Explorant: A Code 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:

Prof. Robert J. Walls
Prof. Gary Pollice

Date: 2022-12-16

Abstract

Gaining a deep understanding of large codebases is difficult and time-consuming. In this report, we present Explorant, a novel trace-based code exploration tool. Explorant allows developers to quickly create state diagrams of other people's code to visualize how major components of the program fit together. It does this without sacrificing the in-depth analysis capabilities of traditional tools by incorporating debuggers and sourcecode viewers. Explorant creates tight experimentation-driven feedback loops that enable new ways of reading and quickly understanding 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 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

Brooks'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 a new developer you impose a 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 standard first-week meetings with mentors drawing diagrams on whiteboards while the new developer rapidly takes notes. The new developer is then subject to multiple weeks wherein they are paranoid that they have misunderstood the stack or that some part of the stack was described wrong as they add new code. Our research shows that it often takes developers between three and nine months until they are fully ramped up [3]. We use the term *onboarding* to describe this multi-month process of becoming an expert in an existing codebase.

In this report, we present Explorant, a code exploration tool designed to enhance and parallelize the onboarding experience. Explorant allows a developer to quickly understand and relate different components of a codebase based on the execution of a trace. With Explorant, senior engineers can annotate their code to produce diagrams that the new engineer can use as a stepping stone for further exploration. We achieve this by integrating state diagrams, source code, RR [4], and GDB [5] into a single cohesive tool that enables users to seamlessly delve into specific code paths or gain a broad overview of the entire program.

2 Background

To understand the details of how Explorant works, we will provide context on certain aspects of the stack that are particularly important for understanding its functionality. Namely, we describe how debuggers help with code understanding, what a state diagram is, and how RR [4] works.

2.1 Debuggers and Breakpoints

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. GDB [5] is one of the most widely known debuggers for statically compiled languages (i.e. not interpreted languages like Javascript). GDB allows a developer to step through a program line by line to see the exact execution path. A developer normally uses a tool like GDB when they know a little bit about how a program works but they are looking to solidify their knowledge and dive deeper.

One of the most important things debuggers can do is set *software-breakpoints*. These work by replacing the first byte of an instruction with 0xcc, also known as `INT3` [6]. When this instruction is executed, the CPU throws an `EXCEPTION_BREAKPOINT` which calls the debugger's exception handler. This allows the debugger to fix the first byte of the instruction and modify the instruction pointer register to act as if the instruction had never been run. At this point, the debugger has full access to the program's memory and the developer can do things like examine memory addresses and data structures. Explorant utilizes this technique to allow the developer to open GDB at any specific instance in time.

