DEVELOPMENT OF VERSION 3.0 OF THE EXTREMELY LOW PROBABILITY OF RUPTURE (XLPR) CODE FRAMEWORK

by

Zane Cox

A senior thesis submitted to the faculty of

Brigham Young University - Idaho

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics

Brigham Young University - Idaho

December 2025

BRIGHAM YOUNG UNIVERSITY - IDAHO

DEPARTMENT APPROVAL

of a senior thesis submitted by

Zane Cox

This thesis has been reviewed by the research committee, senior thesis coordinator, and department chair and has been found to be satisfactory.

Date	Kevin Kelley, Advisor
Date	David Oliphant, Senior Thesis Coordinator
Date	Richard Datwyler, Committee Member
Date	Lance Nelson, Committee Member
Date	Evan Hansen, Chair

ABSTRACT

DEVELOPMENT OF VERSION 3.0 OF THE EXTREMELY LOW PROBABILITY OF RUPTURE (XLPR) CODE FRAMEWORK

Zane Cox

Department of Physics

Bachelor of Science

The nuclear power industry relies on robust and effective safety analysis code to aid engineers in qualifying reactor systems to a reasonable degree of accuracy and in a cost effective manner. One such code is the Extremely Low Probability of Rupture (xLPR) framework developed by the US Nuclear Regulatory Commission (NRC), which is used to model the occurrence and propagation of cracks in nuclear piping systems. During the summer of 2025 I worked as a Nuclear Software Development Intern at Information Systems Laboratories on xLPR framework version 3.0 as contracted by the NRC. During this project I contributed to many parts of the xLPR framework 3.0 code, which will improve performance and maintainability over the previous version. This paper will discuss some of the software design strategies that were used, and how I contributed to the project.

ACKNOWLEDGMENTS

I would like to thank Information Systems Laboratories for providing me with this internship opportunity. I am especially grateful for Lance Larsen and Jack Hanni who were mentors to me during my time at the company. I am ever gratful to my wife, Mallory, for supporting me always. I am also thankful for my senior thesis advisor, Kevin Kelley, as well as committee members Richard Datwyler and Lance Nelson for their guidance during the writing of this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The U.S. Nuclear Regulatory Commission (NRC) requires that the primary piping systems for nuclear power plants must exhibit extremely low probability of rupture (xLPR) in order to prevent dangerous leakage situations. One such way to determine if this condition is met is to conduct conservative deterministic fracture mechanics analyses and implement leakage monitoring systems. The xLPR computational model was developed jointly by the NRC and the the Electric Power Research Institute (EPRI) as a tool to conduct the computational component of this analysis.

The xLPR computational model probabilistically predicts crack occurrence and character, factoring in a variety of pipe degradation mechanisms. It consists of various deterministic models that are fed a combination of user input values, and inputs generated through Monte-Carlo methods. The model is structured such that independent modules complete unique tasks, and an overarching code framework is required to connect these modules. This framework ensures the correct storage, delivery, and transformation of information between the user and all xLPR code modules. The

existing Version 2 of the xLPR framework was implemented using a general purpose simulation software called GoldSim. This software serves to connect the many deterministic models included in xLPR and also functions as a graphical user interface to edit and run xLPR simulations. The objective of this project is to replace the version 2 framework with a Version 3 that is independent of the GoldSim software, but fulfills the same functionality.

The GoldSim software requires that an expensive license be purchased in order to alter the simulation. This makes it difficult for the NRC to maintain the code. It also complicates the user experience, because the user must download third party software to run the code.

1.2 Project Scope

It has been determined that, while effective, there are a number of limitations inherent to the GoldSim implementation of the xLPR framework. Some of the most notable reasons to move away from the existing version are: (1) the need for expensive commercial off-the-shelf software, (2) the difficulty of making modifications, and (3) limited computational capacity. Added flexibility, ease of control, and enhanced performance could be achieved by implementing a new version of the xLPR framework built from the ground up.

Information Systems Laboratories was awarded the contract for this project. Golang (Go) was selected as the programming language of choice mainly due to its simplicity, computational efficiency, and ability to compile the code into an executable to make distribution of the software easy.

