Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Master of Science

TITLE _____

_____

PRESENTED BY _____

ACCEPTED BY THE DEPARTMENT OF

Information Networking Institute

_____ _____
THESIS ADVISOR DATE

_____ _____
ACADEMIC ADVISOR DATE

_____ _____
DEPARTMENT HEAD DATE

APPROVED BY THE COLLEGE COUNCIL

_____ _____
DEAN DATE

Code Modernization Techniques Using Clang-Tidy for C23 Checked Arithmetic

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Security

John S. Samuels, II

B.Sc., Royal Holloway University of London

Carnegie Mellon University
Pittsburgh, PA

May, 2025

Acknowledgements

Thank you to Maverick Woo who initially proposed the idea for this thesis and has guided me throughout this journey. I would also like to thank Patrick Tague for helping me through major milestones throughout this project and providing useful feedback when requested. I am forever grateful for the invaluable support my family and friends have provided me, without which none of this would be possible. This thesis was self-funded.

Abstract

The first major integer safety issue occurred in 1975 with the DATE75 bug, where dates past January 4th 1975 could not be represented using a 12 bit integer. Almost 50 years later, integer safety bugs are still an issue programmers must consider when writing code. The root cause of this issue is related to how computers represent numbers. That is, numbers are represented in binary using a fixed number of bits. In C, when a value is placed in some variable that does not have enough bits to store said value, integer overflow or wraparound occurs. This is essentially a truncation operation and reflects how the underlying hardware operates. Integer overflow is when a signed integer is too small, and wraparound is when an unsigned integer is too small. Although integer wraparound is defined by the C standard, both types of behavior can result in unexpected values and develop into security vulnerabilities. Recently, the C standards committee has come out with a new set of checked arithmetic macros for combating these issues. C23 now has checked arithmetic macros that can be used to ensure that integer operations do not overflow or unintentionally wraparound. This thesis aims to explore the application of these macros, and propose a solution to automatically updating older codebases using a compiler-assisted approach.

There are a number of issues related to using the new checked arithmetic macros, and even more so trying to automatically rewrite expressions. When a rewrite needs to be performed, a set of temporary variables need to be defined based on the types involved in the original expression. These types may be non-obvious, since they may be declared elsewhere or implicitly cast at some point. To address this issue, type information is collected from the compiler to ensure that these temporary variables

reflect the types in the statement itself and its destination. This approach was implemented using the tools and APIs available in the LLVM compiler infrastructure project, and was implemented as a clang-tidy check. Using these tools allowed the clang-tidy check to match on statements that could be rewritten, then suggest a fix that would select the intermediate types based on what the compiler was using.

To evaluate the effectiveness of the clang-tidy plugin, several research questions were posed. This included how comprehensive was the rewrite, what were common reasons to omit lines, and what was the performance overhead. The methodology included writing "debug" checks to match on all possible statements that could overflow to then calculate the proportion of those statements that could be rewritten. Figuring out common reasons to omit lines was purely qualitative and was based on the experience rewriting several targets. Finally, performance overhead was calculated by running tests before and after the rewrite and timing their runtime. Three open-source targets were selected based on their relative size, complexity, and testing infrastructure. The results of the evaluation was coverage ranging from 87% to 53% of statements being rewritten. The experience of rewriting was simple for the first two targets, and unsuccessful for the third due to several issues. Finally, of the two targets that could be tested for performance, performance overhead for checked arithmetic came out to be 12% and 4% respectively.

In all, this thesis explored the possibility of compiler-assisted code modernization and produced a proof-of-concept clang-tidy plugin that can successfully rewrite targets to use new C23 checked arithmetic macros.

Table of Contents

List of Tables

List of Figures

1

Introduction

This thesis would like to explore the idea of using automated techniques for modernizing old C codebases. C standards are published every so often, and can include a number of useful additions that developers may want to take advantage of. This may be the case for C's new checked arithmetic macros. These macros are designed to perform and check arithmetic operations so that bugs related to integer safety are caught and handled at runtime.

The main issue faced by a lot of developers in the real world is that it may take a long time to actually rewrite their codebases to utilize, say, checked arithmetic macros. Modernizing codebases is often plagued by this issue, which may motivate developers to just rewrite their codebases completely. Doing a complete rewrite, especially in a more modern language, has a lot of benefits. However, doing this is just not practical and often misguided. Specifically with integer arithmetic, overflows and wraparounds still occur in modern languages. Thus, it may be more beneficial to perform an automatic rewrite that uses these macros before incurring the cost of a completely new implementation.

It is in the interest of both these developers and society as a whole to adopt

features, especially ones that make code more secure. This is the justification for this thesis, which aims at finding a method for automatically modernizing codebases to use checked arithmetic. Due to the nature of these new macros, a number of challenges need to be overcome to actually be successful.

The main issue related to rewriting statements is the issue of intermediate types. When an arithmetic expression is written, the programmer must use intermediate variables to store and check for overflow. This is simply due to the construction of these macros. Because integer overflow and wraparound depend on the width of the variables involved, selecting the right types of the intermediate variables is imperative to successfully rewriting these statements correctly. Another issue is related to how the macros only operate on a subset of the integer types and operations. Thus, not all operations that can overflow can be rewritten using the checked arithmetic macros.

Finding an automatic method for rewriting statements that tackles these main issues will open the door for developers to harness these new language features until more robust solutions are presented to them in the future.

2

Background

This chapter will explore the background required to fully understand the motivation of this thesis, the current landscape with regard to integer safety, and important details related to the macros introduced in the new C23 standard. It will cover a short history of C and the C standard to provide the reader with some context on the philosophies of C and how new language features are introduced. This will be followed by a discussion on the development pipeline for safety critical systems, which will also be useful for understanding the requirements of the problem space. The next section will cover integer safety. These sections will further describe the problem space of this thesis. Following this description will be a taxonomy of approaches and their relative effectiveness. The chapter will conclude with relevant details of the new C23 standard that motivated this particular project.

2.1 C and the C Standard

C is a general-purpose imperative procedural compiled programming language. It features important tools like a static type system, raw memory manipulation, portable code generation, and a lightweight runtime.

C as a programming language was initially developed by Dennis Ritchie in the early 1970's at Bell Labs [30]. The purpose of the language was to provide a higher-level programming interface to build operating systems like Unix. C was first standardized in 1989 by the American National Standards Institute (ANSI) and later standardized by the International Organization for Standardization (ISO) in 1990 [15][12]. Since then, several more standards have been released to add or remove features from the official language standard. The process of creating a new C standard follows the general ISO standardization process which starts with a proposal, and ends with a publication stage. During this process, there are a series of meetings between stakeholders including the official C standards committee (WG14)[14]. The C standards committee is an international group of experts tasked with maintaining, updating, and refining the C standard.

2.1.1 Unspecified Behavior and Undefined Behavior

The C language specification categorizes the behaviors of programs generated by a C compiler. There are some behaviors, however, that are not explicitly defined by the standard. These include *undefined behavior*, *unspecified behavior*, *implementation-defined behavior*, and *locale-specific behavior* [13]. The two behaviors relevant to this thesis are undefined and unspecified behavior.

Undefined behavior occurs when a program is not well formed and exhibits some statement that is "undefined" [13]. This can include out of bound accesses, null pointer dereferences, and signed integer overflow. When such a statement is written, the standard allows the compiler to do anything it wants (including making demons shoot out of your nose [33]). These statements are often a source of optimization, but can also result in security vulnerabilities.

Unspecified behavior are related to statements that can have multiple outcomes the compiler does not need to document. An example of this is the order of eval-

uation: a statement with multiple subexpressions does not have to be evaluated in specific or consistent order. This allows compilers again to optimize their instructions to maximize performance.

2.2 The Development of Safety Critical Systems

In many cases, it is unacceptable to have a software fault. This would be the case for many safety critical systems like locomotives, aircraft, cars, or power plants, to name a few. It may not be possible to completely eliminate all bugs, however the software development pipelines for these systems is designed to catch and fix bugs as early as possible. There are several regulatory standards imposed on different systems, and we will not be going over all of them. However, it is important to note a common technique that is commonly employed: manual code review [7][2]. That is, when code is written, other developers will need to manually certify that the newly written code does not contain some set of bugs defined in a standard checklist. This will be an important detail as some approaches to code modernization will not take place at the source level, which will need to be reviewed by humans before being certified.

2.3 Integer Safety

The topic of integer safety has a long and interesting history involving a number of different systems. In this section we will discuss that integer safety is, how violations of integer safety occur, and some real world examples of bugs caused by integer overflow.

2.3.1 Integer Representation

When a integer is declared and initialized, it's underlying value will be represented by a collection of bits. The C standard defines a number of different integer types and

can be broken into two different categories: unsigned and signed integers. Unsigned integers represent numbers using raw binary numbers ranging from 0 to $2^{\text{width(type)}} - 1$ where $width(type)$ is the number of bits that represent the type of the integer. Standard unsigned integer types and their widths on x86-64 using the LP64 model include `unsigned char` (8 bits), `unsigned short int` (16 bits), `unsigned int` (32 bits), `unsigned long int` (64 bits), `unsigned long long int` (64 bits) [11].

Signed integers are meant to store positive and negative numbers, something you cannot do using an unsigned integer. Standard signed integer types and their widths on x86-64 using the LP64 model include `signed char` (8 bits), `short int` (16 bits), `int` (32 bits), `long int` (64 bits), `long long int` (64 bits). The C standard also allows compilers to represent signed integers in three different ways: sign and magnitude, one's complement, and two's complement [13]. Virtually all modern compilers will represent signed integers using two's complement, so only the implications of this representation will be discussed. Two's complement works by treating the most significant bit as a *sign bit*, then giving it a weight of -2^{N-1} when calculating the decimal value of each bit. Two's complement is the most popular representation of integers because of properties it has to make arithmetic operations easy and does not have issues like a positive and negative 0 representation like one's complement. One important detail however, is the most negative value that two's complement cannot be represented as a positive number using the same number of bits due to the asymmetry of two's complement.

With any type of integer in C, they are represented using a finite set of bits. This means that depending on the number of bits available for a particular integer type, there is a finite set of numbers that can be represented. This leads to the question: what happens when I try to store an integer that needs more bits than is available in a particular type? The answer is truncation, which can result in a misinterpretation of the newly assigned value. The specifics on how the compiler will react to such a

situation will depend on the types involved.

When a value that is too large for an *unsigned integer* is stored, *wraparound* occurs. Given an unsigned integer of width 16, only the 16 least significant bits of the value will be stored in the unsigned integer. A property of binary representation is this operation becomes a modulus operation, meaning any value placed in an unsigned destination will have the value $n \bmod width(type)$, where n is the value to place in the unsigned integer and *type* is the unsigned integer type of the destination.

Interestingly, this behavior is considered defined behavior in the C language specification [13]. This is a point of controversy within the C community but the rational is because representation itself is always a modulus operation, something that some programmers take advantage of, the concept of an *overflow* in an unsigned type has no meaning. Furthermore, this behavior is closer to how hardware would perform, wrapping around when the maximum or minimum value is reached.

2.3.2 Integer Overflow

An *integer overflow* occurs when an operation involving a *signed integer* results in a value that is too large or small to be represented in the resultant type. The C standard considers integer overflow undefined behavior, which has a number of implications as mentioned earlier in this chapter. Operations that can cause overflow are listed in table 2.1.

Something interesting to note is that division and modulo operations can overflow. This is due to the asymmetry of two's complement as previously mentioned, where the most negative value of two's complement cannot be represented as a positive number. Thus, any operation that can switch the sign (like division and modulo) on the smallest possible value will result in an overflow. Because this overflow depends on the way integers are represented, technically this subset of operations become safe when a different representation is used, like one's complement.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
<<	Left shift
+=	Compound addition assignment
-=	Compound subtraction assignment
*=	Compound multiplication assignment
/=	Compound division assignment
%=	Compound modulo assignment
<<=	Compound left shift assignment
++	Pre/post-increment
--	Pre/post-decrement

Table 2.1: Operations that can Overflow

2.3.3 Integer Conversion

Integers can be converted from one integer type to another explicitly through a type cast, or implicitly within a mixed-typed statement. For example, an implicit cast could be inserted by the compiler in an arithmetic expression, an assignment, or even a function call. The intention of these rules is to make it easy for programmers to write mixed-typed statements without having to worry about explicit casts, so long as they remember the rules that govern them.

There are three concepts that are outlined in the C standard regarding integer conversions: *integer conversion rank*, *integer promotion*, *usual arithmetic conversions* and *default arithmetic conversions*. These concepts are explained in the following subsections:

Integer Conversion Rank

Each integer type is assigned a type of *rank* that is used to govern how conversions are made between different types. The rank assigned to a type is determined by the

rules in Table 2.2 [13].

Rule	Description
1	The ranks of all signed integer types are different and increase with their precision.
2	The ranks of all signed integer types equal the ranks of the corresponding unsigned integer types.
3	The rank of any standard integer type is greater than the rank of any extended integer type or bit-precise integer type of the same size.
4	The rank of `char` equals the rank of `signed char` and the rank of `unsigned char`.
5	The rank of `bool` is less than the rank of any other standard integer type.
6	The rank of any enumerated type equals the rank of its compatible integer type.
7	Ranking is transitive.
8	The rank of a bit-precise signed integer type shall be greater than the rank of any standard integer type with less width or any bit-precise integer type with less width.
9	The rank of any bit-precise integer type relative to an extended integer type of the same width is implementation-defined.
10	Any aspects of relative ranking of extended integer types not covered above are implementation-defined.

Table 2.2: C23 Integer Conversion Rank Rules

The purpose of these rules becomes clear in the next section.

Condition	**Conversion Rule**
Target type can represent the value	The value is unchanged.
Target type is `unsigned` and cannot represent the value	The value 2^b, where b is the number of value bits in the target type, is repeatedly added or subtracted to bring the value into range. In other words, unsigned integers implement modulo arithmetic.
Target type is `signed` and cannot represent the value	The result is implementation-defined behavior (which may include raising a signal).

Table 2.3: Integer Conversion Rules in C23

Integer Promotion

Integer promotion is the implicit conversion of any type with a rank less than an `int` or `bool` to an `int`. Note that the promoted type can be signed or unsigned. Furthermore, these promotions are intended to preserve the value and sign of the initial value. The rules that determine how promotion is handled can be found in Table 2.3.