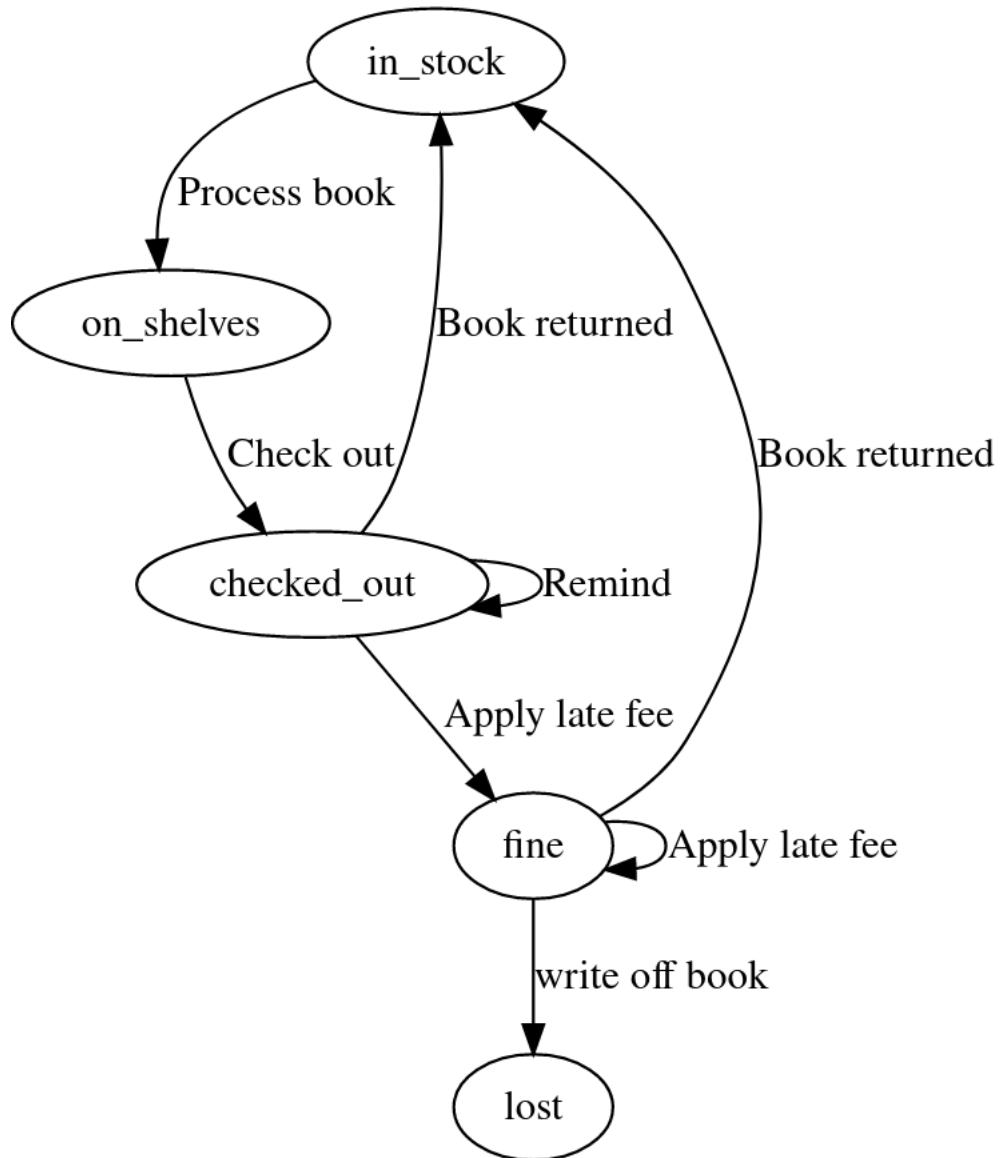## 2.2 State Machines and Diagrams

Figure 2.1: Simple state diagram of a library bookkeeping system

State diagrams are directed graphs that help programmers condense lots of compli-cated domain knowledge into a small number of states. These diagrams show how the states of a program relate to each other [7]. 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, 7].

Each node in a state diagram represents a unique state of the program and each edge represents an action that can occur to transition states. For example, in figure 2.1, a book in the `checked_out` state can be returned, the person can be reminded, or the person can get a late fee.

Finite state machines (FSMs) are visually similar to state diagrams but they apply the node transitions with more rigor [8, p 55]. FSMs specify all of the conditions that lead to a transition and have no concept of global state. They are a very limited concept of computation. Explorant does not directly generate FSMs however it uses graph mining techniques to estimate what an FSM of a given program could look like (See section 4.7.1).

Explorant does not use traditional state diagrams as we do not have the ability to intelligently label edges. All of the state diagrams that Explorant can generate are unlabeled (this may change in the future). However, using techniques described in the graph simplification (4.7) section, we are still able to ensure that the graph is both understandable and usable.

2.3 ELF / DWARF

The ELF (Executable and Linkable Format)[9] is a file format used by many Unix-like operating systems to specify the layout of object files and executables. These files contain a 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)[10], which is a data format that specifies the layout of debugging information. DWARF data is embedded in ELF files and can be accessed by debuggers, such as GDB, to

4

Attribute	Description
DW_AT_external	Whether the function was defined in this program (ex: printf is external)
DW_AT_name	The name of the subprogram
DW_AT_decl_file	The path to the souce file where this function was defined
DW_AT_decl_line	Line in the source file where this function was defined
DW_AT_decl_column	Column in the source file where this function was defined
DW_AT_type	Return type of the function
DW_AT_low_pc	The starting address of the subprogram
DW_AT_high_pc	The ending address of the subprogram
DW_AT_prototyped	Specifies whether the subprogram has a prototype

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

provide valuable information about the execution of a program.

DWARF data is organized into structures called DIEs (Debugging Information Entries), which are composed of key value pairs for the structures in the original source file. Some common DIE tags include `DW_TAG_variable` for variables, `DW_TAG_subprogram` for functions, and `DW_TAG_compile_unit` for compilation units. You can see table 2.1 to see all of the information that goes into one of these DIE entries for a function.

By parsing out these DIE tags, Explorant can access valuable information about the symbols in a program, allowing us to provide useful features such as the ability to correlate lines of code with addresses and figure out which function a particular line is a part of.

2.4 RR Record and Replay Framework

Explorant uses RR [4] extensively and would not be possible without it. The RR record and replay framework is a powerful extension to GDB that enables the user to *continue backwards* as well as forwards while debugging [4]. This allows user 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 occurring in the first place.
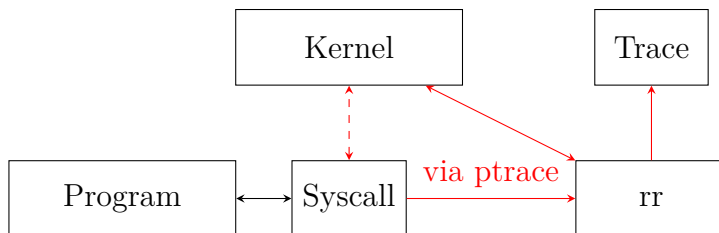
Figure 2.2: Diagram showing how RR intercepts calls into the kernel and records them

One key aspect of RR is its focus on capturing nondeterministic input rather than the whole trace [4]. In this case, nondeterministic inputs are the events that could cause a program's output to vary from run to run. This includes things like syscall inputs and outputs, 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. RR uses a wide variety of tools including the `ptrace` API [11] to attach middleware, overwriting the vDSO (virtual dynamic shared object), limiting a program to a single core, and other more specific techniques in order to accomplish record and replay.