Replacing the GoldSim implementation of xLPR (Figure 1.1) is a sizeable task. Over 4000 unique elements in GoldSim need to be reproduced in Go. New interfaces

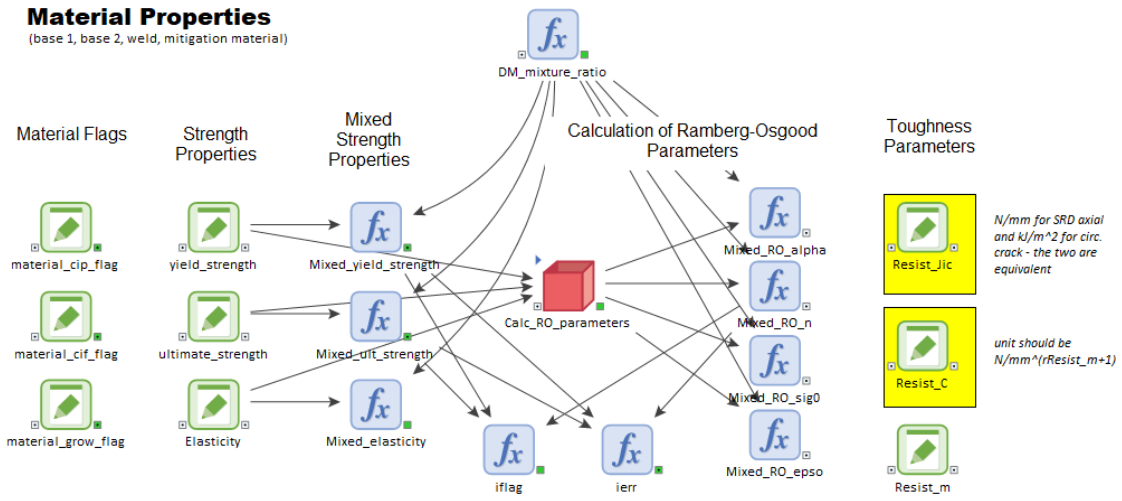

Figure 1.1 A small example of elements in the xLPR framework as implemented in GoldSim. This page displays a combination of inputs, mathematical and logical expressions, and containers. Clicking on an element opens a dialogue box that describes its function.

between parts of the xLPR model that can be reused must be created. The new code must undergo rigorous validation to ensure that the behavior is consistent with the previous version.

1.3 Outcomes

Although at the time of writing, Version 3.0 is not yet complete, the new implementation of the xLPR framework will provide significantly improved performance. Running the xLPR simulation through the GoldSim player was limited to four cores with the free version, and ten cores with the licensed version. Therefore, in addition to running faster regardless of the number of cores used, there will be no restriction on the number of cores used in the Go implementation, making it easier to run a high number of statistical realizations. This results in higher quality statistical results.

Future maintenance of the code will be simplified due to maintainers not needing

to go through a third party software. Not only can the new version be recreated in a simpler manner than the GoldSim implementation, but it will also be free to work on. This grants the NRC more independence and freedom to work on its own code.

The experience will also be improved from a user standpoint, as a custom graphical user interface is in development which will provide a more streamlined experience than the GoldSim implementation. Removing the need for a third party software also simplifies distribution of the code.

Chapter 2

Software Design Strategies

Several strategies were used by the xLPR team to avoid common problems in code. These patterns serve as a template for code development. While many approaches were used, I will emphasize a few that were particularly influential in the development process: low coupling, high cohesion, test-driven development, and readability and maintainability.

2.1 Low Coupling, High Cohesion

Low Coupling, High Cohesion is the idea that one part of the code should fulfill a specific purpose, and that all the parts should work well together, but they should also be able to operate such that a change to one part does not break the system as a whole. This is not a specific solution, rather a design approach. When used, it means that interchangeable behaviors are encapsulated such that specific functionality can be swapped inside a larger framework, and none of the surrounding code needs to be modified when this is done. Object oriented programming is one approach that can be used to implement the Low Coupling, High Cohesion principle, because it groups

related behaviors and allows the programmer to swap them out interchangeably.

