# Explorant: Codebase Exploration and Onboarding Tool

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:

Zachary Porter

Project Advisors:

Robert Walls
Gary Pollice

Date: 2022-12-16

# Abstract

Gaining a deep understanding of large codebases is difficult and time intensive. In this paper, we present Explorant, a novel trace-based code exploration tool. Explorant uses rr and gdb to provide in-depth insights into specific code sections and a dynamic state transition showing how the major components of a program fit together. To provide usable graphs, we also incorporate a number of graph simplification techniques. Explorant creates tight experimentation-based feedback loops that enables new ways of reading code.

# Acknowledgements

I would like to express my heartfelt gratitude to my professors for their unwavering support and guidance throughout my MQP. I am particularly grateful to Professor Walls for our deep and passionate discussions, and to Professor Pollice for his invaluable feedback and real-world experience. I am also deeply thankful for my friends' support and my family's love and encouragement.

I'm would never have started such an ambitious project without the support of everyone I have mentioned above, and for that I am eternally grateful.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

Brook's law famously states that ”adding manpower to a late software project makes it even later” [1]. He attributes this to what he calls the ”ramp-up problem” [2] wherein by adding more developers you impose a huge burden on the rest of the team to educate them over the span of multiple months. Our industry experience lines up with this observation. We are deeply familiar with the hours of meetings with mentors drawing diagrams and rapidly taking notes. We are also familiar with the multiple weeks wherein the new developer is paranoid that they have either misunderstood the stack or that some part of the stack was described wrong as they add new code. In our experience, it can take up to a few months before this feeling starts to fade (depending on the complexity of the software). In literature, this phenomeon is often called the ”ramp-up problem” [2] however I will refer to it with the broader term: onboarding. Onboarding varies slightly from company to company or from project to project, and the large efford and time commitment from onboarding will likely never be fully eliminated we may try.

In this paper, we present Explorant, a new code exploration tool that aims to enhance the onboarding experience. With Explorant, senior engineers can annotate their code to produce diagrams that the new engineer can use as a stepping stone for further exploration. Explorant allows developer to quickly understand and relate different components of a codebase based on the execution of a trace rather than the expert knowledge of a senior engineer. We achieve this by integrating finite state machine (FSM) diagrams, source code, and gdb into a single cohesive tool that enables users to easily delve into specific concepts or gain a broad overview of the entire program.

# 2 Background

In order to understand Explorant the details of how Explorant works, we have provided a bit of context on certain aspects of the stack that we consider to be particularly important.

## 2.1 Debugger internals

Debuggers are an essential tool for software developers, as they allow us to pause the execution of a program and inspect its state at any given moment. This is useful for detecting and fixing bugs, as well as for understanding how the program works. In this section, we will introduce some of the internals of debuggers, starting with the INT3 instruction.

The INT3 instruction, also known as the 0xcc byte, is used by debuggers to implement software breakpoints. When a debugger encounters this instruction in the program's code, it will pause the execution of the program and allow the user to inspect its state. This is useful for setting breakpoints at specific locations in the code, and for stepping through the program one instruction at a time.

Another important aspect of debugging is ASLR, or Address Space Layout Randomization. This is a security feature that randomizes the locations of certain parts of the program in memory, making it more difficult for attackers to exploit vulnerabilities. Debuggers must be able to decode the proc-maps file, which contains information about the locations of the various parts of the program, in order to properly interact with the program.

GDB, a popular command line debugger, implements a variety of features to make debugging easier. One feature that I found particularly interesting was how it understands the current stack trace.

GDB reads the stack trace by accessing the memory locations that store the stack frame pointers and function return addresses. The stack frame pointer is a register that points to the current stack frame, which contains the local variables and arguments for the current function. The function return address is the address of the next instruction to be executed when the current function returns.

By following the sequence of stack frame pointers and function return addresses, gdb can reconstruct the sequence of function calls that led to the current point in the program's execution. This information is then displayed to the user in the form of a stack trace, which shows the names and arguments of the functions that were called.

There is a lot to say about debugger internals, but this should be enough information to understand the rest of this paper.

## 2.2 State Diagrams

State diagrams, also known as finite state machines (FSM), are a useful tool in the realm of debugging. They provide a visual representation of the possible states of a program and the transitions between them.

Each node in a state diagram represents a unique state of the program. These states can be thought of as the different possible ways that the program can behave. For example, in a simple program that takes user input, there may be a ”waiting for user input” state and a ”processing user input” state.

Transitions between these states are represented by edges on the state diagram. These transitions are triggered by certain events or actions within the program. For example, in our simple program, the transition from the ”waiting for user input” state to the ”processing user input” state would be triggered by the user entering input.

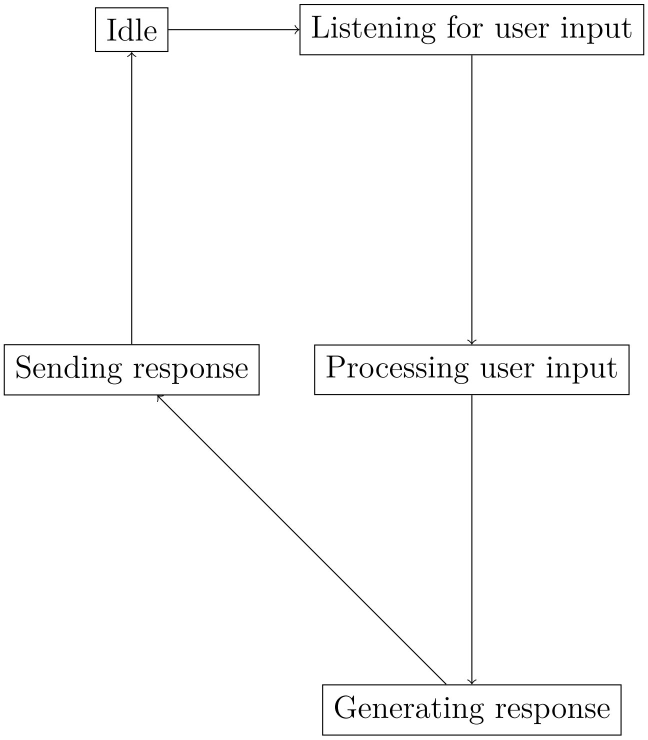


Figure 2.1: Simple state diagram for a bot

Flow control within the program can be easily visualized and understood through the use of state diagrams. For instance, in the case of a conditional statement, the state diagram would split into multiple branches depending on the outcome of the conditional.

