A Study on the Effects of Exception Usage in Open-Source C++ Systems

Kirsten Bradley and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON Canada
{kcpbradl,migod}@uwaterloo.ca

***Abstract*—Exception handling (EH) is a feature common to many modern programming languages, including C++, Java, and Python, that allows error handling in client code to be performed in a way that is both systematic and largely detached from the implementation of the main functionality. However, C++ developers sometimes choose not to use EH, as they feel that its use increases complexity of the resulting code; new control flow paths are added to the code, "stack unwinding" adds extra responsibilities for the developer to worry about, and EH arguably detracts from the modular design of the system. In this paper, we perform an exploratory empirical study on how exceptions are handled in 2721 open source C++ systems taken from GitHub. We observed that the number of edges in a call graph grows, on average, by 22% when edges for exception flow are added to a graph. Additionally, about 8 out of 9 functions that may throw an exception do not initiate that exception. These results suggest that, in practice, the use of C++ EH can add subtle complexity to the design of the system that developers must strive to be aware of.**

***Index Terms*—exceptions, static analysis, C++, exception flow**

I. Introduction

Robust software must be able to recover from a variety of unforeseen error conditions that may arise during runtime. Languages such as C++, Java, and Python provide a flexible language feature for this known as *exception handling* (EH), where functions can interrupt normal execution to allow special error handlers to run and, if possible, recover from the error condition. A key feature of EH is that errors are often handled in a part of the code that is removed from where the problem is detected; this allows developers to place error handling code in key spots of the design, rather than insisting that errors be handled when and where they are first detected.

However, using EH has both costs and risks. For example, EH is known to degrade performance significantly; developers sometimes consider that EH makes code harder to understand [1] [17] and debug [14]; "stack unwinding" adds extra responsibilities to developers to ensure memory does not leak if an exception if thrown; and if an exception is thrown but not caught, the whole program will abort. Google, for example, recommends in its C++ style guide that EH not be used at all, although this is largely because of the redesign costs that would be incurred if EH were to be added to their existing products [3].

Despite these beliefs, many modern programming languages have EH mechanisms and there are known benefits to exception use such as improving error handling across modules [1]. While there may be some inherent difficulty with exception code, this difficulty could come from the tasks for which exceptions are used [20].

Most previous exception studies have looked mainly at Java code, so there is a dearth of empirical results about C++ code. While EH is similar Java and C++, there are also important differences, such as whether an exception specification is used, the need to consider stack unwinding in C++, and the ability of C++ code to throw objects other than explicit exceptions. For these reasons, we chose to only study C++ code. This meant that we had to develop a tool to extract EH information from source code; a secondary contribution of our work is the development of such a tool, which we will release as open source.

This paper has two main contributions. First, we present an exploratory empirical study on understanding EH usage in real-world C++ code taken from GitHub, and we evaluate the potential impact of EH usage that may not be immediately obvious from reading the code; this study helps to shed light on perceived versus observed issues in using EH. Second, we present a static analysis tool — Zelda, which is based on LLVM tooling — that can be used to extract information about EH use in C++ code.

II. Related Research

A. Exception Usage

Previous research has addressed several topics relating to the use of exception handling, particularly in Java and C++ systems. This includes empirical studies of how EH code is written, how EH code affects the overall design of the systems, as well as studies of how developers perceive the benefits and drawbacks of using EH.

Shah et al. explored the question of why developers often choose not to use exception handling at all. They interviewed Java and C++ developers about the topic [18], and also developed and validated a tool for visualizing EH usage. This work did not involve the analysis of existing code for EH usage.

Xie et al. studied how developers handle exceptions once they have occurred [21]. They observed that exception handling is often implemented to handle a conditional case, such as a file not existing, and to handle external cases, for example

checking the results from a call to an API function. Their work categorizes these situations theoretically without examining real-world code or consulting developers.

Marinescu studied versions of Eclipse and showed that code that uses exceptions is more defect-prone than other code [10]. They used the tool iPlasma to inspect code at a class level and determined if there are functions in a class that throw and catch exceptions [9].

B. Exception handling in C++ systems

Bonifacio et al. studied C++ exception usage in detail [1]. They analyzed 65 C++ projects and surveyed several C++ developers with a range of experience. They investigated the most common types of objects caught, what percentage of the code base is dedicated to exception handling, and determining what types of actions are performed in `catch` blocks. The survey included questions about whether developers believe C++ programmers avoid exception usage and why it may be avoided.

Prabhu et al. proposed an algorithm for interprocedural exception analysis for C++ [14]. This is the only other work we are aware of that explores call graphs combined with exceptions for C++ programs. Their goal was to translate C++ code with exceptions into an equivalent exception-free C++ program. We note that the algorithm presented in this research influenced the design of the static analysis tool that we have implemented, Zelda.

C. Exception handling in other languages

Kery et al. examined exception handling in Java [7]. Their work used about 8,000,000 Java repositories on GitHub. They determined that generic types, such as `Exception` and `IOException`, are the most commonly caught types in Java. Additionally, they categorized statements in `catch` blocks to determine what actions are taken to recover from exceptions, and found that rethrowing and terminating are the most common responses.

Nakshatri et al. analyzed patterns of exception handling in Java systems [12]. Over a large corpus of Java code from GitHub, they analyzed types of exceptions and the body of `catch` statements for patterns of how programmers recover from exceptions. While they listed three ways to analyze the types of exceptions, they stated that "an exception thrown [...] will eventually be caught by a caller method using a `try-catch` block". While this may be true in Java due to compilation failure if checked are not handled or specified for a function, there is no similar compilation check in C++.