An example of how this principle is implemented in the xLPR code in Golang is through the use of interfaces. An interface is a type defined as a named set of method signatures. For example, a "Logger" interface might include methods "Log" and "Write". Any type that has methods named"Log" and "Write" is an implicit implementation of the "Logger" Interface. In this example, lets say there are implementations "ToText" and "ToConsole" which are designed to log to a text file or directly to the console. The "Log" and "Write" methods for each implementation carry out a unique functionality, because they are necessarily different to accomplish their individual task. The "Logger" interface now allows us to use either the "ToText" or "ToConsole" implementation of the logger without needing to specify which type it is. If the "Log" method is called on the "Logger" interface, then the correct implementation will be used. This means that either or both of the implementations can be utilized directly using the "Logger" interface, so one part of the code might just tell the logger that something needs to be logged without specifying how that is done, and another part of the code selects the implementation and carries out the correct method. The two parts of the code interact in a lowly-coupled manner, but the full functionality still exists. This approach becomes far more useful when there are many different implementations of an interface.

The actual logger in xLPR version 3.0 works in this manner, making the logger a great example to demonstrate highly cohesive and lowly coupled behavior. The user can select what types of messages to log and where they are displayed. These preferences are stored in a data structure, on which the "Logger" interface is called. The logger then does all the work in deciding how to actually accomplish the specifics of the logging. This makes it very easy to work with the logger from a high level, because any changes in user preferences do not change the way that the logger needs

to be called. If this strategy were not used, a less effective way to accomplish the same thing might be a set of conditional statements that select a logging behavior depending on the user preferences, calling a different function in each outcome.

2.2 Test-Driven Development

When developing a large code base it can become very difficult to ensure correct functionality if an effective testing regime is not followed. Functionality that needs to be tested can inadvertently be overlooked if not tested immediately. The xLPR version 3.0 code was written following the test-driven development approach. Test-driven development entails writing a test for new code before (or at least concurrently with) any new functional code. This strategy helps the developer avoid several problems that are common in code development.

The first way that this approach aids in the creation of good code is that it forces the developer to clearly define the purpose and functionality of each piece of code. It is common for developers to begin writing code with a vague idea of what it should do that evolves to address problems that arise later. If the test is written before the code itself, it forces the coder to define exactly what the code should do and how it connects to adjacent parts of the code. This prevents backtracking and issues of inter-connectivity between existing and new pieces of code.

Test-driven development also requires developers to write concise, easily testable code with clear intent. It is common to see code where a small number of functions perform large scopes of functionality. This causes difficulties with maintainability because it can become very difficult for a developer to tell what the purpose of each function is. It is more effective to have a function with a single purpose, that is clearly identified by the name of the function if possible. If the test is written before the code

itself, as in test-driven approach, then the coder has already identified the purpose and use of a new function, and therefore can neatly place it inside a single function. Simple functions are easier to test because tasks are limited to accomplish an explicit purpose.

An example of how test-driven development was used in the xLPR Version 3.0 project is in the creation of code that replicated mathematical expressions in GoldSim. During this replication process a developer would identify the expression that would be reproduced, write a test for said expression, then write a function that performed the operation exactly how it was done in GoldSim. It may seem trivial to translate a mathematical or logical operation essentially from one coding language to another, but the testing proved very helpful in this process. Differences in syntax, complicated logical trees, and differences in variable typing between the two frameworks were all common issues that required careful treatment from the developers. The test-driven development approach ensured that most problems were caught immediately before moving on to the next chunk of code.

2.3 Readability and Maintainability

A large code that is developed and maintained by a group of people needs to be written in such a way that one developer can look at a portion of the code that they have never worked with before and understand what it does without too much trouble. Many less experienced developers rely only on comments to communicate the purpose of code, but this method has some severe limitations. Comments work well if the code is complete and will no longer be changing, or if care is taken to update them alongside the code, but it is often the case that comments are not updated along with changes in the code itself. As such, the first line of defense should be to write

code that is very readable.