Integer promotion only apply to the following:

- Usual arithmetic conversions
- Default argument promotions
- Operands of the `++` and `--` operators
- Operand of the `~` operator
- Both operands of the `<<` and `>>` operators

The next section will cover the case most relevant to this project, usual arithmetic conversions.

Usual Arithmetic Conversions and Default Argument Promotions

When a mix-typed statement is written, the compiler will need to calculate the *common real type* of the operation itself, which will be used to complete the operation. Operations that would trigger a conversion include binary arithmetic, relational operations, bitwise operations, and the conditional operator. The rules for how the common real type is generated can be found in appendix A.

As mentioned in the integer promotion subsection, default argument promotions can also trigger integer promotion. This occurs when the arguments are passed to a variadic function and are matched to the ellipsis parameter.

Now will these concepts explained, it is important to understand how these rules can be a source of error. The implication is that these casts can result in a loss of value (i.e. conversion where the magnitude of original value cannot be preserved) or a loss of sign (i.e. converting from a signed value to an unsigned value will mean negative values are no longer represented).

Consider the following example [31] :

```
unsigned int ui = ULONG_MAX;
signed char c = -1;
if (c == ui) {
    puts("Why is -1 = 4,294,967,295???");
}
```

In the code above, the signed char `c` is compared to signed int `ui`. As a result, `c` is promoted to an `unsigned int` and now represents a very large unsigned value.

2.3.4 Case Studies

There have been a number of real-world bugs that have been caused by integer overflow and wraparound. One of the most famous examples is the Ariane 5 rocket disaster. The disaster was caused by an integer overflow that resulted erroneous horizontal velocity values, leading to the rocket to self-destruct [17]. The damage exceeded $370 million dollars [37].

Another prominent example was the Therac-25 radiation therapy machine. This machine was meant to deliver a dosage of radiation to patients to treat illnesses like cancer. However, there were a number of software related issues including an arithmetic overflow that caused certain safety checks to be bypassed [1]. The result of these errors resulted in radiation doses that exceeded 100 times the intended amount in a smaller area. Patients developed radiation burns and three individuals lost their lives.

One thing to note is this issue is not constrained to C. The overflow in the Ariane 5 was in Ada. In all, these examples should demonstrate the consequences of poor integer safety.

2.4 Previous Approaches to Ensuring Integer Safety

Given how long integer safety has been an issue for C programmers, there are a number of approaches that have been adopted by the community. They each have pros and cons and are discussed in the following subsections. Note that from this point on we only consider approaches to integer safety that do not involve arbitrary precision types.

2.4.1 General Approaches for Detecting Overflow and Wraparound

There are a number of general approaches that can be used to detect when overflow or wraparound occurs that involve modifying or instrumenting code to catch these cases. These approaches don't depend on any particular build system or language as they are implemented by the programmer manually. These are discussed below:

Pre and post condition testing

The most basic approach to detecting overflow and wraparound involve introducing checks before and after the risky operation. For example, take the following operation:

```
int a, b, c;
// initialize a and b
c = a + b;
```

The line `c = a + b` may overflow. Using a precondition check, we can test if the current values of `a` and `b` will indeed trigger an overflow before performing the operation. This could be written like so [31]:

```
int a, b, c;
// initialize a and b
if ((b > 0 && a > INT_MAX - b) ||
    (b < 0 && a < INT_MIN - b)) {
   // OVERFLOW
} else {
   c = a + b;
}
```

In english, the code above will test that there is room between the current value of `a` and the upper and lower bounds of signed integer representation.

There is a notion of a post-condition test as well. Below is an example of how one would right a post-condition check for wraparound for two unsigned integers:

```
unsigned a, b, c;
// initialize a and b
c = a + b;
if (c < a) {
   // WRAPAROUND
}
```

Although easy to implement, there are a number of issues with this approach. One is related to performance: constantly checking for overflow can introduce overhead and damage performance. Another is it becomes difficult to write multi-operation arithmetic statements like `int c = a + b + c`, which would need to perform two operations and thus two checks. Although this is a simple example, these statements can get quite complex and difficult to rewrite. This issue is further complicated by the fact that the order of evaluation is unspecified behavior and may vary between different compilers.

Using Status Flags

In x86, there exist instructions that will perform a type of postcondition check for overflow once an operation is performed. The overflow (`OF`) and carry (`CF`) flags will be set after an instructions like `add`, `sub`, and `mul` if overflow or wraparound occurs

[6]. Instructions like `jo` and `jc` can then perform jumps if these bits are set to an overflow or wraparound handler.

This approach also introduces overhead similar to the general checks previously mentioned. The performance issue is actually worse in this case, since the compiler likely won't be able to optimize this assembly. Furthermore, this approach is not portable and depends on both the architecture of the system and the compiler extensions supported to write inline assembly (the most likely approach to injecting these instructions). It again suffers from issue of potentially writing overly complex statements to check multi-operation statements.

Casting the result to a larger type

A simple solution to detecting overflow and wraparound in addition and multiplication is to first cast the operands to a larger type, perform the operation and check for overflow, then downcast to the original resultant type. For addition, any result can be represented by $w + 1$ bits, where w is the width of the largest operand. This rule is the same for multiplication, however each operand needs to be at least $2w + 1$.

Once the operation is complete, the result can then be range checked in a type of postcondition check, then downcasted to the intended type if no overflow or wraparound occurred. This approach is simple however it does depend on the compiler since C does not guarantee any standard type is larger than any other standard type.

2.4.2 Language-based Approaches

C is a very old language and its ability to produce integer arithmetic errors largely stems from this fact. Since C was created, many other languages have been developed and have addressed the issue of integer safety in different ways. The following are several notable examples.

2.4.3 Python

Python was initially created as a learning language by Guido van Rossum in 1991 as a type of learning language. It is a dynamically typed, mutli-paradigm scripting language. In Python, integers are represented using arbitrary precision, meaning integers can have as many bits as they need to be represented granted there is enough memory on the system [28]. As a result, for integer types there is no notion of an "overflow" or "wraparound".

Interestingly, there is still an `OverflowError` exception that can be raised if an integer is too large to be represented [29]. This error cannot occur for integers however, and instead a `MemoryError` is raised. According to the documentation it may still be raised if an integer is outside some range due to historical reasons. For completeness, floating-point operations *can* overflow, which is due to a lack of standardization for handling floating-point exceptions in C.

Java

Integers are represented in Java using two's complement arithmetic and will wraparound when a value is too large to be represent a particular value [24]. To address this, Java 8 provides a set of functions that can perform operations and either return the result of the operation or throw an exception. These functions are: `addExact`, `subtractExact`, `multiplyExact`, `incrementExact`, `decrementExact`, and `negateExact` [25].

Another option that Java provides is the `BigInteger` type, which allows for arbitrary precision integers [26].

Rust

Rust is often used as a modern replacement for C and C++ given it's capability as a systems programming language while still supporting many safety mechanisms. Of

these features is integer safety. When a Rust program is compiled in debug mode, checks are inserted in the binary to detect integer overflow [16]. When an overflow is detected, a *panic* is triggered. An important detail here is that these checks are removed when a rust binary is compiled in *release* mode. To address this, Rust provides a set of `checked*` functions to detect integer overflow due to arithmetic operations [36]. These will ensure that checks are performed even in release builds. Downsides of this approach are rewrites of complex operations can be tricky and there is a performance overhead.

Perhaps the most dangerous aspect of this is what most developers think Rust is capable of. Namely, that as long as the `unsafe` keyword is not being used, integer overflow is not something to worry about. This is simply not the case, as integer safety is still an issue in release builds.

2.4.4 Checked Arithmetic Libraries

Another approach to making C arithmetic safe is to use an external library to perform arithmetic in a safe manner. The exact approach for how code should be written or rewritten will depend on the library, and can also issues related to compatibility. The following subsections cover some modern safe arithmetic libraries for C and C++.

SafeInt

The SafeInt library approaches integer safety by providing functions to explicitly cast integers and perform arithmetic operations. This is a great approach since these are the sources of integer errors. In effect this library provides safe functions for assignment, casting, comparison, arithmetic, and logical operations. Below is an example given in their documentation:

```
int main()
{
   int divisor = 3;
   int dividend = 6;
   int result;
   bool success = SafeDivide(dividend, divisor, result);
   success = SafeDivide(dividend, 0, result);
}
```

This library was originally written for C++, however now supports C via the `safe_math.h` and `safe_math_impl.h` header files.

Intel Safe Arithmetic

The Intel Safe Arithmetic library approaches the issue of integer safety via compile-time checks. It provides safe versions of common types, putting them in a type of "safe arithmetic environment". This means any operations performed on these new types are checked for unsafe behavior including signed overflow and unsigned wraparound.

This approach has a massive benefit: there is no runtime overhead. Since these checks are performed at compile-time and any potential errors generate warnings, no instrumentation or handling needs to be inserted to detect and handle overflows. Furthermore, so long as the types are changed existing code can remain relatively similar and reap the benefits of integer safety. The downside to this library is it only supports C++.

safe_numerics

The safe_numerics library uses a hybrid compile and runtime approach. Similar to Intel's Safe Arithmetic library, new safe types are provided that can be used like any other builtin type. When enough information can be determined that an overflow may occur at compile-time, a warning is generated. If not, and an overflow occurs during runtime, the overflow is handled via runtime exception that the developer

can handle. This has the added benefit of being more complete than only using a compile-time approach, however there is a small runtime overhead due to the added runtime checks. This library is also only written for C++.

2.4.5 Compiler-based Approaches

Compiler engineers have developed techniques to automatically mitigate integer overflow and wraparound vulnerabilities. These include inserting instrumentation when an overflow is detected and providing functions for developers to easily perform precondition tests.

The `-ftrapv` flag

Compilers like GCC and Clang provide a number of compiler flags that instruct the compiler on how to deal with integer overflow and wraparound. The `-ftrapv` flag will instruct the compiler to insert instrumentation to detect and handle signed integer overflow [10].

The downsides of this approach is there is again an overhead to inserting these checks and this flag is known for having some implementation issues. These issues include the conditional branches used to handle the trap (which can be expensive) and will only detect overflow for a subset of operations that can produce an overflow. Another is this flag will not detect unsigned wraparound, which may be unintended behavior.

Furthermore, this approach also has the limitation of not inserting changes at the source level. This could be an issue for development pipelines that need to verify changes made by any automatic tools including passes by a compiler like a safety critical system.

One interesting note is that GCC seems to implement handling by calling `abort()`, which is not always the correct approach to handling overflow or any kind of error.

Clang however does have a `-ftrapv-handler` flag that allows developers to specify a function to call when an error is detected [19].

Undefined Behavior and Integer Overflow Sanitizers

Both GCC and Clang provide an Undefined Behavior Sanitizer (UBSan), which will instrument the compiled binary to detect a number of undefined behaviors in a program [27][35]. This is similar to using the `-ftrapv` flag but spans a number of other patterns that can result in undefined behavior. This includes, out-of-bounds array subscripts, bitwize shifts that are too large, null-pointer dereferences, and signed integer overflow.

UBSan is slightly better than using the `-ftrapv` flag since it can also be configured to check for unsigned integer overflow. Besides this benefit, this approach suffers from the same issues using `-ftrapv` incurs.

Overflow Checking Functions

GCC and Clang offer language extensions for checking if an arithmetic operation will overflow[8][3]. The general structure of these functions are to pass both arguments and the destination for the operation to one of these functions. This function will then both perform the operation and check for overflow using the promoted operands. It will then return 1 if an overflow is detected and 0 otherwise. The programmer is meant to check the return value of this operation then handle the case where an overflow is detected.

For the sake of space, only a subset of Clang's extensions are listed below:

```
bool __builtin_add_overflow(type1 x, type2 y, type3 *sum);
bool __builtin_sub_overflow(type1 x, type2 y, type3 *diff);
bool __builtin_mul_overflow(type1 x, type2 y, type3 *prod);
```

With regards to making source-level checks to detect integer overflow, this is the best approach. There are still downsides around readability and ensuring programmers call these functions and handle errors correctly, which are shared with issues pre and post condition checks have. Furthermore, these are compiler *extensions*, and are not part of the C standard. This means there is a dependence on both compiler support and how the compiler implements these functions (the names and signatures of these functions are very different between Clang and GCC).

Notably, both GCC and Clang offer signed and unsigned versions of these functions, and implement the generic check function (i.e. `builtin_add_overflow(type1 x, type2, type3, *sum)` such that it will detect both signed integer overflow and unsigned integer wraparound.

2.4.6 Testing and Analysis

Software testing and using automated or manual analysis is likely the most common approach to avoiding integer arithmetic errors. These techniques can range from simple unit tests to complex guided fuzzing or even formal verification. Because the range of approaches is quite wide and the pros and cons of this approach really depend on the technique applied here. The approaches we will consider are static analysis, testing, and manual review.

Static analysis tools can be quite powerful and will generally involve running the tool, then rewriting code to prevent or remove a line of source that could be unsafe [31]. The benefit is that these tools can integrate nicely in a development environment, however the main downside will be false positives and negatives. Static analysis tools are generally more prone to producing false positives. When the number of false positives is reduced, this may also result in false negatives. It is very difficult to build a static analysis tool that is both sound and complete, so developers should not rely solely on this approach.

Testing is a good practice for any software development, and can quickly locate bugs before they become bigger issues. Unfortunately, this approach cannot make guarantees about how code will behave in the field and it is difficult to ensure all possible inputs are tested.

Finally, the last approach we will consider is a manual source code audit. This is also common practice, and can be quite effective for locating bugs. However this approach may fail to locate bugs due to the error-prone humans it relies on.

The biggest downside to all of these approaches are how much more involved humans must be, which suffers from issues related to human error and the cost of labor.

2.5 The C23 Approach

In December 2023, the C23 standard was officially published [13]. Among the changes to the C standard was a set of checked arithmetic functions for testing if certain operations would overflow [34]. These functions were essentially inspired by the language extensions GCC and Clang have for detecting integer overflow and wraparound. As a result, the GCC and Clang compilers have essentially just provided implementations that wrap their existing extensions.