Other research has addressed concerns about the robustness of exception handling. For example, Kechagia et al. studied context-dependent exceptions in Java code [6], while Oliveira et al. explored exception handling in Android and Java applications [13]. Robillard et al. performed static analysis of exceptions and their flow in Java [15]. They introduced the notions of breadth and depth of exceptions as meaningful ways to reason about global exception flow; they also included two case studies and discussed how the code would be improved in by reducing both exception depth and breadth.

A concept used in the analysis in this paper is augmenting a call graph to include exception flow. Previous work by Sinha et al. and Choi et al. suggested augmentation of Java flow graphs that represents exception usage [19] [2]. Sinha proposed a control flow graph that condensed potential exception instructions to have less complicated graphs. Choi augmented the graph call by adding edges and nodes that represent exceptions at the function level that could be used to create a graph for the interprocedural flow. Both performed an empirical study on their proposed graphs with Choi examining how a typical flow graph is effected by their augmentation and Sinha examining the change in graph size.

D. Static Analysis

To ensure accuracy of the call graphs we produced, we compared our results to the commercial tool Understand by Scitools. Understand is a static analysis tool that supports several tasks to understand code, such as metric calculation, dependency analysis, and control graphs, in languages including C++, Java, and Python. While the tool outputs detailed call graphs, they do not consider exception flow. Additionally, call and flow graphs are produced as visuals with no easily interpreted textual option available.

III. Exception Discussion

A common theme that was mentioned by previous researchers — but notably without empirical evidence — is that exceptions make code more complicated. This section summarizes what various researchers, individuals, and companies have stated about how exceptions might affect design clarity.

As stated by Robillard, exception handling can improve error recovery across modules [1]. Its use allows the developer to be notified of incorrect usage of a module and to specify recovery actions at a location in the system's design that the developer feels is most appropriate (i.e., not necessarily where the error is detected). The use of EH can also decrease the amount of error checking required when returning from a function, as the error can be thrown and the appropriate handler will be invoked [3]. However, developers from another survey expressed that the flow of exceptions had a negative impact on modularity [1].

An important task while programming is knowing the state of the program during a function call. This requires a developer to be aware the control flow of the code, including the expectations from any called functions. In a survey of developers [1], respondents stated exception flow and handlers add new control flow paths to the code, and may run counter to their natural intuition. This suggests that exceptions have an impact on the flow of a program, which makes it more complicated to understand.

The use of exceptions may also impact the design style of the code [3]. For instance, if a programmer knows that exceptions will be thrown when code is misused, they may decide to forgo checking the validity of their input and to place

their error handling code in `catch` statements. Furthermore, to account for stack unwinding when an exception occurs, either intermediate functions have to be prepared to catch all exceptions to free their resources, or code has to be written that strictly adheres to the Resource Acquisition Is Initialization (RAII) metaphor. Apart from RAII, there are other important best practices concerning EH that developers must be aware of, such as "destructors should never throw".

How exceptions are used may also influence how they are perceived by developers. Due to stack unwinding, they are useful when returning to a stack frame that may be far away in the call chain, such as recovery from error situations. As reported by a survey participant, "error and exception handling (code) is hard" [1]. While it is possible that both are difficult to write and understand, Stroustrup has stated that "exceptions make the complexity of the error handling visible. However, exceptions are not the cause of complexity" [20]. It is possible that the perceived difficulty of exceptions comes from how they are used and not that they are used.

In Java, the developer is forced to handle an exception or express that it may be thrown as part of the signature of a function, which is seen as a form of documentation. Java also allows for unchecked exceptions, which do not need the same documentation; however, research has shown that unchecked exceptions result in many program crashes in Android applications [6]. C++ does not enforce any exception restrictions. While a function can be specified as to whether it throws, it is up to the developer to label functions in this manner. Doing so does not force the caller to handle the exceptions, but forces the program to terminate if an unspecified exception is thrown.

This discussion serves to motivate our exploration of EH practice in C++ systems. We have designed and implemented a static analysis tool, called Zelda, to extract exception existence and flow information from C++ code. Using the results from our tool, we have investigated four research questions:

RQ1 *How localized is exception throwing and catching?*

RQ2 *How does the use of exceptions impact the control flow of a program?*

RQ3 *How does the use of exceptions impact the implementation of a program?*

RQ4 *Do C++ exception specifications affect the outcome of exception handling efforts?*

IV. Background

A. C++ Exceptions

Exception handling mechanisms vary between programming languages but typically are used as a way to interrupt control flow when an event — an exceptional event! — occurs that makes continued normal control flow impossible. For example, reading from a file that does not exist or accessing an array element that is out of bounds are actions that simply cannot be performed; EH allows the program designer to consider what to do in these situations.

The rules about which objects can be thrown and how thrown objects must be handled differ between programming languages. There are four components to exception handling in C++: thrown data, code blocks that throw and handle exceptions, function declaration of exceptions, and stack unwinding.

1) Types of Exceptions: In C++, all objects and primitive instances can be thrown. Additionally, C++ provides several pre-defined exception classes, all of which descend from the minimal `std::exception` class. Developers can choose either to create instances of the pre-defined exceptions classes, such as `std::out_of_range`, or create their own exception classes that inherit from them. According to Stroustrup, the intermediate exception classes should be inherited from for different purposes [20]. For example, `std::logic_error` is intended to be used when the error could be caught before executing the code and `std::runtime_error` is for all other exceptions.