If the code is well structured, such that each function has a specific purpose, then it is not too difficult to give each function a descriptive name. Then, even if the inner workings of the function are modified, the name still describes what the outcome it. If comments are used that describe how the function works, then they may become outdated if particular care is not taken. If a function is so complex that it becomes difficult to choose a name for it, it is probable that the function can be divided into multiple clearly named functions. Both having a specific name, and being simple enough that the function is easy to name are good indicators of readable code.

Another safeguard against having code that is difficult to maintain is having good documentation. In xLPR Version 3.0 creating the documentation is similar to writing comments in the code, but a web version is automatically generated that is easy to view. Documentation should primarily describe what code does, as opposed to how it works. The same guidelines exist for writing good documentation as for good comments. Care must be taken that the documentation matches the current behavior of the code.

Chapter 3

Project Contributions

The xLPR 3.0 project was large in both scope and duration. My contributions were largely limited to a few major parts of the code, though I was exposed to many parts that I will not describe here. My early contributions were more basic, but throughout the duration of my internship I was able to accomplish more complex tasks as my skills grew.

3.1 Reproducing Expressions

A large component of the work needed on the xLPR Framework Version 3.0 project was reproducing expressions in the Go code that were previously implemented in GoldSim (Figure 3.1). This work can be fairly repetitive and tends to require less skill than many other sections of the project. This made reproducing the expressions a good place to start for inexperienced developers. This task is useful to give a new developer time to become familiar with the project as a whole and come in contact with many parts of the math driving the simulation.

The scope of the project included the translation of over four thousand elements

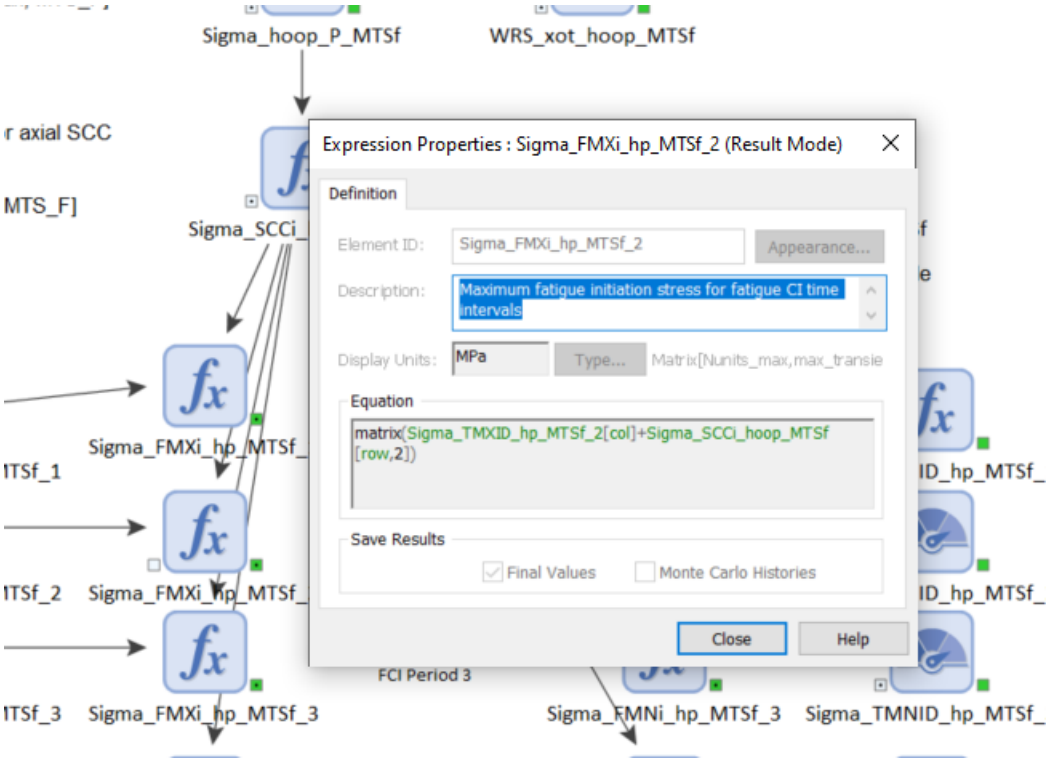

Figure 3.1 An example of a GoldSim element that was reproduced in Go. Pictured is the dialogue box that opens when a user clicks on an element in the graphical interface. In this case the expression includes a Element ID, description, display units, and equation (which describes the actual functionality of the element). The corresponding Go code snippet is included later.

