Quantifying Fault Introductions Across Software Versions

Devin Pohl
Department of Electrical and Computer Engineering
Under Professor Dr. Yashwant Malaiya
CS 530: Fault Tolerant Computing
Email: Devin.Pohl@colostate.edu

***Abstract*—When a software project is updated, the act of modifying a module can introduce faults. For example, a developer may re-implement a sorting algorithm in a more efficient way, but make a mistake due to the increased complexity of the new implementation. This project quantifies the occurrences of this type of fault. This has primarily been done through the development of an automatic white-box back-to-back testing tool which operates via symbolic execution of LLVM. These dependencies will need to be employed in a non-standard way in order to solve the problem of program equivalence for practical domains, as no effective tools yet exist for this exact objective. While not an ideal approach, providing a functional demo is more than enough to highlight upcoming issues with this type of tool, and point further development in the right direction. Contributions include extensions of a public library for symbolic execution, and authoring of a completely novel prototype for practical programming equivalence testing. This report includes research on several of the challenges and shortcomings discovered during this process.**

I. Introduction

Software faults generated between software updates, known as *regressions*, is a fundamental problem which has plagued developers since at least 1988 [1]. But despite this, the problem is yet to be fully solved. Today's industry standard methods of back-to-back testing are entirely black-box based approaches [2], which fundamentally cannot uncover *all* faults in a piece of software. This project presents a possible solution to this issue, relative to the domain of back-to-back testing. Although only a proof of concept, this project is intended to highlight potential future issues in a robust implementation, as well as show just how advantageous such a tool would be. Marked success has been demonstrated in this regard, and a list of research topics prepared.

But before implementation and methodology is discussed, it is useful to take the time to describe *exactly what* back-to-back testing is. The intent of back-to-back testing is to discover regression faults. This type of fault occurs when a pre-existing module (function) is modified incorrectly. Sources of these faults can include incorrectly performed optimizations, updating of dependency versions (where either the dependency itself, or its application is done incorrectly), addition of flawed redundancy code, or errant reimplementation to improve cleanliness. Through these actions, it is possible for developers to introduce faults to previously working software.

The task, then, is clear: move away from the black-box testing approach and towards a white-box approach. Luckily, this can be done somewhat simply by testing *program equivalence* between the previous and new versions. If the two modules are equivalent – that is for every input they have the same behavior/output – then it is impossible for any faults to have been introduced. The issue with program equivalence is that it is fundamentally an unsolvable problem. Due to this roadblock, there has been an extreme lack of research in regards to white-box approaches. In fact, the methods practiced in this report **have never been done before**. No tools exist for program equivalence testing.

However, tools do exist that solve an adjacent problem: symbolic execution. This is the practice of passing *symbolic* values to a function, then running control-flow analysis and calculating every possible behavior of a function. Symbolic execution is fundamentally different from brute-force testing, as values exist as ranges until split up by branches [3]. In this way, computational complexity is reduced from number of possible inputs, to number of possible branches. Symbolic tools can be used to solve program equivalence problems in practical domains, but because they are not optimized specifically for program equivalence, a large amount of performance is left on the table. Therefore, this approach serves as a good proof of concept for highlighting potential future issues in the field, but not for a robust implementation. It is intended that this paper will outline areas of research which must be completed before the ideal robust implementation can exist.

To facilitate this goal, the following unique contributions have been made:

1) Modification to an existing symbolic execution engine to facilitate program equivalence reporting
2) Authoring of a proof-of concept white-box back-to-back testing tool
3) Notation and exploration of several required research areas

In regards to the findings of the project, the following problems are highlighted. These problems must be solved before a robust implementation can exist; this report not only brings attention to these areas of research, but also presents cursory research on potential solutions. The problems discussed are:

- How can divergent behavior be effectively communicated to the programmer?
- What are the advantages and disadvantages of performing symbolic execution of LLVM?
- How can object and memory preconditions successfully be initialized during test setup?
- How can symbolic execution results be correctly and quickly compared?