2.5.1 The ckd_add, ckd_sub, and ckd_mul Macros

C23 defines the set of checked arithmetic macros in the `<stdckdint.h>` as follows:

```
bool ckd_add(type1* result, type2 a, type3 b);
bool ckd_sub(type1* result, type2 a, type3 b);
bool ckd_mul(type1* result, type2 a, type3 b);
```

Each function will perform the operation as if an infinite-precision model is being used, then return `1` if an overflow occurs. Note that these are "type-generic" macros

and can take several types. The standard says that "`type2` and `type3` shall be any integer type other than "plain" `char`, `bool`, a bit-precise integer type, or an enumerated type, and they need not be the same. `*result` shall be a modifiable lvalue of any integer type other than "plain" `char`, `bool`, a bit-precise integer type, or an enumerated type" [34].

2.5.2 Open Questions

These macros are a welcome addition to the C programming language and hopefully will be used to ensure critical operations are checked. However, there are still some open questions about this approach, which are discussed below:

Should these functions work for signed and unsigned types?

As we have seen in the language extensions provided by GCC and Clang, signed and unsigned macros have been provided to be specific with the type of behavior that is being detected. So far there is no distinction here in the C23 standard, and the standard does not discuss this detail. Tests have revealed that these macros do detect unsigned wraparound, however this is likely implementation defined behavior and could result in issues if different compilers are used.

How do you ensure the correct types are used?

The type of the destination and operands are very important for determining if an operation will result in an overflow or wraparound. Furthermore, C has a number of complex rules that govern implicit casts as mentioned previously. The concern here is should the resultant type be determined exclusively by the developer? If not, could the compiler be used to determine if there is going to be an overflow?

How is a two argument signature going to change things?

There is a proposal to move from a three argument macro to a two argument macro that just contains the operands of an argument. This is supposedly a future plan but can also suffer from a similar issue to the typing issue: how will the compiler determine the resultant type and thus correctly detect overflow?

What about other operations and types?

A final question is how will these macros evolve to detect overflow in other operations. The restrictions the C23 standard places on the types of the destination and operands is quite limiting, and makes these macros ineffective against a very large proportion of operations that may overflow.

3

Design

This chapter will cover the process by which the code rewriting methodology employed by this thesis was designed. The chapter will start by defining the requirements and scope, followed by approaches that were considered and how the compiler-based approach was selected. This then leads into further design choices related to tools used to complete the deliverable. Finally, this chapter ends with the intended user flow for the rewrite tool and it's limitations.

3.1 Requirements

There is a basic set of requirements the methodology designed by this thesis must adhere to in order to be relatively realistic approach to the problem of code modernization. They are as follows:

1. Code rewrites must not introduce, change, or remove side effects of rewritten operations

This is a basic requirement that should be adhered to as much as possible. Approaches that increase the likelihood this rule is violated could make the tool difficult

to use or even completely useless. Some important caveats related to this requirement are discussed in 3.2.

2. The approach should be automatic

A common barrier to modernizing codebases is the combination of the complexity of the old code and the shear size of the codebase itself. Modern codebases can range from thousands of lines of code to millions. For example, the popular machine learning library TensorFlow is made up of about 4 million lines of code [23]. Code written for modern vehicles is estimated to be about 100 million lines of code [22]. Even with a large team, manually modernizing even a moderately large codebase could be intractable on a reasonable timescale and could increase the likelihood other bugs associated with the rewrite are introduced. With this in mind, automating the modernization process as much as possible should be a defining requirement of the deliverable.

3. Modifications must be reflected at the source level

As discussed in Chapter 2, there are domain specific development processes that require source to be reviewed before it can pass to the next step of the development pipeline. As a result, any approach that introduces checks for arithmetic operations must be able to show these changes at the source level for human review. Note that this technically implies the approach could introduce checks in code represented in some non-source (e.g. as an AST, IR, machine code, etc.) representation as long as the source could be reconstructed with the newly written checks.

4. Users must be able to tailor how overflow detections are handled

Handling integer overflows is completely context dependent and is thus impossible to write a one-size-fits-all handler. For example, it may be a very good idea to stop program execution when integer overflow is detected in some banking software, but

not such a good idea in an engine controller. As such, the tool must allow the user to provide some rewrite code that acts as the error handler to tailor the system's response to any overflow detections.

5. Users must be able to exempt lines from rewrites

This is a usability requirement given some lines cannot be rewritten for some reason. This could be that there is an operation that is being rewritten that is out of scope of the tool, there is an error in the rewrite for a particular line, or some other unforeseen complication.

6. Comments must be preserved

Comments often contain important information for both humans or other processes in a development pipeline. Some approaches may remove comments since they operate using source stripped of any comments or on artifacts later in the compilation pipeline that don't contain source (and thus comments).

3.2 Scope

With the requirements defined for the code modernization tool, this section will consider the scope of the rewrite tool. This includes the types of operations that will be rewritten, types of operands that will be considered, and the types of build systems this tool will focus on.

3.2.1 Target Operations

The new C23 checked arithmetic headers only include checks for addition, subtraction, and multiplication operations [34]. As mentioned in Chapter 2, there are other operations and patterns that can trigger in integer overflow. Because of this, the rewrite tool will only be concerned with checked operations supported by C23 and

their different forms. For addition, this includes the forms + (non-assignment operator), += (assignment operator), and ++ (unary operator). Furthermore, the tool will ignore any overflows due to other operations or casting errors not directly associated with these operations. The rational behind this is as long as the expression's type is the same in the AST before and after the rewrite, no casting errors can be introduced in the code. The downside is if there is a casting error higher up in the AST, the rewrite would preserve that error.

3.2.2 Types and Other Language Features

Again due to C23's checked arithmetic macros, arguments must be "an integer type other than plain char, bool, a bit-precise integer type, or an enumeration type" [34]. This further limits the type of expressions that can be written with these macros, and thus limits the types of expressions the tool can support. In addition to this, a type that cannot be used is the "bit-precise integer type". This is in reference to the `_BitInt(n)` type also introduced in C23. This is a new integer type and thus not commonly in use especially by codebases written before the C23 standard was published. As a result, how the tool rewrites or does not rewrite expressions involving bit-precise integer types is considered out of scope for this project.

Macros and other expansion features [5] used by the preprocessor included in the C programming language could complicate a checked arithmetic rewriting tool. These include `#includes`, `#defines` and `#/##` directives, just to name a few. These features are used heavily in modern codebases, however in the context of this thesis it complicates the problem beyond what is needed to solve the core questions of this exploration. Furthermore, it is believed that if a good rewrite can be achieved without considering macros it very may well be possible to do so with those considerations. As a result, it is left as future research as described in Chapter 6.

A final note, our approach will not be considering type qualifiers for the same

reason macros are not considered: this is something to consider once a proof-of-concept is complete.

3.2.3 Build Systems

There is a plethora of compilers and tooling available for C programmers. Moreover, the type of compiler a developer uses can have an enormous impact on the features available to a developer and the tools they use. For example, the `_BitInt(n)` integer type is not supported by any version MSVC and only partially supported by GCC 14 [4].

To build a compiler-based approach to performing rewrites will also require the compiler to be on some level extensible and preferably open source for such a tool to be implemented. A good candidate compiler would also be relatively popular to make any tools written for it accessible and usable on a large scale.

With these requirements in mind, the scope of build systems that will be considered for this investigation are Clang 19.1 [18] and GCC 14.2 [9]. These compilers meet all the aforementioned requirements: they are popular, open source compilers that are extensible. Clang is especially extensible since it exists within the LLVM compiler infrastructure project. Clang 19 will be the focus for any tool written, and both Clang 19 and GCC 14.2 are considered valid compilers for the rewritten code. This means any language features supported by these versions of Clang and GCC can be incorporated into any rewritten statement.

3.3 Approaches

In this section we will explore the different approaches that can be taken for rewriting a codebase to use checked arithmetic. This will illustrate the pros and cons of each approach, and why the compiler-based approach was ultimately chosen as a legitimate option.

3.3.1 Manual Rewrite

Although we require that our solution needs to be automatic, we should humor manual rewrites to understand the tradeoffs of using the approaches discussed later in this chapter. Moreover, the manual approach is the only approach currently available to developers looking to migrate to C23.

The steps for a manual rewrite would be getting a developer to go through and identify operations that should and can be rewritten, then rewrite them with the correct checked arithmetic function and error handling. This might seem simple, but there are a number of pitfalls that the developer could fall into.

The first has to do with actually identifying the operations to rewrite. Besides failing to identify an operation that should be rewritten, not every addition, subtraction, or multiplication operation can be rewritten. For example, a + operation between a `char` and an `int` cannot be rewritten since this is not supported by the checked macros. This is all to say the developer would need to ensure all the types of the involved operands are valid, which increases overhead.

Second is how the rewrite is performed. Consider the following expression:

```
long a = LONG_MAX - 1;
long b = 1;
int c = a + b;
```

In this example, the overflow is not due to the + operation but the assignment operation. The common real type between the two operands is `long`, so this would be the type of the operand. However, when the assignment to `c` is made, this result is cast to an `int`. Now consider how you would rewrite this. One improper way could be as follows:

```
long a = LONG_MAX - 1;
long b = 1;
long res;
if(ckd_add(res, a, b)) {
// HANDLE ERROR
}
int c = res;
```

This would be an incorrect rewrite, since we failed to capture the truncation that happens at the assignment. This may seem like a simple mistake but consider the situation where both types aren't clearly declared close to the `ckd_add` statement. Furthermore, choosing this type automatically can be difficult since it would not depend on the operands themselves.

There are some benefits to performing a manual rewrite however. A skilled programmer could avoid issues revolving around incorrect rewrites, even writing them in a very readable manner with tailored error handling. These benefits are in direct contrast to using an automated approach which will produce code that is less readable, fails to capture some contextual information, and could inject a bug into the source. Furthermore, this approach adhears to the other requirements trivially such as modifications being made at the source level, comments being preserved, and exempting lines from rewrites. For small codebases, a manual rewrite will almost always be the best approach; however it will be the most expensive option for any moderately size codebase.

3.3.2 Using Regex

We believed the next realistic approach is using a simple regex matcher to find all the operations, then apply some basic transformation to those matches. This approach may be possible, however to get this right would require a significant amount of development.

The regex approach involves writing a script that takes in files, applies a regex

matcher to locate all operations that could be rewritten, then modifies the files such that these operations are now checked. The main issue with this approach has to do with types. The rewrite must be able to identify which operations can be rewritten based on types, then select intermediate types to use when a rewrite is performed.

Since types of the operands and any sub-expressions aren't always written out in the source, the regex tool would need to infer these values in the same way the compiler does, which would be very complicated. As mentioned in Chapter 2, integer promotion rules and casts are complex and a source of integer arithmetic errors themselves. A regex tool would not only need to perform complex parsing for each operand (which could include scopes, files, declarations and casts) but calculate how the compiler would change the types of the operands during the operation being rewritten in order to be accurate. It may be possible with more capable tools like semgrep [32], which has existing capabilities for matching expressions based on types, however even writing these rules would be difficult considering you would need to write matching involving all of the integer types.

Another issue with this that is shared by any automatic approach is rewrites will likely not be as readable as a human rewrite. Certain coding standards can be adheared to, however writing readable code is contex dependant and is unlikely to be performed well by an automatic tool that will need to insert multi-line rewrites for operations like +, which could be embedded in all kinds of other expressions.

The issue of preserving comments or using them as a means to exempt a line from a rewrite seems quite possible. However, the tool would need to keep track of any changes if they are made inline which could also become a complex challenge.

A regex tool would indeed be automatic, which would be an upgrade from the manual approach, however even a basic rewriting tool would be very complex to implement correctly using regex.

3.3.3 Compiler Tools

The final approach and the approach that was ultimately the focus of this thesis is the compiler approach. This approach involves writing a tool that is integrated in the compiler toolchain to detect and rewrite statements with other parsing information like the AST and associated types.

This approach solves the largest issue that the other approaches, even the manual one, struggle to address. This is detecting the type information of a particular statement to both find and rewrite it accurately. The issue of calculating the types and selecting intermediate types in an expression is essentially a solved issue, solved by the compiler itself.

Although comments are considered to be "just for humans", compilers can still parse them and some compiler tools even interpret them for their own purposes. Because of this we know that it is plausible for compiler tooling to have access to comment information in some form.

Another benefit of this approach is it becomes easy to integrate with the broader build system. Build systems can be complex and fragile, and building a tool that integrates with a compiler already present in the build chain means there is less overhead than an entirely new tool and is less likely to introduce issues to the broader build system.

Besides the issue of readability shared with any automatic rewrite, the downsides to the compiler approach is compilers are very complex and there is a large learning curve to developing tools for them. That being said, this overhead is insignificant considering the issues with the other approaches.

It should be clear why the compiler approach was selected as the means to solve this problem over other approaches. The next section will discuss which compiler technologies were used to create a rewriter and why they were selected.

3.4 LLVM

At this point, it is established that our approach will be a compiler based one using Clang. The next step will be to investigate what tools and APIs are available to develop a transpiler using Clang.

Clang is the premier C/C++ language family compiler for the LLVM project. LLVM is an open source compiler infrastructure project that is home to a number of tools related to building compilers and toolchains. Before we explore the technologies provided by LLVM, it needs to be established what exactly is required to create the transpiler.

The first requirement is we have access to the AST. This allows us to get type information for the rewritten code and even locate operations that can be rewritten. An important note here is when different designs were being investigated, actually being able to rewrite the AST was seen as a requirement. The idea was in the case where types were collected from the AST, and we couldn't manipulate the source easily, we would manipulate the AST instead and convert that back into some sort of readable source. It was agreed that if a better solution could be found this should be scrapped, which it was. It is still important to understand that having full control of the AST was initially considered a requirement just in case the AST had to be rewritten.

The second requirement will be that we have access to the source from the AST, and that changes can be made to the source once a rewrite is generated. There is also a less firm third requirement, which is there is a good level of documentation on how to use the interface to help with development.