This state diagram provides a clear visual representation of a generic bot's behavior, allowing developers to easily understand how the program is executing and how it will respond to different inputs. State diagrams are particularly useful for onboarding new developers, as they provide a high-level overview of the program's flow without requiring a deep understanding of the code.

## 2.3 ELF / DWARF

The ELF (Executable and Linkable Format) is a file format used by many Unix-like operating systems to specify the layout of object files and executables. These files contain a

| **Attribue** | **Description** |
|---|---|
| DW_AT_name | The name of the subprogram |
| DW_AT_low_pc | The starting address of the subprogram |
| DW_AT_high_pc | The ending address of the subprogram |
| DW_AT_decl_file | The file in which the subprogram is declared |
| DW_AT_decl_line | The line on which the subprogram is declared |
| DW_AT_prototyped | Specifies whether the subprogram has a prototype |

Table 2.1: A table of attributes for a DW_TAG_subprogram DIE.

wealth of information about the compiled code, including symbols, debugging information, and other metadata.

One component of ELF files is DWARF (Debugging With Attributed Record Formats), which is a debugging data format that specifies the format of debugging information. DWARF data is embedded in ELF files and can be accessed by debuggers, such as gdb, to provide valuable information about the execution of a program.

DWARF data is organized into structures called DIEs (Debugging Information Entries), which contain tags identifying the type of information they contain. Some common DIE tags include DW_TAG_variable for variables, DW_TAG_subprogram for functions, and DW_TAG_compile_unit for compilation units.

By parsing out these DIE tags, debuggers can access valuable information about the symbols in a program, allowing them to provide useful features such as the ability to inspect and modify variables during execution.

In the context of my live-debugger, Execumap, DWARF data plays a crucial role in my ability to create a state-diagram of the program's execution. By accessing and parsing DWARF data from the recorded trace, I am able to identify the symbols and their relationships in the program.

## 2.4 RR Record and Replay Framework

The rr record and replay framework is a powerful tool for debugging programs. It allows users to record a program's execution and then replay it at a later time. This allows users to easily and quickly reproduce the conditions that led to a bug. This is particularly useful when debugging rare or timing-sensitive bugs like race conditions where gdb might prevent the bug from ever occuring in the first place.

One key aspect of rr is its focus on capturing nondeterministic input rather than the whole trace [3]. In this case, nondeterministic input are the events that could cause a program's output to vary from run to run. This includes things like syscalls, process-switching timings, or even some non-deterministic instructions. By capturing and replaying these inputs, rr avoids having to walk through and instrument the vast majority of executed instructions. To accomplish this, rr uses a wide variety of tools, including the 'ptrace' interface to attach middleware, overwriting the the vDSO (virtual dynamic shared object), limiting a program to a single core, and other more specific techniques.

To intercept many simple system calls, rr can simply overwrite the vDSO which is a use-space code-segement that the kernel exports for code ”that does not necessarily have to run in kernel space” [4] However some code directly calls syscalls. As such, ”when the tracee makes a system call, RR is notified via a ptrace rap and it tries to rewrite the system-call instruction to call into [their] interception library” [5, p. 8]. A diagram of this modification can be seen in figure 2.1.

(TODO) During the replay phase, rr uses this information to replay the system call, allowing the program to execute as if the original system call had been made. This allows rr to accurately reproduce the program's execution, including any system calls that were made. (TODO)

In order to efficently ”continue-backwards”, rr utilizes a checkpointing system. The

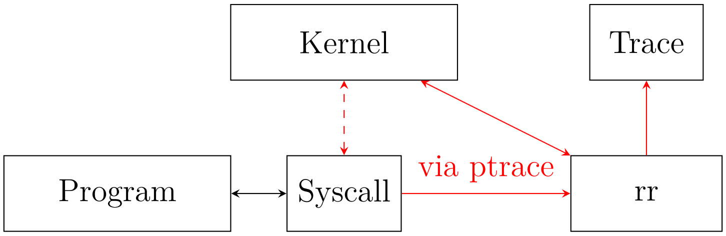


Figure 2.2: Diagram showing how rr intercepts calls into the kernel and records them

checkpointing system works by

`fork`

ing the process to cheaply copy the address space [5, p. 15]. This is efficient because ”fork is (mostly) ’copy-on-write’ and is very well optimized on Linux, so creating a checkpoint typically takes less than ten milliseconds.” [5, p. 15]. This allows rr to quickly and easily restore the program to the state it was in at the time of the checkpoint during the replay phase. Then it can ”continue-forwards” until it reaches the desired location.

The default interface to rr is a gdb server using the gdb messaging protocol. However, this is not a performant solution for a programmatic interface that might be doing queries across the entire execution of a program. As such, we developed librr, a Rust library to interact with the C++ internals and provide nice abstractions such as reading registers, writing bytes in memory, setting breakpoints, etc.

Execumap uses RR (and librr) extensively and would not be possible without them. The ability to gather more information about a programs execution after it happened is has been useful as I am not able to predict all of the locations that a user might want to visit. Similarly, without these tools I would be unable to allow the user to open up the trace to any location within the program’s execution.

# 3 Understanding Code

Understanding code can be difficult, and blindly building a tool to improve the process is likely to get nowhere. As such, in order to learn how we can help build a tool, we conducted a case study wherein we spent a week studying and understanding a code base while recording our observations. The goal was to use only the codebase itself and no video-lectures or explanations that might not exist for all codebases (especially if they are proprietary). The codebase that we chose was the glibc memory allocator, commonly referred to by its most used function, malloc. The memory allocator uses a wide array of complicated datatypes, intricate macros, and C pointer tricks. The memory allocator also is extremely well documented with comments long expository comments. These comments act as a stand-in for a mentor at a company or someone who knows the codebase well.

In order to tackle this problem, we first conducted a bit of research on tools in this area and were fairly dissapointed. Almost all resources we could find online reccomended the same advice: step through the code with a debugger, write some simple code that interacts with the library to explore its API, keep notes, and just read the source code (with tools provide features like jumping to definitions and collapsing modules). (TODO: src)

After our study, we arrived at a the following conclusions:

1. Malloc is more complicated than we had imagined.

2. Gdb is perfect for examining a single function execution in high detail, however it fails at helping the programmer easily understand how functions fit together. In order to understand how functions fit together, we set huge numbers of breakpoints and played around with the execution, jumping around and seeing where things took us. This was useful to understand the sections, however we noticed that we had a hard time dealing

with the signal to noise ratio of gdb.