The remainder of this document includes explanations on methodology of research and development, detail on unique contributions, and analysis of findings. By the end of this report, the advantages of a white-box approach will be clear, and the remaining roadblocks behind its proper implementation enumerated.

II. Methodology

As previously mentioned, this is a heavily development-focused project. Concepts are explored via organic discovery; construction of a solution is attempted, challenges noted, and research performed on potential ways forward. This section of the report covers *how* the research and development of this project has been completed.

A. Assumptions

In order to provide useful analysis, some simplifications of the problem space were required. These simplifications were only made to lower the problem complexity – they do not fundamentally change the problem, and the analyses gathered are still worth further investigation. With that said, there were three major assumptions/simplifications made during the development portion of this project. The first simplification is excluding analysis of multi-file code. That is, code wherein multiple source files compile to a single binary. The reason for this limitation is purely one of effort allocation; inclusion of this feature does not improve the quality of generated research topics. The second assumption made is that code is dependency-free. This means that any code analyzed cannot use libraries or tools. This limitation is an extension of the single-file restriction. And although this feature could have easily been supported in the tool, it was deemed as unnecessary for the main goals of the project. A third assumption is that functions are without side-effects; this means that functions do not modify the global state. Analysis of side effects is a topic so complex as to detract from the topic of this work. The final assumption made is the largest and most impactful: the only functions allowed to be analyzed are functions which take primitive data types as input, and return a single primitive data type as output. Primitive data types include integers of different bit-widths, floating point numbers, and generally any data which can fit into a single register and passed by value: no objects or pointers. This assumption significantly limits the analysis capabilities of the tool. However, the reason for this assumption is an important finding which will be discussed in further sections of this report.

B. Objectives

In the introduction section of this report, it was stated that the white-box analysis tool will operate on the goal of automatically proving program equivalence. To give a concrete example of program equivalence, Listings 1 and 2 show examples of equivalent programs. Although these two programs are implemented entirely differently – one uses an iterative approach and the other uses a recursive approach – they are *equivalent*. That means that for every input of the `fibbonacci` function, the outputs are the same for both functions. For functions without side-effects, this definition is sufficient. For functions with side-effects (which, as previously mentioned, are outside the scope of this project), the definition includes checking that the side-effects are the same as well.

```
#include <stdint.h>

uint8_t fibbonacci(uint8_t x) {
  if(x==0)
    return 0;
  if(x==1)
    return 1;
  return fibbonacci(x-2)
    + fibbonacci(x-1);
}
```

Listing 1: Recursive Solution

```
#include <stdint.h>

uint8_t fibbonacci(uint8_t x) {
  uint8_t n2=0, n1=1;
  for(uint8_t i=0; i<x; ++i) {
    uint8_t tmp = n2+n1;
    n2 = n1;
    n1 = tmp;
  }
  return n2;
}
```

Listing 2: Iterative Solution

To be specific about the goals of the aforementioned program equivalence testing tool, it is intended that the tool will be provided two files. The first file will contain the old, reference implementation which is assumed to have correct behavior. The second file will contain the new, updated code which is under scrutiny. The tool is to cross-reference between the two files, find common functions, and analyze them for program equivalence. For each function pair, each function is separately loaded into the symbolic execution framework. Then, every possible function output is reverse-associated with the corresponding input which produces the relevant output. Because this is done through symbolic execution as opposed to brute force, this is computationally reasonable for a proof of concept tool. Finally, the input-output associations for each

function are compared. If the associations are the same, then the programs are equivalent. If the associations are different, then the tool is to output minimal debug info pointing to *how* the two functions are different. This analysis is completed for every common function in the two files. In this way, the goal of the tool is to *fully and completely* test for fault introduction across software versions.

