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

Type abstract here.

Second paragraph of abstract

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 housemates and their support and to my family for my 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

Chapter 1

Introduction

1.1 Todo

Chapter 2

Background

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 exam-

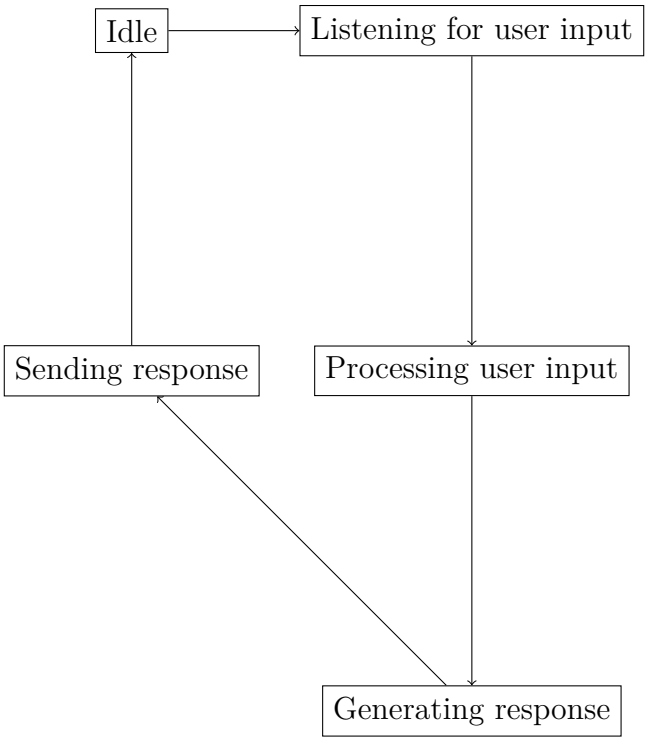


Figure 2.1: Simple state diagram for a bot

ple, 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.

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.

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.

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 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 pars-

ing 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 [1]. 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" [2] 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" [3, p. 8]. A diagram of this modification can be seen in figure 2.2.

(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

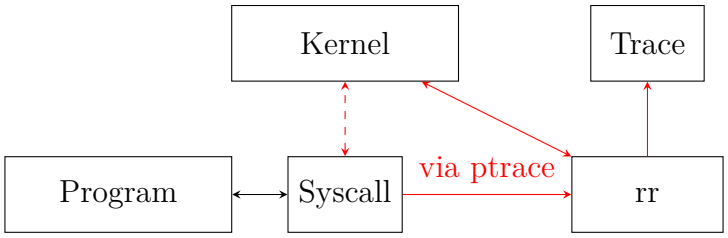


Figure 2.2: Diagram showing how rr intercepts calls into the kernel and records them

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 checkpointing system works by

`fork`

ing the process to cheaply copy the address space [3, 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." [3, 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.

Chapter 3

Design

We spent a significant amount of time designing the user interaction for our application. Initially, we planned to build a gdb plugin that would allow the user to view the state machine graph from the terminal and add annotations directly from the terminal. However, after attempting to write simple gdb plugins, we concluded that it was outside the scope of this project. Therefore, we chose to develop a more GUI-focused application that allows the user to interact with the graph.

To create a responsive and interactive frontend for our application, we decided to split it into a HTTP server backend and a frontend architecture. This decision was not related to the eventual choice of a web-based interface, but rather to simplify the backend. This avoided the need for a message queuing or threadpool system, which would have been necessary in a single unified system to prevent the frontend from freezing whenever the user clicked anything.

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 us 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 increases the overall complexity and overhead of the project, as there are now two separate components that need to be maintained and developed. It also introduces more dependencies for the project, which can potentially cause issues. Overall, while the benefits of a separated architecture are significant, it is important to carefully consider the potential drawbacks before implementing it in a project.

3.1 Frontend Platform

As a Rust developer, I had always been on the lookout for native libraries that would allow me to create robust, efficient, and visually appealing front-end applications. When I stumbled upon Druid, it seemed like the perfect fit.

Druid promised a react-like component-based approach to front-end development, making it easy to modularize and reuse code. Its API was intuitive and well-documented, and it had a growing community of users.

However, as I started using Druid, I quickly realized that it was not as seamless as I had hoped. While the component-based approach was great for building simple, static interfaces, it lacked the ability to easily integrate with other libraries.

For example, I wanted to use a syntax highlighter in my application, but Druid did not have any built-in support for it. I would have had to write my own syntax highlighter from scratch, which was not something I had the time or expertise to do.