to the Go code. Nearly two thousand of these were mathematical expressions and the rest involved initializing, storing, and organizing information. Each expression needs to not only be reproduced, but be documented as well. With so many pieces of code it is important to stay organized.

Each of these new functions in the Go code needed to be linked in to the simulation chain correctly. They also needed to have the correct dependencies and inputs set up. To ensure correct behavior, testing must also be created to catch mistakes and errors.

An example of a GoldSim expression that has been reproduced in Go code is included here. This is a fairly typical example. The code snippet includes documentation explaining what the expression does, as well as the logic of the expression itself.

```
1  // GOLDSIM: Sigma_FMXi_hp_MTSf_2
2  //
3  // Maximum fatigue initiation stress for fatigue CI time intervals
4  func SigmaFmxiHpMtsf2Trafo(input pc.ParamCollector) {
5      result := utils.Matrix[utils.Real](input.Int(prm.NunitsMax),
6      input.Int(prm.MaxTransients))
7      for row := range input.Int(prm.NunitsMax) {
8          for col := range input.Int(prm.MaxTransients) {
9              result[row][col] = input.Real1D(prm.SigmaTMXIDHpMtsf2)
10             [col] + input.Real2D(prm.SigmaScciHoopMtsf)[row][1]
11         }
12     }
13     input.SetValue(prm.SigmaFmxiHpMtsf2, result)
14 }
```

3.2 Unit Testing

With such a large volume of code being translated from one format to another, it is very important to conduct testing of the code as it is being written. If the testing is left until the end of the project it can become very difficult to track down small issues. Test-driven development is a strategy that was used to prevent this issue (see Section 2.2). The test-driven development approach involves writing tests for new code concurrently or even before the functional code itself. This ensures that the developer has a clear definition of what the code should be doing before the code is even written.

GoConvey was used to carry out testing for this project. GoConvey allows the developer to combine a string description of the test with a set of assertions that check the code for the desired qualities. In VSCode (the development environment used) Go code can be tested in .go files with names ending in _test, which makes it easy to run a single test without compiling and running the entire code. The test code can call a function that includes the code being tested and check that once the fuction is run certain test criteria are met.

Though all the expressions being implemented were unique, there were several types of tests that are important to include to prevent common developer errors. The most obvious of these is to check that the expected value is correct with a given set of inputs. This ensures that the values or logical behavior in the Go code remain consistent with GoldSim after the expression is called. It is also very useful to check variable type and array lengths to make sure that they match the expected results. Since type was not always explicitly declared in the GoldSim implementation of xLPR, it is important that the types selected in Go will function correctly in all cases where each variable is used. Testing helps catch inconsistent type use or cases where the

type chosen does not work correctly. In cases where more complicated operations are programmed it is very important to test each possible case carefully to avoid error. A difficulty encountered with this style of unit testing is that in some cases it is not possible to compare code behavior directly with GoldSim and the developer must then test the Go code against what they think should happen. Errors that were not caught during this stage of development were caught later on during the more thorough integrated and regression testing regimes.