With these two requirements established, the next section will cover the available development interfaces provided by LLVM:

LibClang

LibClang is a high-level C API to interface with clang. LibClang is available for languages other than C and supports basic operations like iterating through the AST, however it does not grant full access to the AST. It also does not seem to support clang AST matchers, which would introduce some overhead parsing through the AST to locate operations that need to be rewritten. Furthermore, LibClang does not support source manipulation, which would need to be handled manually by the developer. There seemed to be a somewhat decent amount of documentation on how to use LibClang. The idea of writing the tanspiler in another language like Python was very appealing initially, however the benefits provided by some of the later tools made the cost of developing in C++ worth the extra work. Writing the transpiler with LibClang does seem possible, however it was not seriously considered given these limitations.

Clang Plugins

The Clang Plugin interface allows developers to insert operations as part of the compilation process. This could be to perform linting or generate new build artifacts. The plugin interface would provide full access to the AST, an upgrade from the LibClang approach. Additionally, getting and manipulating source was possible through the `SourceManager` and `Rewriter` interfaces, however this approach did seem a little more unstable compared to options that LibTooling provides. Clang Plugins also offer access to AST matchers but the canonical method for traversing the AST is through the `RecusriveASTVisitor` pattern. This was the approach that was outlined in the documentation and came up with a few issues. The first one was without AST matchers, the developer needed to match on nodes using a series of complex conditions. This made the matching process quite similar to just using LibClang and failed to resolve that overhead. The other issue was it proved to be

quite difficult to get the types of different nodes involved in each operation, which was eventually why this approach was scrapped.

The Clang Plugin approach was the first approach considered given it's promising features, however ultimately was not used due to the aforementioned issues.

LibTooling

LibTooling is Clang's advanced interface for standalone tools. LibTooling is the foundation for popular developer tools provided by Clang including clang-check, clang-fixit, clang-format, and the tool this project eventually extended: clang-tidy. Like Clang Plugins, LibTooling gives full control of the AST to the developer with the added benefit of being designed to run on specific files instead of all the files processed by the compiler (which is the case for Clang Plugins). Furthermore, LibTooling contains functions for both suggesting and applying fixes to code inline using the Diagnostics interface. Finally, the canonical method for interacting with the AST in LibTooling was via AST matchers, making it feasable to write sussinct matchers to find valid operations to rewrite.

LibTooling as an interface was already promising as an means to make a rewriter, however it was clang-tidy that was extended to perform rewrites, which is discussed in detail in the next section.

3.4.1 Clang-tidy

clang-tidy is a clang-based C++ "linter" tool [21]. It is designed to be used by developers to find and quickly fix programming errors that can be found via static analysis. These errors can range from code style errors to misusing programming interfaces. Clang-tidy is run as a standalone tool that takes in files and runs "checks" specified by the user. A check will find some pattern in the source files provided and suggest some change to the source to fix the error associated with that pattern.

This usage is already what we had in mind for a rewrite tool, so all that is left is to write a new "check" that finds and fixes binary operations so they use checked arithmetic. To further support this approach, there exist modernization checks that aim to update codebases by performing transformations. Some examples are the `modernize-use-auto` (will rewrite statements to use the `auto` keyword where possible), `modernize-use-nullptr` (rewrite so that `NULL` is written as `nullptr`), and `cert-*` (a number of checks for secure coding, however they do not generate fixes).

Writing a clang-tidy check involves two main steps: writing and registering AST matchers, and writing a check that produces some diagnostic output based on matches. Since clang-tidy is build using LibTooling, any new check that is added to the clang-tidy will have full access to the LibTooling interface.

Some final benefits to using the clang-tidy is it is very easy to omit lines from being rewritten via the `//NOLINT` directives, satisfying requirement 5. Additionally, clang-tidy makes it easy to provide arguments we can use in our check, which gives us the ability to inject custom handlers and satisfy requirement 4.

3.5 Rewriting a Statement

3.6 User Flow

Now armed with the design that was settled on for this tool, we need to illustrate the steps an end-user would perform to use our checked arithmetic clang-tidy plugin.

1. Write a clang-tidy configuration

When a statement is rewritten, it will contain a check that calls the checked arithmetic function. If this function returns true, an overflow has been detected and an error will occur. As mentioned in the requirements, this error handling functionality is entirely context dependent and will need to be written by the developer themselves. More specifically, in order for the clang-tidy to perform a write auto-

matically, it needs to have code provided by the developer to inject in the case of a detected overflow. This is the purpose of the clang-tidy configuration file. The developer should provide a string that contains the source to inject into the handler, as well as any header files that should be added to the modified file. The intended pattern here is to provide a line like `handle_error()`, which would be a function present in the header file also specified. An example clang-tidy configuration can be found in Appendix B.

2. Prepare the source

This step is optional but would be required if the developer knows there are certain operations that there are operations that they do not want to modify. They would proceed to the lines containing operations they do not want to modify and use an appropriate clang-tidy ignore directive to omit them from any changes.

3. Run the check

Running the check will simply involve running clang-tidy with the `modernize-use-checked-arithmetic` plugin with the appropriate configuration defined in the first step. Note that for lines that contain multiple operations that could be rewritten, only one rewrite will take place, leaving others on that line untouched. This may seem like a limitation that prevents rewrites of more complex statements, however there is a simple solution: run the check more than once. This means any remaining statements in the source, even source that uses the checked arithmetic will get rewritten in successive passes. However, before running clang-tidy again, it is suggested to continue to the next step.

4. Run a formatting tool

Unfortunately, there is no way to avoid formatting issues when creating multi-line `FixItHint` modifications. There are ways to interface with Clang's Lexer to cal-

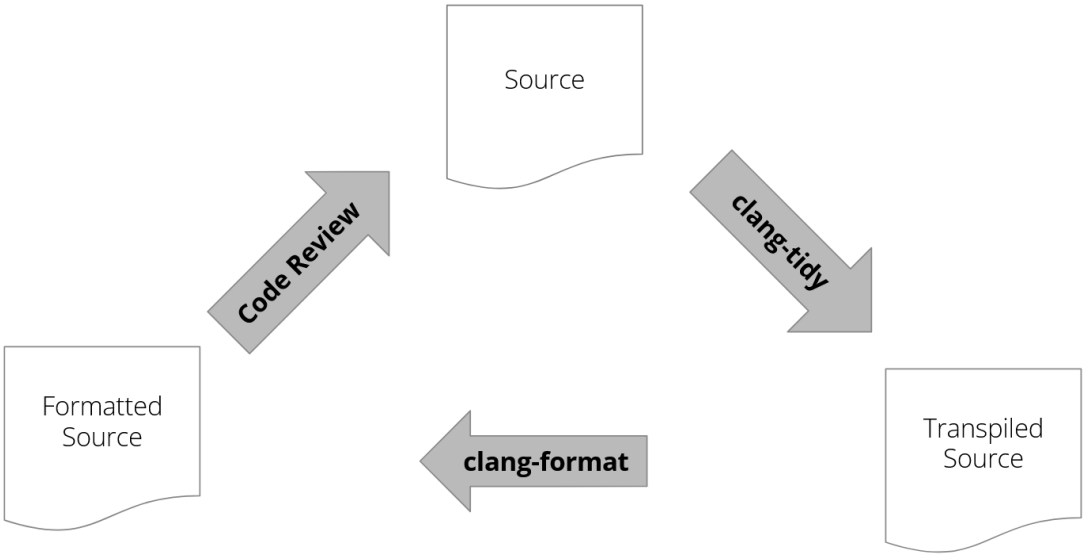

Figure 3.1: Steps involved in transforming code with clang-tidy

culate line indentation before modifications are created, however there is a simpler solution: run clang-format on the modified files. As long as the rewrites are structured correctly, clang-format will automatically fix any formatting issues introduced by the check.

5. Refine

Given there are a number of cases that aren't in scope for this rewrite tool, there may be statements that are rewritten incorrectly and must be either fixed or restored and omitted from other clang-tidy rewrites. This will generally involve running any tests or even performing a manual audit of the changes (which is possible given the changes are introduced at the source level). Once this is done, the user would return to step 3 to continue rewriting checks that were missed by earlier passes.

These steps are visualized in the figure Figure 3.1.

3.7 Limitations

Although the design outlined in this chapter is feasible and satisfies our requirements, there are some limitations with this approach that should be addressed.

The largest issue is a limitation imposed by the C23 checked arithmetic headers: rewrites cannot be performed on all operations that could result in overflow. This is either due to type restrictions or the operations themselves. Approaches for how these limitations can be overcome are discussed in Chapter 6.

This approach is also specific to the LLVM family of compiler infrastructure, which means this approach will not work for codebases that require other compilers. Clang-tidy can technically work with C files meant for other compilers even if clang itself is unable to compiler them. The issue is the type information collected from the AST could error under certain circumstances, rendering the check ineffective. This issue is out of the scope of this project, however solving this design issue would catapult the effectiveness of such a tool.

It was mentioned that lines can be omitted from any changes from a clang-tidy check via the `//NOLINT` directive, however this will prevent changes to the entire line when a developer may want only a single expression to be omitted. This is not a major issue, however this could be a point for improvement in future work.

According to the Clang documentation, the AST matcher interface is not stable and is subject to change [20]. Although this is a research endeavor and not a commercial project meant to have a long shelf life, this could be a deal breaker for a longer running project.

Finally, this approach will inherit the issue of difficult to read translations due to the automated nature of the tool. This is an unavoidable problem when designing an automated tool, however some ideas are also presented in Chapter 6.

4

Implementation

This chapter will go into detail about how the `use-checked-arithmetic` clang-tidy plugin was developed. This includes the general steps for writing a check, the domain specific language used for matching on AST nodes, and specific challenges that were overcome with relation to this particular check. At the end of the chapter, there will be walkthroughs of specific rewrites for different types of operations and how they work.

4.1 How to Write a Clang-Tidy Check

Clang-tidy is an advanced linting tool that utilizes all of the features LibTooling provides for detecting patterns in source to provide warnings and potential fixes. The most basic check is made up of two parts: a set of matchers and the check itself.

Matchers are constructed using the LibASTMatcher interface, which allows programmers to us a declarative domain specific language (DSL) to describe patterns in the AST to "match". When a match is found, the matched expression is passed to the `check()` function to handle the rest. These matchers are registered to a check via the `registerMatchers()` function.

The `check()` function can really be anything, however the most basic check will have a diagnostic output that prints the matched expression followed by an explanation of what the check has found. This is a very basic example, and really the possibilities for what the check can do from here are endless. The main functionality of a code modernization check will be to generate a rewrite of the matched statement. LibTooling also provides a convenient `FixItHint` interface for generating suggested code fixes beside their warnings that a user can apply with the `--fix` or `--fix-errors` flag passed to clang-tidy. Of course it goes without saying that any helper functions or entire check implementations that exist elsewhere can be called within the `check()` function, which is often the case for many of the clang-tidy checks that already exist.

These are the two components a developer needs to implement to create a clang-tidy check, however there are some fundamental questions that need to be answered. The first is where the code for the check actually exists? The answer is in the clang-tidy binary and source tree itself. This has a number of implications, namely that a developer needs to check out the LLVM project in order to build a clang-tidy check and proceed with a build of LLVM (technically not all of it, mostly clang and it's libraries). This can be a daunting task but once completed should not be a barrier to further development.

Clang-tidy is part of the Clang Tools Extra project and thus lives in the `clang-tools-extra/clang-tidy` subdirectory of the LLVM project. In this directory there are subdirectories that organize the different checks available to clang-tidy by their main purpose. To name a few these include: performance for source-level performance optimizations, portability for writing portable code, and modernize for updating old codebases. The check we are writing will be in the modernization subdirectory.

A check writer could create source and header files in one of these directories

to contain their check, then register their check within their specific module via the `registerCheck()` function. Luckly there is a nice script that LLVM provides named `add_new_check.py` that automatically handles creating template files, registering the check, and creating documentation. The developer is then left to write their matchers and check without having to deal with any other overhead. Additionally, there is a `rename_check.py` file that will rename the check, which also involves editing the check registration.

Now with this overview, let's discuss how to write a matcher.

4.2 Writing AST Matchers

When Clang compiles a program, it will eventually generate an Abstract Syntax Tree (AST) that captures the semantics of the program it is compiling in a structured format to be used later. The AST is made up of a set of *nodes*, with the topmost being a translation unit declaration. The rest of the tree will be made up of `Type`, `Decl`, `DeclContext`, and `Stmt` nodes (or nodes that derive from these types). Type nodes are related to C statements for creating types like `typedef`. `Decl` nodes represent a declaration or definition like a variable, struct, or function. `DeclContext` is a special base-class used by other nodes to store other declarations they may contain. Finally, `Stmt` nodes will represent a single statement. It is important to note that arithmetic operations will be represented by `Expr` nodes, which is a subclass of `Stmt`.

AST matchers allow programmers to write out AST patterns to locate in a concise and declarative manner. The basic structure of an AST matcher is a *creator function*, which can contain more matchers within it to make the matcher more specific. For example, say we would want to make a matcher that would match on all binary operations in the AST. This would be written as `binaryOperator()`. We would then make this matcher more specific by adding more matchers to the binary operator matcher like arguments. If we want to only match on additions, we would write this

as `binaryOperator(hasOperatorName("+"))`. The matcher can quickly become expressive with generic matchers `allOf()` (similar to `&&`), `anyOf()` (`||`), `unless()` (!), just to name a few. These matchers can be specific as well like the `gnuNullExpr` matcher, which will match on GNU `__null` expressions.

4.3 Implementing a Check

4.3.1 Writing the Matcher

As outlined in previous chapters, the clang-tidy check needs to locate all of the arithmetic operations that could be rewritten using checked arithmetic. With this information, the first step should be to write out the type of node we would want to match on, which would be a binary operator and would be written as:

```
binaryOperator()
```

This will match on *any* binary operation, including comparison operators like `<`, `!=` and logical operators like `&&`. We will only rewrite the +, -, and * operators, so we need to narrow our matcher to only these binary operations like so:

```
binaryOperator(hasOperatorName("+", "-", "-"))
```

This seems like a valid matcher, however it is missing an important restriction: the types of the operators. This is a restriction imposed by the checked arithmetic functions, which may only operate on integers. We need to now update our matcher to check both operands are integers using the `isInteger()` matcher like so:

```
binaryOperator(
        hasOperatorName("+", "-", "*"),
        hasLHS(ignoringImplicitCasts(hasType(isInteger()))),
        hasRHS(ignoringImplicitCasts(hasType(isInteger())))
)
```

Note that we need to introduce the `ignoringImplicitCasts()` matcher to get to the child nodes we want to interact with. This is because the AST contains

`ImplicitCastExpr` when an implicit cast is generated by a particular statement, which we do not want to match on. There are ways to automatically ignore these nodes by setting the traversal mode of the matcher, however these nodes were seen as potential resources for gathering type information and thus matchers were developed with them in mind.