To intercept many simple system calls, RR can simply overwrite the vDSO which is a user-space code-segment that the kernel exports for code "that does not necessarily have to run in kernel space" [12]. However some code can directly execute syscalls in assembly. As such, "when the tracee makes a system call, RR is notified via a ptrace trap and it tries to rewrite the system-call instruction to call into [their] interception library" [13, p. 8]. A diagram of this modification can be seen in figure 2.1. A similar process happens during replay and all of the nondeterministic calls are replaced with lookups to get the deterministic input from the recording.

In order to efficiently *continue-backwards*, RR utilizes a checkpointing system. The checkpointing system works by `fork`ing the process to cheaply copy the address space [13, 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" [13, p. 15].

This allows RR to quickly and easily restore the program to the state it was in at the time of the checkpoint. Then it can continue-forwards until it reaches the desired location.

The default interface to RR is a GDB server using the GDB remote serial protocol [14]. 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.

3 Understanding Code Onboarding

In order to design Explorant as well as possible, we conducted a case study wherein we spent a week studying and understanding a code base while recording our observations about our onboarding process. The codebase that we chose was the glibc [15] memory allocator, commonly referred to by its most used function, `malloc`. `malloc` uses a wide array of complicated datatypes, intricate macros, and C-style memory optimizations. The goal of this case study was to use only the codebase itself and no video lectures or online explanations. This was done so that we might arrive at a better understanding of what it is like to onboard with codebases that have less documentation. However, we will use many detailed source code comments in `malloc` as a stand-in for internal documentation or a mentor who knows the codebase well.

To tackle our `malloc` case study, we first researched tools for code onboarding. Almost all resources we could find online recommended the same advice: read the source code (with tools that provide features like jumping to definitions and collapsing modules), write some simple code to explore the behavior of the underlying system, step through the code with a debugger while keeping notes and building diagrams, and repeat. [16, 2]. This strategy worked well for us during our case study.

From our study, we made the following observations:

1. 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 had to set many breakpoints. This allowed us to gain an in-depth understanding of a few functions however it did not always help with the larger picture. We also found that GDB showed large amounts of information

that was mentally taxing to wade through and not very useful (like error handling code).

2. `malloc`'s comments are extremely detailed and any changes that are made to `malloc` will require a great deal of attention to the documentation that is scattered around the source code. We didn't find any instances of incorrect comments but we noticed how certain data structures and ideas were referenced in comments throughout the program, yet they were only defined in a single location. We can easily envision how documentation that is separate from the implementation could become outdated and be hard to maintain.

3. Manually writing down a lot of the internals of `malloc` was tiresome and distracting. This custom documentation was very useful to have because we referenced it later, though it interrupted our flow of thought and we found that we often wrote down many inconsequential implementation details that were unimportant.

4. 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.

5. After having completed our exploration, we thought we had a fairly deep understanding of `malloc`. However, once we started describing certain high-level interactions, we realized that we had misunderstood how multiple key components related to each other. By diving deep into certain areas, we had convinced ourselves that we understood the whole codebase at a similar depth which was clearly wrong.

Most importantly, this case study validated our suspicion that there is room to improve this time-consuming and laborious process.

4 Design

Our experiment with `malloc` and the experience we gained laid the foundation for the design of Explorant. Specifically, our experiment showed us the importance of building a temporal map that relates different segments of the codebase. When we were working with `malloc`, this map was created in our heads and on paper, however, Explorant was designed to help translate from important locations in the code to a readable state diagram of the program. To do this, we built a system that works on events organized inside modules.

4.1   Events

Explorant uses Events as a core building block. Events are similar to states in a FSM (Section 2.2) 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 thought of as a breakpoint (though they are not implemented that way. See section 5). 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 trace in the form of a graph.

We envision that these events can serve as a form of accurate documentation that senior engineers can easily provide to junior engineers. We did this by allowing there to be two ways to define events: the first is through adding a source code annotation like:

```
int main(){
    // [[{type:"event", name:"::entry"}]]
```

```
    printf("Hi!")

    return 0;

}
```

The other option is to have the person who is exploring save all of their events to a JSON file which stays separate from the codebase but can easily be imported and exported from Explorant to allow for different profiles or investigation paths within the same codebase.

4.2   Modules

To achieve greater organization and categorization, we developed a module system. 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;
```

11

```
    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;