An example of a unit test Go code snippet is included here. This is the test corresponding with the Expression in previous section.

```
// GOLDSIM: Sigma_FMXi_hp_MTSf_2; Pos 214
func TestSigmaFmxiHpMtsf2Trafo(t *testing.T) {
    pc := MakeDefaultParameterCollection()
    pc.SetValue(NunitsMax, 2)
    pc.SetValue(MaxTransients, 2)
    pc.SetValue(SigmaScciHoopMtsf, [][]Real{{1, 2}, {3, 4}})
    pc.SetValue(SigmaTMXIDHpMtsf2, []Real{1, 2})
    Convey(PCgivens(pc, true), t, func() {
        Convey(WhenTrafoCalled(SigmaFmxiHpMtsf2Trafo), func() {
            SigmaFmxiHpMtsf2Trafo(pc)
            exp := [][]Real{{3, 4}, {5, 6}}
            Convey(Sfmt("THEN %s is added to the PC and is equal to
            the expected value", SigmaFmxiHpMtsf2), func() {
                So(pc.HasParam(SigmaFmxiHpMtsf2), ShouldBeTrue)
                for row := range pc.Int(NunitsMax) {
                    for col := range pc.Int(MaxTransients) {
                        So(pc.Real2D(SigmaFmxiHpMtsf2)[row][col],
                        ShouldEqual, exp[row][col])
                    }
                }
            })
        })
    })
}
```

3.3 Integrated Testing and Regression Testing

Integrated testing involved stringing together sections of code and testing the behavior of the entire chunk against the same chunk in GoldSim. This was a time consuming process due to the difficulty in extracting values from the GoldSim simulation. The method used involved several steps of going back and forth between the Go code and the GoldSim implementation to make sure the correct parameters were being tested at the correct part of the simulation.

To conduct one of these tests, the developer would:

- Run a section of code through some integrated testing functions that identify which variables were inputs and outputs for that section of code.
- Create elements in GoldSim specifying inputs and outputs that are needed for the testing and when to sample their values; these elements generate a file with the values of each of the specified variables as they are sent to and from that section of expressions.
- Run the integrated testing code which passed the input values from GoldSim to the code in Go, collected the Go outputs, and compared them with the GoldSim outputs; the test code would display names and values of any variables that did not match between the two codes.
- Use the information from the test to recheck functions where problems occured and make fixes.

One of the limitations of this process is that the list of needed variables was generated in Go, where errors were expected. This means that if errors existed and the wrong section of code was accessed, or if parts should have been accessed but were

not, then the test would proceed in the wrong direction. The wrong variables could be tested, and possibly even pass the test without ever identifying all the underlying issues. Additionally, GoldSim inputs were passed to the Go functions because incorrect Go inputs were not directly addressed (except possibly in a separate integrated test), therefore occasionally causing the wrong test variables to be identified by following the wrong logic paths through the code. Because the initialization of the code was not tested using integrated testing, but regardless contained several errors, many errors appeared in integrated testing that were difficult to find. Often an error would appear and the development team would spend large amounts of time following the error backwards through dozens of expressions only to find the source of the error in the initialization code.

Regression testing was very similar to the integrated testing, except it tested the entire code rather than a single section. This phase of testing was completed very quickly as nearly all of the errors had already been caught. Where integrated testing tested many sections of code each using a single simulation input set, the regression testing verified that the entire code functioned correctly with a large variety of input sets.

3.4 Logging and Error Handling

By the time I started working on the logging and error handling section of the code I had plenty of experience reproducing existing expressions in Go and creating and carrying out testing. This new section of the code was an opportunity for me to write new code and practice implementing effective software development strategies.

During the running of an xLPR simulation there are various levels of warnings and errors that can appear to a user. It is important that the code respond correctly

to these events and display the behavior and its cause to the user. The logging system that would describe the behavior of the simulation to the user was developed concurrently with the code that checked for and addressed these error conditions.

The logger was developed so that the user can specify a number of locations to which this information is logged. This can be to a file or directly to the terminal. The user can also specify the level of verbosity of the logger, thus choosing what types of behavior they want to be notified of. I was involved with the design of the logger, and in writing code to implement it inside the simulation. I also added the capability of the logger to display messages at multiple locations, rather than only one.

In GoldSim there are elements that check for certain conditions and can modify the behavior of the simulation in response. For example, a statistical realization can be skipped if there is a condition that might render it invalid. I created functions in Go that match the behavior from these GoldSim elements. They are linked in to the simulation the same way a mathematical expression might be (the simulation execution consists of running a chain of functions that each pull from a dynamic parameter collection). When these error functions are called at the correct time during the simulation, they read values from the parameter collection and send a message to either continue the simulation or break the chain and alter its behavior. A string is also sent to the logger to alert the user if the simulation behavior was altered by this function and why.

