implementation consists of the `QuicErrorCode` declaration along with a set of functions of switch cases such as `QuicErrorCodeToString` that allow the return of the string for an error. We believe a better solution would be to declare and use a dictionary from the start rather than declaring and using the enumeration to create the dictionary.

This solution would not require the crossing of type boundaries, since the type of the dictionary can be declared accordingly to the types of the values. Also, the solution would bring improved efficiency. Enumerations are efficient since they are resolved at compile time and converted into integral literals at the bitcode level. The enumerations are used along with switch cases and iterations over the enumerations, which present a linear efficiency. This efficiency performs well on a small number of cases, which is not the case for `QuicErrorCode` since it consists of 199 cases. On the other hand, the selection of a key in a dictionary would have a logarithmic efficiency. We are not sure if `QuicErrorCode` is used in any other part of the application, but dictionaries should generally perform better than large enumerations. Our solution would also ease the code maintainability process. Each time `QuicErrorCode` needs to be updated, it requires modifications at the declaration and at each function with switch cases. It would be easier to maintain a dictionary since the only modification required would be at the declaration. This example shows a need for tools that identify if the cast of types is essential and if the cast is done correctly. It is crucial to ensure that the crossing type boundaries are beneficial from a software engineering point of view, allowing code reuse without confounding the uses of types and operators for those types.

2.3. Motivation

In this research, we hypothesise that in large and mature projects such as Chromium, where code is reviewed before it is merged in the application, there are hints in programme identifiers that point to their purpose. We aim to use this natural language information in identifiers to understand if *named casts* are being used for good software engineering reasons. If this is not the case then we aim to identify when the named casts are being poorly implemented, such as when they are part of unoptimised or buggy code. For example, the actual to formal binding for the method `SetInteger` binds `error` of type `QuicErrorCode`to a formal named `in_value` of type `int`. A perfunctory check of the names for the variables and the types may seem that these variables are disparate. However, one may notice upon close inspection that `SetInteger` is a modifier of a dictionary. Therefore, it is essential that formal arguments of this modifier are named generically. In this work, we combine an automated analyser with human inspection to classify cases where *named casts* are used to point out both good and poor practices in using *named casts*.

In a *named cast* situation, precise names are meaningful names that reflect the relation between the *source* and *destination*. The choice of the identifiers is not only vital during development, but also during maintenance. Precise names reflect that the developers had a good understanding of the problem that they solved. The same precise names allow other developers to gain a faster and more comprehensive understanding of the code. Thus, the reusability and maintenance of the code is made easier. If the relation between *source* and *destination* does not exist, developers may be misled by the names and overlook some cases which could be dangerous during code testing and maintenance. For instance, consider the following set of pairs of imprecise identifier names: `tag` with `chars[i]`, `levels` with `fparams[0]`, `param` with `bufSize`, `t` with `output_cursor`, `val` with `p[i]`, `frames` with `out_trace`. All of these pairs exhibit no meaningful connection or semantic similarity, and this could potentially lead to misunderstanding. Those cases need to be identified and refactored with meaningful names. Our tool uses the information-theoretic analysis to discover imprecise names given the *source* expression and *destination* variable.

3. Methodology

Our objective is to analyse if natural language identifiers are indicative of the purpose of the cast. For this, we focus on assignment expressions where the right hand side is a named cast expression and on actual-to-formal bindings in method calls where the argument to the method is a named cast expression. In both cases, the expression that is cast to a new type is referred to as the *source* and the identifier to which the cast expression is bound is called the *destination*.

3.1. Software architecture of the proposed tool

Fig. 1 presents an overview of our tool. We rely on a Clang plugin to traverse the abstract syntax tree (AST) of source files. Our plugin traverses every node to discover named cast expressions and then determines if the expression is part of a larger sub-tree representing an assignment operation or a method call expression. Details of this process can be found in Section 3.2. We detail the analysis of the extracted dataset in Section 3.3 and the results of our manual investigation can be found in Section 4.6.

Our corpus is generated from the Chromium project (Google, 2019a). Chromium is an extensive system written in C++ and it only supports the Clang compiler for building. Chromium uses the Ninja build system and GN (Google, 2019b) as a meta-build system that generates Ninja build files. The Ninja files run the Clang compiler, for which our analysis plugin is written, on the C++ files. Therefore, we modified the meta-build system to use

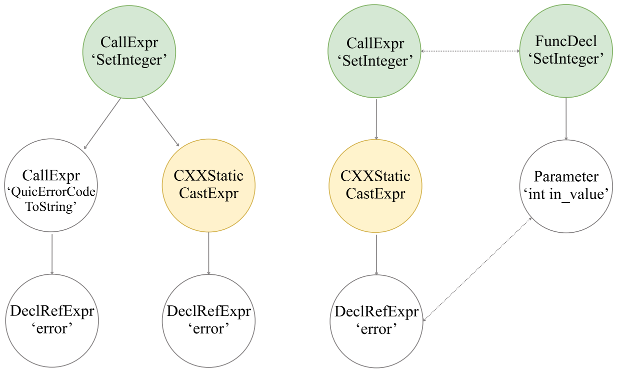


Fig. 2. Abstract syntax tree representation for our motivating example; we selected only the nodes of interest. The left side shows the function call, *SetInteger*. The right side presents the mapping between the function call and the function definition.

a local version of Clang that is compatible with our plugin. The output generated by our modified compilation phase is a JSON file containing the named cast information for every C++ file that is compiled. These named casts constitute the dataset for our analysis which is described next.

3.2. Extraction of named casts

In Fig. 2, we present an example of how our plugin analyses a named cast from the *Net* sub-system in Chromium. After Clang parses the source file and produces an AST for the file *net_log_util.cc*, the plugin traverses the tree and searches for named casts that are a part of either assignments or call expressions. On the left in Fig. 2, the syntax tree for the function call `SetInteger` is shown. The node `CallExpr` has a child `CXXStaticCastExpr` which represents the node for `static_cast` implying that the named cast is used as an argument for a function call. The plugin then follows the call to find the method definition. A projection of the AST for the method definition is shown on the right in Fig. 2. The plugin then links the formal parameter to the actual parameter for `SetInteger` and discovers that the *source* variable is `error` and the *destination* variable is `in_value`. All the macro names in the code will be replaced with actual code at the compilation stage (Cplusplus, 2019). However, the physical location of the named casts would still point to the macro's call. To solve this, our plugin is designed to follow macro definitions, post their expansion, to discover named casts inside macro definitions as well. For each C++ file analysed, the Clang plugin generates a JSON file with information about named casts. Each JSON entry in the file consists of the type of named cast i.e. `static_cast`, `dynamic_cast` `reinterpret_cast` or `const_cast`. It additionally contains the type and the subtokens for the *source* and the *destination* expression. To generate the subtokens, we extract all tokens from each expression and we preserve only identifiers, keywords and literals tokens. Those tokens are split in subtokens based on the camelcase and snakecase separators.

3.3. Data analysis

In this research, we study if the identifiers convey the reason for the use of a named cast. We do this by comparing the *source* expression subtokens with the *destination* variable subtokens. Our comparison is based on a notion of entropy — the amount of information in names. We find cases where *source* subtokens are significantly different from the *destination* subtokens. The difference is measured using conditional entropy which computes the number of additional bits that would be required to represent the *destination* given the subtokens in the *source*. While we have access to the type information, we do not use this information in the calculation of the conditional entropy. The reason for this is that, during development and sometimes in static time, the type of a variable is not always visible to the human. That is why including the type in our analysis would make it different than the way a human would view code.

Table 1
Examples for conditional entropy calculations; S - *Source*, D - *Destination*, H - Entropy.

S	D	H(S)	H(S, D)	H(D| S)
baz	foo	0	1	1
baz	fooBar	0	1.58	1.56
baz	fooBaz	0	0.92	−0.08
bazGoo	foo	1	1.58	0.58
bazGoo	goo	1	0.92	−0.08
bazGoo	bazGoo	1	1	0
bazGoo	fooBar	1	2	1
bazGooFoo	fooBar	1.58	1.92	0.34

Next, we show how we compute the conditional entropy of `fooBar` given the entropy for `bazGoo` in the named cast `fooBar = static_cast<Quux*> bazGoo`. (1) presents the standard Shannon's formula for computing the entropy (Shannon, 1948), which is the negative sum of the probabilities multiplied with the logarithm value of the probability. Here, X represents `bazGoo` and x_i represents the probabilities for `baz` and `Goo` which are the subtokens of the identifier. The subtokens' probabilities have a value of $\frac{1}{2}$ since there are only two possible options. Thus, $H(bazGoo) = -(2 * \frac{1}{2} * log_2 \frac{1}{2}) = -[1 * (-1)] = 1$. In other words, we need only one bit to represent the two possible options for the *source* subtokens.

We then compute the conditional entropy as shown in (2) (Cover and Thomas, 2005). The conditional entropy is the amount of information (in bits) required to express the outcome of a random variable knowing the outcome of another random variable. In (2), `Y` is a placeholder for the subtokens from `foo` and `Bar` in our example. We try to compute the conditional entropy of `Y` given `X` based on the chain rule. Thus, the conditional entropy value is the entropy value of the *source*'s subtokens subtracted from the joint entropy value of both *source* and *destination* subtokens. In current example, the joint entropy is computed for all the subtokens `baz`, `Goo`, `foo` and `Bar`. $H(bazGoo, fooBar) = -(4 * \frac{1}{4} * log_2 \frac{1}{4}) = -[1 * (-2)] = 2$. The conditional entropy tells how many more bits are needed to represent the additional subtokens that the *destination* identifiers bring knowing the *source*'s subtokens. In the example, the conditional entropy equals with the difference between the joint entropy and entropy of the `bazGoo` and it has value one. Thus, the *destination* `fooBar` identifier will require an additional bit in order to represent the two new additional subtokens. Examples of entropy values for identifiers can be found in Table 1.

$$H(X) = -\sum_{i=1}^{n} P(x_i) * \log P(x_i) \qquad (1)$$

$$H(Y|X) = H(X, Y) - H(X) \qquad (2)$$

The role of conditional entropy value is to discover how different a *destination* expression is, compared to the *source* expression used in a named cast. Therefore, we compare the subtokens of the *destination* expression with the subtokens of the *source* expression for each named cast operation we collected from Chromium. If we were to consider the subtokens across multiple named cast cases in the conditional entropy calculation for each case, then the result would not be the difference between *source* and *destination*. The comparison would instead identify if the *destination* expression contains unique subtokens compared to *source* subtokens from all the cases. The chances that some of the *destination* subtokens appear in the subtokens from *source* expression increases with the addition of multiple *source* expressions in the calculation of the conditional entropy.

The conditional entropy values of the *destination* given the *source* enables the identification of cases where the *source* looks significantly different from the *destination*. A low conditional entropy value implies that *source* and *destination* subtokens are similar. On the other hand, a high conditional entropy value means they have few subtokens in common. If identifiers are used for different purposes, under the assumption that names are chosen carefully, their subtokens will also be different. We are interested in the cases where the conditional entropy is high. Those cases should generally point to clear instances where disparate names are used in the *source* and the *destination* expressions. This is indicative of the *destination* variable serving a different purpose than the *source* expression.

One may wonder why we did not use a simpler distance metric such as Levenshtein Distance (LD) instead of conditional entropy. LD uses three operations: insertion, deletion, substitution and the edit distance is the number of operations used to transform the input string into the output string. It is sensitive to the ordering of subtokens. Subtoken ordering is not important to us as we want only to check if the subtokens are being reused from the *source* in the *destination*. Whether an identifier is called `thrown_type` or `type_thrown` is immaterial to us, but it affects the Levenshtein distance.