2) Exception Handling Code: There are three language features specific to exceptions: `throw` expressions, `try` statements, and `catch` statements. A `throw` expression is used to activate exception handling. If two exceptions are active at the same time, the program terminates. A `try` statement surrounds a block of code that could throw an exception directly, or be propagated from a function call. A `catch` statement follows a `try` statement and surrounds a block of code that is to be run if an exception occurs. If an exception from the `try` block matches the type of the `catch` statement, the associated code runs and the program continues running after the last `catch` statement associated with the `try` block that threw the exception.

3) Function Exception Declaration: In C++ there are two language features that can be used to indicate that a function may throw an exception. A `throw` clause lists all the types of exceptions that might be thrown from a function. However, `throw` clauses are deprecated as of C++14. Declaring whether a functions can throw any exceptions is preferred with a `noexcept` clause, which asserts a Boolean indicating if any exceptions can be thrown. If a function violates an exception specification, the program will abort.

4) Stack Unwinding: Stack unwinding is the automatic process of stack frames, including function calls, being removed from the call stack when an exception is thrown to find a `catch` statement that can handle the exception. When a stack frame goes out of scope, whether due to exception mechanisms or the scope being finished, the destructor will be called for each stack based object. The difference with stack unwinding is that the remainder of the code in the scope will not be executed. Due to destructors being called while the stack is unwinding, destructors should not throw exceptions.

V. Methodology

A. Static Analysis Tool

To analyze the presence and usage of exceptions in a code base, a static analysis tool was needed. We could not find an existing tool that met our technical needs of performing C++ exception flow analysis and giving textual output. We thus creates a static analysis tool that we called Zelda —

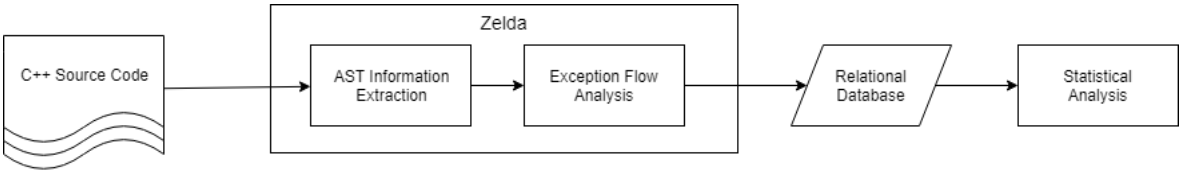

Fig. 1. Data processing during analysis.

Zee[1] Exception Length and Destination Analyzer — based on LLVM tooling infrastructure. Basically, Zelda consists of a set of AST tree walkers that extract and output information about exception handling, including the flow of exceptions, and the basic classification of expressions [8]. The capabilities and implementation details of Zelda discussed further below.[2]

The first task was to be able to determine the presence of exception code. As discussed in IV-A2, this involves the detection of `try` statements, `catch` statements, and `throw` expressions. Each of these has a specific type of AST node, making finding all instances of these nodes a simple task in a walk of the AST. Furthermore, the declaration of the `catch` statement and expression of the `throw` expression are stored, which simplifies determining the types.

The second task was to collect data about other aspects of the code in the corpus, specifically line counts and statement classification. The line counts of functions and `catch` statements were calculated by counting the number of newline characters present in the pretty print of the body from LLVM. Statement classification involved recording the types of statements present. AST nodes of interest were simple to detect due to having unique nodes in the structure, including `returns`, `breaks`, `continues`, `throws`, and `deletes`. Further classification involved looking at function calls, such as calls to `operator<<` and `printf` were classified as prints.

The final task for Zelda was to extract and map out exception flow through a system. This involves knowing where exceptions occur, the types of thrown exceptions and `catch` statements, and call information. All of this data can be determined from walking the AST; however, more information is needed than a typical call graph. Specifically, knowing about the presence of a `throw` statement or a function call from a function is not enough information to understand exception flow. For this purpose, an augmented call graph, that we call a context graph, is used in this analysis. A context graph associates a call with the calling function and the context within the calling function. For our purposes, the context can be either a `throw` statement, `catch` statement, or function.

While the first two tasks are relatively simple to implement, tracking exceptions flow through a program is more complicated. To ensure accuracy, the call graph has to be correct. Given the size of the corpus, it was infeasible to check the accuracy of the call graphs on a large sample of projects. Thus 10 projects from the corpus where chosen at random and their call graphs were compared to those produced by the commercial tool Understand by SciTools [16]. Of the 10 projects that were analyzed, the majority of functions were reported by both tools, although both reported functions that the other did not detect. Each of the programs reported calls to library functions that the other did not report. Since we are not concerned about exceptions thrown from libraries, this is not a concern. Understand reported calls to parent constructors, destructors, overridden methods, and templated functions that Zelda did not report. Zelda does not report destructor calls as developers are discouraged from throwing exceptions from destructors due to a combination of a program terminating if two exceptions are active at the same time and the fact that destructors are implicitly called while the stack unwinds. The remaining calls that Zelda does not report may be important to know about and could be found by improving the tool.

Once the call graph is determined, the context and exception information is combined to determine the flow of exceptions through a program. This is accomplished by looking at the context of each `throw` statement with the following algorithm:

Find all `throw` statements that are not rethrows
For each `throw` statement t:
  Find all context edges C for t
  For each context c in C:
    If c is a `catch` and t is a `throw` in c
      Add edge(context(c),t) to C
    If c is a `catch` and c contains a rethrow
      Add edge(context(c),t) to C
    If c is a `catch` and c does not contain rethrow
      Continue
    If c is a `try`
      Find first `catch` c_1 of c of type t
      If no such `catch` exists
        Add edge(context(c),t) to C
      else
        Add edge(c_1,t) to C
    If c is a function
      Find all function calls to c
      For each function f that calls c: add edge(callContext(c,f),t) to C

Through these steps, all exceptions are traced to either the `catch` statements that catch it, or to the last function that throws it.