}
```

As you can see in figure 4.1, 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 4.2 how these modules can be collapsed to simplify and hide complexity in the graph.

We recommend keeping the number of modules to a minimum to prevent complicated and difficult-to-understand graphs. With many modules, our layout engine (Graphviz [17]) 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.
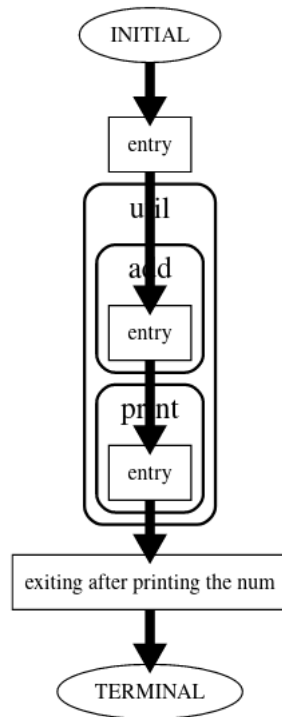
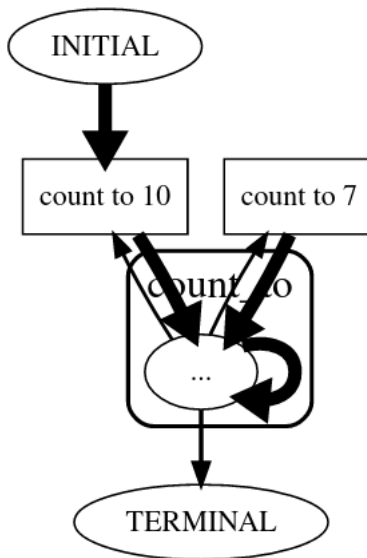Figure 4.1: Graph of simple module system

Figure 4.2: A collapsed module for a different program

13

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 4.3 shows the completed user interface that is actively examining a trace. In this figure, we can see that we have selected the `print` event and all of the components have been updated to highlight the event and show information about that event.
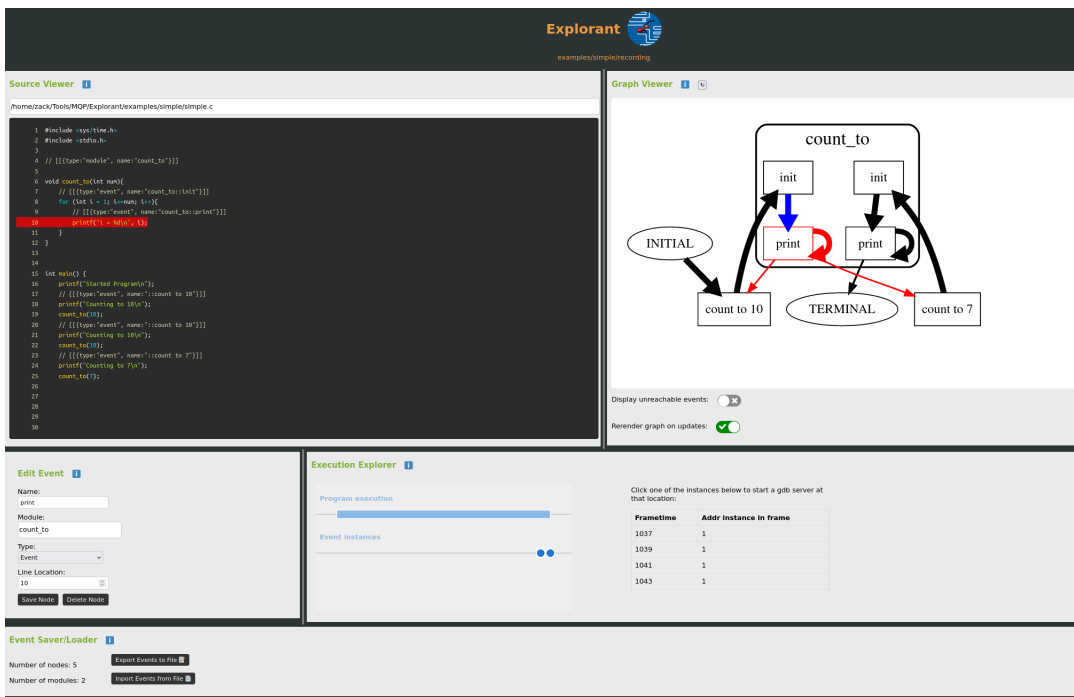

Figure 4.3: Image of the UI for Explorant

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:
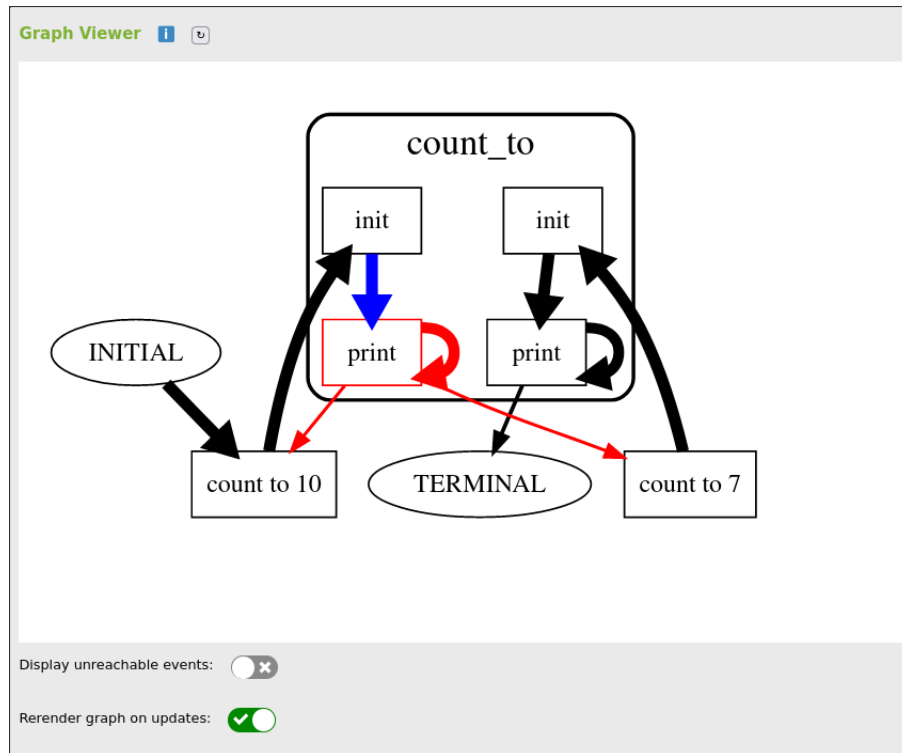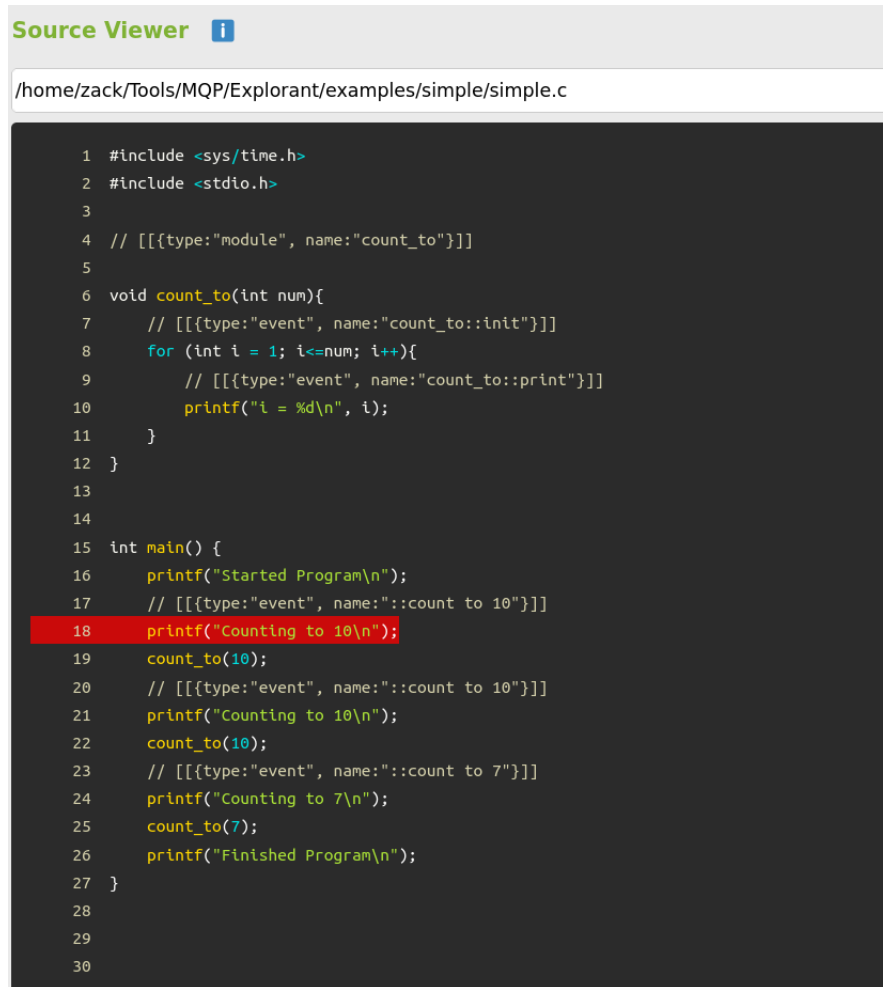
14

Figure 4.4: Image of the graph visualizer

- The currently selected event is highlighted and all of its incoming and exiting nodes are colored.

- The size of the edges are determined based on how probable that edge is to be traversed.

- Events are grouped into modules (described in section 4.2)

- 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 4.4 including the highlighted node `print`, the module `count_to`, the varying sized edges, and program's entry and exit nodes, `INTIAL` and `TERMINAL`.

4.4   Source Viewer

Figure 4.5: Image of the source viewing component

Another critical design choice we made was to ensure that it is easy for the developer to relate the high-level state diagram to the source code easily. We addressed this by envisioning a simple source code viewer that contains features like syntax highlighting while also coloring the line with the currently selected event in red. This allows developers to easily move back and forth between the 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 4.5 to see how this was implemented.