3.4. *Flagging casts for inspection*

Our tool flags casts for inspection if the conditional entropy of a *destination* variable, given the *source* expression, is above a prespecified threshold. We refer to these casts as flagged casts. The tool is parametric in the threshold: a user can tune the threshold to their needs. For our experiments, we optimise the threshold based on the precision–recall curve, as detailed in Section 4.4.

3.4.1. *Shortlisting casts*

We apply a set of preprocessing filters before computing the conditional entropy for a cast. These filters are essential to reduce the noise in entropy calculations, which may artificially inflate conditional entropy values and introduce false positives. Our filters are described below.

Casts from literals. The universe of literals is large and literals are likely to differ from well named identifiers that they are being cast into. This increases the conditional entropy values of the *destination* given the *source*, introducing false positives. We check if the *source* is a literal and if it is, the cast is not flagged by the tool. The possible literals in C++ are: integers in base 2, 10 or 16 (e.g. `42`, `033`, `0x00`), floating point numeric values (e.g. `0.5`, `-0.26E\tmin5`), characters (e.g. `’a’`, `’\{’`), escape sequences (e.g. `\\n`, `\\r`), and strings (e.g. `"HelloWorld"`).

Descriptive variables and types. We check if the *source* is part of the *destination* using the substring operation. In these cases, the name of the *destination* is usually a more descriptive name for the *source*, while retaining the *source* token. Therefore, we do not flag these casts. We also check if the *source*’s type is a substring of the *destination*’s type. If the *destination* type contains the *source* type, we assume that the two types have similar semantics and refrain from flagging these casts. For instance, our tool would not flag the third example from Table 1 because the *source* `baz` can be found in the destination `fooBaz`.

We generate a ranked list for the named casts based on their conditional entropy value in order to select identifiers where the expressions in *source* and *destination* are disparate. This was done to generate the corpus which is comprised of all four categories of named casts: `const_cast`, `dynamic_cast`, `reinterpret_cast` and `static_cast`.

4. Evaluation

We now present the results of our evaluation of our tool on the Chromium corpus. Through our experiments, we try to address the following research questions.

Rq$_1$. How prevalent is the usage of named casts in large and mainstream open-source projects such as the Chromium project? (Section 4.1)

Rq$_2$. Is our tool effective at identifying poor practices, in both how casts are used and how identifiers in the casts are named? (Section 4.4)

Rq$_3$. Are longer *source* expressions, which are harder to comprehend, correlated with poor cast usage? (Section 4.4)

Rq$_4$. What is the nature of some of the poor practices in choosing names and usage of casts identified by the tool? (Section 4.6)

The goal for **RQ$_1$** is to assess how often the named cast operators are used in large projects. This helps to identify if there would be a need for a tool to validate named cast operators. **RQ$_2$** checks that the collected named cast operations can be validated using the identifiers information. For **RQ$_3$**, we look into long *source* expressions (those of 100+ characters) which are harder to comprehend by developers. **RQ$_3$** tries to observe if the relation between *source* length and conditional entropy value can lead to detecting casts misuse. The scope of **RQ$_4$** is to generate insights about each named cast operator’s usages and the identifier names used.

4.1. *Quantitative analysis (RQ$_1$)*

To answer **RQ$_1$**, we present the usage patterns of named cast operators in Chromium’s components. Table 2 shows the distribution of the named casts in various components of Chromium. Our corpus consists of 36,298 named casts. Table 2 shows the frequency for each category of named casts for individual modules in the Chromium corpus. Overall, 63.62% are `static_cast` s, 32.68% are `reinterpret_cast`s, 0.25% are `dynamic_cast`s and 3.45% are `const_cast`s. As discussed in Section 3.2, we consider named casts that are a part of either assignments or actual-to-formal parameter binding in function calls. The proportion of named casts that are a part of assignments is 64.46% (23,395 casts) while 35.54% (12,903 casts) are in call expressions. Overall, there are 1.3 named cast operations per one thousand lines of code across Chromium. Table 2 shows that the number of casts per KLOC varies and it can take values from 0.1 up to 86.7 depending on the module’s scope. This shows that some developers, particularly those on the higher end of cast usage that we observed, could benefit from a tool to vet their named cast operations.

It is observed from Table 2 that `dynamic_cast` and `const_cast` operators are used rarely. The `dynamic_cast` operator uses Run-Time Type Identification (RTTI) to verify that the types can be converted at runtime, which is an expensive operation. It is likely that the cost of checking prohibits their widespread use. `const_cast` operators are used to set or remove the constness or volatility of variables. Such variables are rare themselves which explains why so few instances of `const_cast` are present in our dataset. `static_cast` can be used to cast up or down objects. A check on the class inheritance hierarchy evaluates if the

Table 2
C++ Corpus from Google Chromium. Represents the distribution of cast types and the frequency of usage of each conversion operator (S - `static_cast`, R - `reinterpret_cast`, D - `dynamic_cast`, C - `const_cast`).

Name	Description	KLoc	Assignment Expressions				Call Expressions				Total	Casts per
			S	R	D	C	S	R	D	C		KLoc
V8	JavaScript Engine	1,359	1,262	1,649	0	8	1,592	353	0	4	4,868	3.6
Net	Networking Protocols	766	616	1,153	0	26	693	770	0	15	3,273	4.3
gpu	Graphics Stack	277	1,386	307	0	10	171	100	0	56	2,030	7.3
UI	UI Frameworks	179	197	823	0	5	689	36	0	4	1,754	9.8
Media	Media Components	370	450	700	0	20	358	207	0	3	1,738	4.7
Blink	Browser Engine	1,524	1,081	120	0	0	138	0	0	0	1,339	0.9
Chrome	Application Layer	2,385	776	199	0	22	256	3	0	0	1,256	0.5
Webrtc	Communications API	634	482	78	0	9	541	33	0	1	1,144	1.8
Skia	Graphics Library	665	349	274	0	20	208	179	0	33	1,063	1.6
Device	Sensor Communication	134	469	376	0	0	116	30	0	0	991	7.4
Policy	Policy Settings	39	121	34	0	353	314	34	0	0	856	22.2
Perfetto	Tracing Service	205	297	7	0	54	454	1	0	0	813	4
SafeBrowse	URL Check Protocol	9	162	57	0	79	440	46	0	0	784	86.7
Dawn	WebGPU	66	125	542	0	0	25	3	0	0	695	10.5
Protobuf	Serialising Struct Data	227	160	77	0	17	394	10	0	15	673	3
Common	Application Layer	40	341	319	0	1	9	0	0	0	670	16.8
Base	Core Components	278	192	220	0	7	129	102	0	6	656	2.4
Pdfium	PDF Library	484	369	62	0	1	181	20	0	0	633	1.3
ICU	Unicode Components	325	285	63	75	40	79	14	1	5	562	1.7
VIZ	Visual Subservices	84	176	235	0	0	51	57	0	0	519	6.2
MetricsProto	Data Analysis	75	165	0	0	47	304	0	0	0	516	6.9
Sync	Sync Implementation	140	92	1	0	84	313	3	0	0	493	3.5
Angle	Graphics Engine	2,381	175	28	0	3	230	19	0	0	455	0.2
Buildtools	Buildtools Chromium	510	187	153	13	2	25	7	0	3	390	0.8
Audio	Audio System	34	43	202	0	0	33	50	0	0	328	9.6
Swiftshader	Graphics Library	2,166	160	87	0	5	62	6	0	0	320	0.1
Extensions	Core Parts Extension	224	312	4	0	0	0	0	0	0	316	1.4
CC	Compositor Renderer	198	117	17	0	0	167	6	0	2	309	1.6
RemoteCocoa	Cocoa Front-End	4	137	158	0	0	5	1	0	0	301	70.7
Logging	Logs Implementation	43	90	0	0	6	176	0	0	0	272	6.3
Rest of Corpus	Components < 250	11,175	2,238	1,284	0	247	1,925	545	0	42	6,281	0.6
	Total Casts	27,000	13,012	9,229	88	1,066	10,078	2,635	1	189	36,298	1.3

conversion between the object and *destination* type is possible. Therefore, `static_cast` is safer than `reinterpret_cast` which is extremely permissive, allowing arbitrary type conversions. Indeed, best practice is to use `static_cast` over `reinterpret_cast` and this is reflected in the prevalence of `static_cast` operations in our corpus. It is noticed from Table 2 that the larger and performance-critical modules such as the JavaScript compiler *V8*, networking (*Net*), GPU, user interface (*UI*), the *Media* libraries, etc. have the most casts. Interestingly, none of these modules uses the runtime intensive `dynamic_cast` cast operators. Only International Components for Unicode *(ICU)* and *Buildtools* components contain a total of 88 `dynamic_cast` operators. Neither of these components are central to the user experience of the browser and thus they can potentially tolerate runtime overheads.

4.2. Establishing ground truth

We performed a manual investigation of a random-uniform sample of the corpus to establish the ground truth. The ground truth is essential to present trade-offs at different thresholds above which casts are flagged by our tool. Using the central limit theorem (Anderson, 2010), we targeted a 90% confidence with a 5% margin of error for the sampled set. The set comprised of 271 data points with a breakdown of 163 `static_cast`, 97 `reinterpret_cast`, 11 `const_cast` and 0 `dynamic_cast` operations. There are no `dynamic_cast` operations in the sampled set because the casts corpus contains only 89 of these cases out of 36,298, thus the probability of randomly selecting any of these cases is small.

Raters. We used three raters to establish the ground truth. All raters were male residents of the United Kingdom. Their occupations were: full-time undergraduate student (R1), full-time Ph.D. student (R2) and Lecturer (R3), with 3, 7 and 20 years of programming experience respectively. The raters were provided with full access to the *source* code for the casts to mark casts as true or false. The true cases represent casts that are poorly implemented or they have imprecise names for identifiers. In contrast, false cases are correct and efficient implementations with meaningful names for the identifiers.

Guidelines and protocol. The raters were looking for instances when the casts were misused or part of inefficient or buggy code. To understand if identifiers are descriptive and concordant with their meaning, the raters were advised to look at usages of the variable as well as statements before and after the cast expression. To identify poor implementation of casts, the raters inspected the code that was referring to the cast, the variables' definitions, the variables' usages, the user-defined type definitions and the git logs for changes related to the casts. The raters consulted ISO C++ standards (ISO, 2020) to revise their understandings of each named cast operator usage.

Inter-rater agreement. Overall, the human evaluators classified on average 71 out of 271 cases as being true noting that 26.9% of cases as casts were misuses and 73.1% as cases with imprecise names. The individual ratings are 73, 71 and 71 for R1, R2 and R3, respectively. The inter-rater agreement, also called Cohen's Kappa coefficient (Cohen, 1960), was used to measure agreement level. Kappa can take values between -1 and 1. If Kappa has a value of 1, it means that the raters are in perfect agreement. A negative Kappa value means that the raters are in disagreement. Kappa coefficient is calculated as the mean value between the kappa coefficient between any two raters. The Cohen's Kappa coefficient for this evaluation between raters had the following values: 0.85, 0.89 and 0.96. This means that raters had high levels of agreement between them.

Table 3
A taxonomy of casts.

(a) Degrees of Freedom. For each degree of freedom, a value * is used if any value can be used.

Attributes	Description	Values
Cast Type	The type of named cast used	static_cast, dynamic_cast,const_cast, reinterpret_cast
Identifiers	Usage of meaningful identifiers indicative of semantics	{Specific (S), Generic (G), Value (V) eg. literals, Not Indicative/Imprecise (N)}
Direction	Whether a cast is a value or expression is being upcast or downcast	{Downcast (D), Upcast(U), Coercion-like conversion (C)}

(b) Categorisation of casts. The values within the tuple < , > represents the expressiveness of *source* and *destination*.