C. Choosing an Engine

As alluded to in the introduction of this report, the largest setback in regards to program equivalence testing is that no dedicated tools for the task exist yet. Of course, that is the goal of this project. However, in order to avoid time-consuming implementation, it was required to use a symbolic execution engine. This approach has the disadvantage of decreased performance, but does allow the project to be completed in its entirety. With that in mind, preliminary research in this project focused on choosing what symbolic execution engine to choose. Through the process of this research, a several options were considered. The most promising of those options, with a short list of pros and cons, are as follows:

- KLEE [3]: industry-standard, LLVM symbolic execution
  Pros: Robust operation, fast execution
  Cons: Difficult to implement, ineffective documentation
- Haybale [4]: Rust based, LLVM symbolic execution
  Pros: Fairly robust, easy to manipulate
  Cons: Requires modification to test program equivalence
- Seer [5]: Miri [6] fork supporting symbolic execution
  Pros: Tightly integrated with Rust ecosystem
  Cons: Severely out of date, only supports Rust
- Otter [7]: symbolic execution of C code
  Pros: Complete support for all language functions
  Cons: Prohibitively complex to implement
- PyExZ3 [8]: symbolic execution of Python code
  Pros: Extremely easy to work with
  Cons: Python is seldom used for fault-sensitive code

In order to produce a genuine list of pros and cons, simple analysis programs were written with each of the above options to test ease-of-implementation. After this process, it was decided that Haybale would be used for final implementation. This option was chosen for a few reasons: this tool is fairly up to date, supports multiple programming languages, can be manipulated quickly, and is built on top of Rust. The last point is of particular importance, as Rust offers advanced and mature tools for dependency management, build tooling, automatic testing, and application bundling. These factors all contribute to highly efficient development of the tool. The only notable downside to using Haybale was the need to modify its source code to support program equivalence testing – this is expanded upon in the next section of this report.

III. Contributions

Considering that this project is heavily development based, and that this exact problem lacks any existing development whatsoever, multiple novel contributions were made. These contributions include modifications to Haybale to support program equivalence debugging, authoring of a tool which utilizes Haybale's symbolic execution engine to actually perform back-to-back white-box testing, as well as reporting minimal debug info when the tool detects faults. Additionally, contributions include preliminary research on several of the challenges inherent to this type of testing. In this section, only the first two contributions are detailed. The next major section of this report is dedicated to the final contribution.

A. Modifications to Haybale

1) Task: As stated previously, the symbolic execution engine chosen for this project, Haybale, required some modifications in order to give useful debug information during program equivalence testing. Performing these modifications was the first major goal of this project. However, these modifications were not purely for functionality; development in this regard shed light on the issue of effectively communicating divergent behavior to the programmer. In this way, the goal of modifying Haybale was instrumental to several aspects of this project.

But before explaining the modifications performed on Haybale, it bares explanation of exactly *what* Haybale is. This is best explained with examples. Listing 3 contains code taken from the Haybale GitHub README [4]. It is shown in the README that using provided analysis functions, Haybale is able to find the *zeros* of this function, or the input values for which this function returns zero. This is only one example of what a symbolic execution engine can do; another analysis function provided by Haybale can provide a simple list of all the possible return values of a function.

```
int foo(int a, int b) {
    if (a > b) {
        return (a-1) * (b-1);
    } else {
        return (a + b) % 3 + 10;
    }
}
```

Listing 3: Sample C++ Code

The limitation of Haybale, as it stands, is its lack of debug information. Specifically, when the program under execution crashes or throws an error, little is known about the error details. Prior to this project's contribution, the only information included was the value of the error thrown, or whether not the program crashed.

In order to built a tool which is effective at reporting deviant behavior between two versions of software, communicating program flow via a stack-trace is required. Additional information on the program state during error generation would potentially be helpful as well. With this goal in mind, the first task for this project was clear: attach debug information to crashes.

2) Work: With the task of attaching debug information to program crashes clear, implementation could begin. The first step in this regard was identifying the data structures internal

to Haybale responsible for holding symbolic states. It was found that Haybale holds computation values as an enum, with possible options of a normal value, the void type, a thrown value, and an abort. The normal and thrown value variants include a symbolic bitvector representing the memory of this value. This symbolic bitvector is implemented with Boolector [9]. This is a library which solves Satisfiability Modulo Theories of arbitrary-length bit-vectors. This means that the contents of a bit-vector are represented as a series of equations, and manipulations of data is done by algebraically manipulating the representative equations. Then, analysis operations (such as computing possible values, or equality testing of two statements) are done by mathematically solving the provided theories under certain assumptions. Haybale performs symbolic execution by manipulating Boolector bit-vectors, and provides useful analysis by solving the resulting systems of equations under input assumptions.