3. The comments are extremely detailed and any changes that are made to malloc will require careful changes to the documentation that is scattered around the source code. We didn't find any instances of incorrect comments but can easily envision how documentation that is seperate from the implementation could become outdated

4. Writing everything down is painful and tedious however it was very helpful for our understanding as it allowed us to focus on the high level concepts while also explaining the low level implemenation.

5. Jumping around the source code with definitions and code collapsing was critical to keep the amount of information in our "working-memory" focused on understanding an individual task.

# 4 Design

We spent a significant amount of time designing the user interaction for our application. We considered multiple platforms such as a gdb or ghidra plugin or even just a terminal application. However we eventually decided on a more GUI-focused application that would give us more creative freedom to address many of our design goals. This section details many of the decisions we made and the capabilities afforded by those decisions.

## 4.1 Frontend Platform

As Rust developers, we originally designed and planned for a Rust-native gui library like Druid [6] however as time went by, we rapidly realized that unfortunately, the ecosystem is not ready yet. Many packages that we required for our project did not exist and we would have had to create them.

We decided on a web-based frontend that communicates over standard POST requests. This had many unforseen benefits like the creation of a well-designed split between frontend and backend logic as well as creating an async-first architecture which is important for a responsive frontend that might be waiting on computations that take a long time in the backend (like re-running a trace).

The separated networked architecture also enables us to support remote debugging without major ergonomic impacts. This is because the backend and frontend can communicate over the network, allowing future users to debug and troubleshoot issues without being in the same physical location as the application.

In addition, splitting the frontend and backend gave us more flexibility in designing the frontend. We were able to focus on creating a user-friendly and intuitive interface without worrying about the underlying logic and functionality of the application.

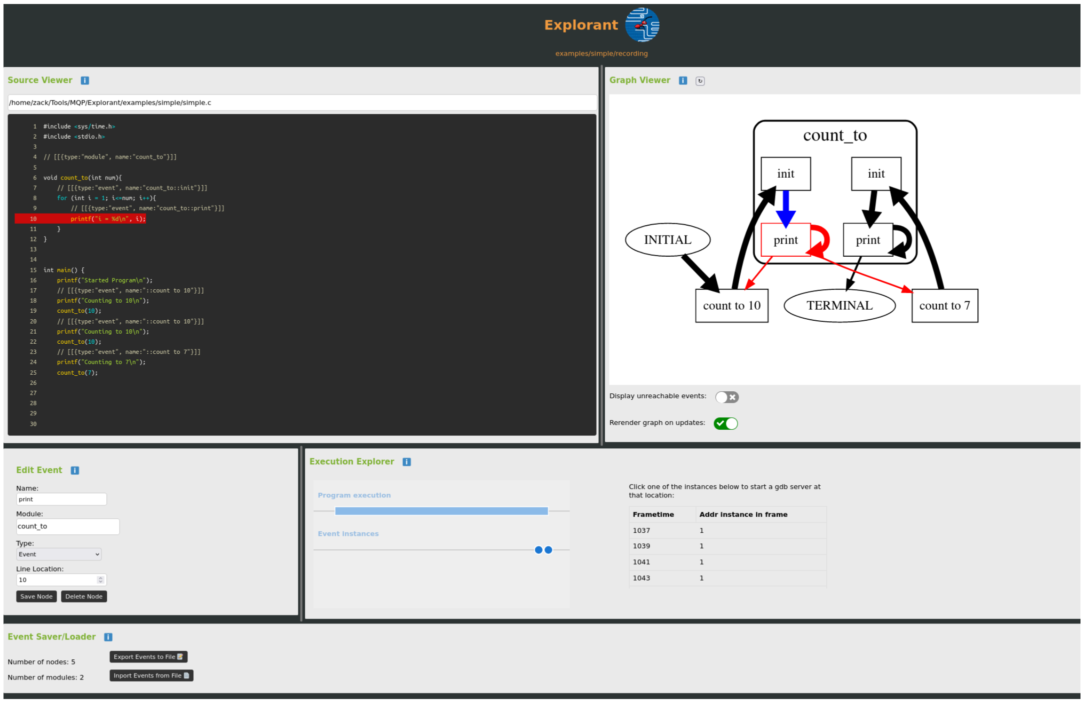
However, this architecture also has some drawbacks. It significantly increased the overall complexity and overhead of the project by adding a new language to the language and a whole new build infastructure for that code. It also introduces more dependencies for the project, which can potentially cause issues. Overall, while the benefits of a separated architecture are significant, given the chance, we think it would still be better to rewrite to a native-first application later on.

## 4.2 Events

One of the core ideas we employed when designing Explorant were events. Events are similar to states in a FSM however they are slightly more basic and less assuming. A real state in a FSM represents a unique global state of the program, though we do not have enough information to create such states without a much deeper understanding of the code. As such, we limit ourselves to discussing events. Each event can be effectively though of as a breakpoint (though they are not implemented that way. See section 6). As the developer adds events on various lines throughout the program, we are able to create diagrams that relate these events and allow a developer to rapidly explore a graph.

## 4.3 Major components of the UX

In this section, we will delve into the real implementation of major components of the software in order to provide concrete examples of how we solved many of the issues we encountered during our case study. Figure 4.1 shows the completed user interface that is actively examining a trace. In this figure, we can see how we have selected the `print` event and how all of the components around the graph have updated to show information about that event.