4.5 Event Adding
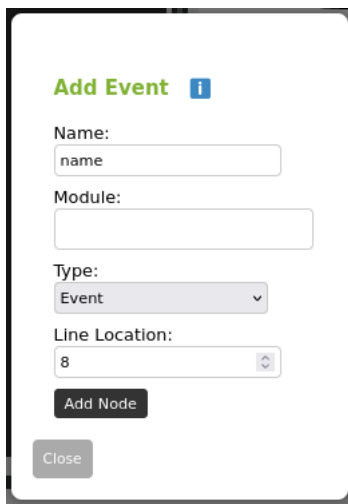
Figure 4.6: Image of the event adding component

Our case study helped highlight how important real-time feedback and experimentation was to the onboarding experience. To ensure Explorant offered this kind of tight feedback loop, we allow the developer to add new events after a trace has already been recorded. We can leverage RR to replay the trace as if the new event had been there the whole time. 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. Figure 4.6 shows this component and how it allows the user to define events, determine what module they reside in, and what line they pertain to.

4.6 Execution Explorer

The last and perhaps most important constraint we considered when designing the UX of Explorant was the ability to dive deep when necessary. 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 an event was
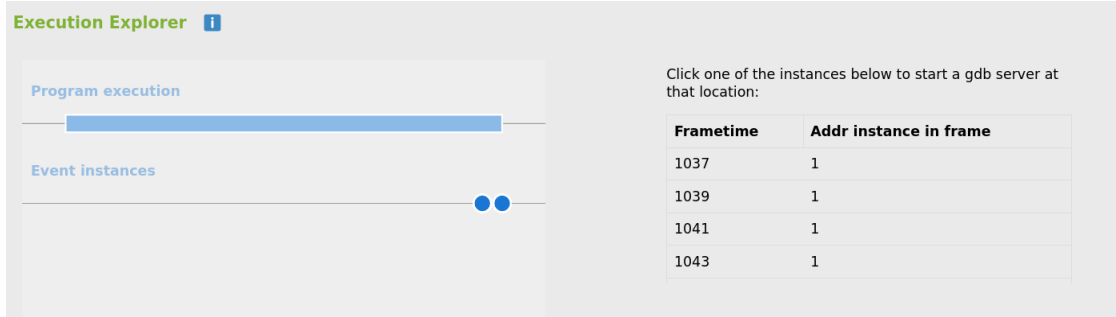
Figure 4.7: Image of the execution explorer component

reached and when it occurred 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 4.7 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 Graph Generation

Even with this GUI, A naive Explorant Event -> Graph Node mapping did not create understandable graphs. As such, we applied multiple filters to simplify the graphs for the developer. Without these filters, the graphs were rendered as giant nests of events, with each event having many edges in different parts of the codebase. The methods we employed were: a FSM miner (see section 2.2), a module system (see section 4.2), and node-grouping techniques to simplify them.

4.7.1 Synoptic

The primary tool we leveraged to simplify the graph was Synoptic [18], a finite state machine (FSM) miner. Developed by the University of Washington, Synoptic is described in their paper "Leveraging Existing Instrumentation to Automatically Infer

18