Now that the fundamental operation of Haybale was understood, modifications could safely occur. As previously mentioned, Haybale runs on LLVM code. At any point in execution, the symbolic state of the LLVM program can be analyzed. In order to perform the required modifications, code was added to detect aborts and instances of thrown errors. This code then attached a copy of the program state to the internal value enum. Additional code was added to properly handle propagating these values all the way to program termination. The result of these modifications was proper debug information on program crashes: when a program aborts or throws a value, the programmer can see *where* and *how* the error occurred.

3) Results: These modifications, thankfully, were successful. Listing 4 shows a simple example of the type of code that the above modifications affect. This code has the possibility to `throw` an error. It is now possible for a simple analysis function written on top of Haybale to not only detect the presence of a possible error (as was the previous behavior), but point to the exact line of the error and conditions for which it can occur. These modifications have laid the groundwork needed to discuss how a program equivalence testing tool can effectively communicate divergent behaviors to the programmer. This topic is discussed further in the next major section of this report.

```
unsigned short try_clamp(unsigned int a) {
    if(a > USHORT_MAX)
      throw a;
    return a;
}
```

Listing 4: Code to Test Haybale Modifications

B. Authoring of `rust-eq`

With Haybale prepped for use, the coveted program equivalence tool was ready to be written. This tool, titled `rust-eq` [10], can successfully perform back-to-back white-box testing. This section will outline its construction, operation, and capabilities.

1) Construction: As `rust-eq` is built in Haybale, which is itself written on Boolector, significant work was required to provide an analysis environment. The first step in this process was setting up the program interface. It was decided that, via command line arguments, two files would be provided. An additional command line flag `--complexity` was included to bound the computation time of equivalence tests.

With the program initialization settled, the second stage of development focused on generating the LLVM bit code. To do this, the tool creates a temporary directory, then runs either `rustc`, `clang`, or `clang++` depending on the file extension. This compiler pass generates the files for analysis. In the current scheme, each of the two input files generates a separate bit code file – the pros and cons of this approach is discussed in the next major section of this report.

The next step in development was performing the symbolic execution. First, an LLVM parsing library was used to interpret the bit code. Then, the parsed bit code was passed to Haybale for symbolic execution.

Finally, the execution results can be compared. This was a significant challenge due to how Boolector works; each Boolector system is an independent environment. Because there are two separate bit code files, Haybale generated two separate Boolector systems. And given that Boolector cannot perform manipulations or comparisons across systems, a workaround was required to still provide functionality. This workaround came in the form of partial enumeration on concrete ranges. Essentially, each return value (range) was mapped to an input value (range). Then, these two maps were compared. This was a non-ideal approach, but was quick to implement. A potentially better method is discussed in the next major section of this report.

2) Results: As the proof of concept tool has been completed, it bares showing some examples of what it can do. The first example was actually shown near the beginning of the paper: Listings 1 and 2 can be proven equivalent with this tool. The tool can also handle Rust code: Listings 5 and 6 are proven equivalent. Even *cross-language* analysis can be performed. Listing 7 shows an implementation of a clamp function in Rust, where Listing 8 shows an alternative implementation in C. The tool shows that these two functions are equivalent, even despite subtle differences between how the two languages handle common operations.

Of course, the provided examples are simple, toy programs which are of little consequence to literature. It needs to be stressed that larger, more complex programs can be analyzed too. Programs which manipulate linked lists, sort arrays, and search binary trees have been proven equivalent across different implementations with this tool.