In an assignment operation, we also need to have type checks on the result of the operation. To do this, we can use the `hasParent()` matcher to traverse to the parent of our binary operation, then check the type of that node. This will allow us to capture any casts or assignments to a different type for us to correctly check if the resultant will actually fit in the destination. The final matcher is as follows:

```
binaryOperator(
        hasAnyOperatorName("+", "-", "*"),
        hasParent(expr(hasType(isInteger()))),
        hasLHS(ignoringImpCasts(hasType(isInteger()))),
        hasRHS(ignoringImpCasts(hasType(isInteger())))
)
```

LLVM also provides the clang-query tool, which will give developers a type of REPL to run matchers on an AST given by a particular source. This tool was invaluable for iterating on the matcher and quickly testing it on examples that were written by hand without having to rebuild clang-query each time.

4.3.2 Binding to Subexpressions

It is one thing to match to an expression, however the check will need to retrieve specific nodes to perform the rewrite. This is done by calling the `bind()` method like so `binaryOperator().bind("binary-operation")`, where the string `binary-operation` will serve as an identifier for the matched binary operator node. This introduces an issue with the existing query: we cannot call `bind` on the `isInteger()` matcher. A solution to this is to use a new `hasLHS` and `hasRHS` matcher specifically for binding to the subexpression like so:

```
StatementMatcher makeNonAssignmentMatcher() {
    return binaryOperation(
    hasParent(expr(hasType(isInteger()))).bind("parent-expr"),
        hasAnyOperatorName("+", "-", "*"),
        hasLHS(ignoringImpCasts(hasType(isInteger()))),
        hasLHS(ignoringImpCasts(expr().bind("argOne"))),
        hasRHS(ignoringImpCasts(hasType(isInteger()))),
        hasRHS(ignoringImpCasts(expr().bind("argTwo"))))
    .bind("Non-AssignmentOp");
}
```

Notice that there are now `bind()` calls on the whole binary operator statement and each argument. With this, we can now match on non-assignment operations and bind to the specific arguments we will need to perform our rewrite.

4.3.3 Registering the Matcher

Now that the matcher is written, we can register our matcher with the `registerMatchers()` function. In our implementation, the `addMatcher()` function is called on the `MatchFinder` object passed to the function to register matchers. Note we will need to match three different matchers: the non-assignment matcher, the assignment matcher, and the unary matcher. To simplify this, three helper functions were created to actually generate these matchers: `makeNonAssignmentMatcher()`, `makeAssignmentMatcher()`, and `makeUnaryMatcher()`. Each of these functions are called to generate a matcher, which is subsequently registered.

4.3.4 Implementing `check`

Once a match is is found, the `check` function will be called to handle the rest of clang-tidy's behavior. Since we are handling multiple types of expressions, the code will need to determine what kind of expression was matched to know how to rewrite it. This is done by calling the node with `getNodeAs<BinaryOperator>("TYPE")`, where `"TYPE"` is the type of binary operator we are testing for. For example, we

wrote a matcher that binds the matched node to `"Non-AssignmentOp"`. To test if we have this operation, we would use:

```
if (Result.Nodes.getNodeAs<BinaryOperator>("Non-AssignmentOp"))
{
    // handle non-assignment operator
}
```

We can use this to switch on the match type and call the appropriate rewrite handler.

4.3.5 Implementing the rewrite

Once we have a matched statement, we need to suggest a rewritten version of that statement. So far we have been talking about the process for writing a check for a non-assignment operation, so this section will cover implementing a rewrite for that type of statement.

To reiterate how a rewrite will be performed, say we have the following statement:

```
int a = INT_MAX;
int b = 5;
int c = a + b;
```

This has an overflow in it when 5 is added to `INT_MAX`. To detect this overflow, we need to rewrite it using `ckd_add()` in a statement expression like so:

```
int a = INT_MAX;
int b = 5;
int c = ({
   int dest;
   int argOne = a;
   int argTwo = b;
   if (ckd_add(&dest, argOne, argTwo)) {
      assert(0);
   };
   dest;
});
```

Note that we need the types of `a`, `b`, and `c` in order to write this out. Furthermore, we actually need the expressions on the LHS and RHS of the addition, and the addition itself (the operation).

All of the nodes that we bound to in the matcher have a `getType()` function that allows us to retrieve both the type and any type qualifiers of that node.

A note about the resultant type of the expression (the type of `dest`), there are actually two different matchers that are used: one for operations that don't have an implicit cast and one for matchers that do. They will use the same fix, however this is used to detect if the result of the addition is being cast to a new type, potentially resulting in an overflow. If there is no cast, it is safe to just use the resultant type of the binary operation as outlined in the design chapter.

Once all of the types are collected, the source of the LHS and RHS needs to be retrieved. This is done through a helper function `getExprSourceString`, which will use the bounds of the passed expression and get the resultant source using the Lexer.

The next step is to determine which checked arithmetic function will need to be used. This is also accomplished through a helper function, which will simply match on the type of the operator and return a string containing the appropriate function to use.

The final step is to collect the handler code to place inside the if statement. This is simply a global string that is populated from the configuration at the start of the `check()` function.

With all these components, the replacement string can then be constructed like so:

```
auto replacement = "({ " + resultType + " dest;\n";
replacement += argOneType + " argOne = " + argOneSource + ";\n";
replacement += argTwoType + " argTwo = " + argTwoSource + ";\n";
replacement += "if(" + ckdFunc + "(&dest, argOne, argTwo)) {";
replacement += HandleCode + "\n};";
replacement += "dest;})";
```

This string is then used to create a `FitItHint`, which will apply the replacement across the source range of the original arithmetic expression.

4.4 Walkthrough of Assignment Operator Rewrites

The full matcher for locating assignment operations to rewrite is as follows:

```
return binaryOperation(
    hasAnyOperatorName("+=," "-=," "*="),
isAssignmentOperator(),
    hasLHS(ignoringImpCasts(hasType(isInteger()))),
    hasLHS(ignoringImpCasts(
        anyOf(expr().bind("dest"), integerLiteral().bind("dest")))),
    hasRHS(ignoringImpCasts(hasType(isInteger()))),
    hasRHS(ignoringImpCasts(
        anyOf(expr().bind("arg"), integerLiteral().bind("arg"))))
).bind("AssignmentOp");
```

The difference with this matcher as opposed to the non-assignment matcher is there is no need for matchers related to the parent and the operator names. We don't need to match to the parent because we can already get the destination type based on the LHS, which will be the same as the destination.

We need to capture the side-effect of the assignment operation, whatever it may be. This is different from the non-assignment rewrite because the destination is not an operand in the operation itself like it is here. Thus, this rewrite will collect a pointer to the destination, then dereference that pointer as an operand to the checked arithmetic macro. Consider the following code:

```
unsigned int a = 1;
unsigned int b = UINT_MAX;
a += b;
```

This approach will rewrite the code like so:

```
unsigned int a = 1;
unsigned int b = UINT_MAX;
({
    unsigned int* dest = &a;
    unsigned int arg = b;
    if(ckd_add(dest, *dest, arg)) {
        // Handle error
    }
    *dest;
});
```

This rewrite is captured in the check with the following code:

```
auto replacement = "({ " + destType + "* dest = " + "&" + destSource
+ ";\n";
replacement += argType + " arg = " + argSource + ";\n";
replacement += "if(" + ckdFunc + "(dest, *dest, arg)) {";
replacement += HandleCode + "\n};";
replacement += "*dest;})";
```

4.5 Walkthrough of Unary Operator Rewrites

Matching on unary operations is simple and done with the following matcher:

```
return unaryOperator(
    hasAnyOperatorName("++", "--"),
    hasUnaryOperand(ignoringImpCasts(hasType(isInteger()))),
    hasUnaryOperand(ignoringImpCasts(
        expr().bind("arg"))))
).bind("UnaryOp");
```

Unary operations will come in the form of a pre and post decrement and increment. The difference between increments and decrements will only be the checked macro to be used, however the order of the operation will make a big difference. For a post decrement, the value of the variable is evaluated before the operation takes place. This is essentially how a normal arithmetic operation will take place and thus the rewrite will be very similar to an assignment operation with on the RHS. Consider the following code that has a post-increment:

```
unsigned int a = 1;
++a;
```

This would be rewritten like so:

```
unsigned int a = 1;
({
    unsigned int* dest = &a;
    if(ckd_add(dest, *dest, 1)) {
        // Handle error
    }
    *dest;
});
```

This approach is using the same technique where we preserve the side effect of the operation by collecting the pointer of the destination then dereferencing it as an operand.

Now consider the following post-increment:

```
int a = 1;
a++;
```

In this case, we need to preserve the original value before the increment so that when it is "evaluated" the value before the operation is returned. Using this approach we would get the following rewrite:

```
int a = 1;
({
    int* dest = &a;
    int oldDest = *dest;
    if(ckd_add(dest, *dest, 1)) {
        // Handle error
    }
    oldDest;
});
```

The check itself is written to use a single matcher, so a switch statement is used to differentiate between the different operations and perform the correct transformation like so:

```
switch (MatchedExpr->getOpcode()) {
    case clang::UO_PreInc:
    case clang::UO_PreDec:
        // handle fix for prefix
        replacement = "({ " + argType + "* tmp = &" + argSource +
";\n";
        replacement += "if(" + ckdFunc + "(tmp, *tmp, 1)) {\n";
        replacement += HandleCode + "\n};";
        replacement += "*tmp;})";
        break;

    case clang::UO_PostInc:
    case clang::UO_PostDec:
        // handle fix for postfix
        replacement = "({ " + argType + "* tmp = &" + argSource +
";\n";
        replacement += argType + " oldTmp = *tmp;\n";
        replacement += "if(" + ckdFunc + "(tmp, *tmp, 1)) {\n";
        replacement += HandleCode + "\n};";
        replacement += "oldTmp;})";
        break;

    default:
        llvm::errs() <<"Unknown unary operator:  "
            <<MatchedExpr->getOpcodeStr(MatchedExpr->getOpcode())
<<"\n";
        return;
}
```

A complete copy of all the code written for this thesis, including the debug checks discussed later can be found in Appendix C

5

Evaluation

This chapter is related to the evaluation of the proposed design and subsequent implementation of the clang-tidy tidy check described in this thesis. It begins with defining several research questions to be answered by the evaluation, followed by a methodology for how these questions will be answered. It will finally end with real results observed from several real-world codebases to answer these research questions based on the metrics defined.

5.1 Research Questions

The following are a set of research questions that are posed as part of this evaluation.

RQ1: How comprehensive is the rewrite?

Fundamentally there is a question of how many expressions can the clang-tidy check rewrite relative to the total number of operations that can overflow. This is essentially measuring how much more secure a rewrite is given each rewrite is performed correctly. The implications of this will determine how effective or ineffective such a tool would be. An interesting detail here is figuring out how many rewrites are not

possible given the current checked arithmetic macros supplied by the C23 standard.

RQ2: What are common reasons lines are omitted from rewrites?

As with any static analysis tool, the issue of false positives and false negatives will be an important factor in how useful such a tool will be. This question will need to be investigated to determine how practical it is to perform rewrites without much manual intervention (i.e. commenting certain lines so they are exempt). Finding common reasons for exceptions is also a detail this question is trying to solve, which may reveal further limitations.

RQ3: What is the performance overhead of using checked arithmetic?

Once a tool or library is rewritten using checked arithmetic, what kind of performance overhead does this induce at runtime? As with the previous questions, this detail can be an important factor as to how practical such a tool is. This is especially true in safety critical systems, which often run as a real-time operating system and must meet certain performance requirements.

With these questions now defined, we move on to the methodology for answering said questions.

5.2 Evaluation Methodology

Finding answers to these questions will start with locating a set of targets to perform tests on. Once this is done, the evaluation itself will consist of several approaches. The main approach will utilize a script that will automatically calculate statistics based on warnings and outputs from clang-tidy. Beyond this, manual testing will be used to collect performance overhead statistics as well as qualitative measurements on the experience of performing a full rewrite and noting which expressions must be omitted.

5.2.1 Selecting targets

A set of target codebases to rewrite should be selected to evaluate the effectiveness of the clang-tidy check. Due to the nature of the check and the questions being answered, the targets should have source available and have a test suite to ensure that the rewritten code does not introduce any bugs. That being said, three targets were selected. The first is a relatively popular strings library *str.h*, which contains many complex statements that should be able to test the rewrite quite broadly. The second is *Monocypher*, a cryptography library that also contains a lot of complex operations. Given how close cryptography and security are, having a checked cryptography library is something that may be desired. Finally, *Binutils* was rewritten using the check. Binutils is GNU's tool suite for operating on binary files and is often the subject of testing software for research. These targets should be sufficiently complex to measure the effectiveness of the clang-tidy check.

5.2.2 Evaluating comprehensiveness

Comprehensiveness will be measured as how many arithmetic operations exist that can overflow verses how many operations are actually rewritten. Given the type restrictions of arguments passed to the checked arithmetic macros, it is expected that there should almost always be fewer rewrites than there are operations that may overflow.

The main approach to collecting this information is to write three clang-tidy checks: one "debug" check that will match on all $+$, $-$, and $*$ operations (and their variations) without type restrictions, another "debug" check with with type restrictions, and the real check which will also have type restrictions.

Note that there was some discussion about why even measure statements you cannot write given the current C23 macro definitions. This I believe highlights a major limitation of the current state of these macros, and yields a more accurate measure-

ment of how much more secure a codebase is. The debug check with type restrictions will measure "of the statements you can rewrite, how many get rewritten".