I also wanted to include a graph viewer in my application, but again, Druid did not have any built-in support for it. I ended up writing my own graph viewer, but the result

was not pretty. It was clunky and difficult to use, and it didn't have the same level of polish as other graph viewers I had seen.

I tried other Rust native libraries like iced and gtk, but they had similar limitations. They were great for building simple interfaces, but they lacked the flexibility and integration capabilities of React.

In the end, I reluctantly fell back to using React for my front-end application. While it was not my ideal choice, it allowed me to quickly and easily integrate with other libraries, and it gave me the ability to create a more polished, user-friendly interface.

Chapter 4

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.

4.1 Solution

Chapter 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" [4]. 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.

(TODO image)

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 ”malloc” being a child of ”glibc”. In this case, the entry point of ”free” can be specified as ”free::entry” and the entry point of ”malloc” can be specified as ”malloc::entry”.

This might look like the following in code:

(TODO MAKE THIS A CODE BLOCK)

```
// The following metadata sets up the modules used in this example
// [[{type:"module", name:"glibc"}]]
// [[{type:"module", name:"malloc", parent:"glibc"}]]
// [[{type:"module", name:"free", parent:"glibc"}]]


uptr_t malloc(...){
    // Define a start event that gets expanded into ::glibc::malloc::entry
    // [[{type:"event", name:"malloc::entry"}]]
    ...
}
```

```
void free(...){
    // Define a start event that gets expanded into ::glibc::free::entry
    // [[{type:"event", name:"free::entry"}]]
    ...
}
```

TODO image of a graph created by module system

We recommend keeping the number of modules to a minimum in order to prevent complicated and difficult-to-understand graphs. Careful placement of a few modules, however, can make the graph much more legible. Additionally, modules allow for viewers to click on any module and expand or collapse the nodes within it, providing a high-level overview of complicated programs.

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. You can see an example of these groups below:

TODO graph on / off

Chapter 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.

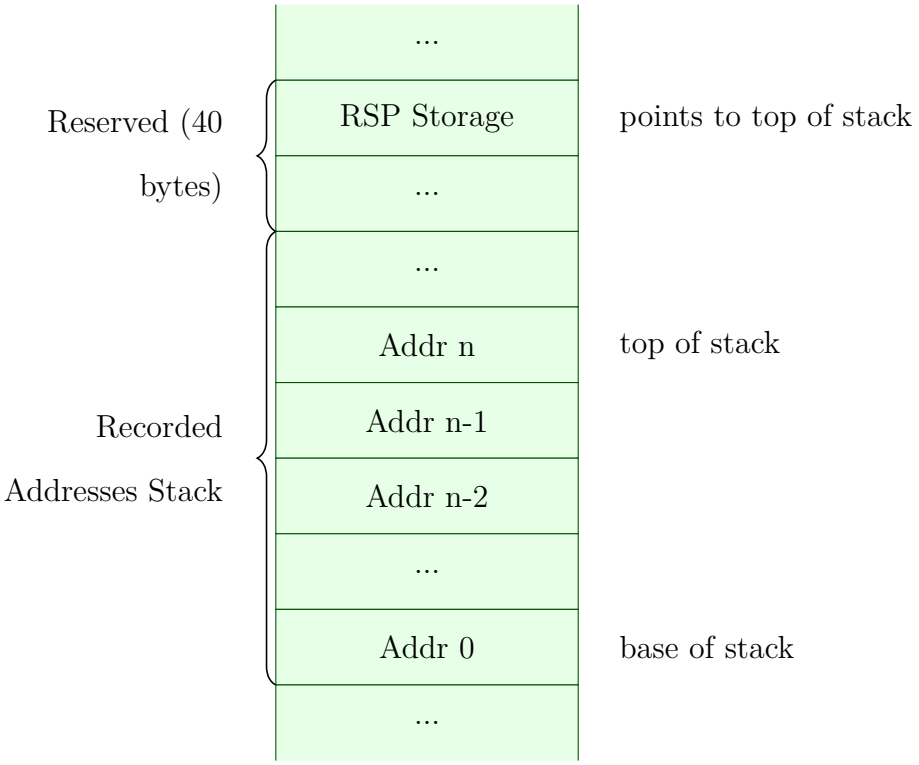
6.1 Naive Implementation

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.