For our empirical study, the graphs and code information are output to a relational database. The database is queried for

[1]Pronounced with an outrageous French accent [5].

[2]We will release Zelda as open source once our paper is accepted.

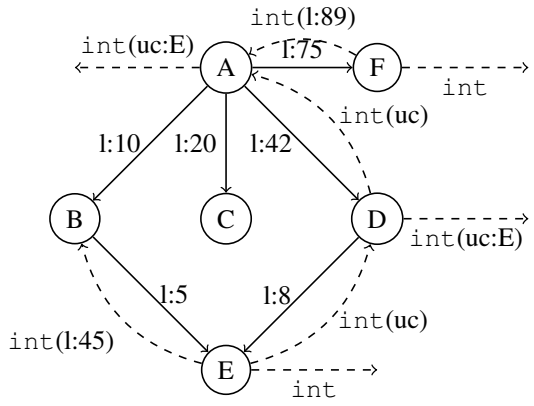

Fig. 2. Exception graph. Solid lines indicate a function call and the line the call is from is a label on the edge. Dashed lines indicate a thrown exception. Dashed lines that are not directed at a node indicates the function throws the exception.

the data necessary to address various aspects of the programs.

B. Exception Graphs

Exceptions unwind the stack of an arbitrary number of function calls, which means the control flow through a program can be drastically changed based on an exception being thrown. However, in a typical call graph, the semantics of returning to the calling function is implied by the graph. Thus, call graphs do not account for an exception to occur and the flow is inaccurately expressed. However, due to exceptions travelling up the call stack, if the developer knows a function can throw an exception, it would not typically be difficult to trace the exception through the call graph.

Consider a flow graph that includes both the calls between functions and the returns from a function. Calling a function that does not throw an exception will result in the return to the caller to be to the same location in the call to the function. Thus, return edges were not added and are understood to be implicit.

Consider an exception that is caught in the function that throws it. While this does activate stack unwinding, within a function unwinding the stack is effectively equivalent to breaking out of one or more nested conditionals. Additionally, since control flow remains in the function, the graph would not be altered outside of the throwing function.

Now consider if an exception is not caught in the throwing function. An edge will be added from the function that does not have a destination to indicate the function throws that type. If the throwing function is called, an edge annotated with the exception type and where it is caught are added between the nodes. The exception being caught will travel back to the call location and then unwind the stack based to find the matching `catch` statement. While this is occurring, destructors will run for all stack allocated objects, which means that other function calls will be occurring that should be added to the graph. Eventually, the exception is caught and the control flow is at a different place in the calling function than where the function call occurred. Thus, this edge, while still returning to the calling function, would be separate from the call edge. Additionally, each exception could have a unique `catch` location.

Any function that throws an exception being called results in an edge being added to the exception graph. Furthermore, these functions need to implement an exception handler or the except will further propagate resulting in more edges being added to the graph.

Figure 2 depicts an exception graph for some arbitrary function. From `A`, there are calls to `B`, `C`, `D`, and `F`. There are calls to `E` from `B` and `D`. The functions `E` and `F` each throw an exception of type `int`. From `E`, the function `B` catches the exception on line 45, while `D` does not catch the exception and an exception edge being added from `D`, annotated with "uc" for uncaught followed by `E` to indicate where the exception originated. Finally, `A` catches the exception from `F` on line 89 and does not catch the exception from `E`.

	Contain exceptions	No exceptions
Number of projects	1539	1182
Lines of code	23,159,082	74,079,530
Throw expressions	0	78,403
Try statements	0	78,565
Catch statements	0	101,854

TABLE I
SIMPLE DATA EXTRACTED FROM THE CORPUS.

VI. Corpus and Data

The projects used for analysis are C++ projects that are publicly posted on GitHub. The projects analyzed have a main language of C++, are at least a year old, have had more than 100 commits in their lifetime, and had been committed to within the past year. The projects were selected using a mirror of GHTorrent [4] from February 2017 with the code being the current versions from July 2017. There were 3,686 projects that matched these criteria. Removing instances of repeated projects, there was a total of 2,721 projects.

Various subsets of the corpus are used to address our research questions. These subsets are determined based on what aspects of code is being addressed. For example, projects that use a specific feature may be compared to projects that lack that feature, or features are checked between different aspects of the same project. If the projects studied are not specified for a specific result, the entire corpus was studied. Additional data from the corpus is presented in Table I.

VII. Exception Metrics

The distance an exception travels is defined by the number of unique functions that the exception will travel through before it is caught. This includes all functions through any path the exception could flow through. If an exception is rethrown, it is considered to be the same exception and further propagation increases the distance it travels, while an exception thrown from a `catch` statement is a unique exception.

The metric is influenced by MaCabe's cyclomatic complexity [11]. Cyclomatic complexity states that more potential paths through a program means the program is more complex. The existence of additional paths due to exceptions suggests that the program may be more complex reflects the previously expressed concern from developers. Similar to cyclomatic complexity, exception distance is about the flow of the program. However, exception flow works in the reverse order of function calls. Additionally, cyclomatic complexity is calculated as the sum of the number of flows through each function, which means each function is considered individually. Exception distance is calculated by investigating all the paths exceptions could take through a program and the order of called functions is crucial to this measurement unlike in cyclomatic complexity. While the order of function calls is important to the tracking the flow of exceptions, the metric is concerned with the number of functions visited and not the order of visitation.