/home/zack/Tools/MQP/Explorant/examples/simple/simple.c

```
#include <sys/time.h>
#include <stdio.h>

// [[{type:"module", name:"count_to"}]]

void count_to(int num){
    // [[{type:"event", name:"count_to::init"}]]
    for (int i = 1; i<=num; i++){
        // [[{type:"event", name:"count_to::print"}]]
        printf("i = %d\n", i);
    }
}


int main() {
    printf("Started Program\n");
    // [[{type:"event", name:"::count to 10"}]]
    printf("Counting to 10\n");
    count_to(10);
    // [[{type:"event", name:"::count to 10"}]]
    printf("Counting to 10\n");
    count_to(10);
    // [[{type:"event", name:"::count to 7"}]]
    printf("Counting to 7\n");
    count_to(7);
```

Display unreachable events: ☒

Rerender graph on updates: ✓

Name: print

Module: count_to

Type: Event

Line Location: 10

Click one of the instances below to start a gdb server at that location:

| Frametime | Addr instance in frame |
|---|---|
| 1037 | 1 |
| 1039 | 1 |
| 1041 | 1 |
| 1043 | 1 |

Number of nodes: 5

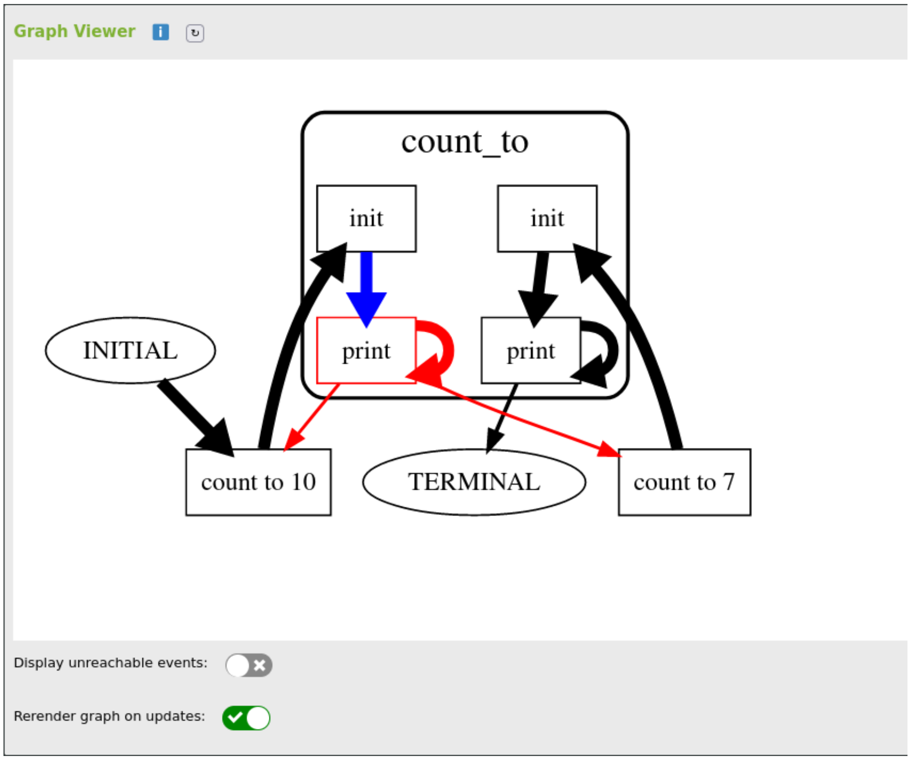
Number of modules: 2

Figure 4.1: Image of the UI for Explorant

### 4.3.1 Graph Viewer

One of the most important components in Explorant is the graph viewer. The graph viewer describes the relationships between events across the entire execution of a trace. The graph viewer tries to convey as much information as possible. Some of the most important of these techniques are:

- The currently selected event is highlighted and all of its incoming and exiting nodes are colored.
- The size of the edges are colored based on how probable that edge is to be traversed.
- Events are grouped into modules (Described in section 5.2)



Display unreachable events: ☒

Rerender graph on updates: ✓

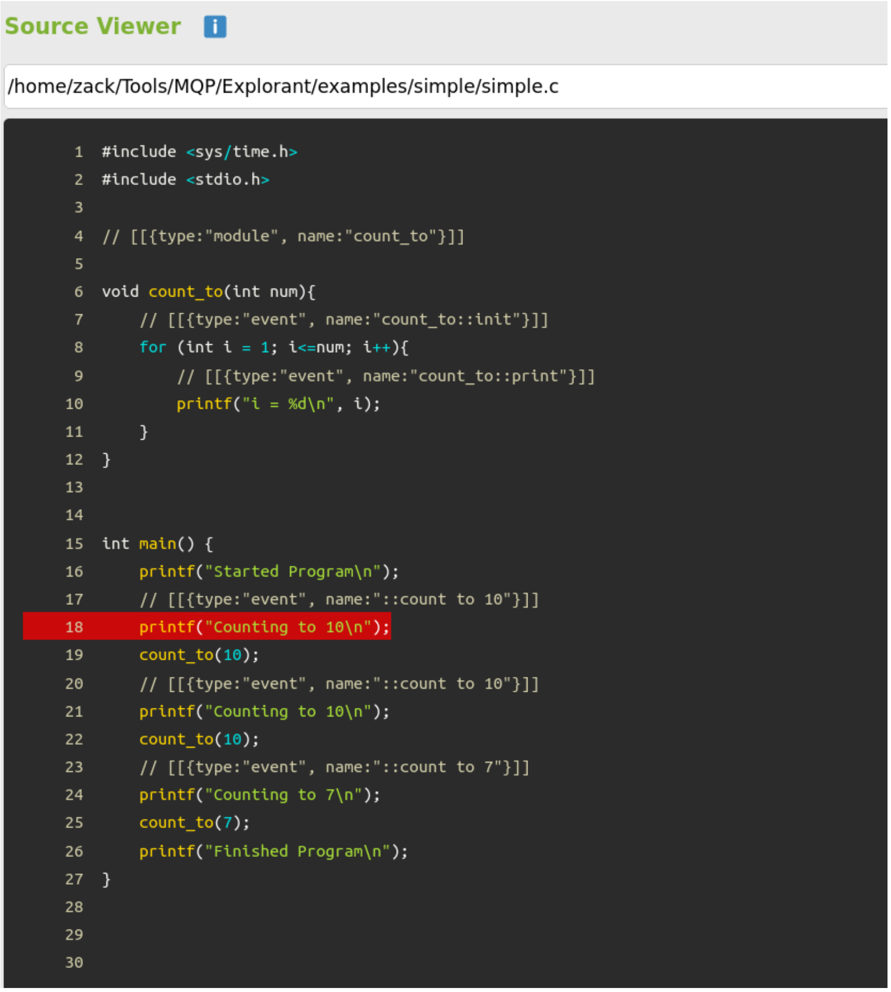
Figure 4.2: Image of the graph visualizer

- Unreachable (Un-run) events can be toggled to only show the happy path that was actually executed during the trace.
- Hovering over an event shows what function it was defined in

You can see examples of these techniques in figure 4.2 including the highlighted node `print`, the module `count_to`, the varying sized edges, and the toggled unreachable events.