Invariant-Constrained Models" [19]. 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 preceded it. Then Synoptic 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 separate nodes that are used for multiple purposes (like `printf`) into multiple copies of the same node that represent different execution paths. This results in graphs with many more nodes but each node follows a clear and unique execution path. Consider the graph in figure 4.4 where you can see how synoptic was able to break apart `print` and `init` in `count_to` into two separate sets of nodes depending on whether or not they were called from `count to 10` or `count to 7`

4.7.2   Grouping Strictly Sequential Nodes

Grouping nodes that are strictly sequential also significantly simplified the graphs Explorant built. 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 drawn with a dotted border. We created figure 4.8 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.
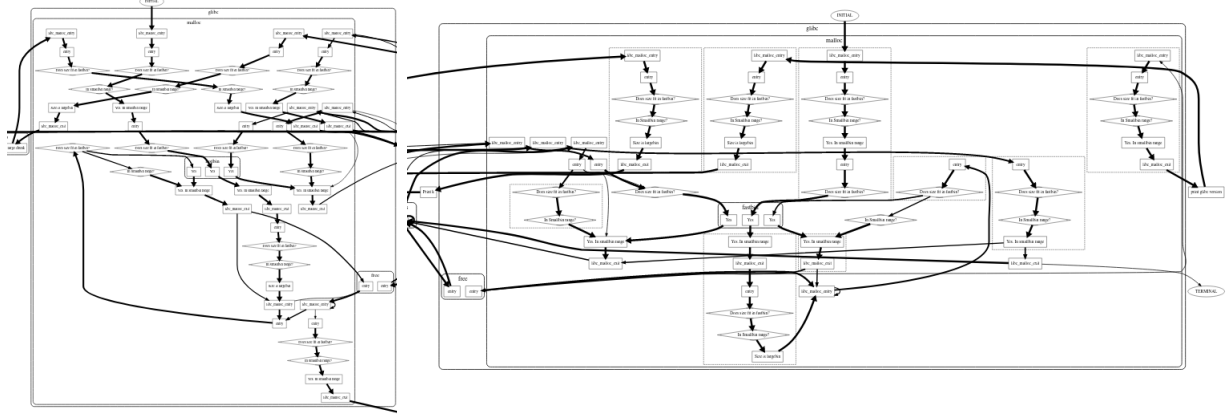
Figure 4.8: Off/On comparison showing strictly sequential grouping

4.8 Design Limitations

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

JIT compilation / self-modifying code JITs [20] 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. Unfortunately, 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 always translate to code very 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 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 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 [21]) then functions can be inlined, loops can be unrolled, 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 [22]), meaning that sometimes annotations inside of `malloc` do not behave as we expect them to.

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

5 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 (See section 2.1).

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 $25 - 100\mu s$ 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.

5.1   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:

Figure 5.1: Naive efficient address recording stack layout

5.1.1 Address-segment diagram

5.1.2 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)
```