IV. Analysis of Findings

The major goal of this project is to highlight areas of research which need to be completed before a production-ready white-box back-to-back testing tool can be written.

```
fn foo(x: u8) -> u8 {
    if x%4 == 0 {
^^I0
    } else if x%2 == 0 {
^^I2
    } else if x%4 == 1 {
^^I1
    } else {
^^I3
    }
}
```

Listing 5: A Rust Test

```
fn foo(x: u8) -> u8 {
    x%4
}
```

Listing 6: Refactored Rust Test

While the tool described in this report, `rust-eq`, is not the production ready version, it has undergone enough development to encounter several issues. These issues, fundamental to the process of program equivalence testing, were then given precursory research. This section outlines these findings, anecdotal evidence of how they block development, and research on potential solutions.

A. Divergent Behavior

1) Overview: The first challenge was discovered before development even began: how can divergent behavior be effectively communicated to the programmer? Properly formatted debugug information can help to quickly pinpoint where and how faults are introduced, but is difficult to be done correctly.

```
fn clamp(x: u8, y: u8) -> u8 {
    if x > y {
^^Iy
    } else {
^^Ix
    }
}
```

Listing 7: Clamp Function in Rust

```
#include <stdint.h>

uint8_t clamp(uint8_t x,
^^I      uint8_t y) {
  if(x>y) {
    return y;
  } else {
    return x;
  }
}
```

Listing 8: Clamp Function in C

This difficulty stems from the fact that debug information is generated from program execution, while effective communication must occur at the source code level. In regards to this type of project, analysis is done on LLVM code but debug information needs to be printed on top of the C/C++/Rust/etc code in order to facilitate maximum communication. This complicates the generation of debug information significantly.

With this in mind, it is useful to prototype what effective debug output could look like. Listing 9 shows one such example. This example is inspired by the way `rustc` manipulates and points to source code upon errors in compilation.

```
error: Crash behavior is not equivalent
 --> old.cpp:5:6
  |
1 |   int foo(int x) {
2 |     if(x<10)
  |        ^^^^--------------------------------------\
3 |       throw x;                                   |
  | /-----^^^^^^^ throws in different conditions     |
 --> new.cpp:5:6                                     |
  | |                                               |
1 | | int foo(int x) {                              |
2 | |   if(x<20)                                    |
  | |      ^^^^--------------------------------------|
3 | |     throw x;                                  |
  | \-----^^^^^^^ compared to this                   |
  = note: Due to different behavior in branches ->-/
  = help: They differ on input values x:[11-20]

error: aborting due to previous error
```

Listing 9: Debug Output Prototype

The output in Listing 9 is something reasonably attainable with the modifications to Haybale previously mentioned in this report. The current behavior of `rust-eq` does include some output on error locations and control flow, but does not take the next step of providing graphical output. This was deemed outside the scope of this project.

2) Research: With the problem introduced, this section introduces some research on the topic of good debug output. Given that this debug information is presented on the source code level, the most relevant topics in literature address the effectiveness of compiler error messages. And although the topic of comparing two source code files has yet to be discussed, extensive research on effective communication of errors in a single file exists. The following paragraphs will detail existing literature in the field, and extrapolate the techniques discussed to the problem at hand.

The first paper to discuss relates on the testability of error message effectiveness [11]. The obvious conclusion is that better error messages allow programmers to more quickly identify and fix errors. However, this paper also found that the signals of effectiveness are rather weak. In essence, although there is a strong correlation between better error messages and better developer response, the trend has dilute impact. The reason for this, cited in the paper, is that most prior studies focus on the number of errors *introduced* by programmers, not how effectively those errors could be fixed by programmers. However, significant difficulties in the second approach were

observed. Namely, it is particularly hard to normalize results based on programmer skill and familiarity with source code. What may take an unfamiliar programmer an hour to fix, may be immediately understood by the programmer who wrote the code. The conclusion gathered from this paper is that although an effective error format exists, determining exactly what that error format is will require *extensive* study including a *large* sample size of programmers. Without such a large dataset, it will be unexpectedly difficult to make headway in the task of producing quality error messages. Recent studies such as [11] have partially addressed this, but more work will be needed to solve this problem in the context of program equivalence.