Category	Combination	Description	Applications or Examples
Assign Value	* , <V, S>, C	Assign values, generally of primitive type, to variables, pointers or objects	Assign default values (min, max, size, etc.), calculation results, bitwise operations, etc.
Conversions Between Classes	*, <*, *>, {D, U}	Objects or pointer of objects being upcast, downcast or the bytes being reinterpreted in related class types	Access of related class methods or members.
Specialisation	* , <*, *>, D	A generic pointer, of the type void*, being cast into a type with more specific properties	Initialisation of iterators.
Rapid Prototyping	* , <*, *>, C	These cases indicate the casts are being used to cross type boundaries for Software Engineering reasons	Serialisation or deserialisation of objects, or e.g. fileBufferto networkBuffer
Modifiability	const_cast , <*, *>, *	Modifying the const or volatileproperty of a variable	Calling non-const functions on const variables e.g. print
Check and Cast	dynamic_cast , <*, *>, *	The dynamic_cast operator ensures that the source can be converted to the destination type	Validate user input and rely on the operator for safety
Imprecise Naming	* , {<N, *>, <*, N>}, *	Any case where the source or destination does not sufficiently describe the data or the operation, which can cause confusion	Some examples are presented in Section 2.3
	* , <G, G>, *	Any case where the source or destination is too generic and does not improve the clarity of the code	

4.3. Cast taxonomy

The usage of casts can vary across projects, but for every cast operation there are a few degrees of freedom. These include, but are not limited to, the type of the cast operator, the use of variables or expressions in the *source* and *destination* and the names of these variables, as well as the tokens in the expressions. We also consider the direction of the cast as part of the degrees of freedom. The directions of the cast can be Upcast or Downcast when there is a hierarchical class relation between the variables. In addition, there will also be Coercion-like conversions between variables with no class relation, for example, cast between numerical values. The degrees of freedom are presented in Table 3. Our research hypothesis is that identifiers used for variable names and those in expressions carry semantics. Therefore, in our research, and subsequently, our taxonomy, we treat choice of identifiers as a degree of freedom. In addition, whether a cast is being used to upcast or downcast the *source* potentially influences the cast's role in the wider code; hence, in our categorisation, we treat this as an input.

Named cast operations have many applications, and we have split these into six major categories: Assign Values, Conversions Between Classes, Specialisation, Rapid Prototyping, Modifiability and Check and Cast. As it can be seen from the combination of inputs, Specialisation category is a more specific usage of Conversion Between Classes of void* pointers. On top of the six categories, there is the additional negative category, Imprecise Naming, which we capture through our tool. We established these categories after consulting background information about each named cast operator, as presented in Section 2.1. This is in addition to careful consideration following first-hand observations of cast usage during our manual evaluation of the Chromium corpus. Table 3 presents the combination of the degrees of freedom for each category, along with a description and some applications.

Casts usage in the ground truth set. To better understand the ground truth set, the raters grouped the named cast operations using the taxonomy classification. Table 4 presents the frequency of named cast cases for each category. To provide a better description of the ground truth set, some additional applications have been added to the following categories: Assign Value and Conversion Between Classes. Out of the 271 cases, 121 are in Assign Value, 53 are in Conversion Between Classes, 64 are in Specialisation, 23 are in Prototyping, 10 are in Modifiability and none are in Check and Cast. Of all the cases, only 71 cases belong in the Imprecise Names category. Based on the identifiers' quality, there are 138 cases where *source* and *destination* are specific, 20 cases where the *source* is a value and *destination* is specific, 50 cases of generic identifiers and 64 cases where the identifiers are not indicative.

4.4. Precision–recall curve (RQ_2)

To answer $\mathbf{RQ_2}$, we established a ground truth on a sample dataset in order to quantify the results of our tool. We investigate the optimal setting for the conditional entropy threshold and report the tool's performance. For the sampled dataset, the conditional entropy ranges from -0.08 to 2.58, with two cases where the conditional entropy was 1.58 and above. The Conditional Entropy is negative for cases where the joint entropy is smaller than the *source*'s entropy because the *source* subtokens can be found in the *destination*, while the *destination* is unique. To optimise the threshold we selected values between -0.2 and 1.4 in increments of 0.2. For each level, we computed the precision, recall, accuracy and F1-score.

We first give the definition of our metrics before presenting our results. Precision, presented in (3), is the proportion of the actual flagged cases from all the cases marked as flagged. (4) shows the recall, which is the proportion of flagged cases that our tool

Table 4
Sampled dataset quantified based on the taxonomy.

Categories of named cast usage	# of cases
Assign values	
Create a local copy of a value	50
Default values (max, min, size, etc.)	46
Value calculations result	10
Enumerations conversions	9
Bitwise operations result	4
Assign default address	2
Conversion between classes	
Address to address or pointer to pointer conversions	34
Upcast/Downcast	12
From address to specific data structure	7
Specialisation	
Void type pointers conversions	53
Iterator Conversions	11
Rapid prototyping	
Serialisation/Deserialisation of objects	23
Modifiability	
Modifying const property of a variable	10
Check and Cast	0
Imprecise Naming	71

Categories of identifiers’ quality	# of cases
Specific/Precise identifiers	
*Source*and *destination* are related	125
*Source*and *destination* are the same	13
***Source* is value and *destination* is specific**	
*Source*is an operation	11
*Source*is a value of a primitive type	9
Generic identifiers	
Identifiers are generic	50
Not indicative/imprecise Identifiers	
*Source*and *destination* are unrelated	13
Identifiers are generic	51

detects from all actual flagged cases. Accuracy is presented in (5) and it represents the percentage of cases that our tool correctly classifies. F-measure is the harmonic mean between precision and recall (van Rijsbergen, 2004). For our study, we utilise the F1-score presented in (6) to evaluate the tool’s performance.

$$Precision = \frac{TP}{TP + FP} \quad (3)$$

$$Recall = \frac{TP}{TP + FN} \quad (4)$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (5)$$

$$F_1 score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (6)$$

Fig. 3(a) shows the precision, recall, accuracy and F1-score based on the different threshold levels. The conditional entropy threshold represents the value when our tool starts marking a case as true. At the lowest threshold value, our tool marks all cases as true except those marked false by the preprocessing filters. With the increase of the threshold value, the precision and the F1-score also increase, while the recall decreases. When the threshold takes the value of 1.2, the F1-score and recall begin to fall, and the precision halves. This phenomenon happens because the casts are distributed based on the conditional entropy and the high number of true cases between the threshold values of 1 and 1.2. In the corpus, there are only 1,405 cases with a conditional entropy higher than 1.2. Thus, when the sampled dataset was collected, only 14 cases were selected with a conditional entropy higher than 1.2. Based on the F1-score, we select the **value 1** as the **conditional entropy threshold** for our tool. Fig. 3(a) shows that, on the sampled dataset of 271 cases, the tool has a peak precision of 81%, a recall of 90% and an accuracy of 92% in identifying correctly flagged cases.

Impact of filters. Figs. 3(b)–3(d) show the difference between precision, recall and accuracy for the tool with and without the preprocessing filters. The use of preprocessing filters improves the Precision value from 0.67 up to 0.81, and Accuracy from 0.86 up to 0.92. However, the recall is decreased from 0.93 to 0.9. The Cast from Literals filter selects 9 cases and the Descriptive Variables and Type filter selects another 25 cases. The filters prevent our tool from wrongly flagging 31 out of the 34 selected cases, while 3 cases which should have been flagged by the tool become false negatives. The tool’s reliability is improved with the small price paid of a few false negatives being inserted. The filters thus prevent the tool from flagging many more false positives.

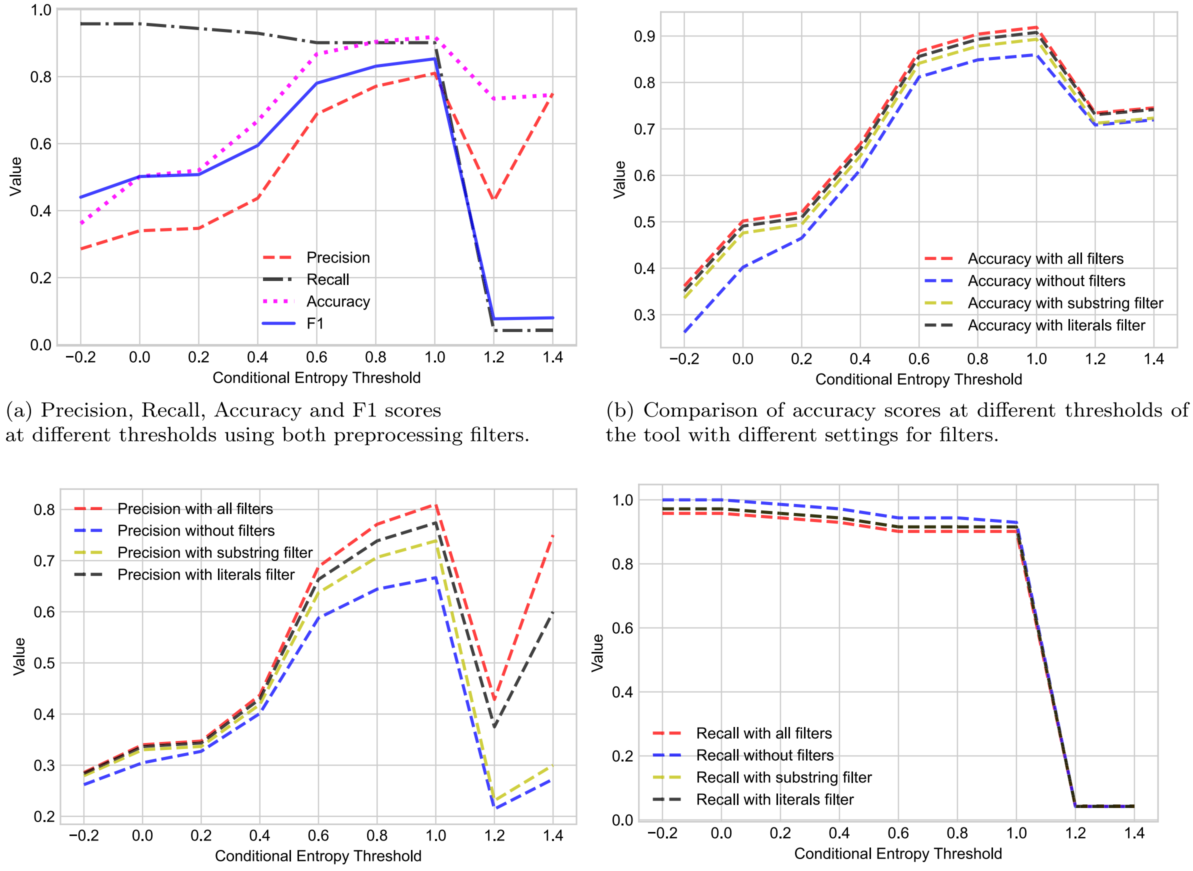
4.5. Confounding factors in usage of casts (RQ_3)

We associate the poor usage of casts with cases that present a higher conditional entropy value. Thus, to answer **RQ_3**, we inspect some long source expressions cases and we perform a correlation analysis between source length and conditional entropy value. Poor usage of casts may occur when developers have difficulty interpreting the *source* expression. This often happens for long and more complex *source* expressions. We used the number of characters in the *source* expression as a proxy of how complex a *source* is, and we investigated if casts with long *source* expressions are flagged by the tool. We chose characters over subtokens in the *source* to better capture the semantics in the identifiers. For instance, `relativeVelocity` and `relV` have the same amount of subtokens but `relativeVelocity` has more characters and is more descriptive.