VIII. Results

RQ1 *How localized is exception throwing and catching?*

Exceptions are an encouraged way to express that an error has occurred between modules. When used in this way, exceptions force the developer to be aware of and respond to improper use of code without having to check for returns signifying an error after each call. Within this work, we define a module to be all the functions written in a single file. An intermodule throw is a `throw` statement that unwinds the stack to or past a function that is part of a different module. If an exception is caught in at least one module it did not originate from, the catch is an intermodule catch, or the exception is caught intermodularly.

Intermodular exceptions occur infrequently within the corpus. Of the 78,403 possible exceptions, 9,241 (11.8%) are intermodular. This suggests that the majority of functions are thrown and caught within the same module. Furthermore, of the intermodule exceptions, 2,356 (25.5%) can be caught intermodularly, while 6,203 (9.9%) of exceptions that are not intermodule can be caught. Using a chi-squared test, the thrown exceptions were split by whether they were intermodule and whether they were caught. With a p-value of < 0.0001, we conclude that intermodularity and being caught are dependent variables with exceptions being caught more often if they are thrown intermodularly.

We conclude that exceptions are used infrequently between modules. However, exceptions that are intermodule are about 2.5 times as likely to be caught than other exceptions. This suggests that developers do respond to exceptions from other modules more commonly than within a module.

RQ2 *How does the use of exceptions impact the control flow of a program?*

A major concern about using exceptions is the increased complexity of control flow throughout a program. While it is clear when an exception is thrown, it is not obvious where an exception is thrown if it is not caught immediately. The exception could be caught in any function on the stack when an exception is thrown. If a function does not catch an exception from a function it calls, it effectively also throws the exception. This could lead to functions that throw exceptions that do not have `throw` statements and are not obvious candidates for exception analysis. We studied this concern in two ways: looking at functions that throw exceptions that originated in another function and edges that would be added to a flow graph to express paths that are due to exceptions.

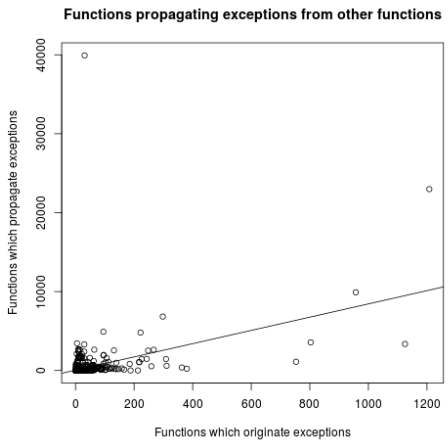

Fig. 3. Functions that throw exceptions that originate from another function.

1) Throwing Functions: In general, we were curious about how many calls occur to throwing functions. We consider exceptions to be thrown directly if they originate from the function being considered, and indirectly otherwise. We addressed the following questions:

1) How many functions throw directly or indirectly?
2) How many throwing functions are called?
3) How many calls exist to throwing functions?

All numerical results for these questions are in Table II.

First, we inspected the number of throwing functions in the project. While most of the results vary drastically between directly and indirectly throwing functions, they have similar means. Considering the median is 19 when combined, more than half of the projects have fewer than 20 throwing exceptions present. Furthermore, we compare the count of each within a project in Figure 3. From a linear regression with a p-value of < 0.0001, there are 8.39 functions that propagate an exception thrown from a function. This suggests that there are approximately 8 functions that indirectly throw an exception for every directly throwing function.

Knowing that there are many occurrences of throwing functions across the corpus, we wanted to know how many functions call throwing functions. When considering which functions call these functions, we are not concerned if the caller throws as well. Additionally, a function with multiple calls to throwing functions is counted as one function. We observed that the median number of calls to directly throwing functions is higher than indirectly throwing calls. This is interesting since there are generally more indirectly throwing

	Direct	Indirect	Combined	Call to Direct	Call to Indirect	Throwing Calls	All Calls
Mean	22.89	237.9	260.8	37.21	107	208.5	2417.0
Median	7	10	19	5	2	11	493
Standard Deviation	74.25	1491.37	1523.9	100.2	360.2	675.3	5370.3
95th percentile	72.0	1100.2	1260.4	232.8	562.4	1270.6	1996.0

TABLE II
PREVALENCE OF FUNCTIONS THAT THROW EXCEPTIONS AND CALLS TO THROWING FUNCTION PER PROJECT.

functions.

Finally, we consider how many calls there are to each throwing function. Unlike the previous question, this is count takes both caller and callee into account and represents total number of calls. This considers all directly and indirectly throwing functions.

2) Exception Graphs: Given the data involved in determining the information of throwing functions, how a flow graph would be altered due to exceptions can be concluded. In particular, the number of effected nodes and added edges can be determined.

The number of exception edges that do not lead to another node is the number of directly and indirectly throwing functions. This is also the number of nodes that are out-edges for exception edges. The calls to direct and indirect throws is the number of nodes that are in-edges for exception edges. Finally, the number of added edges is the the number of calls to any throwing function plus the number of throwing functions. The number of edges in the graph prior to the augmentation is the number of function calls and is used to normalize the edges.