Chapter 4

Conclusion

4.1 Skills Learned

The nature of the xLPR 3.0 project was such that the skills developed were primarily associated with software development. Even though it is a physics based simulation, the physics was already in place, and mathematical expressions merely needed to be reproduced, not created from scratch. That being said, the proximity to said physics and industry relevant software was useful.

Prior to this internship I did not have very much coding experience, and, in addition to learning to use a new language (Golang), I also learned a lot about effective programming practices. The only experience I had beforehand was using small scripts to perform physics calculations. It is a much different experience to work behind the scenes on a large software project on a team with several people. It is difficult to begin work on a large, existing code base. I now feel much more confident doing so and have a better sense of what the challenges are and how to approach them.

Reading code written by other developers has helped me to understand what

makes code effective. An important part of software development is ensuring that the code is readable, maintainable, and easy to use. When working on small projects that are not intended for other people to use it is easy to set aside good practices and just do what is fastest or most simple, even if it makes the code harder to read or use.

4.2 Personal Challenges and Reflection

While I learned many things during this time, there are still many areas where I have a lot to learn. I have had some practice implementing effective programming practices and utilizing more advanced programming concepts, but it is such a broad and complex field that it takes time to become highly skilled.

All of the concepts touched on in this paper were things I learned about and got some practice with, but still have a lot of room to improve on. For example, there are many techniques that I have yet to learn to make code highly cohesive. I learned a few approaches that have improved the quality of my code compared to what it was before, but there are so many things I am not yet aware of that can improve the quality of my work.

Another struggle that was difficult during this internship was not knowing what to ask. There were many times when I would grapple with a problem for a long time, only to later learn that there is an easier solution, or that somebody had already solved the problem somewhere but I did not know where to look. This is a challenging problem to address, but it can be very helpful to check in with more experienced developers frequently to get feedback and advice.

4.3 Professional Growth

I am aiming for a career in the nuclear engineering field, particularly in the safety analysis space. This work often heavily relies on the use of safety analysis software to assess risk of physical nuclear power equipment. The use of many widely used safety analysis codes in this field involves scripting or some other form of programming to get maximal use out of the tool. The code development experience that I have gained from this internship will be extremely useful for a future utilizing this type of software. Not only will I have a better understanding of how they work behind the scenes, but I will feel more confident using them.

During my time at ISL I have also learned a lot about the nuclear engineering industry from coworkers and mentors, many of whom conduct nuclear safety analyses frequently as part of their jobs. I have a much better idea of what the work is like, and what knowledge and skills are useful in the field than I did before. I am also much more confident going into big projects that can feel overwhelming at first.

At the time of writing, the xLPR 3.0 project is wrapping up and I am starting work on TRACE, another nuclear safety analysis code produced by the NRC and widely used in industry. Working on this new project is very helpful to expand my experience and knowledge of the industry. I am eager to continue learning and advancing my career.

Bibliography

[1] P. D. Mattie, C. J. Sallaberry, J. C. Helton, and D. A. Kalinich, "Development, Analysis, and Evaluation of a Commercial Software Framework for the Study of Extremely Low Probability of Rupture (xLPR) Events at Nuclear Power Plants," https://www.osti.gov/servlets/purl/1005032 (Accessed October 1, 2025).

[2] P. E. Mariner, "xLPR Sim Editor 1.0 User's Guide," https://www.osti.gov/servlets/purl/1365467 (Accessed October 1, 2025).

[3] "xLPR V2.0 EXAMPLES OF IT APPLICATIONS," https://www.osti.gov/servlets/purl/1465913 (Accessed October 1, 2025).

[4] C. Nellis, H. Mendoza, "Extremely Low Probability of Rupture Code NRC Update 2025," https://www.nrc.gov/docs/ML2516/ML25161A162.pdf (Accessed October 1, 2025).