Figs. 4(a)–4(d) show the conditional entropy against the number of characters in the *source*. The highlighted cases are those with a higher conditional entropy value than the threshold and are flagged by the tool. The *source* expression lengths range from 1 to 223 characters. Longer *sources* are a consequence of conditional blocks that the conditional operator ‘‘?’’, or macro functions that generate long expressions after compiler preprocessing.

We performed a correlation analysis between the length of the source and conditional entropy value. We computed the values of the following two metrics: Pearson’s correlation coefficient (r) and Spearman’s correlation coefficient (r_s). Pearson’s correlation coefficient measures if two variables are linearly related (Pearson, 1920), while Spearman’s correlation coefficient measures the rank correlation of two variables (Spearman, 1961). These coefficients can take values between -1 and 1. If the value is 0, it indicates no correlation between the variables. As the coefficient gets closer to either -1 or 1, the stronger the correlation is between variables. In our case, r has a value of -0.474, while r_s is -0.587 and both have $p < 0.001$. As the r_s value is closer to -1 than r, we can notice that the trend of the cases preserves a more monotonic correlation rather than a linear correlation. The correlation coefficient values show that the correlation between length of the *source* and conditional entropy is weak.

As can be seen in Figs. 4(a)–4(d), conditional entropy of the *destination* somewhat decreases as the *source* length increases because longer identifiers hold more information. This trend is also presented through the weak negative correlation of the two



(a) Precision, Recall, Accuracy and F1 scores at different thresholds using both preprocessing filters.

(b) Comparison of accuracy scores at different thresholds of the tool with different settings for filters.

(c) Comparison of precision scores at different thresholds of the tool with different settings for filters.

(d) Comparison of recall scores at different thresholds of the tool with different settings for filters.

Fig. 3. Tool's performance under different settings.

variables. Surprisingly, the flagged cases are largely limited to smaller *source* expressions. A likely interpretation for this is that the developers deliberately cross type boundaries, using casts to change types. It is also possible that the shorter *sources* are not expressive enough and therefore, might be misinterpreted by the developer. To better understand the nature of developers' choices, we performed an analysis of hand-picked named cast operations which is presented next.

4.6. Qualitative analysis (RQ$_4$)

To answer **RQ$_4$**, we present a number of interesting cases selected by our raters, followed by a summary of how each operator was used. We grouped these cases into the following categories: good implementations with precise names, poor implementations with precise names, good implementations with imprecise names and poor implementations with imprecise names.

Good implementations with precise names. The cast operations presented in this category are examples of good implementations of the cast operators. The names of the source and destination variables present a connection between them and the code such that they are precise to the context. The cases have the following conditional entropy values: Case 1 - 1.58, Case 2 - 2, Case 3 - 0.92, Case 4 - 1.5 and Case 5 - 0.92. Even if some cases present a large conditional entropy value, most of the time, those cases will not be flagged due to the preprocessing filters. Cases 1 and 4 are detected by the filter Casts from Literals by observing that the source is a literal. Cases 2, 3, 4 and 5 are detected through the filter Descriptive Variables and Type by noticing that the source identifier is part of the destination identifier.

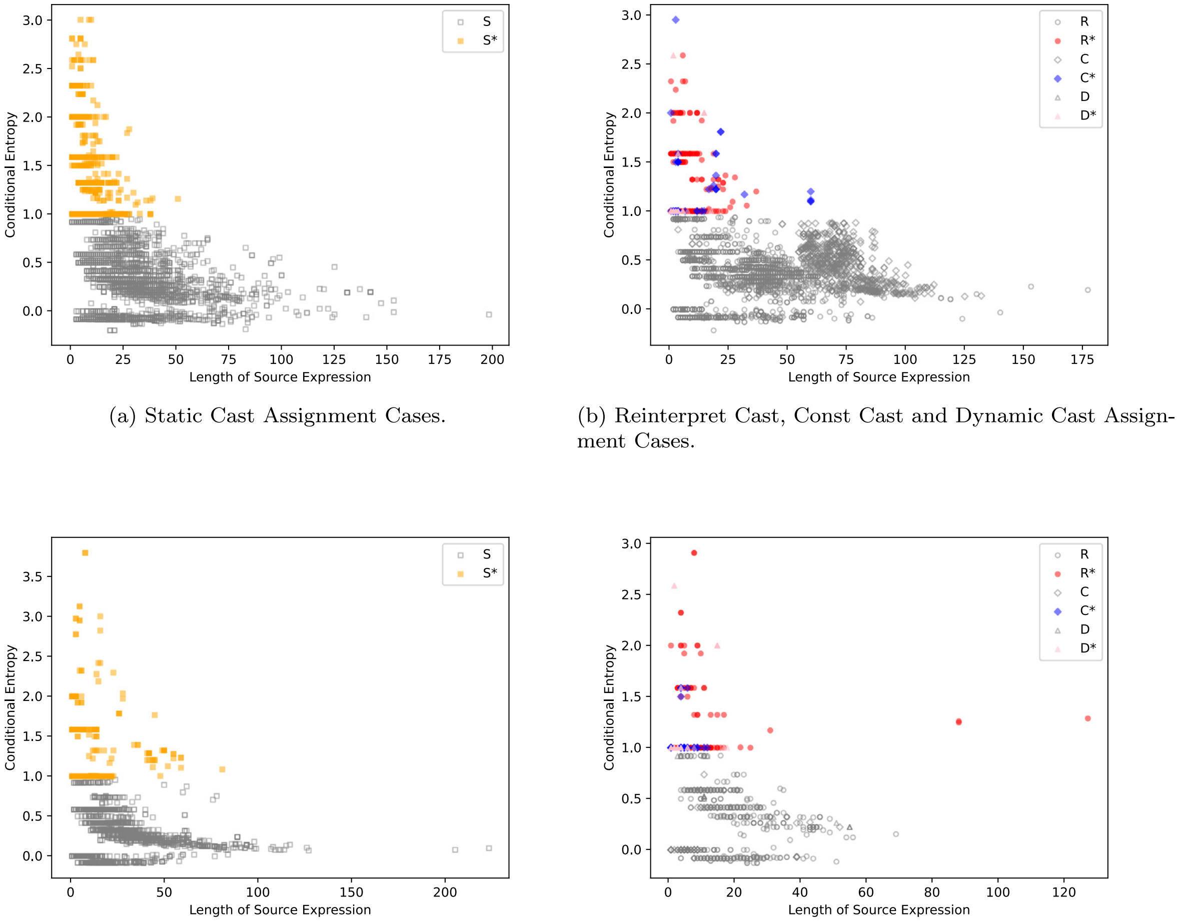
Case 1 - `static_cast` case

```
webrtc::metrics::Histogram* prev_pointer =
  rtc::AtomicOps::CompareAndSwapPtr(
        &atomic_histogram_pointer,
  static_cast<webrtc::metrics::Histogram*> (nullptr),
  histogram_pointer);

static T* CompareAndSwapPtr(T* volatile* ptr, T* old_value,
      T* new_value)
{ return static_cast<T*>(
      ::InterlockedCompareExchangePointer(
        reinterpret_cast<PVOID volatile*>(ptr), old_value,
              new_value));
}
```

Listing 7: The `static_cast` operator is required from a portability view to compare a pointer of type `nullptr_t` with another pointer of type `Histogram`. This example presents a good utilisation of `static_cast` with precise names to perform a safe pointer initialisation for `atomic_histogram_pointer`. This case has a conditional entropy of 1.58.

An example of a `static_cast` where the *source* (`nullptr`) and *destination*(`old_value`) look different is presented in Listing 7. This is an Assign Value case with *source* being a value and *destination* being specific. The Listing contains a call to `CompareAndSwapPtr` as well as the definition for the same. This method is actually called from within a macro function definition, `RTC_HISTOGRAM_COMMON_BLOCK`. The purpose of this macro function is to add the information passed to the `histogram_pointer` safely. If the memory where `histogram_pointer` points is empty, then the



(a) Static Cast Assignment Cases.

(b) Reinterpret Cast, Const Cast and Dynamic Cast Assignment Cases.

(c) Static Cast Function Call Cases.

(d) Reinterpret Cast, Const Cast and Dynamic Cast Function Call Cases.

Fig. 4. Type conversions represented by *source* expression length and conditional entropy. The star cases are the outliers.

pointer will be changed to point to the new memory address. Otherwise, the code from Lines 1–4 will ensure that it points to a `nullptr`.

The `static_cast` used on Line 3 in Listing 7 is passed as a parameter to the function `CompareAndSwapPtr`. The function call is part of a pointer declaration. The newly declared pointer `prev_pointer` will become the output of the method `CompareAndSwapPtr`. This function makes use of the API `Interlocked CompareExchangePointer` from `Windows` which is used to perform a pointer comparison and swap atomically. The code has to clear `atomic_histogram_pointer`. So, the API call ultimately will compare the pointer with a `nullptr`. If those two pointers contain different values, then it will store the value of `nullptr` in the address of `atomic_histogram_pointer`. The `static_cast` converts the `nullptr` to the type `webrtc::metrics:Histogram`* for consistency.

Since the code from Listing 7 tries to validate if `atomic_histogram _pointer` is `NULL`, it is required to compare the pointer with a null pointer literal: `nullptr`. In order to compare two pointers, they need to be of the same type and therefore, a `static_cast` is used as it is the only named cast operator which allows casts from `nullptr` to a different type. The *destination* identifier to which the named cast is bound is `old_value`. While `old_value` looks different to `nullptr` and that is why our information-theoretic analysis identified it, the method `CompareAndSwapPtr` is likely designed to be generic and accepting of many different pointer types. Therefore, this use of named cast is sound. Our tool does not flag this case based on the preprocessing filters, since `nullptr` is a literal. Additionally, the *source* and *destination* identifiers are specific to the context. Thus, we consider this case as having precise names.

Case 2 - `dynamic_cast` case

```
const __pointer_to_member_type_info* member_ptr_type =
   dynamic_cast<const __pointer_to_member_type_info*>
        (__pointee);
```

Listing 8: In order to implement an exception handler, a `dynamic_cast` is necessary to prove a cast is valid. Generic names for the variables are reasonable since the code is part of the Application Binary Interface for C++. This case has a conditional entropy of 2.

Since the sampled dataset had only one instance of `dynamic_cast`, we expanded our investigation to the entire dataset and analysed a total of 11 cases. We present two cases of Check and Cast. The first instance can be found in Listing 8. It has been extracted from *private_typeinfo.cpp* and it is part of *libc++abi* library. The use of the `dynamic_cast` operator appears in variable declarations in methods `can_catch` and `can_catch_nested`. These methods are used for exception handling and report mismatches during type conversions by checking if the result is null or not. If it is, the methods return an exception. The *source* variable, in our example, has the identifier `__pointee` , which is of the type `const __shim_type_info`*. The *destination* variable is `member_ptr_type` , which is a `const` pointer to `__pointer_to_member_type_info`, which itself is derived from the class `__pbase_type_info` a sub-class of `std::type_info` which contains information about types for variables. The names in this cast are generic, however they are still relatable to each other. Thus, we consider them to be precise. *libc++abi* implements the Application Binary Interface for *C++* and is expected to be generic to fit in with a wide spectrum of low-level transactions between the application, libraries and the operating system. The `dynamic_cast` operator is used in this case to check at runtime if the *destination* variable can take the *source*'s type while keeping the natural language identifiers as generic as possible.

Case 3 - `dynamic_cast` *case*

```
const DecimalFormat *decFmt = dynamic_cast<const
      DecimalFormat *>(&fmt);
```

Listing 9: An adequate example of `dynamic_cast` performing a down-cast conversion with meaningful names for *source* and *destination*. This case has a conditional entropy of 0.92.