Finally, we investigated how the flow graph is changed due to exceptions. The mean growth of the call graph is 22.1% with a median of 5.1%. From a linear regression with a p-value of < 0.0001, there is a slight positive correlation with a slope of 0.060. Comparing the number of edges added to the graph to the number of directly throwing functions, a linear regression with a p-value of < 0.0001 shows a positive correlation with a slope of 14.2. Thus, we conclude that the number of edges added to the exception graph is correlated to both the number of functions present and the number of throwing functions present.

RQ3 *How does the use of exceptions impact the implementation of a program?*

There are several ways that using exceptions in a project may result in a global change to the structure of code. The most obvious change is the presence of exception handling mechanisms. Aside from these mechanisms, the style of the code may change. Where a user may have used conditional statements to check for invalid input before a function call, they may write a handler for the exception. Finally, if exceptions are used, developers are encouraged to follow RAII suggests releasing of resources to occur in destructors, and if not, it is likely to occur in `catch` statements.

Before looking at if exceptions being used affects the implementation of a project, the prevalence of exception handling code is investigated. This includes both the number of projects that use exceptions and how much of a project is dedicated to exception handling. Exception handling code was defined to be the code within `catch` blocks. The median percent of code in a project with exceptions that is dedicated to exception handling is 0.64% and a median across all projects of 0.02%. Comparing to the previous result from Bonifacio [1] of 0.03% of their corpus being dedicated to exception recovery. The medians across all projects is comparable between the two studies.

Statement	exception	non-exception	p-value
`break`	0.0054	0.0060	0.2068
`continue`	0.0008	0.0008	0.5453
`delete`	0.0007	0.0008	0.5054
`do`	0.0032	0.0018	< 0.0001
`for`	0.0074	0.0090	0.0028
`if`	0.0417	0.0430	0.4565
`return`	0.0270	0.0283	0.3095
`switch`	0.0083	0.0077	0.3955
`while`	0.0020	0.0250	0.0028

TABLE III
OCCURRENCES OF CONTROL STATEMENTS USED IN PROJECTS WITH AND WITHOUT EXCEPTIONS NORMALIZED BY NUMBER OF LINES. P-VALUE FROM T-TEST

3) Control Flow Statements: Comparing across projects in the corpus, we studied whether there was a difference in control statement usage based on whether any exceptions were used. The measure used is the total number of occurrences of each statement normalized by the number of lines in the project. When comparing exception code within a project to other code in the project, both are normalized by the number of lines of code present for the respective category.

The results in Table III compare the usage of statements between projects that use exceptions and those that do not. The difference in control statements used is not statistically significant for most types of statements with the exception of `do` and `while` statements. This shows that the presence of exception usage does not have a significant global effect on the control structures used, although code without exceptions use `for` and `while` loops more frequently and `do-while` loops are used more frequently in exceptional code.

Looking within a project, the use of control structures is separated into occurring within `catch` statements, `try` statements, or neither, and is presented in Table IV. From the results, it is clear that these control structures contribute to the amount of code in each category differently. There are several observations that can be made about these differences.

The first category of statements considered was conditional statements. `for` and `while` loops occur significantly less often in `catch` statements. `do-while` loops occurred equally often in `try` and `catch` statements while occurring significantly less in non-exceptional code. These difference

Category	nonExcept	try	catch	nonExcept vs. try	nonExcept vs. catch	try vs. catch
break	0.0051	0.0094	0.0018	< 0.0001	< 0.0001	< 0.0001
continue	0.0008	0.0012	0.0016	0.0308	0.0601	0.3809
delete	0.0007	0.0031	0.0046	< 0.0001	< 0.0001	0.0267
do	0.0026	0.0099	0.0098	< 0.0001	< 0.0001	0.9099
for	0.0073	0.0096	0.0006	0.0003	< 0.0001	< 0.0001
if	0.0400	0.1044	0.0161	< 0.0001	< 0.0001	< 0.0001
return	0.0268	0.0942	0.0815	0.0907	< 0.0001	0.751
switch	0.0081	0.0094	0.0006	0.1663	< 0.0001	< 0.0001
throw	0.0007	0.0089	0.0233	< 0.0001	< 0.0001	< 0.0001
while	0.0020	0.0041	0.0002	< 0.0001	< 0.0001	< 0.0001

TABLE IV
CAPTION

suggest that the use of do-while loops in catch statements could potentially be reversing the work done by the same loops in try statements. Loops were also the only type of statements to statistically be used differently between projects with and without exceptions which suggests that loops may be related to to exception usage. While the presence of continue statements is consistent over the three, the use of break statements is more prevalent in try statements than anywhere else which could be related to the higher occurrence rate of loops. Finally, if statements occur most frequently in try statements that could reflect extra error checking before executing code that may throw an exception.

delete, throw, and return statements were the other statements considered. delete statements were considered due to the potential for catch statements to be used for clean-up of heap allocated variables and were found to be most prevalent in try and catch statements. This result is interesting due to RAII being recommended to developers using exceptions that should mitigate the heap allocated memory that needs to be freed. However, this may suggest that developers use exceptions to release resources manually. throw statements occurred most often in catch blocks, which suggests that once an exception is thrown, it is likely for further exceptions to be thrown or for the exception to be rethrown. The latter suggests that the function cleans up what it can and continues the propagation of the exception. The only instance that was statistically significant for the difference in return statements was between catch statements and non-exceptional code. This suggests that functions are often exited upon catching an exception.

We conclude that while using exceptions is not correlated with the overall structure of a program, the structure of try and catch statements are distinct from other code in a program. Overall, how control flow is handled in catch statements is not similar to general code. Furthermore, try statements share similarities to catch statements and general code which could be due to the developer being aware of the potential for exceptions to occur while programming.