5.1.3 Limitations

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)

5.2   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 instructions can increment the x86 retired branch counter as otherwise there will be diversions from the recording. This means that we 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.

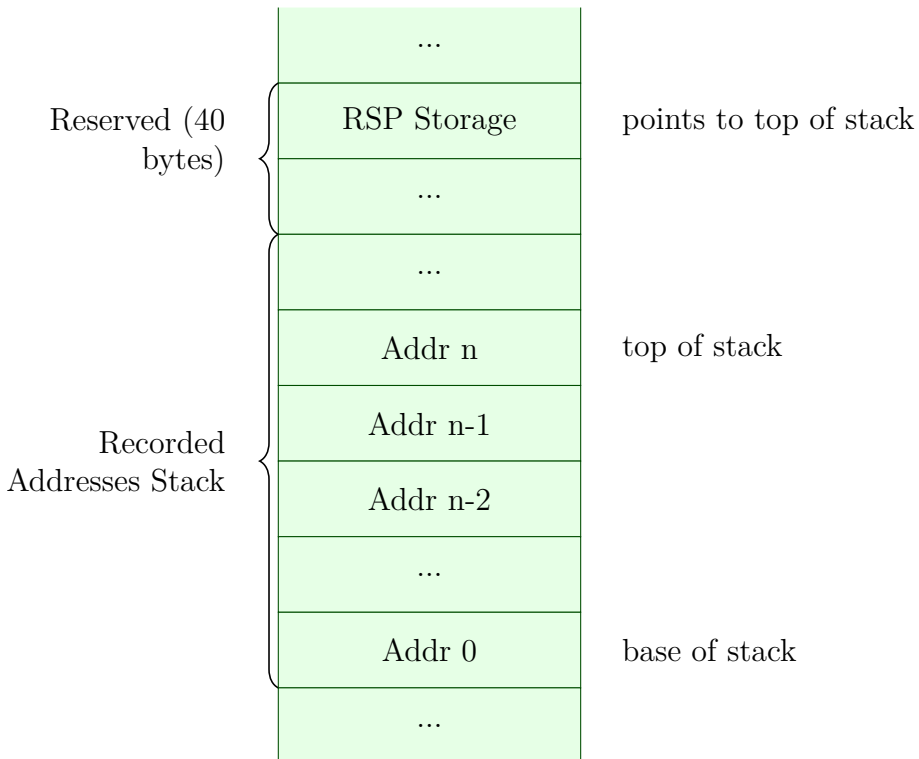- The NOP (0x90) instruction is the same size as INT3 (1 byte)

5.2.1   Address-segment diagram

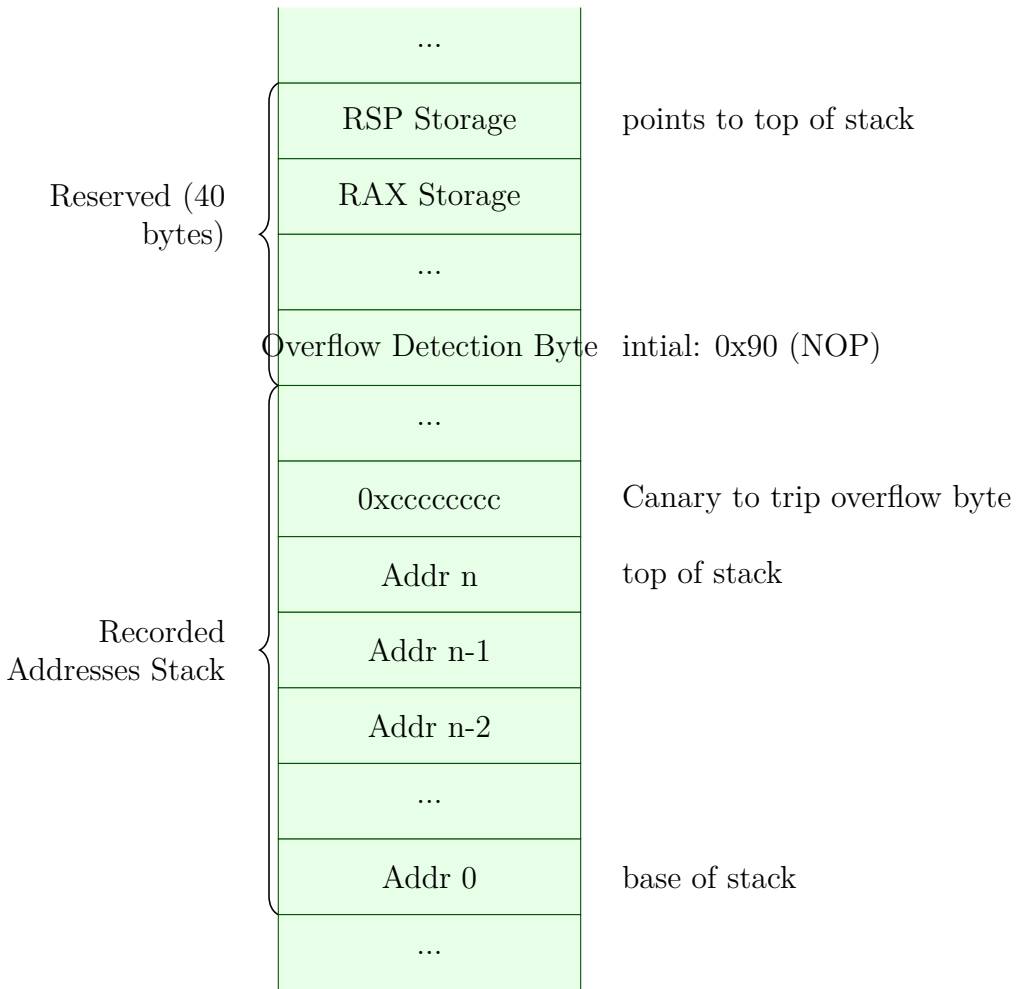Figure 5.2: Naive efficient address recording stack layout