5.2.3 Evaluating common exceptions

Common exceptions will generally come down to the user experience: when a rewrite breaks something, what is a general pattern that causes this issue. This is difficult to quantitatively measure, so the evaluation of this question will be qualitative. That is, as different targets are rewritten, I will go through and locate issues I notice and have to manually address through an exemption comment. I will then take note of the issues to be used in the evaluation.

5.2.4 Evaluating performance overhead

Measuring performance will generally take the shape of running a test suite on a target and noting the time, performing a full rewrite, then running the test suite again and recording the time. Note that each run of the test suite under a timer should be performed ten times, then averaged to get a more accurate timing metric.

5.3 Results

Below are comments on the findings on the aforementioned targets.

5.3.1 str.h

This was the first target to be rewritten and was relatively straight forward. The statistics on how the plugin performed can be found in Table 5.1. The main metric for comprehensiveness is the fixes over potential replacements, which came out to 80.84%. This is a relatively high number of replacements but it is not 100%. We know that this is due to the type restrictions imposed on the checked arithmetic macros, since the typed debug plugin detected the same number of potential replacements as the number of replacements actually performed. Other interesting metrics are the

number of overlapping fixes, which sits at 2. When clang-tidy applies a fix, then encounters a fix on the same line, only one fix will be applied at a time. This is why the check may need to run multiple times. Since there are two overlapping fixes, we should expect to run the check at most 2 more times to get a complete rewrite.

Table 5.1 contains other interesting metrics like the distribution of different operations, which may give a better picture on the type of statements being rewritten and how much more coverage the clang-tidy check achieves by implementing rewrites for these statements.

Table 5.1: Plugin Statistics for str.h

Plugin	**Warnings**	**Result**
Debug Plugin	Potential replacements	595
	Unary operations	59 (9.92%)
	Assignment operations	271 (45.55%)
	Non-assignment operations	265 (44.54%)
Typed Debug Plugin	Potential replacements	481
	Unary operations	17 (3.53%)
	Assignment operations	233 (48.44%)
	Non-assignment operations	231 (48.02%)
Real Plugin	Replacements	481
	Unary operations	17 (3.53%)
	Assignment operations	233 (48.44%)
	Non-assignment operations	231 (48.02%)
	Overlapping fixes	2 (0.42%)
Total Coverage	Fixes / Potential Replacements	481 / 595 (80.84%)
	Unary Ops Fixed	17 / 59 (28.81%)
	Assignment Ops Fixed	233 / 271 (85.98%)
	Non-assignment Ops Fixed	231 / 265 (87.17%)

The rewrite experience was very straight forward for this library even considering the complex arithmetic involved in string manipulation. The plugin was run on the source directory three times to rewrite all possible statements, however notably macro related rewrites were not going through. At the time of writing it is unknown why

this is, however it is technically out of scope. Once the rewrites were performed and formatting was applied the library was built and test ran without error as if nothing had changed.

Performance was measured to come out to a 12% performance hit on average when checks were in place. This seems like a small number, however this could be too large an overhead in some applications. Although not in scope, the reason for this performance overhead could be the way the rewrite was implemented and the implementation of the check macros themselves.

5.3.2 Monocypher

The performance statistics for the rewrite of Monocypher can be found in Table 5.2. Compared with str.h, the clang-tidy check performed better with a total coverage of 85.06% of statements rewritten. There are also a number of dramatic differences between the number of overlapping fixes and the proportion of unary operations fixed. This means there are a lot of complex nested arithmetic statements, and a high number of unary operations that can be fixed by the check.

The process of rewriting Monocypher was not without challenge. This is because many of the statements that were rewritten actually took advantage of unsigned integer wraparound. This became an issue when running the tests with the default rewrite handlers, which would trigger `assert(0)` when an overflow was detected. This was good and bad news: potential integer overflows and wraparounds were being detected, but many were false positives. The easiest thing to do was to run the tests and find which assert was being triggered, then comment out the assert statement. This however quickly became a time consuming process given how many statements had intended wraparound. To get the rewrite to still work, an empty handler function was defined in the clang-tidy config for the check to use. This resulted in the rewrite being performed, but if overflow was detected, nothing would

happen and execution would continue. This highlights the important differences between overflow and wraparound, and that the plugin in it's current state does not differentiate between the two. This is partly due to the macros treating both overflow and wraparound the same, but the plugin could have an option to only match and rewrite signed arithmetic statements.

The performance measurement came out to about 4%. This is a dramatic difference from the rewrite of str.h, and could be due to the way that measurements are being taken. If this is an accurate reading however, this would be an interesting avenue for further investigation.

Table 5.2: Plugin Statistics for Monocypher

Plugin	**Warnings**	**Result**
Debug Plugin	Potential replacements	1185
	Unary operations	112 (9.45%)
	Assignment operations	583 (49.20%)
	Non-assignment operations	490 (41.35%)
Typed Debug Plugin	Potential replacements	1008
	Unary operations	111 (11.01%)
	Assignment operations	489 (48.51%)
	Non-assignment operations	408 (40.48%)
Real Plugin	Replacements	1008
	Unary operations	111 (11.01%)
	Assignment operations	489 (48.51%)
	Non-assignment operations	408 (40.48%)
	Overlapping fixes	192 (19.05%)
Total Coverage	Fixes / Potential Replacements	1008 / 1185 (85.06%)
	Unary Ops Fixed	111 / 112 (99.11%)
	Assignment Ops Fixed	489 / 583 (83.88%)
	Non-assignment Ops Fixed	408 / 490 (83.27%)

5.3.3 Binutils

Of the targets selected, Binutils is by a large margin the largest and most complex codebase to be rewritten. As explained in the following subsection, the rewrite was not successfully built. Nonetheless, a rewrite was attempted and the statistics can be found in Table 5.3.

These statistics demonstrate how large and complex the codebase is, with 174 overlapping fixes and 50.11% rewrite coverage. The overlapping fixes is much larger than the previous targets, and becomes a more serious issue when considering the amount of time clang-tidy ran to achieve a single rewrite pass: 5 minutes. If 174 rewrites are required to ensure all statements are rewritten, this would take 14 hours to complete. This is a worse case scenario, however even a fraction of this time is not insignificant. The lower number of expressions rewritten demonstrates how much the impact these type restrictions have on the rewrite, which is at 100% for expressions with supported types.

The rewrite itself was not successful however. There were several cases that caused the tool to break, these included issues with clang-format, variable names, and cases where array accesses had arithmetic expressions in them.

The issue with clang-format was that Binutils expects a very specific ordering of header file imports. Since clang-format was being run each time changes were made, the ordering of imports was being changed which caused a number of build issues.

Variable names became an issue, where temporary variables like `dest` or `argOne` were shadowed. The compiler complained about these, and they were too numerous to change by hand. The solution to this is to generate random temporary names, however this was not implemented in the time alloted.

Finally, there were cases where array indexes where being modified. I believe this was an issue with selecting types, where an array subscript was given the type `char`

when it should have been some other integer type. This is likely a bug in the AST matcher.

No performance metrics were collected since no binary was produced to run performance metrics on.

Table 5.3: Plugin Statistics for Binutils

Plugin	**Warnings**	**Result**
Debug Plugin	Potential replacements	6974
	Unary operations	1243 (17.82%)
	Assignment operations	3439 (49.31%)
	Non-assignment operations	2292 (32.86%)
Typed Debug Plugin	Potential replacements	3495
	Unary operations	684 (19.57%)
	Assignment operations	1599 (45.75%)
	Non-assignment operations	1212 (34.68%)
Real Plugin	Replacements	3495
	Unary operations	684 (19.57%)
	Assignment operations	1599 (45.75%)
	Non-assignment operations	1212 (34.68%)
	Overlapping fixes	174 (4.98%)
Total Coverage	Fixes / Potential Replacements	3495 / 6974 (50.11%)
	Unary Ops Fixed	684 / 1243 (55.03%)
	Assignment Ops Fixed	1599 / 3439 (46.50%)
	Non-assignment Ops Fixed	1212 / 2292 (52.88%)

This chapter demonstrates that while this approach does work in some capacity, however there are a number of issues related the implementation, type restrictions, the time it takes to actually perform the rewrites. Although the rewrite of Binutils was not successful, the success of the other two targets should demonstrate the applicability of this approach when done right. Furthermore, metrics on how type restrictions affect the theoretical capabilities of such a tool were collected.

6

Concluding Remarks

This thesis was a deep exploration into the world of language design, integer safety, and compiler infrastructure. In this chapter, we will conclude this work with an overview of the final deliverables, future work, the learning outcomes, and a personal reflection on the thesis.

6.1 Deliverables

As covered in the previous chapter, the main deliverable is a clang-tidy plugin that will automatically locate and rewrite statements to use the new C23 checked arithmetic. In addition to this a testing script is provided that was used for the evaluation of the plugin. These deliverables are supplemented by log files generated by the test script containing the generating warnings and final statistics for each plugin, as well as some basic C files that were written as part of initial testing. All of these deliverables have been placed in a Git repository located here.

6.2 Future Work

Integer safety is an ongoing problem for software security and there are many aspects of this issue that have yet to be explored. With regard to the approach this thesis took, the following are some topics that would be relevant next steps for exploring this issue.

6.2.1 Multithreading

Concurrency complicates the use of checked arithmetic functions significantly. Race conditions would need to be considered for every expression involved in a rewrite, and even the checked arithmetic functions themselves. Coming up with a solution may involve using mutexes or semaphores in strategically placed locations in the rewrite, but this could introduce overhead and could be dependent on the system itself. Solving this issue would significantly improve the practicality and real-world application of this approach.

6.2.2 Updates to C23

The C23 standard mentions future plans for rewriting the checked arithmetic macros to take just two arguments. If this were the case, the current design of the plugin would need to be updated and questions about the resultant type and how to calculate correctly would be a major question.

6.2.3 Maximizing the Utility of Checked Functions

Since the checked arithmetic macros only take a specific set of types, they are unable to detect overflow for the majority of arithmetic operations. There may be a way to use carefully constructed casts to cast these types to be used in the checked arithmetic functions before being recast to their original values. The main problems to solve with this approach would be selecting the correct types to both preserve

original values, detect overflow, and preserve the semantics of the original operation.

6.2.4 Working Through Modernizing Larger Codebases

The current plugin does have issues can does need to have more work done on it to be appropriate for completely automatic rewrites of larger codebases. This would involve going through the cases in which the current rewrite rules do not work, and either fixing the rewrite or coming up with better matchers to avoid rewriting statements that are not supported.

6.2.5 Non-Statement Expression Rewrites

Statement expressions are not currently part of the C standard, and thus aren't supported by a number of compilers. Statement expressions are a keystone for the approach taken by this thesis, however future work could try to use approaches that don't require statement expressions to achieve a more portable solution. This would likely involve more complex rewrites that are less generic.

6.3 Learning Outcomes

A number of interesting and important skills were collected and exercised during this thesis. Working with LLVM and Clang was a major theme of this thesis and is a marketable skill. I learned very specific details about how Clang works, the AST and AST matchers, and how to interact with Clang's C++ APIs. Furthermore, using Visual Studio code and building LLVM was a major learning experience, especially on Windows.

Learning about integer safety was another learning outcome. I learned not only in-depth details on how integers work according to the C standard but many of the cases where issues can arise and how they can be mitigated. Besides integers I learned a lot about the C standard, which I believe has made me a better C programmer.

6.4 Reflection

The student that started this thesis is not the student that is finishing it. This thesis has been one of the most difficult but rewarding experiences I have had at Carnegie Mellon. The initial ideas about what this thesis would cover were very broad and the direction of the project has narrowed significantly since then. I am happy with the result of this thesis, especially considering the struggles went through to complete it. I believe the main hurdles have been related to this narrowing of scope. Initially, my main interest was to work on embedded systems security and what the topic turned out to be felt like a departure from the initial motivations I had. Initially this made the project difficult to complete from both a personal and technical level. I did not have much experience with C++ and the process of even building LLVM, as opposed to my experience with embedded systems, security design, and ARM assembly. Even so, I still managed to make a clang-tidy plugin, learn enough C++ to interact with LLVMs APIs, and I now have a much deeper understanding of C as a language. I see this project as a success and plan to interact with the LLVM community in the future.

Bibliography

[1] S. Baase, *A Gift of Fire: Social, Legal, and Ethical Issues for Computing and the Internet*, ser. Alternative Etext Formats. Pearson Prentice Hall, 2008. [Online]. Available: https://books.google.com/books?id=GcBaS87q74gC

[2] B. Brosgol and C. Comar, "Do-178c: A new standard for software safety certification," in *Proceedings of the 22nd Systems and Software Technology Conference (SSTC)*. Salt Lake City, Utah: AdaCore, Apr. 2010, sponsored in part by the USAF. Approved for public release; distribution unlimited; Accessed on 2025-04-26. [Online]. Available: https://apps.dtic.mil/sti/pdfs/ADA558107.pdf

[3] Clang Developer Community, *Clang Language Extensions*, LLVM Project, 2025, accessed on 2025-04-26. [Online]. Available: https://clang.llvm.org/docs/LanguageExtensions.html

[4] cppreference.com contributors. (2022) Compiler support for c23 - cppreference.com. Accessed on 2025-04-26. [Online]. Available: https://en.cppreference.com/w/c/compiler_support/23

[5] ——. (2025) Replacing text macros - c preprocessor. Accessed on 2025-04-26. [Online]. Available: https://en.cppreference.com/w/c/preprocessor/replace

[6] Domars, "X86 architecture - windows drivers," accessed on 2025-04-26. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

[7] Federal Aviation Administration, *Software Approval Guidelines*, U.S. Department of Transportation, Federal Aviation Administration Std. Order 8110.49A, Mar. 2018, office of Primary Responsibility: AIR-600, Policy and Innovation Division; Accessed on 2025-04-26. [Online]. Available: https://www.faa.gov/regulations_policies/orders_notices/index.cfm/go/document.information/documentID/1032976

[8] GCC Developer Community, *GCC Built-in Functions to Perform Arithmetic with Overflow Checking*, GNU Project, 2025, accessed on 2025-04-26. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

[9] GCC Team. (2024) Gcc 14 release series. Accessed on 2025-04-26. [Online]. Available: https://gcc.gnu.org/gcc-14/

[10] GNU Project, *GCC Code Generation Options*, https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html, 2024, accessed on 2025-04-26.