The second example of Check and Cast is presented in Listing 9. The snippet is from the file *upluralrules.cpp* in the *ICU* (International Components for Unicode) module. The *source* variable is `fmt` with the type `const class icu_64::NumberFormat*` which captures the format of the expression. The *destination* variable is `decFmt` and it has the type `const class icu_64::DecimalFormat*` . The *destination*'s type class `DecimalFormat` inherits from *source*'s type class `NumberFormat` (Unicode, 2020) and this is an example of a down-cast operation which is verified at runtime. If the checks fail and `decFmt` is `NULL`, the method continues to check for other known formats. The *ICU* module handles a wide variety of data types. Even for numerics, which is the focus of our example, there are several different types that need checking: `int32_t`, `double` and `FixedDecimal`. Most of these values are only available at runtime and therefore, the developers prefer to insert explicit checks through the `dynamic_cast` operator. The identifiers in this case reflect the type specialisation that is happening through the `dynamic_cast` operator. This specialisation provides us with precise names for the context. This is an example where type conversions are used judiciously with clear objectives and the names reflect the type conversion that is taking place. Further, the use of `dynamic_cast` operator makes the type conversion safe at runtime. The preprocessing filters result in our tool not flagging this case due to the *source* identifier being a substring of the *destination* identifier.

Case 4 - `const_cast` *case*

```
TimeZoneNamesImpl *nonConstThis =
      const_cast<TimeZoneNamesImpl *>(this);
```

Listing 10: A fair example of how `const_cast` is used to obtain a non `const` object from the `const` pointer `this`. The generic identifiers for *source* and *destination* indicate the code's purpose. This case has a conditional entropy of 1.5.

There were only five cases of Modifiability in the sampled dataset. Four cases belong to the library *ICU* in two different files: *tznames_impl.cpp* and *tzfmt.cpp*. For these cases, the *source* identifiers are generic and partially different compared to the *destination* identifiers. Listing 10 presents one of the four cases from the file *tznames_impl*.cpp. The *source* variable is the pointer `this` which is an instance of the class encapsulating the statement and has the type `const TimeZoneNamesImpl*`. The *destination* variable is a pointer called `nonConstThis` which does not have the qualifier `const` in its type. The chosen identifiers for *source* and *destination* reinforces our hypothesis that identifiers carry meaning. Here, the getters in the encapsulating class need to maintain the integrity of the original object. Thus, the desired values need to be extracted from a non `const` object derived from the pointer `this` using a `const_cast` operator. This is an instance where explicit casting is being used judiciously, clearly indicating its purpose through meaningful identifiers. Our tool does not flag this case due to the preprocessing filters recognising that the *source* identifier is a substring of the *destination* identifier.

Case 5 - `reinterpret_cast` *case*

```
1 auto memberBuffer = reinterpret_cast<DawnTextureFormat*>
      (*buffer);
2
3 for (size_t i = 0; i < memberLength; ++i) {
4    memberBuffer[i] = record.colorFormats[i];
5 }
```

Listing 11: An example of `reinterpret_cast` that is used in serialisation/deserialisation to be able to iterate over an enumeration to reduce the amount of code developers had to write. The identifiers used are precise and the cast is well implemented. This case has a conditional entropy of 0.92.

Listing 11 presents a case of Rapid Prototyping which uses `reinterpret_cast`. This snippet is from component Dawn in file *WireCmd_autogen.cpp* and is one of 13 similar cases. The file is generated from *WireCmd.cpp* using the build system and contains serialisation and deserialisation functions. The generated file is large with 14,000 lines of code and has a total of 200 type conversions which have the same identifier for *source* variables and also for the *destination* variables. The *source* identifier is the string `buffer` and in most cases, it is a pointer to a pointer for `char`. There are cases when the *source* variables have additional type qualifiers such as `const volatile`. The *destination* variable is `memberBuffer` and it is declared with the type `auto`. We observed that the *destination* type varies from pointers to numeric types such as `unsigned long long` to pointers for structures and enumerations. The casts are part of assignment expressions in which the `memberBuffer` is initialised with a part of the `buffer`. We consider the names to be precise because the *destination* identifier is more specific than the *source* and both names provide context to the cast. As with the previous two cases, the *source* identifier is a substring of the *destination* identifier and as such, the tool does not flag this case.

The purpose of these casts is to serialise and deserialise a variety of different structures for the component *Dawn*. In other words, the methods provide the functionality to convert objects to streams of bytes and recreate the objects when needed. Since the universe of types to be serialised is large, developers have relied on macros to serialise/deserialise objects. The example selected in Listing 11 presents the `buffer` which is converted in the type `DawnTextureFormat`. The target type is an enumeration. Similar to the example from Section 2.2, Lines 2–4 iterate over the enumeration. While the use of macros is preferred for serialisation and deserialisation, given the massive number of types that need to be serialised or deserialised, macros provide little insight into the actual role of the casts. Nonetheless, the generated file can be created from only 700 lines of code which contain macros. The use of `reinterpret_cast` in this case is clearly beneficial from a software reuse point of view and leads to a decrease in the amount of code. On the other hand, the named cast operator is used to bypass the lack of an iterator for the enumeration type, which if not done correctly, can be pernicious as `reinterpret_cast` comes with no semantic checks at all and as discussed above, enum types may not be contiguous in the first place.

Poor implementations with precise names. The cast operations presented in this category are examples of poor implementations of the cast operators, while the names are still precise to the context. The cases have the following conditional entropy values: Case 1–1.32, Case 2–2, Case 3–1.58 and Case 4–1.58. The tool flagged all the cases due to a higher conditional entropy than the threshold value of 1. The only exception is Case 4, which due to the Casts from Literals filter, would be erroneously not flagged.

Case 1 - `static_cast` case.

```
address.bytes_[i++] = static_cast<uint8_t>(next_octet);
```

Listing 12: An example of `static_cast` operator used in function `ParseV4` to parse an IPv4 address. The identifiers are precise, but the cast was unnecessarily implemented. The `ParseV4` function has been refactored and the conversion has been replaced with the function `sscanf` to parse the values. This case has a conditional entropy of 1.32.

Listing 12 presents an Assign Value case of the `static_cast` operator in the component *Base*, in file *ip_address.cc* inside the method `ParseV4`. This method is used as part of the constructor for the class `IPAddress` to extract the IPv4 address from a string. The named cast operation in Listing 12 is part of a variable assignment. Although the *source* and *destination* identifiers are selected because they look different, we need to understand how they are used to assess whether a named cast is necessary here. We studied how the *source* and *destination* identifiers are used and found that the input string for `ParseV4` is split in octets in order to be parsed and added to the IPv4 address. The *source* identifier is `next_octet` of type `uint16_t`, which represents one byte of the IPv4 address. The destination variable is `address.bytes_` where `bytes_` is a member of the class `IPv4`. Specifically, it is an array of type `array<uint8_t, 16>`. The array has the length 16 since `IPAddress` can also have the IPv6 format. These identifiers are self-explanatory within the code, thus we consider them precise. The implementation of `ParseV4` does not seem to be erroneous. However, the use of the `static_cast` operator is unnecessary since the conversion from string to octets can be done using the built-in type transformation type. Developers can use functions such as `sscanf` to read parts of the formatted string and return directly the desired output. In fact, this is exactly what the developers did in later versions of the implementation: the `ParseV4` function has now been refactored (Chromium, 2020) and updated to use `sscanf`.

Case 2 - `reinterpret_cast` case

This case presents two similar cases that appear in different components. The cases belong to Conversions Between Classes category and use the `reinterpret_cast` operator. Two different *source* identifiers (`&data`, `&hbi`) are bound to the same *destination* identifier (`host_info_out`). Listing 13 contains the calls and the signature for the function `host_statistics` and `host_info`. These method calls have been collected from the files *process_metrics _mac.cc* from *Base* component and *audio_low_latency_input_mac.cc* from *Media* component. The functions `host_statistics` and `host_info` are defined in the *Mach* library which contains services and primitives for the OS X kernel.

The role of the functions `host_statistics` and `host_info` is to retrieve host-specific information. The function `host_statistics` in Line 2 obtains information about virtual memory for a host. The `host_info` method in Line 10 retrieves basic information about a host such as the number of current physical processors for the host. Both methods return a variable `kr` of type `kern_return_t`. This variable is an integer which maps to a list of generic errors. If the method is successful, then `kr` would have the value `0`. Otherwise, it would have a different value which represents a specific error. Most of the methods from the *Mach* library follow the same coding conventions and they have a similar format to these two functions.

```
1 //check the total number of pages currently in use and
       pageable.
2 kern_return_t kr = host_statistics(host.get(),
       HOST_VM_INFO,
3    reinterpret_cast<host_info_t>(&data), &count);
4
5 kern_return_t host_statistics(host_t host_priv,
       host_flavor_t
6    flavor, host_info_t host_info_out,
7    mach_msg_type_number_t *host_info_outCnt);
8
9 //retrieve the number of current physical processors
10 kern_return_t kr = host_info(mach_host.get(),
       HOST_BASIC_INFO,
11    reinterpret_cast<host_info_t>(&hbi), &info_count);
12
13 kern_return_t host_info (host_t host, host_flavor_t flavor,
14    host_info_t host_info_out,
15    mach_msg_type_number_t *host_info_outCnt)
```

Listing 13: An example of `reinterpret_cast` operators with precise identifiers. These are used to allow functions to take pointer parameters that can point to two different data structures. Thus, the function returns a different result based on the variable `flavor`. It is not desirable to have pointers which point at different types structures. This case has a conditional entropy of 2.

The *source* variable for the first case has the generic identifier `&data`. Its type is `vm_statistics_data_t` which is a pointer to the structure `vm_statistics` and contains statistics on the kernel's use of virtual memory. The *source* identifier for the cast from Line 10 is `&hbi` which is the acronym for its type, `host_basic_info`. `&hbi` is the address of a structure `host_basic_info` which is used to present basic information about a host. The two casts from Listing 13 have the same *destination* identifier: `host_info_out` with type `host_info_t`.

`host_statistic` can hold two different types of structure: `vm_statistics` for virtual memory information and `host_load_info` for host processor load information. The `flavor` keeps track of the type of statistics desired. In this way, the functions will treat each *destination* variable differently based on the variable `flavor`. Implementing the functions in this manner allows them to perform different operations based on the parameters passed. The *destination* identifiers are identical since the functions `host_statistics` and `host_info` follow the same coding conventions and have a similar format. Unfortunately, if the developer is not careful to pass the correct match between the type and the `flavor` as parameters to the functions, it may lead to a crash. Despite the *source* and *destination* identifiers being meaningful for the context, this is a case where rigorously adhering to a coding convention can cause confusion during development.

Case 3 - `reinterpret_cast` case

The code from Listing 14 presents a Rapid Prototyping case through the use of a `reinterpret_cast` in Line 4. The snippet is collected from component V8 in file *api.cc*. The *source* variable is a `void*` pointer with the identifier `info` , while the *destination* variable is a shared pointer with the identifier `bs_indirection` of type `std::shared_ptr<i::BackingStore>*`. To understand this case, first, we need to understand what the type `BackingStore` is. In caching, a backing store is represented by the copy of a data in the memory, more specific in our case, a copy to an

```
// The backing store deleter just deletes the indirection,
   which downrefs
// the shared pointer. It will get collected normally.
void BackingStoreDeleter(... void* info) {
      std::shared_ptr<i::BackingStore>* bs_indirection =
          reinterpret_cast<std::shared_ptr
          <i::BackingStore>*> (info);
      ...
      delete bs_indirection;
}
```

Listing 14: An example of reinterpret_cast that allows deletion of the indirection of the shared pointer. The case has precise identifier names taking into account the context. Due to the complex design of the BackingStore, the code was refactored and the cast was removed. This case has a conditional entropy of 1.58.