The second piece of literature to examine in this regard is an article researching techniques to improve error message efficiency [12]. This study analyzes the affects of technical term use, message length, sentence structure, and word choice. Building off of recently developed evaluation methods from [11], the new work tested a litany of possible factors. Surprisingly, a few of these factors were determined to be very effective. The strongest indicator of a good error message is its length; shorter messages were almost universally better understood by developers. The second strongest indicator is in the use of jargon. This report found that error messages with less technical language – save for a few technical terms common to the language producing the error – were far more effective. The conclusions of this study suggest to view error messages not as an explanation of an error, but a means of progressively disclosing information as quickly as possible. This same mechanical approach can, no doubt, be applied to the problem of noting divergent behavior in program equivalence testing. Studies such as these will serve as a useful base for developing robust error messages in a future back-to-back tool.

B. Limitations of LLVM

The second research objective for this domain is in regards to tooling limitations. Symbolic execution engines, of course, need to run on a language input. Analysis can be run directly on high level language, on binary, or anywhere in between. However, based on the specifications of these languages, significant limitations may occur. If a proper white-box back-to-back testing tool is written, its execution language must be chosen carefully. The following paragraphs will explain backing criteria which factor into this decision. Future research focusing on this topic may, then, consider these criteria for implementation.

The first criteria is language specification size. If a tool is supposed to compute program equivalence, it should support a reasonable subset of the language. Ideally, every possible function should be able to analyze, but a few limitations may be reasonable. Examples of this include seldom used features, machine-dependent code, and language edition quirks – a production-ready tool may reasonably omit these aspects for project-complexity reasons. In this way, a lower level programming language may seem better: implementing analysis in MIPS assembly may seem like a good idea. However, the lower level the analysis language, the more difficult the error reporting becomes. Thus, a balance must be struck between a high level or low level analysis language. LLVM seems a popular choice for existing projects solving similar problems [3]–[5,]. However, some additional research needs to be done to ensure this choice can effectively carry over to the problem of equivalence testing.

The second criteria is in analysis accuracy. Aside from bugs in the final tool, is possible never have a false positive (stating that no faults are introduced, but in reality some are). In this way, the tool can always be *correct* in saying that no faults are introduced. However, reporting false negatives is far more acceptable (stating that faults were introduced, when in fact none were). The reason for introducing false negatives are numerous: simplification of implementation, increase of execution speed via approximate computing, etc.. But the largest reason is that some operations simply cannot be symbolically executed. During development of this project, it was discovered that Haybale's symbolic execution engine could not reason about certain possible behaviors of a program. Take, for example, the code in Listing 10. A programmer may be able to reason that the `checked_add` will never experience overflow, as the overflow was manually anticipated beforehand. However, when this function is compiled to LLVM, it emits a call to `llvm.uadd.with.overflow.i32`. This function, according to the LLVM spec [13], this call is allowed to signal overflow with any inputs – even those that would make no sense such as `0+0`. In this way, a symbolic execution needs machine-dependent knowledge to provide a fully constrained analysis.

```
fn clamp_increment(a: i32) -> i32 {
    if a == i32::MAX {
^^Ireturn a;
    }
    return a.checked_add(1).unwrap();
}
```

Listing 10: Partially Non-Computable Function

However, the tool implementer may choose not to provide this machine-dependent information by only running symbolic execution on LLVM. This will provide an unconstrained analysis, but is much more simple to implement. The result is more false negatives – introducing checks may incorrectly signal fault introductions. This is a particularly tricky problem which will need significant research in order for a proper tool to be written. Pros and cons of different backing languages must be considered, and an analysis of how much effort can be expended addressing all language quirks performed.

C. Memory Preconditions

The third research objective relates to a fundamental issue with both equivalence and back-to-back testing: what do function inputs look like? In all the examples in this report, the only functions considered are those which accept integers and return integers. All passed-by-value parameters work with the

approach outlined in this paper. However, analysis falls apart as soon as a value is passed by reference. Object-oriented code cannot be automatically analyzed without significant research. And this research *must* be completed; without analysis of object oriented code, the usefulness of this tool is questionable.