## 4.4 Source Viewer

Another critical design criteria is that it must be easy for the developer to relate the high-level design to the source code easily. We addressed this by envisioning a simple source code viewer that contains features like syntax highlighting while also highlighting the line with the currently selected event in red. This allows developers to easily move back and



/home/zack/Tools/MQP/Explorant/examples/simple/simple.c

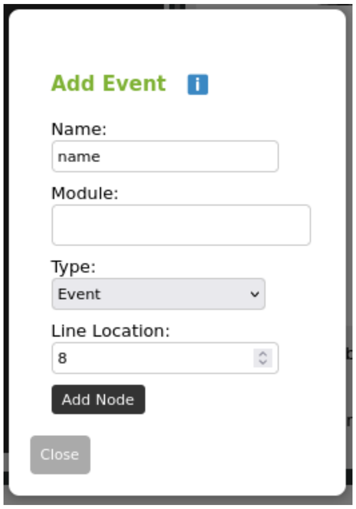
```c
#include <sys/time.h>
#include <stdio.h>

// [[{type:"module", name:"count_to"}]]

void count_to(int num){
    // [[{type:"event", name:"count_to::init"}]]
    for (int i = 1; i<=num; i++){
        // [[{type:"event", name:"count_to::print"}]]
        printf("i = %d\n", i);
    }
}


int main() {
    printf("Started Program\n");
    // [[{type:"event", name:"::count to 10"}]]
    printf("Counting to 10\n");
    count_to(10);
    // [[{type:"event", name:"::count to 10"}]]
    printf("Counting to 10\n");
    count_to(10);
    // [[{type:"event", name:"::count to 7"}]]
    printf("Counting to 7\n");
    count_to(7);
    printf("Finished Program\n");
}
```

Figure 4.3: Image of the source viewing component

forth between source code and the graph. The source viewer also can be right clicked to allow the developer to add new events to the graph in real time. You can see figure 4.3 to see how this was implemented.

## 4.5 Event Adding

Our case study showed that it is critical that the developer is to be able to add new events after a trace has already been recorded. This ensures that the developer has a tight feedback loop that does not entail recompiling and rerunning a program every time they want to experiment and add to the graph. This component allows the user to define


Add Event

Name: name

Module:

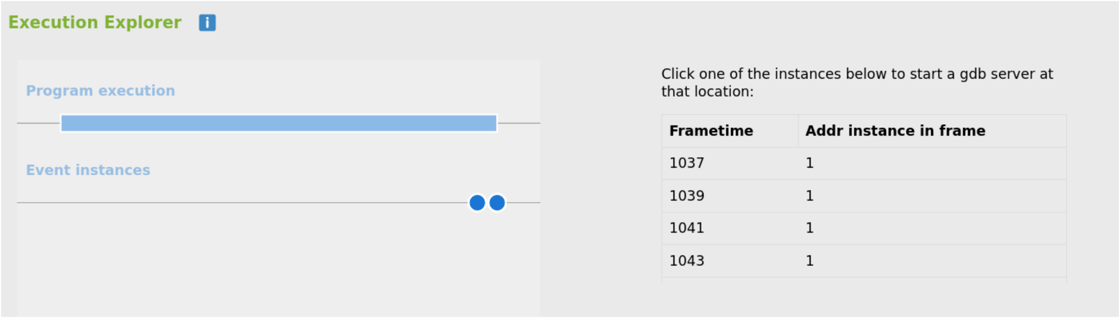
Type: Event

Line Location: 8

Figure 4.4: Image of the event adding component

events, determine what module they reside in, and what line they pertain to. See figure 4.4 to see how it was actually implemented.

## 4.6 Execution Explorer


Execution Explorer

Click one of the instances below to start a gdb server at that location:

| Frametime | Addr instance in frame |
|---|---|
| 1037 | 1 |
| 1039 | 1 |
| 1041 | 1 |
| 1043 | 1 |

Figure 4.5: Image of the execution explorer component

The last and perhaps most important constraint we considered when designing Explorant was the ability to dive deep when neccessary. A debugger like GDB is capable of providing the developer with the means to get a very close look at a particular moment in time. As such, we allow the developer to see a list of every time the event was reached, when it occured relative to the start of the program, and if they click on the particular instance of the event, it opens up gdb at that exact location. You can see figure 4.5 to see an image of how this was implemented. Note the timeline which shows all of the times the event was reached (with blue dots). The timeline also stacks some of these events because they occur so close together in time.

## 4.7 Design Limitations

This design has a few major limitations that are worth examining:

**JIT compilation / self modifying code** JITs [7] do not expose a standard method to access their location in the source code in the same way that static compilers provide DWARF data. This means that it is currently not feasible to instrument an arbitrary JIT compiler or code that it is running. Unfortuately, this means that many common languages like Java, Javascript, and Python will not work with Explorant.

**Macro heavy code** Macros hide a lot of information that DWARF data is unable to process as the macro is expanded in many locations and does not translate well.

**Long running programs** Because we must run the whole trace every time we rebuild the graph (in case an address was executed in a spot we didn't expect), working with long running programs is difficult and painful. We could address this by either allowing the user to only analyze a certain time-range within the trace (which is equivalent to just ignoring the problem), or we could employ a much more advanced strategy where we instrument all function calls and then build heatmaps for where a new event could have been run and then only rerun those time segments. In either case, the current design and does not allow for efficient manipulation of long living programs.

**Optimization levels** If a program is compiled with high optimization levels (like O3 in gcc [8]) then functions can be inlined, loops can be unrolled, and the DWARF data becomes harder to parse, and every line is no longer guaranteed to have assembly instructions associated with it. As such, this design means that the user must be sure to compile the program without optimization. This is particularly important because some programs like glibc cannot be compiled without optimization (glibc requires at least O2), meaning that sometimes annotations inside of `malloc` do not behave as we expect them to.

**Browser dependent** The frontend is rendered in a browser. While this enables a fast development cycle, it also means that the final UI is much slower and heavier than a native app.