ArrayBuffer (Google, 2021). The named cast operator is used to retrieve the shared pointer for BackingStore data, which will be deleted later in the same function. The BackingStore pointer is a shared pointer that can be accessed from the V8 and the Embedder components of Chromium and generates a lifetime management problem when both components hold pointers to the backing store data. The code complexity is increased since the components can resize the shared memory or transfer ownership from one component to another. The unsafe ownership model of BackingStore is prone to errors, such as memory leaks and access of the pointers after deleting them, which has eventually led to various bugs (Chromium, 2021d,e).

The problems have been solved by refactoring the ownership model and making the BackingStore to own the shared pointers (Google, 2021; Chromium, 2021a). The previous implementation required each component to delete its shared pointer instance through the method BackingStoreDeleter. The new version of the BackingStore class counts the shared pointers references and if the count reaches zero, then the BackingStore will delete the pointer. The named cast operation, along with the function BackingStoreDeleter , was removed in the new implementation (Chromium, 2021a). While the named cast operation was not directly causing the bugs, we can definitely say that it added complexity to the code by asking each component to delete its shared pointer instance, and eventually the code led to bugs. Even though the *source* and *destination* identifiers (info and bs_indirection) are different, we can notice there is a semantic relation between the identifiers and they are precise for the context. info refers to the data and bs_indirection refers to backing store pointer which is the copy of the data. If a semantic perspective were to be considered, it is likely that this case would not have been identified.

Case 4 - reinterpret_cast case

```
1 // Old implementation
2 #define F(name, str)
3 ...
4   HashMap::Entry* entry =
5       string_table_.InsertNew(name##_string_,
            name##_string_->Hash());
6   entry->value = reinterpret_cast<void*>(1);
7
8 // New Implementation
9 #define F(name, str)
10 ...
11   string_table_.InsertNew(name##_string_,
         name##_string_->Hash());
```

Listing 15: An example of reinterpret_cast using precise identifier names. This case was used to add a default value to the HashMap entries due to the lack of empty value behaviour. The code was refactored by implementing empty value behaviour and the cast was removed. This case has a conditional entropy of 1.58.

Listing 15 presents two versions of a macro function *F* collected from the file *ast-value-factory.cc* of component AST. The first version contains an Assign Value case using the reinterpret_cast operator on Line 6. The *source* expression is an integer literal representing the value 1. The *destination* variable is a void* pointer with the identifier entry->value and it points to the value of an entry in a HashMap. The function *F* is used in the initialisation of HashMap objects and each entry is initialised with value 1. The *destination* identifier is self-explanatory in the context, and so we consider this case as having precise names.

The second version of the macro function *F*, which is a refactored version (Chromium, 2021c), does not contain the named cast operation. With the lack of the named cast operation along with the information from the commit, we can tell that the new implementation of the HashMap supports objects with empty values without causing any errors. The named cast operation in the first version was a workaround, without a proper way of defining the behaviour if the entries did not have values. This means that the code in the first version was error-prone in the case of empty values. A proper implementation shows that the named cast operation is not needed in the current case. Due to the preprocessing filters detecting a literal, this case would be erroneously not flagged.

Good implementations with imprecise names. The cast operation presented in this category is an example of good implementation of the cast operator, while the names are imprecise. The tool flags Case 1 because it has a conditional entropy value of 1.58.

Case 1 - const_cast

```
1 size_t frame_count = base::debug::TraceStackFramePointers(
2     const_cast<const void**>(frames),
3     max_entries, skip_frames);
4
5 size_t TraceStackFramePointers(const void** out_trace,
6     size_t max_depth, size_t skip_initial)
```

Listing 16: An adequate example of how a const_cast operator is used to add the const qualifier to a variable. The *destination* identifier could be renamed to be more meaningful. This case has a conditional entropy of 1.58.

Listing 16 presents a fair example of a Modifiability case with Non Indicative identifiers. This example is taken from component *Base* and belongs to the method *CaptureStackTrace* which is used to collect frames in the execution stack. It is interesting and complements the one discussed in Listing 10 because the type qualifier const is being added to a value in this case. In this case, the type conversion is a parameter for the function call TraceStackFramePointers. The function in Lines 1–3 returns the total number of the frames for the stack. The *source* identifier is frames which has the type void** and it represents the pointer to the stack frames. Line 5 of Listing 16 shows the function declaration. The *destination* identifier is out_trace with the type const void**. Being able to check the stack is vital for debugging but at the same time, the stack should be protected during debugging. The const_cast is required in this case to protect the stack frames from inadvertent manipulation while the developer is inspecting the stack. Here, we see an instance where the cast is necessary but the identifier for the *destination* is not descriptive enough. A more meaningful *destination* identifier would allow for greater clarity in the code.

Poor implementations with imprecise names. The cast operations presented in this category are examples of poor implementations of the cast operators, while the names are also imprecise. Case 1 and Case 2 have conditional entropy values of 2.32 and 1.58. Since the values are higher than the threshold value, those cases are flagged by the tool.

Case 1 - `static_cast` case

```
((unsigned int*)element)[0] = static_cast<unsigned int>(r);
((unsigned int*)element)[1] = static_cast<unsigned int>(g);
((unsigned int*)element)[2] = static_cast<unsigned int>(b);
((unsigned int*)element)[3] = static_cast<unsigned int>(a);
```

Listing 17: Example of how a `static_cast` is used on primitive types. The *destination* variable is originally a `void` pointer and may potentially be misused if the developer is unaware of the various types it can represent. The lack of precise *destination* identifier increases the chance of such misuse. This case has a conditional entropy of 2.32.

The code from Listing 17 presents a set of four Rapid Prototyping cases through `static_cast` conversions collected from the component *Swiftshader* from the file *Surface*.cpp. We identified the casts because the *source* identifiers are very short compared to the *destination* identifiers. These casts are inside a method `write` which contains a `switch` statement that writes the colour values (RGBA format) to a data structure. The *source* identifiers are `r`, `g`, `b` and `a` of type `float` , which represent the colours red, green and blue, and the opacity value (alpha). The *destination* identifiers, which are originally `void` pointers, have the generic name `element` because it may point to arbitrary data types. However, notice in Listing 17, `element` has been implicitly cast to point to an `unsigned int` to match the type for the desired *destination* type. Implicitly casting `void` pointers at the point of use can be confusing. This could lead to the variable `element` being treated differently, assuming it has another type. Further to this, the *destination* identifier is not precise enough which further adds to the confusion. We have found 45 similar conversions in the same `switch` statement.

Case 2 -`static_cast` case

```
1 template <typename T>
2 std::vector<Path *> GatherPaths(..., const void *paths
3 ...
4 const auto *nameArray = static_cast<const T *>(paths);
```

Listing 18: Example of a `static_cast` case with an imprecise *destination* identifier. This cast was part of complex code that led to inefficient code. Under specific circumstances, the code crashed by retrieving information from an empty pointer. This code was refactored and the cast was no longer used. This case has a conditional entropy of 1.58.

Another Rapid Prototyping case using `static_cast` is presented in Line 4 in Listing 18, which belongs to the file *Context.cpp* from component *libANGLE*. The *source* variable is a pointer of type `const void*` with the identifier `paths` and it represents a vector of potential paths from the Render Tree. The *destination* variable is a pointer of type `const auto*` with the identifier `nameArray`. This conversion is required to allow the conversion of the `paths` vector in a target template type. The template type is used as an argument to the named cast operator in Line 4 and it appears in the function template declaration on Lines 1–2 in Listing 18. The role of the function `GatherPaths` is to iterate through all the `paths` and returns their names. This case belongs to a larger and more complex piece of code that has the functionality to validate the command buffer at path rendering. The developers decided to stop supporting this feature since this rendering method had a worse performance compared to the other rendering methods (Chromium, 2021b). In addition, under specific circumstances this functionality was trying to retrieve information from an empty pointer which was leading to a crash (Chromium, 2021f). This example shows that a named cast conversion can be used correctly, but it might also add complexity to the code, leading to inefficient and error-prone code. In addition, the *destination* identifier is imprecise because it is generic compared to the *source* identifier. This further decreases the code's clarity.

Usage of casts (RQ$_4$). In this work, we presented a summary of the findings from the named cast operators study. We have identified: two cases of iteration over enumeration types (Listing 6 and Listing 11), two cases of poorly named variables (Listings 17 and 11), two instances of anti-patterns that have been refactored in later versions of the software so that the named cast operators were no longer used (Listings 12 and 15), two cases that increased the complexity of the code which led to poor quality code and bugs (Listings 18 and 14), two cases that enabled a function to and two good programming practices for protecting values stored in variables (Listings 10 and 16).

The operator `static_cast` is the most versatile and most widely used operator for explicit type conversions. In Listing 6, we discovered the use of the `static_cast` to iterate over an enumeration, which is an abuse of the enumeration type and an inefficient implementation. Listing 7 presents a good use of `static_cast`, demonstrating how it can be used to provide safety during pointer initialisations. We also found examples where named casts were used as a quick workaround. The case from Listing 12 showed a cast which has been removed in recent versions. The case from Listing 17 shows conversions between primitive types, which in most cases is harmless. However, the *destination* variable is a `void` pointer which can point to many types and lead to type confusions. The last case from Listing 18 shows a correct use of the `static_cast` operator being part of complex code that led to inefficient code and even to a bug.

The `reinterpret_cast` operator is used mostly for pointer to pointer conversions as it is the most permissive. Listing 13 presented two examples of conversions of two different pointer types bound to a *destination* which has the same name. Using the same name to store data of different kinds is not desirable and we believe the code can benefit from variable renaming. In Listing 11, we presented an example of serialisation/deserialisation where the developers have relied on `reinterpret_cast` to be able to deal with a diversity of objects. There is a strong software engineering reason to do so as it is essential to keep the interface to the serialiser and deserialiser generic to be able to deal with any data type. The case from Listing 14 shows another example where complex code led to bugs. After the bugs were solved, the code was refactored and the named cast was completely removed. Last case shows the use of a `reinterpret_cast` as a quick workaround to not develop the behaviour for empty values case for entries of a HashMap. This named cast operation was also removed in the recent versions.

`dynamic_cast` operators are used infrequently. They are used when the developer is unsure if a conversion is possible or not. In this way, the runtime checks will confirm whether the casts are valid. An example where it is mandatory to prove a cast is valid appears in the implementation of an exception handler showed in Listing 8. Another essential use-case of `dynamic_cast` operator is for downcasts. The component ICU contains the most dynamic conversions and they are used for downcasts. Section 6 discusses some solutions to avoid the expensive dynamic cast. However, the question of why from all Chromium's components only ICU has implemented its downcasts with `dynamic_cast` remains unanswered.

The operator const_cast is used for software engineering reasons and security reasons. Even if this operator can introduce undefined behaviour as presented in Section 2, the analysed cases were adequately implemented. We have identified two const_cast usage patterns from the analysis. One pattern appears when an object tries to access itself through the pointer this in a function declared with the qualifier const. The const functions will make the pointer this also have the qualifier const. However, there are times when the const this pointer needs to be passed as a parameter to non-const functions. Listing 10 shows an example where an explicit conversion was performed in a getter to obtain information from an object. Another use-case appears when some non-const variables need to be protected against modification in specific methods. In order to do so, the const_cast is used to add the const qualifier. Listing 16 shows how a stack is passed as a parameter to a function after the conversion. The motivation behind the use of some const type conversions comes from the use of third party libraries.

5. Threats to validity

Internal threats. The results of the manual investigation and the findings of the named casts operators usages are influenced by the subjective experience of the raters. We tried to minimise this bias by using three raters with experience in C++. As mentioned in Section 4.2, each rater consulted the ISO C++ Standard (ISO, 2020) to understand how the named cast operators should be used, and only after this did the raters provide feedback on the sample data. After each rater individually performed their initial evaluation, they selected together the interesting cases presented in Section 4.5.