RQ4 *Do C++ exception specifications affect the outcome of exception handling efforts?*

Documentation can alleviate some of the requirement for developers to know which functions can throw exceptions. While a project may have style guides that dictate how exceptions should be documented, there are built-in features in C++ to annotate function discussed in Section IV-A3. Both throw and noexcept document and enforce exception usage and we investigate them together.

The presence of these features was first considered across the whole corpus. There were 462 projects that throw exceptions and have at least one function marked with exception information. There were also 71 projects that did not use exceptions and documented functions. It was unexpected to find exception specifications in projects that do not throw exceptions.

We are interested in if there is a difference in exception distance based on whether the functions are annotated to throw changes how far an the thrown exception travels within a project. In particular, whether a function indicates exception usage is used to divide exceptions and the distance they travel is analyzed. To determine if exceptions travel further if the throwing function was annotated, the 66 projects that had throwing functions, where there was annotated and non-annotated, were investigated. The average throw length from a documented and undocumented function was 3.167 and 3.746 functions respectively. Comparing with a paired t-test gives resulted in a p-value of 0.35. Thus, we cannot conclude that documenting a function affects the number of functions an exception travels through.

After determining that exceptions documentation does not affect throw length, we considered whether exception documentation existing within a project influences throw length. The projects that throw were categorized by whether they have any functions documented. The mean throw length was 9.6 functions if there were no documented functions, and 23.1 functions if there were documented functions, which is a statistically significant difference in means with a p-value of 0.0006.

Overall, annotating functions within a project does not effect how far exceptions will travel. However, projects that annotate functions tend to have exceptions travel to more functions.

IX. Threats to Validity

The corpus studied is taken from the portion of GitHub that is publicly available. The projects in the corpus had a main language of C++, existed for at least one year, had at least 100 commits, and had been committed to in the last year to ensure the projects were real projects. Additionally, projects

that had similar names were examined to ensure there were not repeated projects in the corpus.

The C++ standard library has many classes that developers commonly use such as `string`, `vector`, input and output, as well as other common utilities. Many of these utilities are designed to throw exceptions when used inappropriately. However, exceptions from built-in libraries are not considered. Thus, the results in the paper only reflect user exceptions and may be inaccurate if the standard library is also considered. We decided that this was a good trade-off, as we are primarily interested in how ordinary C++ developers use exceptions, rather that C++ library designers.

Similarly, exceptions originating from third-party libraries are not considered in the analysis unless the code is included within a project. This is due to the large number of libraries that exist and the difficulty of ensuring all required information would be provided, such as compiler flags. While this information could be determined from `Makefile`s and other compilation systems, there are many systems used within the corpus making determining the required libraries difficult.

Thus, only the code present and exceptions written within the project are analyzed. While this does not ensure that the code is written as part of the project, it does reflect the code that is used. This also means that if projects do include code for a library, the project is analyzed with the version of the code that it would use.

Templated functions are also not analyzed by Zelda. This is due to the unique AST structure of such functions. We would also have to consider `throw` statements from multiple instances of templated functions should be considered different exceptions. There is uncertainty as to how these exceptions should be considered and there is added complexity that makes handling these functions more difficult. It is expected that templated functions would not greatly change the results unless they are implemented significantly differently than typical functions.

Our analysis also does not account for the use of function pointers. Thus, it is possible that functions are called through pointers that we were unable to take into consideration. However, this is a general problem when working with function pointers. The only ways to address this concern is to either keep track of the possible variables that have been assigned to a function pointer and assume that a call can be to any such function, or to assume that any function whose signature matches the pointer could be called. Either would not be overly accurate due to over estimating the number of functions that could be called.

For the analysis of intermodule exceptions, we defined a module to be all code written within a particular file. This definition may not match the definition of module to other individuals. However, this definition reflects that a developer would have to look outside of a file to determine exception information about code.

Alternatively, functions within a header file could be a similar definition to a source code file. However, using source code files has the benefit that any static functions are considered part of the module. Classes and namespaces could be used which results in different rules about modules being considered. For instance, would two classes in the same namespace be in the same module despite potentially having no relation? Would nested classes be considered to be in the same module as the class they are nested? Thus, we chose to used files as our module definition.

While Zelda was tested against another known tool for correctness of call graphs, further accuracy was not easily tested because there is no tool available to compare the results against. However, between the described algorithm, testing during the development process, and checking results from projects in the corpus, we are confident the tool works as described and intended. However, there is the possibility that Zelda does not work in situations that we have not encountered.

X. Future Work

A. Tool Improvement

While Zelda works as intended, there are aspects of C++ that are not addressed with the tool currently. Specifically the analysis of templated functions, calls to virtual methods and parent constructors, and implicit destructors calls would require additional information about class hierarchies than the current analysis.

Another point of improvement is to distinguish between functions that are marked to throw exceptions and those that will never throw exceptions. As the C++20 standard, the only language component will be `noexcept(expr)`, where `expr` is an expression that evaluates to true or false. Further analysis of `noexcept` statements would determine if the function is marked to never throw, potentially throw, or throw based on some property of a class. This would both determine how this program feature is used and improve exception flow analysis.

B. Dead Code Analysis

At this point, the analysis process assumes any code that is present will be executed at some point. However, there is certainly code that is not executed included in the analysis. For example, there are functions that are never called and conditions that can never be met. Adding basic dead code detection could be used determine if `throw` statements will ever actually occur. With the exception analysis involved, `catch` statements and code after a `throw` statement could also be detected as dead code.