# 5 Graph Simplification

Once we started generating graphs, we noticed that the graphs we generated were nearly unusable for all but the simplest programs. These graphs were rendered as giant nests of events, with each event having many edges in different parts of the codebase. To improve the clarity of the graphs, we implemented a FSM miner, a module system, and node-grouping techniques to simplify them.

## 5.1 Synoptic

The first tool we used to simplify the graph was Synoptic, a finite state machine (FSM) miner. Developed by the University of Washington, Synoptic is described in their paper ”Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models” [9]. Synoptic accepts a series of events that it uses to refine the graph. First, it creates a compact model where each node exists only once and is connected to all of the nodes that directly followed or preceeded it. Then Synoptic and then mines invariants from the graph. In this context, an invariant is a statement such as ”x is always followed by y”, ”x always precedes y”, or ”x is never followed by y”. By mining invariants, Synoptic is able to refine the graph and seperate nodes that are otherwise abiguous into multiple copies of the same node that represent different execution paths. This results in graphs with many more nodes but with each node follows a clear and unique execution path. Consider the graph in figure 4.2 where you can see how synoptic was able to break apart `print` and `init` in `count_to` into two seperate sets of nodes depending on whether or not they were called from `count to 10` or `count to 7`

## 5.2 Modules

Secondly, we employed a module system for events in order to achieve greater organization and simplicity. This system allows for the creation of namespaces, similar to popular programming languages. For example, instead of naming two ”entry” points of functions as func1_entry and func2_entry, we can define a module with a unique name and a single parent, such as ”add” being a child of ”util”. In this case, the entry point of ”add” can be specified as ”add::entry” and the entry point of another function like ”print” can be specified as ”print::entry”.

This might look like the following in code:

```
#include <stdio.h>
// [[{type:"module", name:"util"}]]
// [[{type:"module", name:"print", parent_module:"util"}]]
// [[{type:"module", name:"add", parent_module:"util"}]]

int increment(int a){
    // Define a start event that gets expanded into ::util::add::entry
    // [[{type:"event", name:"add::entry"}]]
    a = a+1;
    return a;
}
void print_num(int a){
    // Define a start event that gets expanded into ::util::print::entry
    // [[{type:"event", name:"print::entry"}]]
    printf("val: %d\n", a);
}


int main(){
    // [[{type:"event", name:"::entry"}]]
    int a = increment(2);
    print_num(a);
    // [[{type:"event", name:"::exiting after printing the num"}]]
    return 0;
}
```

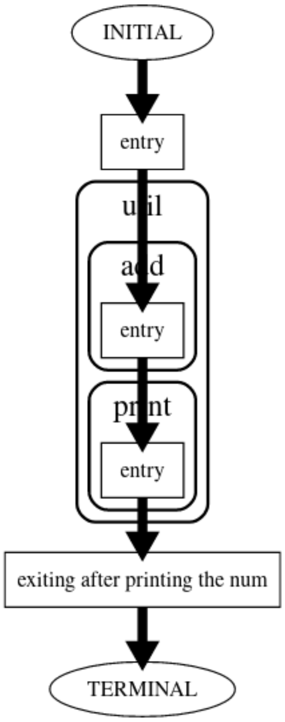


Figure 5.1: Graph of simple module system

As you can see in figure 5.1, this sample code generated a simple graph containing multiple entry nodes all grouped into multiple different modules. This example is very contrived however it demonstrates multiple aspects of the module system. We can also see in figure 5.2 how these modules can be collapsed to simplify and hide complexity in the graph

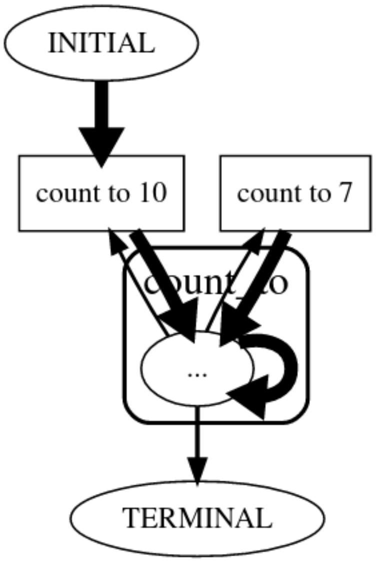


Figure 5.2: A collapsed module for a different program

We recommend keeping the number of modules to a minimum in order to prevent complicated and difficult-to-understand graphs. With many modules, graphviz is more constrained during graph layout and this can cause the creation of many extra extremely long edges. However, careful placement of a few modules can make the graph much more legible and easily navigable.

## 5.3 Grouping Strictly Sequential Nodes

Finally, we used the idea of grouping nodes that are strictly sequential to simplify the graphs in our project. We define strictly sequential nodes A and B as having the only edge out of A directed to B and the only edge into B coming from A. Similarly, a set of nodes (A,B,C) is strictly sequential if A and B are strictly sequential and B and C are also strictly sequential. This technique is particularly useful for Synoptic graph simplification because

Synoptic produces a large number of nodes and this helps to constrain and visually separate the graph. These groups are automatically highlighted to the user with a dotted border. We created figure 5.3 to show the effect of sequential grouping. These are two photos of the same graph, one with grouping enabled, and the other with it disabled. In this case, the grouping allows the user to quickly see the relationships between nodes that were otherwise hidden due to how they were displayed in the image on the left.