[11] IBM, "Ilp32 and lp64 data models and data type sizes," https://www.ibm.com/docs/en/zos/2.3.0?topic=environments-ilp32-lp64-data-models-data-type-sizes, accessed on 2025-04-26.

[12] *Programming languages – C*, International Organization for Standardization Standard 9899, 1990.

[13] *Information technology — Programming languages — C*, International Organization for Standardization Std. ISO/IEC 9899:2024, 2024, accessed on 2025-04-26. [Online]. Available: https://www.iso.org/standard/82075.html

[14] ISO/IEC JTC1/SC22/WG14, "Iso/iec jtc1/sc22/wg14 — c," 2025, official working group for the C programming language standardization; Accessed on 2025-04-26. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/

[15] B. Kelechava, "The origin of ansi c and iso c," Nov 2024, accessed on 2025-04-26. [Online]. Available: https://blog.ansi.org/2017/09/origin-ansi-c-iso-c/

[16] S. Klabnik and C. Nichols, *The Rust Programming Language*, 2nd ed. San Francisco, CA: No Starch Press, 2023.

[17] J.-L. Lions and I. Board, "Report by the inquiry board on the ariane 5 flight 501 failure," 1996, accessed on 2025-04-26. [Online]. Available: https://web.archive.org/web/20000815230639/http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html

[18] LLVM Project. (2024) Clang 19.1.0 release notes. Accessed on 2025-04-26. [Online]. Available: https://releases.llvm.org/19.1.0/tools/clang/docs/ReleaseNotes.html

[19] ——, *Clang Command Line Argument Reference*, https://clang.llvm.org/docs/ClangCommandLineReference.html, 2024, accessed on 2025-04-26.

[20] ——, "Clang documentation: Choosing the right interface for your application," https://clang.llvm.org/docs/Tooling.html, 2025, accessed on 2025-04-26.

[21] ——, "Extra clang tools documentation: Clang-tidy," https://clang.llvm.org/extra/clang-tidy/, 2025, accessed on 2025-04-26.

[22] D. McCandless, "Million lines of code," https://informationisbeautiful.net/visualizations/million-lines-of-code/, 2013, accessed on 2025-04-26.

[23] OpenHub contributors, "Tensorflow - openhub," https://openhub.net/p/tensorflow, 2025, accessed on 2025-04-26.

[24] Oracle Corporation, *Class Integer*, 2017, accessed on 2025-04-26. [Online]. Available: https://docs.oracle.com/javase/9/docs/api/java/lang/Integer.html#Integer-int-

[25] ——, *Class Math*, 2022, accessed on 2025-04-26. [Online]. Available: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Math.html

[26] ——, "Biginteger class," https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html, 2025, accessed on 2025-04-26.

[27] M. Polacek. (2014) Gcc undefined behavior sanitizer - ubsan. Red Hat Developers. Accessed on 2025-04-26. [Online]. Available: https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan

[28] Python Software Foundation, *Built-in Types — Python 3 Standard Library*, 2024, accessed on 2025-04-26. [Online]. Available: https://docs.python.org/3/library/stdtypes.html

[29] ——, *OverflowError — Python 3 Built-in Exceptions*, 2024, accessed on 2025-04-26. [Online]. Available: https://docs.python.org/3/library/exceptions.html#OverflowError

[30] D. M. Ritchie, "The development of the c language," *SIGPLAN Not.*, vol. 28, no. 3, p. 201–208, Mar. 1993, accessed on 2025-04-26. [Online]. Available: https://doi.org/10.1145/155360.155580

[31] R. C. Seacord, *Secure Coding in C and C++*, 2nd ed. Addison-Wesley Professional, 2013.

[32] Semgrep, Inc., "Semgrep: Lightweight static analysis for many languages," https://semgrep.dev/, 2025, accessed on 2025-04-26.

[33] H. Spencer. (1992, Feb.) Re: Why is this legal? Message posted to comp.std.c; Accessed on 2025-04-26. [Online]. Available: https://groups.google.com/g/comp.std.c/c/ycpVKxTZkgw/m/S2hHdTbv4d8J

[34] D. Svoboda, "Towards integer safety," Mar. 2021, working Draft Proposal; Accessed on 2025-04-26. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2683.pdf

[35] The Clang Team, *UndefinedBehaviorSanitizer*, LLVM Project, 2025, accessed on 2025-04-26. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[36] The Rust Project Developers, "num 0.4.3," https://docs.rs/num/latest/num/index.html, 2024, accessed on 2025-04-26.

[37] G. Waldron, "Arianespace aims high in asia-pacific," Oct 2019, accessed on 2025-04-26. [Online]. Available: https://www.flightglobal.com/arianespace-aims-high-in-asia-pacific/120757.article

Appendix A

Rules on Calculating a Common Real Type

The rules for calculating a common real type for a mix-typed operation are as follows [13]:

1. If one operand has decimal floating type, the other operand shall not have standard floating, complex, or imaginary type.
   - If the type of either operand is `_Decimal128`, the other operand is converted to `_Decimal128`.
   - Otherwise, if the type of either operand is `_Decimal64`, the other operand is converted to `_Decimal64`.
   - Otherwise, if the type of either operand is `_Decimal32`, the other operand is converted to `_Decimal32`.
2. Otherwise, if one operand is `long double`, `long double complex`, or `long double imaginary`, the other operand is implicitly converted as follows:
   - Integer or real floating type to `long double`
   - Complex type to `long double complex`
   - Imaginary type to `long double imaginary`

3. Otherwise, if one operand is `double`, `double complex`, or `double imaginary`, the other operand is implicitly converted as follows:
    - Integer or real floating type to `double`
    - Complex type to `double complex`
    - Imaginary type to `double imaginary`
4. Otherwise, if one operand is `float`, `float complex`, or `float imaginary`, the other operand is implicitly converted as follows:
    - Integer type to `float` (the only real type possible is float, which remains as-is)
    - Complex type remains `float complex`
    - Imaginary type remains `float imaginary`
5. Otherwise, both operands are integers. Both operands undergo integer promotions; then, after promotion, one of the following applies:
    - If the types are the same, that type is the common type.
    - Else, if the types are different:
        - If the types have the same signedness (both signed or both unsigned), the operand with the lesser conversion rank is implicitly converted to the other type.
        - If the types have different signedness:
            * If the unsigned type has conversion rank greater than or equal to the signed type, the signed operand is converted to the unsigned type.
            * Else, if the signed type can represent all values of the unsigned type, the unsigned operand is converted to the signed type.
            * Otherwise, both operands are converted to the unsigned counterpart of the signed operand's type.

Appendix B

Example Clang-tidy Config

```
Checks: '-*, modernize-use-checked-arithmetic'
CheckOptions:
  - key: modernize-use-checked-arithmetic.handleCode
    value: 'printf("ERROR\n");'
  - key: modernize-use-checked-arithmetic.handleImport
    value: '<stdio.h>'
```

Appendix C

Code for clang-tidy check

`UseCheckedArithmeticCheck.cpp`:

```
#include "UseCheckedArithmeticCheck.h"
#include "../utils/IncludeInserter.h"
#include "clang/AST/ASTContext.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Lex/Lexer.h"

using namespace clang::ast_matchers;

namespace clang::tidy::modernize {

void UseCheckedArithmeticCheck::registerPPCallbacks(
    const SourceManager &SM, Preprocessor *PP, Preprocessor *ModuleExpanderPP) {
  IncludeInserter.registerPreprocessor(PP);
}

StatementMatcher makeCastNonAssignmentMatcher() {
    return binaryOperator(hasAnyOperatorName("+", "-", "*"),
                          hasParent(expr(hasType(isInteger()), unless(hasType(isConstQualified()))).bind("parent-expr")),
                          hasLHS(ignoringImpCasts(hasType(isInteger()))),
                          hasLHS(ignoringImpCasts(expr().bind("argOne"))),
                          hasRHS(ignoringImpCasts(hasType(isInteger()))),
```

```
                        hasRHS(ignoringImpCasts(expr().bind("argTwo"))))
            .bind("Non-AssignmentOp");

}

StatementMatcher makeGenericNonAssignmentMatcher() {
    return binaryOperator(hasAnyOperatorName("+", "-", "*"),
                        hasLHS(ignoringImpCasts(hasType(isInteger()))),
                        hasLHS(ignoringImpCasts(expr().bind("argOne"))),
                        hasRHS(ignoringImpCasts(hasType(isInteger()))),
                        hasRHS(ignoringImpCasts(expr().bind("argTwo"))))
            .bind("Non-AssignmentOp");

}

StatementMatcher makeUnaryMatcher() {
  return unaryOperator(hasAnyOperatorName("++", "--"),
                      hasUnaryOperand(ignoringImpCasts(hasType(isInteger()))),
                      hasUnaryOperand(ignoringImpCasts(anyOf(
                          expr().bind("arg"), integerLiteral().bind("arg")))))
      .bind("UnaryOp");
}

StatementMatcher makeAssignmentMatcher() {
  return binaryOperation(
             hasAnyOperatorName("+=", "-=", "*="), isAssignmentOperator(),
             hasLHS(ignoringImpCasts(hasType(isInteger()))),
             hasLHS(ignoringImpCasts(
                 anyOf(expr().bind("dest"), integerLiteral().bind("dest")))),
             hasRHS(ignoringImpCasts(hasType(isInteger()))),
             hasRHS(ignoringImpCasts(
                 anyOf(expr().bind("arg"), integerLiteral().bind("arg")))))
      .bind("AssignmentOp");
}

std::string getCkdFunction(llvm::StringRef opStr) {
  if (opStr == "+" || opStr == "++" || opStr == "+=") {
    return "ckd_add";
  } else if (opStr == "-" || opStr == "--" || opStr == "-=") {
```

```
    return "ckd_sub";
  } else if (opStr == "*" || opStr == "*=") {
    return "ckd_mul";
  } else {
    llvm::errs() << "Unknown operation to convert: " << opStr << "\n";
    return "";
  }
}

void UseCheckedArithmeticCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(traverse(TK_AsIs, makeCastNonAssignmentMatcher()), this);
  Finder->addMatcher(traverse(TK_AsIs, makeGenericNonAssignmentMatcher()), this);
  Finder->addMatcher(traverse(TK_AsIs, makeAssignmentMatcher()), this);
  Finder->addMatcher(traverse(TK_AsIs, makeUnaryMatcher()), this);
}

std::string getExprSourceString(const Expr *expr, const SourceManager &sm,
                                const LangOptions &langOpts) {
  const auto range = expr->getSourceRange();

  const auto start = sm.getSpellingLoc(range.getBegin());
  const auto end = sm.getSpellingLoc(range.getEnd());

  return Lexer::getSourceText(CharSourceRange::getTokenRange(start, end), sm,
                              langOpts)
      .str();
}

void UseCheckedArithmeticCheck::fixAssignmentOp(
    const MatchFinder::MatchResult &Result) {
  const auto *MatchedExpr =
      Result.Nodes.getNodeAs<BinaryOperator>("AssignmentOp");

  const auto resultType = MatchedExpr->getType().getAsString();

  const SourceManager &sm = *Result.SourceManager;
  const LangOptions &langOpts = Result.Context->getLangOpts();
  std::string destSource, argSource;
  std::string destType, argType;
```

```
const clang::Expr *dest;
const clang::Expr *arg;

// If it's an expression
if (Result.Nodes.getNodeAs<Expr>("dest")) {
  dest = Result.Nodes.getNodeAs<Expr>("dest");
} else { // if it's an integer literal
  dest = Result.Nodes.getNodeAs<IntegerLiteral>("dest");
}

// Same logic for argTwo
if (Result.Nodes.getNodeAs<Expr>("arg")) {
  arg = Result.Nodes.getNodeAs<Expr>("arg");
} else {
  arg = Result.Nodes.getNodeAs<IntegerLiteral>("arg");
}

// Get appropriate checked function
const std::string ckdFunc = getCkdFunction(MatchedExpr->getOpcodeStr());

if (ckdFunc == "") {
  llvm::errs() << "Error converting operation:" << MatchedExpr->getOpcodeStr()
               << "\n";
  return;
}

// Extract the source for dest and arg
destSource = getExprSourceString(dest, sm, langOpts);
argSource = getExprSourceString(arg, sm, langOpts);

// Get types of dest and source
destType = dest->getType().getAsString();
argType = arg->getType().getAsString();


auto replacement = "({ " + destType + "* dest = " + "&" + destSource + ";\n";
replacement += argType + " arg = " + argSource + ";\n";
replacement += "if(" + ckdFunc + "(dest, *dest, arg)) {";
replacement += HandleCode + "\n};";
replacement += "*dest;})";
```

```
  DiagnosticBuilder Diag =
      diag(MatchedExpr->getBeginLoc(), "assignment operation can be rewritten to use checked arithmetic");

  Diag << FixItHint::CreateReplacement(MatchedExpr->getSourceRange(),
                                       replacement);

  Diag << IncludeInserter.createIncludeInsertion(
      Result.Context->getSourceManager().getFileID(MatchedExpr->getBeginLoc()),
      "<stdckdint.h>");
  Diag << IncludeInserter.createIncludeInsertion(
      Result.Context->getSourceManager().getFileID(MatchedExpr->getBeginLoc()),
      HandleImport);
  return;
}

void UseCheckedArithmeticCheck::fixUnaryOp(
    const MatchFinder::MatchResult &Result) {
  const auto *MatchedExpr = Result.Nodes.getNodeAs<UnaryOperator>("UnaryOp");

  const SourceManager &sm = *Result.SourceManager;
  const LangOptions &langOpts = Result.Context->getLangOpts();
  std::string argSource, argType;

  const clang::Expr *arg = Result.Nodes.getNodeAs<Expr>("arg");

  // Extract the source with this expression
  argSource = getExprSourceString(arg, sm, langOpts);
  argType = arg->getType().getAsString();

  const std::string ckdFunc =
      getCkdFunction(MatchedExpr->getOpcodeStr(MatchedExpr->getOpcode()));

  if (ckdFunc == "") {
    llvm::errs() << "Error converting operation.\n";
    return;
  }

  std::string replacement;

  switch (MatchedExpr->getOpcode()) {
```

```
case clang::UO_PreInc:
case clang::UO_PreDec:
  // handle fix for prefix
  replacement = "({ " + argType + "* tmp = &" + argSource + ";\n";
  replacement += "if(" + ckdFunc + "(tmp, *tmp, 1)) {\n";
  replacement += HandleCode + "\n};";
  replacement += "*tmp;})";

  break;

case clang::UO_PostInc:
case clang::UO_PostDec:
  // handle fix for postfix
  replacement = "({ " + argType + "* tmp = &" + argSource + ";\n";
  replacement += argType + " oldTmp = *tmp;\n";
  replacement += "if(" + ckdFunc + "(tmp, *tmp, 1)) {\n";
  replacement += HandleCode + "\n};";
  replacement += "oldTmp;})";

  break;

default:
  llvm::errs() << "Unknown unary operator: "
               << MatchedExpr->getOpcodeStr(MatchedExpr->getOpcode()) << "\n";
  return;
}

DiagnosticBuilder Diag =
    diag(MatchedExpr->getBeginLoc(), "unary operation can be rewritten to use checked arithmetic");
Diag << FixItHint::CreateReplacement(MatchedExpr->getSourceRange(),
                                     replacement);

Diag << IncludeInserter.createIncludeInsertion(
    Result.Context->getSourceManager().getFileID(MatchedExpr->getBeginLoc()),
    "<stdckdint.h>");
Diag << IncludeInserter.createIncludeInsertion(
    Result.Context->getSourceManager().getFileID(MatchedExpr->getBeginLoc()),
    HandleImport);
return;
```

```
}

void UseCheckedArithmeticCheck::fixNonAssignmentOp(
    const MatchFinder::MatchResult &Result) {
  const auto *MatchedExpr =
      Result.Nodes.getNodeAs<BinaryOperator>("Non-AssignmentOp");

  std::string resultType;

  // If there is a cast above this node we need to use that type as the dest
  if(Result.Nodes.getNodeAs<Expr>("parent-expr")) {
    const auto result = Result.Nodes.getNodeAs<Expr>("parent-expr");
    resultType = result->getType().getAsString();

  } else { // otherwise we can just use the type it's going into
      resultType = MatchedExpr->getType().getAsString();
  }

  const SourceManager &sm = *Result.SourceManager;
  const LangOptions &langOpts = Result.Context->getLangOpts();
  std::string argOneSource, argTwoSource;
  std::string argOneType, argTwoType;

  const clang::Expr *argOne;
  const clang::Expr *argTwo;

  argOne = Result.Nodes.getNodeAs<Expr>("argOne");
  argTwo = Result.Nodes.getNodeAs<Expr>("argTwo");

  // Extract the source with this expression
  argOneSource = getExprSourceString(argOne, sm, langOpts);
  argTwoSource = getExprSourceString(argTwo, sm, langOpts);

  argOneType = argOne->getType().getAsString();
  argTwoType = argTwo->getType().getAsString();

  const std::string ckdFunc = getCkdFunction(MatchedExpr->getOpcodeStr());

  if (ckdFunc == "") {
    llvm::errs() << "Error converting operation:" << MatchedExpr->getOpcodeStr()
                 << "\n";
```

```
    return;
  }

  auto replacement = "({ " + resultType + " dest;\n";
  replacement += argOneType + " argOne = " + argOneSource + ";\n";
  replacement += argTwoType + " argTwo = " + argTwoSource + ";\n";
  replacement += "if(" + ckdFunc + "(&dest, argOne, argTwo)) {";
  replacement += HandleCode + "\n};";
  replacement += "dest;})";

  DiagnosticBuilder Diag =
      diag(MatchedExpr->getBeginLoc(), "non-assignment operation can be rewritten to use checked arithmetic");
  Diag << FixItHint::CreateReplacement(MatchedExpr->getSourceRange(),
                                       replacement);

  Diag << IncludeInserter.createIncludeInsertion(
      Result.Context->getSourceManager().getFileID(MatchedExpr->getBeginLoc()),
      "<stdckdint.h>");
  Diag << IncludeInserter.createIncludeInsertion(
      Result.Context->getSourceManager().getFileID(MatchedExpr->getBeginLoc()),
      HandleImport);
  return;
}

void UseCheckedArithmeticCheck::check(const MatchFinder::MatchResult &Result) {

  if (HandleImport == "__unset") {
    HandleImport = "<assert.h>";
  }

  if (HandleCode == "__unset") {
    HandleCode = "assert(0);";
  }

  if (Result.Nodes.getNodeAs<BinaryOperator>("Non-AssignmentOp")) {
    fixNonAssignmentOp(Result);
  } else if (Result.Nodes.getNodeAs<BinaryOperator>("AssignmentOp")) {
    fixAssignmentOp(Result);
  } else if (Result.Nodes.getNodeAs<UnaryOperator>("UnaryOp")) {
```

```
    fixUnaryOp(Result);
  }

  return;
}
} // namespace clang::tidy::modernize
```