External threats. Our tool is subject to analyse code where variable names are chosen carelessly. In an ideal world, the natural language channel provides enough context to understand the code's purpose. Our approach relies on the connection between the identifiers to detect cast misuses and the tool performs better if the identifiers are meaningful. In a scenario where the names are chosen carelessly, our tool might identify fewer cases of casts misuses, but it will identify more cases of imprecise names. In many cases, cast misuse can be overshadowed by imprecise naming. This is overcome by initially identifying imprecise naming, essentially forming the first stage of a two stage refactoring — clarification of intent followed by validation of intent. However, our tool will also detect some false positives based on the nature of the approach. Developers might decide in some cases that generic or different names are appropriate for the *source* and *destination* identifiers. In such cases, these casts would be flagged despite the identifiers being meaningful to the code.

6. Related work

Research into type systems accelerated with Luca Cardelli's seminal and accessible papers on type theory (Cardelli, 1991; Cardelli and Wegner, 1985; Cardelli, 1988). He lucidly explained how type systems could help us write better programmes with fewer bugs. Some of that research also discusses properties of types in object-oriented programming. Wise (1996) presented an analysis of the explicit type casts operators for C++ with details of each type of operator. Gibbs and Stroustrup (2006) proposed a method to implement dynamic casts, which is an expensive operation, for systems where performance is critical. Dechev et al. (2008) have demonstrated the efficiency of the Gibbs and Stroustrup implementation by using it as a baseline while also improving the performance by a factor of two.

Type casting studies. In term of the effects, there are a significant number of research papers that present the study of the undefined behaviour introduced by type conversions (Hathhorn et al., 2015; D'Silva et al., 2015; Dietz et al., 2012). Undefined behaviour can have many causes and some of them are due to type conversions. For instance, during the execution of a dynamic_cast, the programme needs to check the pointer's type. This is done by the dereferencing the pointer, and this case is undefined behaviour (LLVM, 2019; Regehr, 2019). Compilers will capture some cases of undefined behaviour for which they will generate warnings, but not all of them (Hathhorn et al., 2015). For this reason, developers need tools and techniques to verify their code.

Pradel and Sen (2015) have done an empirical study over the implicit casts for JavaScript. They proved that those type conversions are in general harmless and developers use them correctly. This can be translated as most of the times, implicit casts are safe to use. However, there is contradicting evidence that unrestrained named casts or explicit casts can have undesirable effects. Tools have been researched and developed to detect such casts. Lee et al. (2015) present CAVER, which is a tool to identify poor practices in casting and also discussed their security implications. The tool analyses C++ code and focuses on the unsafe uses of the static_cast and dynamic_cast. This work has provided a good background to understand how named casts can go wrong. Their tool's evaluation, much like ours, is performed on the code from Chromium. Jeon et al. (2017) provide another tool HexType that performs well at detecting badly implemented casts. They have implemented HexType using low-overhead data structures and compiler optimisations to minimise the required resources. Mastrangelo et al. (2019) provided an empirical study of type conversions for Java. The target of their research is to discover when and how developers use an explicit cast. This is done through discovering and presenting 25 patterns of cast-usages from real-life Java code. This paper is the closest to our work, but unlike us, it does not use any signal from the natural language identifiers to detect anti-patterns.

Dual-channel research. Knuth (1984) proposed a paradigm shift in programming, which is commonly known as Literate Programming, where writing code to instruct a computer is secondary to presenting it to human beings. In Literate Programming, each programme contains its explanation in natural language intermixed with sections of code. Knuth presented the system WEB, which is a literate programming language comprising of a document formatting language (TEX) and a programming language (PASCAL). Literate programmes contain a human-readable explanation interspersed with code which is automatically picked up by the WEB system to produce an executable. At the same time, WEB enables the inclusion of powerful features such as pictures, equations, tables, and others in the natural language part of Literate programme. Thus, the natural language information remains in harmony with the software itself.

Literate programming laid the foundation for novel research directions in Software Engineering that drew upon advances in Natural Language Processing. Hindle et al. (2012) proposed the *naturalness hypothesis* for software which noted that large programmes can be repetitive and can be modelled with techniques that capture repetition such as n-grams. They noted that code is analogous to natural languages in the way it tends to repeat. Such repetitive patterns can be harvested and interpreted as statistical properties that can be used to develop better software engineering tools. They used this observation to build a statistical language model over a large corpus to improve code completion. An n-gram language model was built using token sequences, which included natural language information in the form of identifiers, from open source code. The model was used in a plugin to complete code for Eclipse IDE which performed better than the Eclipse's completion system at that time.

Source code is normally written for it to run on a device. But, the same code is also written for developers who maintain or improve the application. Therefore, a large part of the code semantics is embedded in the communication channels between developers i.e. the natural language identifiers that are chosen and the comments that are written in the code. Based on this insight, Casalnuovo et al. (2020) described two communication channels in source code: the algorithmic channel (AL) and the natural language channel (NL). The algorithmic channel comprises of all the instructions written by the developers which will be executed by a computer. The natural language channel, which consists of identifiers and comments, provides information about the purpose of the code in a human-readable format. The relation between the AL and NL channel can be utilised to improve software analysis tools.

Pârundefinedachi et al. (2020) have developed a tool called HEDDLE to detect and separate tangled commits into atomic concerns. HEDDLE generates a graph data structure that encodes different versions of the programme and annotates the data flow edges using the natural language information from the source code. HEDDLE performs faster and is more accurate in the detection of tangled commits than the previous state-of-the-art. Pârtachi et al. (2020) have also developed a technique called POSIT, which adapts NLP techniques for tagging between code and natural language. POSIT can generate more accurate tags for both source code tokens and natural language words than the previous state-of-the-art.

Dual-channel research on extracting meaning from names. Identifier names represent the majority of tokens from the source code. Butler et al. (2010) have shown through an empirical study on Java applications that there is a direct relation between the naming quality of identifiers and source code quality. Thus, poor named identifiers show a lack of understanding of the problem, which is translated into poor quality software. The authors measured the quality of identifiers based on identifier naming guidelines and subtokens comparison to Java and application specific terms. Even if the subtokens' semantic meaning is ignored in the analysis, this empirical study proves that the relation between the dual-channel information is not entirely harvested and applied in software analysis tools.

Arnaoudova et al. (2013) defined the first *Linguistic Antipatterns (LAs)* which are a collection of recurring poor practices in names of identifiers, documentation and code. LAs detect inconsistencies based on semantic relations between names of variables, methods signatures and the documentation. Arnaoudova et al. (2016) performed a study to discover how developers perceive LAs based on the familiarity to the code. 69% of the external participants (with no familiarity to the code) and 51% of internal participants (on the development team for the code) rated LAs as poor development practices that should be avoided. In addition, Fakhoury et al. (2018) explored how developers' cognitive load is influenced by the existence of LAs in code snippets. The authors tracked the cognitive load using functional brain imaging and eye movement tracking. The participants had to investigate code snippets and identify semantic faults. The code snippets altered to contain LAs significantly increased participants' cognitive load. Linguistic Antipatterns, as proposed by Arnaoudova et al. motivate the development of automatic tools such as the one we have developed in this paper.

Dash et al. (2018) used dual-channel constraints to mine conceptual types from identifiers and assignment flows between them. Conceptual types are types that are latent in the programme but not explicitly declared by the developer. Generally, conceptual type corresponds to the actual types, but there are cases where they can be latent. For instance, password and username may have the same type, *string*, but their conceptual types are different. If a password, which is generally a highly protected field, was declared the same way as the username, it would lead to a vulnerability.

Pradel and Sen (2018) developed a learning approach, called DeepBugs, for discovering bugs based on the semantic meaning of the identifier names. This approach uses embeddings, a vector representation for identifiers, which preserve the semantic similarities between identifiers. The bug detection is treated as a binary classification problem. DeepBugs approach trains a classifier to distinguish correct code from incorrect code. The training data consist of correct code and incorrect code generated by the authors. The bug detectors use the embeddings from the training phase to discover bugs. Three bug detectors were built based on this approach to discover accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations. The bug detectors have a high accuracy between 89% and 95% to distinguish correct and incorrect code. The bug detectors are also very efficient, taking less than 20 ms to analyse a file. False positives are inevitable in static analysis tools; however, the bug detectors have a 68% true positive rate.

Another approach that makes use of the semantic meaning of the identifier names is presented by Bavishi et al. (2018) and it is called Context2Name. JavaScript code is usually deployed in a minified version in which the identifiers are replaced with short and random names. Context2Name is a deep learning-based technique that predicts identifier names for variables that have a minified name. This technique generates context vectors for each identifier by inspecting five tokens before and after the identifier's occurrence. The context vectors are then summarised in embeddings. Those embeddings are used by a recurrent neural network to predict natural names for the minified variables. Context2Name predicts correct identifiers with a 47.5% accuracy of all minified names and it predicts 5.3% additional identifiers missed by the state-of-art tools.

The improvements made by the dual-channel research shows how much potential the dual-channel information presents for software analysis. Our study uses similar approaches to the work from dual-channel research, but on a different problem. Hints of the developer's intent have been extracted from natural language information to guide the detection of anti-patterns of named casts.

7. Conclusion

Identifiers can add insights into programme semantics and can be used for sanity checking cast operations where the developers cross type boundaries. Our work provides insights into how developers use named casts and can be used to prioritise refactoring for named cast operators. The approaches presented in this work are lightweight and can be easily used during development, as an IDE plugin during development or either at the review stage. While our tool is for any C++ project, it depends on descriptive names to make decisions and therefore, can potentially perform better for projects with well-established coding guidelines. These coding guidelines need to contain precise details for how the identifiers should be named and regular code reviews should ensure that developers are following these guidelines.

The evaluation results and the in-depth analysis demonstrate the potential of the dual channel approach for programme analysis. Despite compelling results, there is still room for improvement in our approach. We did not find value in directly applying techniques from Natural Language Processing for analysing subtokens in identifiers due to the domain-specific nature of our corpus. For this reason, future work on our tool could involve combining domain-specific knowledge with dual channel analysis to improve the reliability of our tool. Despite this, our work also provides a strong foundation to help richer forms of static analysis scale by using novel programme representations derived from the natural language channel in software.

CRediT authorship contribution statement

Constantin Cezar Petrescu: Conceptualization, Methodology, Software, Validation, Writing, Investigation. **Sam Smith:** Data curation, Visualization, Writing – review & editing. **Rafail Giavrimis:** Data curation, Writing – review & editing. **Santanu Kumar Dash:** Supervision, Project administration, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Earl T. Barr has supervised one of the authors.

Data availability

Data is shared on a GitHub repository, and it can be submitted if requested.

Acknowledgements

This research was partly funded by EPSRC, United Kingdom grant number EP/W015927/1.

References

Anderson, Carolyn J., 2010. Central limit theorem. In: The Corsini Encyclopedia of Psychology. http://dx.doi.org/10.1002/9780470479216.corpsy0160.

Arnaoudova, Venera, Di Penta, Massimiliano, Antoniol, Giuliano, 2016. Linguistic antipatterns: what they are and how developers perceive them. Empir. Softw. Eng. 21 (1), 104–158. http://dx.doi.org/10.1007/s10664-014-9350-8.

Arnaoudova, Venera, Di Penta, Massimiliano, Antoniol, Giuliano, Guéhéneuc, Yann-Gaël, 2013. A new family of software anti-patterns: Linguistic anti-patterns. In: 2013 17th European Conference on Software Maintenance and Reengineering. pp. 187–196. http://dx.doi.org/10.1109/CSMR.2013.28.