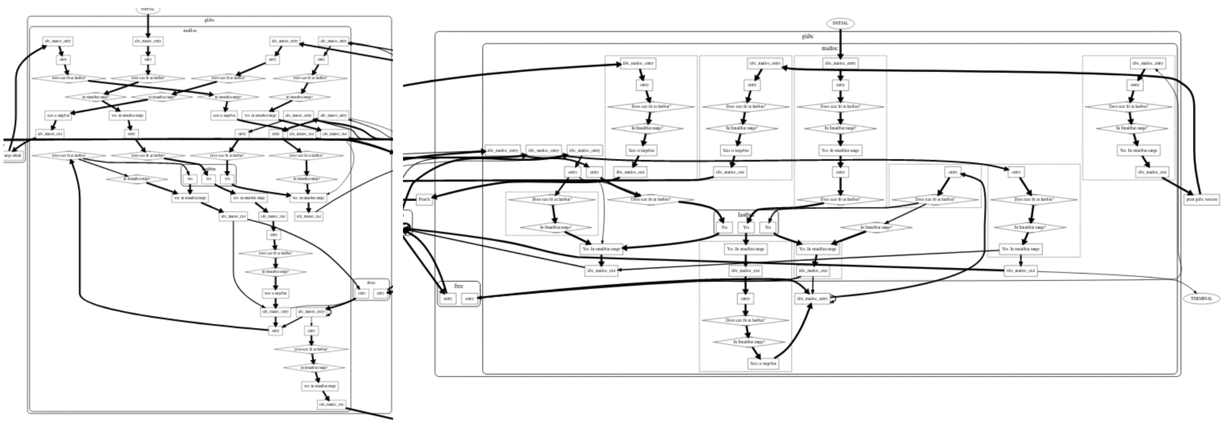


Figure 5.3: Off/On comparison showing strictly sequential grouping

# 6 Efficient Address Recording

To understand the flow of a program, we need to record the path that it takes. For statically compiled binaries, this is equivalent to recording the locations in memory where the CPU is executing instructions. One way to determine this path is to set ”software breakpoints” at various addresses and then continue until the CPU hits one of these breakpoints.

However, when we began our analysis of librr, we quickly noticed that singlestepping and continuing (interrupting the process) incurs a heavy performance penalty. Early measurements indicated overhead on the order of $100\mu s$ per software breakpoint. This resulted in a 100-1000x performance penalty depending on the workload. This is unacceptable for anything but the simplest of programs. As a result, we decided to use a more complicated technique called code stomping to force the underlying program to store its location in the program without using software breakpoints.

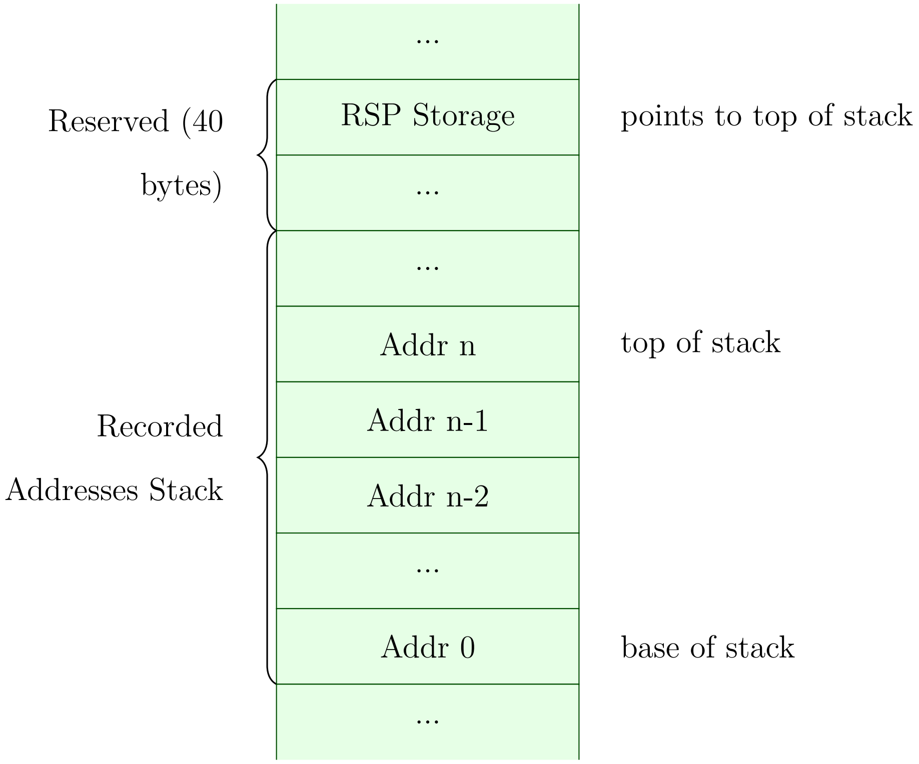
## 6.1 Naive Implementation

To have the program record its own location, we created two new memory regions: the address-segment and the trampoline-segment.

The trampoline segment stores code that acts as a recorder for the addresses that are reached. The address-segment acts as the space that the code in the trampoline-segment can use to store the instrumented addresses. To instrument an address, we simply replace the instruction at that address with a jump to an entry in the trampoline-segment.

The address-segment looks like the following:

### 6.1.1 Address-segment diagram



### 6.1.2 Sample trampoline-segment entry

```
{instruction that was stomped}

XCHG rsp (beginning the address-segment)

PUSH (address of instruction that was skipped)

XCHG rsp (beginning of the address-segment)

JMP (address to go to next instruction in the main program)
```