The problem with pass-by-reference function parameters is how the memory is laid out. With a standard symbolic execution engine such as Haybale, the function inputs are represented as symbolic bit vectors. If a pointer is passed to a function, it is assumed to be a pointer to any memory address. The proper scheme requires the pointer *value* to be concrete, but the memory which the value points to be symbolic.

Furthermore, not all memory values are legal. Take, for example, the following scenarios:

- A boolean is passed to a function, but the symbolic execution engine tests what happens if that boolean is neither true nor false
- A C-style `enum` is passed to a function, but the symbolic execution engine tests for illegal variants
- An object is passed to a function. Due to struct packing rules, several bits of memory will be uninitialized or zero-initialized. The symbolic execution engine tests variations in these bits anyway
- An object contains pointers to other objects. Take, for example, a list. Both the length and contents of the list should be symbolic
- Dependent values. For example, code which does operations on error-free Hamming code words – a symbolic execution engine may pass error-containing code words to this function

The case of illegal memory values is a particularly complex one to solve. The complexity is compounded by the fact that in order to solve the problem automatically, it is required to use higher-level concepts from the source language. In the case of illegal `enum` variants, the concept of an `enum` does not exist at the LLVM level. And even if it did, LLVM `enum` values may not be the same as in the source language. Take for example Rust – an `enum` in this language can have objects with data as variants. In this specific case, the problem can be solved automatically somewhat trivially – use the source language to encode data type information, and filter out illegal parameter values before symbolic execution takes place. However, when combining this case with pass-by-reference values, more work is needed.

This discussion unveils the heart of the issue: an automatic method of encoding preconditions to a function is required. When stated in this general form, it is immediately clear that this will solve all the complications enumerated above. Luckily, this exact scheme has been in development for decades; encoding of preconditions is commonly done through *contracts*. There are several programming languages which support explicitly stated contracts – the most popular of which is Ada, a language used which can semi-automatically prove correctness at compile time. The issue with languages such as Ada is that contracts need to be manually written. This cuts down on faults, but still allows for human error.

Automatically generating contracts, then, is the way to go. Literature in this regard, surprisingly, has many ties to cryptocurrency. *Smart contracts* is the designation for the process of automatically proving an operation. This increases security of a blockchain, but has broad applicability in memory precondition generation for white-box back-to-back testing. Furthermore, there is extensive literature on this topic [14]–[17,], so most work required is only transitory. Although difficult, this portion of research is at least straightforward.

D. Comparison of Execution Results

The final major area of required research uncovered during development of this project relates to how results are gathered and compared. This topic was uncovered when writing the result comparison portion of `rust-eq`. Using Boolector as the solving backend has the advantage of reducing duplication of effort, but has the disadvantage of having certain limitations. The largest of these limitations is that Boolector cannot perform operations across engine instances. For example, if a two separate programs are independently ran under symbolic execution, their results cannot react under Boolector. The only way to analyze function results with Boolector is to have both of those functions compile into the same executable. Indeed, many solving engines have the same limitation. The following paragraphs will discuss the pros and cons of this approach, and point in the direction of further research on efficiently solving the problem of result comparison.

The first approach to discuss involves compiling both programs into the same executable. This means that for both the old and new implementations to be back-to-back tested, they will both end up in the same LLVM bit code file. On the surface, this may not seem like an issue, and would solve the problem of multiple-engine instances. However, for certain classes of programs, this will not work at all. The most simple counter-example is in cross-language testing. Putting both Rust and C++ in the same LLVM bit code file, relying on different runtimes, is difficult. This issue can also occur if the language runtime has changed – for example, testing a C99 program vs a C11 program. This issue can occur even with the same runtime if enough of the project has changed; identifying, on an LLVM level, how duplicate named functions and executable entry points need to be handled increases significant complexity to this task. However, handling both program versions in the same executable is not impossible. It just requires creative compilation techniques.

The second approach to discuss involves compiling each program into separate LLVM bit code files. Although this requires either extensive manipulation or outright source code modification of solver back end programs, it massively simplifies the compilation step. It even allows for separating the compilation step from the final product entirely. This has several benefits, such as the ability to handle large projects hassle-free, less work required to support a new language, and less product development required to reach a prototype. It is for this last reason that `rust-eq` was written with the second approach.