Bavishi, Rohan, Pradel, Michael, Sen, Koushik, 2018. Context2Name: A deep learning-based approach to infer natural variable names from usage contexts. CoRR abs/1809.05193, arXiv:1809.05193. URL http://arxiv.org/abs/1809.05193.

Butler, Simon, Wermelinger, Michel, Yu, Yijun, Sharp, Helen, 2010. Exploring the influence of identifier names on code quality: An empirical study. In: 14th European Conference on Software Maintenance and Reengineering. http://dx.doi.org/10.1109/CSMR.2010.27, URL http://oro.open.ac.uk/19224/.

Cardelli, Luca, 1988. Types for data-oriented languages. In: Schmidt, J.W., Ceri, S., Missikoff, M. (Eds.), Advances in Database Technology—EDBT '88. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–15.

Cardelli, Luca, 1991. Typeful programming. Formal Descr. Program. Concepts URL http://www.lucacardelli.name/Papers/TypefulProg.pdf.

Cardelli, Luca, Wegner, Peter, 1985. On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. 17 (4), 471–523. http://dx.doi.org/10.1145/6041.6042.

Casalnuovo, Casey, Barr, Earl T., Dash, Santanu Kumar, Devanbu, Prem, Morgan, Emily, 2020. A theory of dual channel constraints. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results. In: ICSE-NIER '20, Association for Computing Machinery, New York, NY, USA, pp. 25–28. http://dx.doi.org/10.1145/3377816.3381720.

Chromium, 2020. Chromium commit notes. https://chromium.googlesource.com/openscreen/+/75ea15d1ed9e7a689ef43bdfbc44059d1c7beeab. (Accessed on 2020-09-11).

Chromium, 2021a. Implementation of the new backing store version. https://chromium.googlesource.com/v8/v8.git/+/578f6be. (Accessed on 2021-04-15).

Chromium, 2021b. Performance issue for path rendering case. https://bugs.chromium.org/p/chromium/issues/detail?id=1063193. (Accessed on 2021-04-15).

Chromium, 2021c. Refactored version for HashMap case. https://chromium.googlesource.com/v8/v8.git/+/7bfc2e006. (Accessed on 2021-04-15).

Chromium, 2021d. Reported bug for backing data case. https://bugs.chromium.org/p/chromium/issues/detail?id=951795. (Accessed on 2021-04-15).

Chromium, 2021e. Reported bug for backing data case. https://bugs.chromium.org/p/v8/issues/detail?id=9221. (Accessed on 2021-04-15).

Chromium, 2021f. Reported bug for path rendering case. https://bugs.chromium.org/p/angleproject/issues/detail?id=3619. (Accessed on 2021-04-15).

Cohen, Jacob, 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 20 (1), 37–46. http://dx.doi.org/10.1177/001316446002000104.

Cover, Thomas M., Thomas, Joy A., 2005. Entropy, relative entropy, and mutual information. In: Elements of Information Theory. John Wiley and Sons, Ltd, pp. 13–55. http://dx.doi.org/10.1002/047174882X.ch2, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/047174882X.ch2.

Cplusplus, 2019. Preprocessor directives. http://www.cplusplus.com/doc/tutorial/preprocessor/. (Accessed on 2019-08-20).

Cppreference, 2019a. C++ international open standard from cplusplus. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf. (Accessed on 2019-08-20).

Cppreference, 2019b. Const cast conversion documentation from cppreference. https://en.cppreference.com/w/cpp/language/const_cast. (Accessed on 2019-08-20).

Cppreference, 2019c. Dynamic cast conversion documentation from cppreference. https://en.cppreference.com/w/cpp/language/dynamic_cast. (Accessed on 2019-08-20).

Cppreference, 2019d. Explicit conversions documentation from cppreference. https://en.cppreference.com/w/cpp/language/explicit_cast. (Accessed on 2019-08-20).

Cppreference, 2019e. Implicit conversions documentation from cppreference. https://en.cppreference.com/w/cpp/language/implicit_conversion. (Accessed on 2019-08-20).

Cppreference, 2019f. Reinterpret cast conversion documentation from cppreference. https://en.cppreference.com/w/cpp/language/reinterpret_cast. (Accessed on 2019-08-20).

Cppreference, 2019g. Static cast conversion documentation from cppreference. https://en.cppreference.com/w/cpp/language/static_cast. (Accessed on 2019-08-20).

Cppreference, 2019h. Type conversions documentation. http://www.cplusplus.com/doc/tutorial/typecasting. (Accessed on 2019-08-20).

Dash, Santanu Kumar, Allamanis, Miltiadis, Barr, Earl 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.

Dechev, Damian, Mahapatra, Rabi, Stroustrup, Bjarne, 2008. Practical and verifiable C++ dynamic cast for hard real-time systems. J. Comput. Sci. Eng. 2 (4), URL http://www.stroustrup.com/fdc_jcse.pdf.

Dietz, Will, Li, Peng, Regehr, John, Adve, Vikram, 2012. Understanding integer overflow in C/C++. In: Proceedings of the 34th International Conference on Software Engineering. URL https://www.cs.utah.edu/~regehr/papers/overflow12.pdf.

D'Silva, Vijay, Payer, Mathias, Song, Dawn, 2015. The correctness-security gap in compiler optimization. In: IEEE CS Security and Privacy Workshops. URL https://nebelwelt.net/publications/files/15LangSec.pdf.

Fakhoury, Sarah, Ma, Yuzhan, Arnaoudova, Venera, Adesope, Olusola, 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension. ICPC, pp. 286–28610.

Gibbs, Michael, Stroustrup, Bjarne, 2006. Fast dynamic casting. Softw. - Pract. Exp. URL http://www.stroustrup.com/fast_dynamic_casting.pdf.

Google, 2019a. Chrome project. https://www.chromium.org/. (Accessed on 2019-08-20).

Google, 2019b. What is GN? https://chromium.googlesource.com/chromium/src/tools/gn/+/48062805e19b4697c5fbd926dc649c78b6aaa138/README.md. (Accessed on 2019-08-20).

Google, 2020. QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic. (Accessed on 2020-09-11).

Google, 2021. New ArrayBuffer API explainer. https://docs.google.com/document/d/1sTc_jRL87Fu175Holm5SV0kajkseGl2r8ifGY76G35k/view. (Accessed on 2021-04-15).

Hathhorn, Chris, Ellison, Chucky, Roşu, Grigore, 2015. Defining the undefinedness of C. pp. 336–345. http://dx.doi.org/10.1145/2737924.2737979.

Hindle, Abram, Barr, Earl T., Su, Zhendong, Gabel, Mark, Devanbu, Premkumar, 2012. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering. ICSE '12, IEEE Press, pp. 837–847.

ISO, 2020. Programming languages - C++. Standard, (ISO/IEC 14882:2020), International Organization for Standardization, Geneva, CH, URL https://www.iso.org/standard/79358.html.

Jeon, Yuseok, Biswas, Priyam, Carr, Scott, Lee, Byoungyoung, Payer, Mathias, 2017. HexType: Efficient Detection of Type Confusion Errors for C++. pp. 2373–2387. http://dx.doi.org/10.1145/3133956.3134062.

Knuth, D.E., 1984. Literate programming. Comput. J. 27 (2), 97–111. http://dx.doi.org/10.1093/comjnl/27.2.97, arXiv:https://academic.oup.com/comjnl/article-pdf/27/2/97/981657/270097.pdf.

Lee, Byoungyoung, Song, Chengyu, Kim, Taesoo, Lee, Wenke, 2015. Type casting verification: Stopping an emerging attack vector. In: Proceedings of The24th USENIX Security Symposium. URL https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-lee.pdf.

LLVM, 2019. What every c programmer should know about undefined behavior 1/3. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html. (Accessed on 2019-08-20).

Mastrangelo, Luis, Hauswirth, Matthias, Nystrom, Nathaniel, 2019. Casting about in the dark: an empirical study of cast operations in java programs. Proc. ACM on Program. Lang. 3, 1–31. http://dx.doi.org/10.1145/3360584.

Microsoft, 2019. Enumerations (C++). https://docs.microsoft.com/en-us/cpp/cpp/enumerations-cpp?view=msvc-160. (Accessed on 2021-03-25).

Pârtachi, Profir-Petru, Dash, Santanu Kumar, Treude, Christoph, Barr, Earl T., 2020. POSIT: Simultaneously tagging natural and programming languages. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, Association for Computing Machinery, New York, NY, USA, pp. 1348–1358. http://dx.doi.org/10.1145/3377811.3380440.

Pârundefinedachi, Profir-Petru, Dash, Santanu Kumar, Allamanis, Miltiadis, Barr, Earl T., 2020. Flexeme: Untangling commits using lexical flows. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, pp. 63–74. http://dx.doi.org/10.1145/3368089.3409693.

Pearson, Karl, 1920. Notes on the history of correlation. Biometrika 13 (1), 25–45, URL http://www.jstor.org/stable/2331722.

Pradel, Michael, Sen, Koushik, 2015. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In: Boyland, John Tang (Ed.), 29th European Conference on Object-Oriented Programming (ECOOP 2015). In: Leibniz International Proceedings in Informatics (LIPIcs), vol. 37, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 519–541. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.519, URL http://drops.dagstuhl.de/opus/volltexte/2015/5236.

Pradel, Michael, Sen, Koushik, 2018. DeepBugs: A learning approach to name-based bug detection. Proc. ACM Program. Lang. 2 (OOPSLA), http://dx.doi.org/10.1145/3276517.

Regehr, John, 2019. A guide to undefined behavior in C and C++. https://blog.regehr.org/archives/213. (Accessed on 2019-08-20).

van Rijsbergen, C.J., 2004. The Geometry of Information Retrieval. In: Online access with purchase: Cambridge Books Online, Cambridge University Press, URL https://books.google.co.uk/books?id=b-gjx83aqBQC.

Shannon, C.E., 1948. A mathematical theory of communication. Bell Syst. Tech. J. 27, URL https://ieeexplore.ieee.org/document/6773024.

Spearman, Charles, 1961. The proof and measurement of association between two things. http://dx.doi.org/10.1037/11491-005.

Unicode, 2020. Documentation for NumberFormat class reference. https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classicu_1_1NumberFormat.html. (Accessed on 2020-05-07).

Winther, Johnni, 2011. Guarded type promotion: Eliminating redundant casts in java. In: Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs. FTfJP '11, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/2076674.2076680.

Wise, G. Bowden, 1996. Casting in C++: Bringing safety and smartness to your programs. SIGPLAN Not. 31 (8), 10–15. http://dx.doi.org/10.1145/242903.242912.

Constantin Cezar Petrescu is currently a Ph.D. student in Computer Science at the University of Surrey. He received his B.Sc. in Computer Science in 2018 from University College London and his M.Sc. in Information Security in 2019 from Royal Holloway, University of London. His areas of interest are Program Analysis and Information Security.

Sam Smith is an undergraduate student in his final year studying Computer Science at the University of Surrey. He is due to graduate with a B.Sc. in Computer Science in Summer 2022. He is planning on starting a Ph.D. in Cyber Security in 2022. He is interested in Software Analysis with a focus on its applications in Cyber Security.

Rafail Giavrimis is currently a Ph.D. student in Computer Science at the University of Surrey. He received his B.Sc. in Computer Science in 2021 from the University of Surrey. His research interests include search-based software engineering and AI.

Santanu Kumar Dash is a Lecturer in the Information Security Group at Royal Holloway, University of London. He is interested in applications of Software Analysis to the maintenance and security of large software ecosystems, such as the Android Open Source Project. His work on Bimodal Software Analysis, which combines symbolic and probabilistic techniques in a unified framework, has led to publications in flagship venues (ESEC/FSE'20 and ESEC/FSE'18). He holds a Ph.D. in Type-driven Software Security from University of Hertfordshire.