### 6.1.3 Considerations

1. This has no protection against overflowing the stack in the address-segment

2. You can only place trampolines on instructions that are 5+ bytes

3. Yon can only place trampolines on instructions that do not alter program flow (for now)

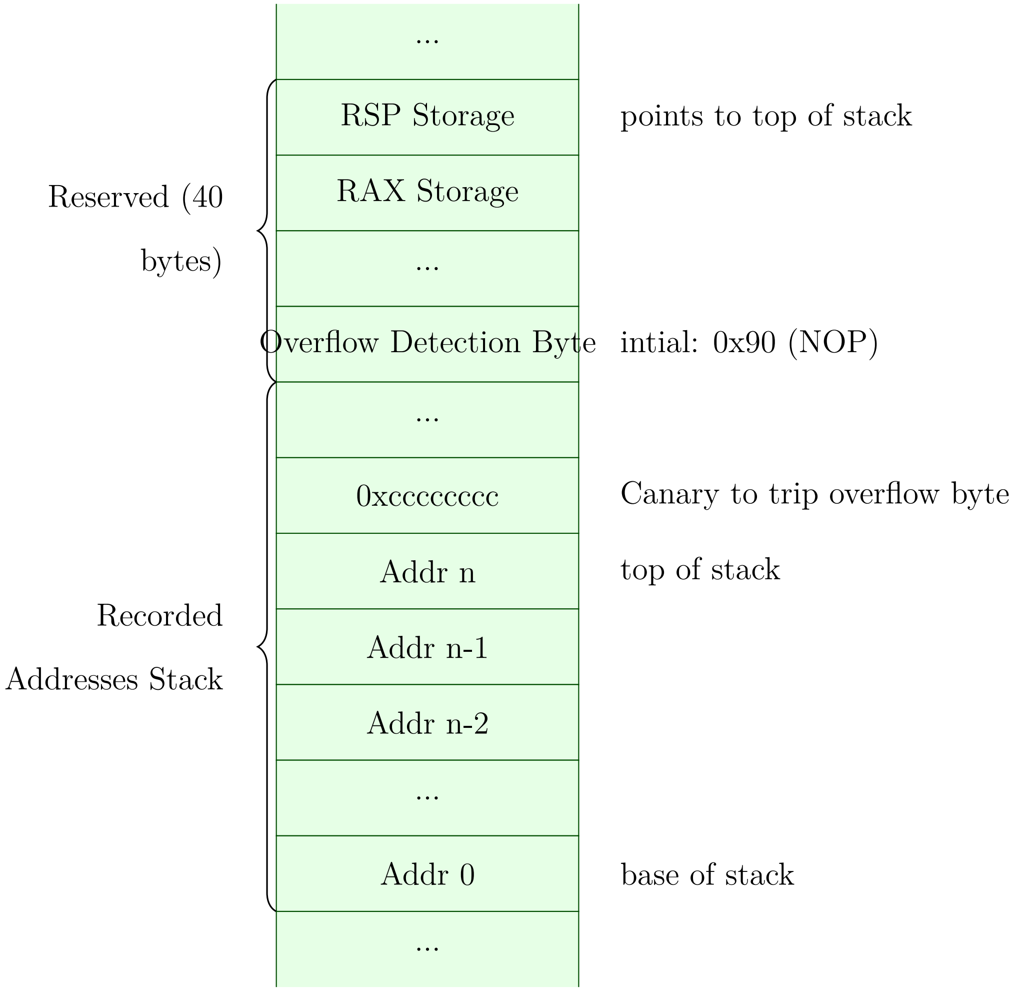
## 6.2 Stack overflow protection

One problem with this implementation is that it wastes stack space and is risky because if the stack overflows, the user will get a segfault that is difficult to debug. Therefore, we also developed another implementation that includes stack overflow protection. This implementation uses a special byte that is overwritten whenever the stack grows into the reserved space, triggering a software breakpoint and an interrupt that allows the main program to reset the stack.

Key pieces of information:

- No code I write here can increment the x86 retired branch counter as otherwise there will be diversions from the recording. This means that I cannot just compare RSP with some value and then interrupt.
- The INT3 (0xcc) instruction triggers an interrupt that the CPU and kernel use to alert the debugger that a software breakpoint was reached.
- The NOP (0x90) instruction is the same size as INT3 (1 byte)

### 6.2.1 Address-segment diagram



### 6.2.2 Sample trampoline-segment entry

```
{instruction that was stomped}

XCHG rsp (beginning of the address-segment)

XCHG rax (beginning of the address-segment + 8)

PUSH (address of instruction that was skipped)

PUSH 0xcccccccc

POP  <- subtracts from RSP without clearing the memory in front of it

MOV AL (overflow detection byte)
MOV (RIP+1) AL

NOP  <- instruction that will be overwritten by overflow detection byte

XCHG rsp (beginning of the address-segment)

XCHG rax (beginning of the address-segment + 8)

JMP (address to go to next instruction in the main program)
```

### 6.2.3 Considerations

1. This requires the trampoline segments to have more than twice as many instructions
2. You can only place trampolines on instructions that are 5+ bytes
3. Yon can only place trampolines on instructions that do not alter program flow (for now)

## 6.3 Final Implementation

After some experimentation, we determined that it was easiest to simply use the naive implementation and give the address-segment a large (256Mib) buffer. We then clear this stack every time a new ”frametime” event triggers, which happens on average every $40,000,000$ instructions [10, Scheduler.h:72]. This allows us to process on the order of $10^9$ instructions per second (at peak thoretical throughput with no saving overhread). We have not conducted in depth benchmarks on this because this system has completelly eliminated the problem of address recording speed.

# 7 Conclusions and Future Work

In this paper, we presented Explorant, a novel onboarding and code exploration tool. While we were not able to conduct an in depth analysis of how users would use our tool, we clearly see how this tool addresses many of the concerns with current methods while incorporating the other tools (like gdb and the source code) directly inside of Explorant.

As time goes on, we hope that Explorant will continue to develop and address some of its main weaknesses. While many of the limitations addressed in the Limitations 4.7 section cannot be fixed (like JIT support), a number of them can. The areas that we think are most important for future work include:

- The ability to drag around nodes and add custom edges and labels within the graph so that the graphs can serve as documentation
- Fixing difficultiies with long running executions
- Developing file-based segmentation techniques
- Possible automatic event definition based on some criteria from the trace that contain the fewest nodes but capture the most important flow paths.

We strongly believe that this tool could serve as a critical resource not only for the new developers understanding a codebase, but also for the senior engineers who can add Explorant annotations and have the junior engineers explore on their own. We hope that others will embrace and extend Explorant to realize the next generation of code exploration.

# Appendices

## .1 Appendix A Title

Input materials for Appendix A. Works the same as regular text, just do not include captions or labels on any tables or figures. Appendices can be referenced in text the same way you reference figures or tables, using the label.

## .2 Appendix B Title

Input materials for Appendix B

# Bibliography

[1] L. Gren, "A fourth explanation to brooks' law - the aspect of group developmental psychology," *CoRR*, vol. abs/1904.02472, 2019. [Online]. Available: http://arxiv.org/abs/1904.02472

[2] L. Pradel, "Quantifying the ramp-up problem in software projects," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2915970.2915975

[3] rr. [Online]. Available: https://rr-project.org/

[4] D. P. Bovet, "Implementing virtual system calls," 2014. [Online]. Available: https://lwn.net/Articles/615809/

[5] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability extended technical report," 2017. [Online]. Available: https://arxiv.org/pdf/1705.05937.pdf

[6] R. Levien. (2022) Druid. [Online]. Available: https://docs.rs/druid/latest/druid/

[7] L. Clark. (2017, 2) A crash course in just-in-time (jit) compilers. [Online]. Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

[8] (2022) Options that control optimization. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[9] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Szeged, Hungary, Sep. 2011, pp. 267–277.

[10] Z. Porter, "librr." [Online]. Available: https://github.com/zaporter/rr/tree/f036bbc37fb63efea9ae99d82051efccf18d68df/src