C. Language Features

Pursuing additional information about the use of language features could lead to a better understanding of when people use exceptions. Instead of looking at what features are used in parts of a program, a study could be performed to address when exceptions are used. For instance, are exceptions commonly thrown from failed conditional statements, or how often is are functions called to facilitate the throwing of exceptions.

Our work has focused on exceptions that are caught and thrown by the same project. While one of our questions addressed if exceptions are thrown between modules, this question could be extended to between libraries. This would involve having the source code for third party libraries available for analysis with projects. Ideally the library code could be checked separate from a project and the exception flow could be analyzed by combining the information from the project and the library. This approach could also facilitate the analysis of exceptions from the standard C++ library.

D. Additional Programming Languages

We chose to focus on C++ exceptions when addressing exception concerns. The results in this work may not reflect how exceptions affect other languages. The research questions from our work could be answered for any other language. Due to exceptions being more restrictive in most programming languages, the results could vary drastically.

E. Code Defects

We focused on whether common concerns about exceptions were present in C++ code without looking at how exceptions may effect the robustness of a program. The analysis performed could be linked to other information about code. For instance, one could ask if code with exceptions is more likely to lead to code defects and address this by comparing bug reports and pull requests with exception usage. Combining the knowledge of exception flow from Zelda with such reports could show that exceptions have a wider impact than it seems due to their ability to travel through the stack in ways that are not immediately obvious.

XI. Conclusion

We have put forth questions derived from the concern of increased complexity of C++ code due to exceptions usage. We focused on the difference to programs that may not be immediately obvious to a developer, such as the localization of exceptions, and their impact on the control flow and structure of a program.

Additionally, we developed a static analysis tool that was used to analyze the corpus. This tool was necessary for our research due to the lack of tools that produce text output about call graphs and exception flow in C++. Without such a tool, the only option is to use graphical output from tools, which is not feasible when performing analysis on large tasks.

We found that exception usage does impact various aspects of C++ programs including exception flow increasing the size of a call graph by an average of 22% and that most exception handling is localized to a file, but exceptions are handled more frequently when exceptions travel between functions. Furthermore, there are about 8 functions that throw exceptions indirectly for every one function that directly throws an exception. Surprisingly, specifying the exceptions of a function made no significant difference to exception distance.

Overall, it seems that the hesitance from developers to use exceptions are founded. The effects of exceptions seem to be significant to several aspects of a program. The flow of exceptions may not be easily noticeable or trackable as systems grow in size. Using software, such as Zelda, can alleviate some of the burden on developers to track exception flow and ensure the robustness of their software.

References

[1] Rodrigo Bonifacio, Fausto Carvalho, Guilherme N. Ramos, Uira Kulesza, and Roberta Coelho. The use of C++ exception handling constructs: A comprehensive study. *Proc. of the 2015 IEEE International Working Conference on Source Code Analysis and Manipulation, (SCAM-2015)*, pages 21–30, 2015.

[2] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. *Proc. of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE-1999)*, pages 21–31, 2004.

[3] Google. Google C++ Style Guide: Exceptions. https://google.github.io/styleguide/cppguide.html#Exceptions.

[4] Georgios Gousios and Diomidis Spinellis. GHTorrent : Github ' s Data from a Firehose. *Proc. of the 9th IEEE Working Conference on Mining Software Repositories (MSR-2012)*, pages 12–21, 2012.

[5] T. Jones and T. Gilliam. *Monty Python and the Holy Grail*, motion picture, Python (Monty) Pictures, 1975.

[6] Maria Kechagia, Tushar Sharma, and Diomidis Spinellis. Towards a context dependent java exceptions hierarchy. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C-2017)*, pages 347–349. IEEE Press, 2017.

[7] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. Examining programmer practices for locally handling exceptions. *Proc. of the 13th International Workshop on Mining Software Repositories (MSR-2016)*, pages 484–487, 2016.

[8] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*.

[9] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE International Conference on Software Maintenance (ICSM Industrial and Tool Volume)*, pages 77–80. Society Press, 2005.

[10] Cristina Marinescu. Are the classes that use exceptions defect prone? In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL-11)*, pages 56–60, New York, New York, USA, 2011. ACM Press.

[11] T J McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[12] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in Java projects. *Proc. of the 13th International Workshop on Mining Software Repositories (MSR-2016)*, pages 500–503, 2016.

[13] Juliana Oliveira, Nelio Cacho, Deise Borges, Thaisa Silva, and Fernando Castor. An Exploratory Study of Exception Handling Behavior in Evolving Android and Java Applications. *Proc. of the 30th Brazilian Symposium on Software Engineering (SBES-2016)*, pages 23–32, 2016.

[14] Prakash Prabhu, Naoto Maeda, Gogul Balakrishnan, Franjo Ivanči, and Aarti Gupta. *Proc. of 25th European Conference on Object-Oriented Programming (ECOOP-2011)*, pages 583–608.

[15] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, pages 191–221, 2003.

[16] "Scitools". Scitools' understand, 2019.

[17] Hina Shah, Carsten Görg, and Mary Jean Harrold. Visualization of exception handling constructs to support program understanding. *Proce. of the 4th ACM Symposium on Software Visualization*, page 19, 2008.

[18] Hina Shah, Carsten Görg, and Mary Jean Harrold. Why Do Developers Neglect Exception Handling? *Proc. of the 4th International Workshop on Exception Handling*, pages 62–68, 2008.

[19] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, pages 849–871, 2000.

[20] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.
[21] Tao Xie and Suresh Thummalapenta. Making Exceptions on Exception Handling. pages 1–3, 2012.