Between the two approaches, this report is recommending the second. While the first approach may be easier to make a somewhat quality tool with, the second approach will produce a tool of much higher overall quality. This conclusion is based on the comparison of pros and cons in the previous paragraphs. However, further research on this topic is still required; for a tool which considers all edge-cases and possibilities of program behavior, due diligence on this topic needs taken.

V. Conclusions

The task of developing a white-box back-to-back testing tool is one that needs to be completed. Completing such a task would greatly reduce fault introduction across software versions, and allow developers peace of mind when modifying existing code. However, this task will be extremely difficult.

Through the contributions in this report, an intimate understanding of these difficulties was achieved. Multiple novel contributions were made, both to existing open source libraries and new tools. These contributions culminated in the presentation of a working white-box back-to-back testing tool. Running symbolic execution on top of LLVM, this tool is far from optimized. But although it is not perfect, its development has clearly highlighted several of the challenges pertinent to the goal. With challenges established, precursory research has been performed and leads on possible answers established. With the contributions in this report, the field is ready for heavy research and development into methods for solving the aforementioned problems and close in on a production quality implementation for white-box back-to-back testing.

In years to come, the field of back-to-back testing may see a resurgence in popularity. Automatic analysis coupled with effective reports of divergent behavior will serve as a welcome tool to fault analysis suites. And with adjacent development on smart contracts and active consideration on analysis languages, tools have the opportunity to be both correct and efficient.

With the above considered, this project has been a great success. It has served as preliminary research and development, essentially laying the first pieces of groundwork on the next generation of back-to-back testing. Further work in this field will no doubt be a promising endeavor.

References

[1] M. Vouk, "On back-to-back testing," in *Computer Assurance, 1988. COMPASS '88*, 1988, pp. 84–91.

[2] D. Beyer, *Status Report on Software Testing: Test-Comp 2021*, ser. Fundamental Approaches to Software Engineering. Springer International Publishing, 2021, pp. 341–357. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-71500-7_17

[3] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," Stanford University, Tech. Rep., 2008. [Online]. Available: https://llvm.org/pubs/2008-12-OSDI-KLEE.pdf

[4] C. Disselkoen and H. Ayers, "Haybale," 2021. [Online]. Available: https://github.com/PLSysSec/haybale

[5] D. Renshaw and A. Canciani, "Seer," 2018. [Online]. Available: https://github.com/dwrensha/seer

[6] The Rust Foundation, "Miri," 2021. [Online]. Available: https://github.com/rust-lang/miri

[7] Y. P. Khoo, "Otter," 2013. [Online]. Available: https://bitbucket.org/khooyp/otter/src/main/

[8] T. Ball, T. Pani, V. von Hof, and P. Chapman, "Pyexz3," 2015. [Online]. Available: https://github.com/thomasjball/PyExZ3

[9] M. Preiner, A. Niemetz, and A. Biere, "Boolector," 2022. [Online]. Available: https://github.com/Boolector/boolector

[10] D. Pohl, "rust-eq," 2022. [Online]. Available: https://github.com/Shizcow/rust-eq

[11] B. A. Becker, K. Goslin, and G. Glanville, "The effects of enhanced compiler error messages on a syntax error debugging test," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2 2018. [Online]. Available: http://dx.doi.org/10.1145/3159450.3159461

[12] P. Denny, J. Prather, B. A. Becker, C. Mooney, J. Homer, Z. C. Albrecht, and G. B. Powell, "On designing programming error messages for novices: Readability and its constituent factors," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 5 2021. [Online]. Available: http://dx.doi.org/10.1145/3411764.3445696

[13] LLVM Project, "Llvm language reference manual," 2022. [Online]. Available: https://llvm.org/docs/LangRef.html

[14] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An overview of smart contract: Architecture, applications, and future trends," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 108–113.

[15] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2021.

[16] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018, pp. 1–4.

[17] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: Foundations, design landscape and research directions," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1608.00771v3