UseCheckedArithmetic.h:

```
#ifndef LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USE_CHECKED_ARITHMETIC_H
#define LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USE_CHECKED_ARITHMETIC_H

#include "../ClangTidyCheck.h"
#include "../utils/IncludeInserter.h"

namespace clang::tidy::modernize {

/// This check will warn on arithemtic expressions that can overflow and suggest
/// using the C23 ckd_* macro
///
class UseCheckedArithmeticCheck : public ClangTidyCheck {
public:
  UseCheckedArithmeticCheck(StringRef Name, ClangTidyContext *Context)
      : ClangTidyCheck(Name, Context),
        HandleImport(Options.get("handleImport", "__unset")),
        HandleCode(Options.get("handleCode", "__unset")),
        IncludeInserter(Options.getLocalOrGlobal("IncludeStyle",
                                                 utils::IncludeSorter::IS_LLVM),
                        areDiagsSelfContained()) {}
  /* TODO maybe specify later
  bool isLanguageVersionSupported(const LangOptions &LangOpts) const override {
    llvm::outs() << "Called lang supported";
    return LangOpts.C23;
  }
  */

  void storeOptions(ClangTidyOptions::OptionMap &Opts) override {
    Options.store(Opts, "handleImport", HandleImport);
    Options.store(Opts, "handleCode", HandleCode);
```

```
  }

  void registerPPCallbacks(const SourceManager &SM, Preprocessor *PP,
                           Preprocessor *ModuleExpanderPP) override;
  void registerMatchers(ast_matchers::MatchFinder *Finder) override;
  void check(const ast_matchers::MatchFinder::MatchResult &Result) override;

private:
  utils::IncludeInserter IncludeInserter;
  std::string HandleImport;
  std::string HandleCode;
  void fixNonAssignmentOp(const ast_matchers::MatchFinder::MatchResult &Result);
  void fixAssignmentOp(const ast_matchers::MatchFinder::MatchResult &Result);
  void fixUnaryOp(const ast_matchers::MatchFinder::MatchResult &Result);
};

} // namespace clang::tidy::modernize

#endif // LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USE_CHECKED_ARITHMETIC_H
```

UseCheckedArithmeticDebugCheck.h:

```
//===--- UseCheckedArithmeticDebugCheck.h - clang-tidy ----------*- C++ -*-===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USECHECKEDARITHMETICDEBUGCHECK_H
#define LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USECHECKEDARITHMETICDEBUGCHECK_H

#include "../ClangTidyCheck.h"

namespace clang::tidy::modernize {

/// FIXME: Write a short description.
///
/// For the user-facing documentation see:
/// http://clang.llvm.org/extra/clang-tidy/checks/modernize/UseCheckedArithmeticDebug.html
```

```
class UseCheckedArithmeticDebugCheck : public ClangTidyCheck {
public:
  UseCheckedArithmeticDebugCheck(StringRef Name, ClangTidyContext *Context)
      : ClangTidyCheck(Name, Context) {}
  void registerMatchers(ast_matchers::MatchFinder *Finder) override;
  void check(const ast_matchers::MatchFinder::MatchResult &Result) override;
};

} // namespace clang::tidy::modernize

#endif // LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USECHECKEDARITHMETICDEBUGCHECK_H
```

UseCheckedArithmeticDebugCheck.cpp:

```
//====--- Usecheckedarithmeticdebugcheck.cpp - clang-tidy ------------------===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//

#include "UseCheckedArithmeticDebugCheck.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"

using namespace clang::ast_matchers;

namespace clang::tidy::modernize {

void UseCheckedArithmeticDebugCheck::registerMatchers(MatchFinder *Finder) {
  // Matcher for binops
  Finder->addMatcher(binaryOperation(hasAnyOperatorName("+", "-", "*"))
                         // hasLHS(ignoringImpCasts(hasType(isInteger()))),
                         // hasRHS(ignoringImpCasts(hasType(isInteger())))
                         .bind("Non-AssignmentOp"),
                     this);

  // Matcher for unary ops
  Finder->addMatcher(
      unaryOperator(hasAnyOperatorName("++", "--")).bind("UnaryOp"), this);
```

```
  // Matcher for compound assignment
  Finder->addMatcher(binaryOperation(isAssignmentOperator(),
                                     hasAnyOperatorName("+=", "-=", "*="))
                         .bind("AssignmentOp"),
                     this);
}

void UseCheckedArithmeticDebugCheck::check(
    const MatchFinder::MatchResult &Result) {
  // FIXME: Add callback implementation.
  if(Result.Nodes.getNodeAs<Expr>("Non-AssignmentOp")) {
   const auto *MatchedOperation = Result.Nodes.getNodeAs<Expr>("Non-AssignmentOp");
   diag(MatchedOperation->getBeginLoc(), "Potential checked operation (non-assignment operation)");
  } else if(Result.Nodes.getNodeAs<Expr>("AssignmentOp")) {
   const auto *MatchedOperation = Result.Nodes.getNodeAs<Expr>("AssignmentOp");
   diag(MatchedOperation->getBeginLoc(), "Potential checked operation (assignment operation)");
  } else if(Result.Nodes.getNodeAs<Expr>("UnaryOp")) {
   const auto *MatchedOperation = Result.Nodes.getNodeAs<Expr>("UnaryOp");
   diag(MatchedOperation->getBeginLoc(), "Potential checked operation (unary operation)");
  }
}

} // namespace clang::tidy::modernize
```

UseCheckedArithmeticTypedDebugCheck.h

```
//====--- UseCheckedArithmeticTypedDebugCheck.h - clang-tidy ----------*- C++ -*-===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USECHECKEDARITHMETICTYPEDDEBUGCHECK_H
#define LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USECHECKEDARITHMETICTYPEDDEBUGCHECK_H

#include "../ClangTidyCheck.h"
```

```
namespace clang::tidy::modernize {

/// FIXME: Write a short description.
///
/// For the user-facing documentation see:
/// http://clang.llvm.org/extra/clang-tidy/checks/modernize/UseCheckedArithmeticTypedDebug.html
class UseCheckedArithmeticTypedDebugCheck : public ClangTidyCheck {
public:
  UseCheckedArithmeticTypedDebugCheck(StringRef Name, ClangTidyContext *Context)
      : ClangTidyCheck(Name, Context) {}
  void registerMatchers(ast_matchers::MatchFinder *Finder) override;
  void check(const ast_matchers::MatchFinder::MatchResult &Result) override;
};

} // namespace clang::tidy::modernize

#endif // LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USECHECKEDARITHMETICTYPEDDEBUGCHECK_H
```

UseCheckedArithmeticTypedDebugCheck.cpp:

```
//===--- UseCheckedArithmeticTyppedDebugCheck.cpp - clang-tidy ------------------===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//

#include "UseCheckedArithmeticTypedDebugCheck.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"

using namespace clang::ast_matchers;

namespace clang::tidy::modernize {

void UseCheckedArithmeticTypedDebugCheck::registerMatchers(MatchFinder *Finder) {
    Finder->addMatcher(
        binaryOperator(hasAnyOperatorName("+", "-", "*"),
                       hasParent(expr(hasType(isInteger()), unless(hasType(isConstQualified())))),
                       hasLHS(ignoringImpCasts(hasType(isInteger()))),
```

```
                        hasRHS(ignoringImpCasts(hasType(isInteger()))))
           .bind("Non-AssignmentOp"),
        this);

    Finder->addMatcher(
        binaryOperator(hasAnyOperatorName("+", "-", "*"),
                        hasLHS(ignoringImpCasts(hasType(isInteger()))),
                        hasRHS(ignoringImpCasts(hasType(isInteger()))))
           .bind("Non-AssignmentOp"),
        this);

    Finder->addMatcher(
        unaryOperator(hasAnyOperatorName("++", "--"),
                      hasUnaryOperand(ignoringImpCasts(hasType(isInteger()))))
      .bind("UnaryOp"), this);

    Finder->addMatcher(binaryOperation(
             hasAnyOperatorName("+=", "-=", "*="), isAssignmentOperator(),
             hasLHS(ignoringImpCasts(hasType(isInteger()))),
             hasRHS(ignoringImpCasts(hasType(isInteger()))))
      .bind("AssignmentOp"), this);

}

void UseCheckedArithmeticTypedDebugCheck::check(
    const MatchFinder::MatchResult &Result) {
  // FIXME: Add callback implementation.
  if(Result.Nodes.getNodeAs<Expr>("Non-AssignmentOp")) {
   const auto *MatchedOperation = Result.Nodes.getNodeAs<Expr>("Non-AssignmentOp");
   diag(MatchedOperation->getBeginLoc(), "Potential checked operation (non-assignment operation)");
  } else if(Result.Nodes.getNodeAs<Expr>("AssignmentOp")) {
   const auto *MatchedOperation = Result.Nodes.getNodeAs<Expr>("AssignmentOp");
   diag(MatchedOperation->getBeginLoc(), "Potential checked operation (assignment operation)");
  } else if(Result.Nodes.getNodeAs<Expr>("UnaryOp")) {
   const auto *MatchedOperation = Result.Nodes.getNodeAs<Expr>("UnaryOp");
   diag(MatchedOperation->getBeginLoc(), "Potential checked operation (unary operation)");
  }
}
```

```
} // namespace clang::tidy::modernize
```