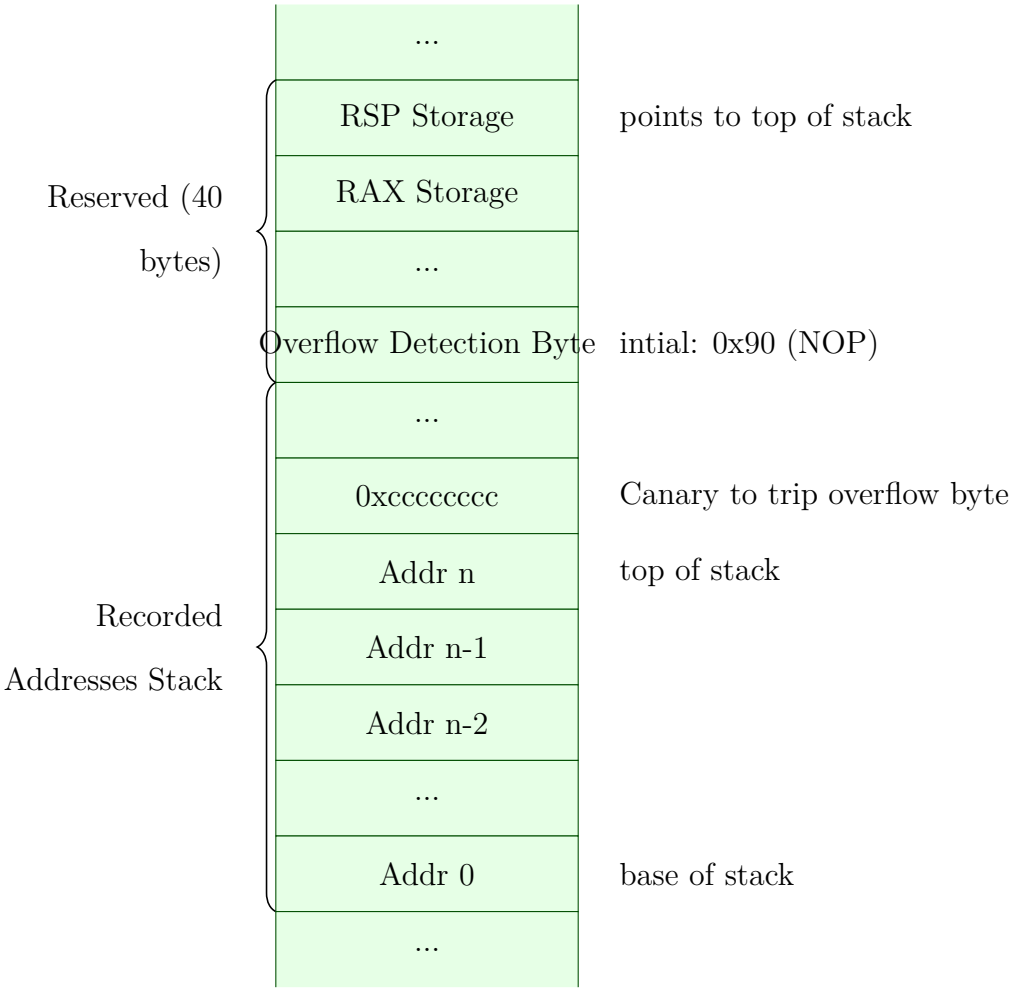
Key pieces of information:

- 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 [5, 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 numbers on this because this system has completelly eliminated the problem of address recording speed.

Chapter 7

Conclusions and Future Work

In this paper, we presented Explorant, a novel onboarding and code exploration tool. It utilizes a number of technical concepts to enable users to explore code without needing to re-run traces. Explorant helps developers gain a comprehensive understanding of a codebase through detailed analysis with gdb and high-level FSM analysis with the graph viewer.

While our target and case study has been malloc, in the end, we but were unable to make the diagrams as simple as we desired. Despite this, we were able to address many of the complaints we had about debugging without Explorant. In particular, Explorant enabled us to easily navigate large code segments, create usable diagrams, and quickly dive into the execution and setup of the program.

Given more time, we believe it would be beneficial to allow users to customize the graph further. This could include options to position nodes and add custom edges and labels. Additionally, restructuring the code to enable analysis on a subset of the whole execution would be helpful, especially for long-running setup-heavy programs.

We believe this tool has the potential to revolutionize programmer onboarding and we hope that other people will embrace and extend Explorant and its ideas.

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] "rr."

[2] D. P. Bovet, "Implementing virtual system calls," 2014.

[3] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability extended technical report," 2017.

[4] 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), pp. 267–277, Sept. 2011.

[5] Z. Porter, "librr."