5.2.2 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)
```

5.2.3   Limitations

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)

5.3   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 [23, Scheduler.h:72]. This allows us to process on the order of 10^9 instructions per second (at peak theoretical throughput with no saving overhead). We have not conducted in-depth benchmarks on this because this system has completely eliminated the problem of address recording speed.

6 Conclusions and Future Work

In this report, we presented Explorant, a novel onboarding and code exploration tool. Explorant was designed to improve many of the issues that we encountered in our case study of `malloc` and we think that it has succeeded at its mission and more.

As time goes on, we hope that Explorant will continue to develop and address some of its main weaknesses. While many of the limitations brought up in section 4.8 cannot be fixed (like JIT support), a number of them can. The areas we think are ripe for future work are:

- A user study evaluating the effectiveness of this tool and its usefulness

- The ability to compare multiple traces at once and build a combined graph from all of them.

- The ability to drag around nodes and add custom edges and labels within the graph so that the graphs can serve as official documentation

- Fixing difficultiies with long running executions using complex function call heatmaps or simply limiting the execution to a smaller range (perhaps both)

- Automaic segmentation of events based off of filesystem hierarchies rather than modules. We think this would be particularly useful for codebases with a large number of disjoint files.

- 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 this tool can serve as a critical resource not only for new

developers understanding a codebase, but also for senior engineers who can add Explorant annotations and have the junior engineers explore on their own. We will continue supporting Explorant and we hope others embrace and extend our work to realize the next generation of code exploration.

Appendices

A Frontend

We spent a significant amount of time designing the user interaction for our application. We considered multiple platforms such as a GDB or Ghidra[24] 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.

As Rust developers, we originally designed and planned for a Rust-native GUI library like Druid[25] 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.

As such, we decided on a web-based frontend that communicates over standard POST requests. This had many unforeseen 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 project and a whole new build infrastructure 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.

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] T. A. Dror, 10 2022. [Online]. Available: https://hackernoon.com/engineer-onboarding-the-ugly-truth-about-ramp-up-time-7e323t9j

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

[5] Gdb: The gnu project debugger. [Online]. Available: https://www.sourceware.org/gdb/

[6] Anti-debug: Assembly instructions. [Online]. Available: https://anti-debug.checkpoint.com/techniques/assembly.html

[7] M. M. Fleck, "Chapter 19: State diagrams," 2021. [Online]. Available: https://mfleck.cs.illinois.edu/building-blocks/version-1.3/state-diagrams.pdf

[8] M. Ben-Ari and F. Mondada, *Finite State Machines*, 01 2018, pp. 55–61. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-62533-1_4

[9] *Tool Interface Standard (TIS) Portable Formats Specification*, 1993. [Online]. Available: https://refspecs.linuxfoundation.org/elf/TIS1.1.pdf

[10] (2022) The dwarf debugging standard. [Online]. Available: https://dwarfstd.org/Home.php

[11] (2021) ptrace(2) — linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man2/ptrace.2.html

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

[13] 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

[14] Gdb remote serial protocol. [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html

[15] "The gnu c library (glibc)," 2022. [Online]. Available: https://www.gnu.org/software/libc/

[16] A. Ju, H. Sajnani, S. Kelly, and K. Herzig, "A case study of onboarding in software teams: Tasks and strategies," *CoRR*, vol. abs/2103.05055, 2021. [Online]. Available: https://arxiv.org/abs/2103.05055

[17] Graphviz. [Online]. Available: https://graphviz.org/

[18] "Synoptic," 2022. [Online]. Available: https://github.com/ModelInference/synoptic

[19] 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.

[20] 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/

[21] (2022) Options that control optimization. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[22] Frequently asked questions about the gnu c library. [Online]. Available: https://sourceware.org/glibc/wiki/FAQ

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

[24] "Ghidra." [Online]. Available: https://ghidra-sre.org/

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