University of Calgary

PRISM Repository **https://prism.ucalgary.ca**

The Vault Open Theses and Dissertations

2023-12-19

FrAG: Framework for the Analysis of Games

Ganesh, Sankarasubramanian

Ganesh, S. (2023). FrAG: Framework for the Analysis of Games (Master's thesis, University of Calgary, Calgary, Canada). Retrieved from https://prism.ucalgary.ca.

https://hdl.handle.net/1880/117793

Downloaded from PRISM Repository, University of Calgary

UNIVERSITY OF CALGARY

FrAG: Framework for the Analysis of Games

by

Sankarasubramanian Ganesh

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

DECEMBER, 2023

© Sankarasubramanian Ganesh 2023

Abstract

Historical games or retrogames ran on constrained systems that required programmers to use various techniques and optimizations. To learn about these techniques, we often study their source code. However, when the only remaining information about the games is their binary image, conventional analysis methods are time-consuming and do not scale. One way to study these techniques is to reverse engineer the binary images. However, conventional approaches to reverse engineer the images often do not provide accurate results as programmers often blurred the lines between code and data, because unlike most modern platforms, these platforms do not distinguish between code and data. We present FrAG, a Framework for the Analysis of Games that dynamically analyzes games at scale with no human interaction using artificial intelligence. FrAG's design allows it to be ported to other platforms. To demonstrate FrAG's capability and to evaluate its efficacy, we use a test suite of eight Atari 2600 games with ground truth available. We also present a novel way of disassembling game ROMs using data collected from the framework. Furthermore, FrAG can also be used as a platform for training artificial intelligence agents for several platforms.

Acknowledgments

My first and foremost thanks go to my supervisor, John Aycock, for his guidance and support throughout my studies. I am sincerely thankful for the opportunity he provided me to work on this project and for his efforts in helping me develop my writing, teaching, and presentation skills. His expertise, encouragement, and vast breadth of knowledge have been instrumental in both my academic and professional growth.

This work would not have been possible without my parents and brother, to whom I'm indebted for their support and encouragement. Their belief in my abilities has been a source of motivation.

I extend my appreciation to Christopher Jiang for his help in debugging and implementing some features of this project.

I also want to thank Dr. Ryan Henry and Dr. Richard Zhao for agreeing to be part of my committee and providing feedback on my work.

To my parents and my brother.

Table of Contents

Abstract . . . . **ii**

Acknowledgments . . . . **iii**

Dedication . . . . **iv**

Table of Contents . . . . **v**

List of Tables . . . . **viii**

List of Figures . . . . **x**

Acronyms . . . . **xii**

1 Introduction **1**

1.1 Contributions of the Thesis 4

1.2 Overview of this Thesis 5

2 Background and Related Work **6**

2.1 Atari 2600 6

2.2 Traditional Methods of Code Analysis 12

2.3 Fuzzing 13

2.4 Artificial intelligence (AI) in Games and Games in AI Research 15

2.5 Related Work 16

2.6 Summary 18

3 Framework **20**
3.1 Multiple Arcade Machine Emulator 20
3.2 FrAG 21
3.3 Extracting Data 33
3.4 Portability 34
3.5 Limitations 34
3.6 Summary 35
4 Framework Implementation: Atari 2600 . . . . **36**
4.1 Atari 2600 Actions 36
4.2 Atari 2600 Interface 39
4.3 Using Extracted Data 46
4.4 Limitations 47
4.5 Summary 48
5 Evaluation **49**
5.1 Test suite: Rationale 49
5.2 Results 54
5.3 Comparing Learning Methods 76
5.4 Summary 82
6 Future Work and Conclusion **84**
6.1 Future Work 84
6.2 Conclusion 87
Bibliography **90**
Ludography **103**
A Modifications & Additions **105**
B Handlers **143**

B.1 Read Handler 143
B.2 Write Handler 147
B.3 PC Handler 149
B.4 Bank Retrieval 150
C RAM Heatmaps 156
D Coverage Jumps 159
E Training Graphs 164

List of Tables

2.1 Memory Map for the Atari 2600. 11
2.2 Bank-switching types 11

3.1 Languages used in Multiple Arcade Machine Emulator (MAME) 0.252. 22

4.1 Selected Atari 2600 joystick actions. 38
4.2 Selected Atari 2600 paddle actions. 38
4.3 Octal numbers and their interpretation used for tracking addresses. 43

5.1 Test suite summary. 53
5.2 Read only memory (ROM) coverage for each iteration of deep q-network (DQN) model training. 55
5.3 Coverage of *Combat* with different runs. 60
5.4 Number of games played by all four instances using DQN. . . . 60
5.5 Confusion Matrices. 66
5.6 Number of times a bank was accessed. 73
5.7 Bank breakdown in bank switching games. 74
5.8 Number of relative branches that were always taken. 75
5.9 ROM coverage comparing learning methods. 78

5.10 Number of games played by all four instances using DQN (15 M frames). 79
5.11 Reward for coverage jumps shown in Figure 5.10. 81

D.1 Coverage changes for DQN 15 M frames. 163

List of Figures

2.1 Atari 2600 with a joystick. 8

2.2 Atari 2600 joystick and paddle controllers. 8

2.3 Secret room in Atari 2600's Adventure. 14

3.1 High-level overview of MAME's execution flow. 25

3.2 High-level UML class diagram of FrAG. 26

3.3 High-level overview of FrAG's execution flow. 26

3.4 Example of a 4-button controller and its ports and fields. 28

3.5 Example mappings from integer to actions. 29

4.1 All actions using the Atari 2600 joystick and paddle. 39

4.2 An example graph depicting how a RAM location graph is built. 45

4.3 Execution graph for the example code in Listing 4.1 46

5.1 MAME debugger disassembly window. 57

5.2 ROM location covered for *Boxing*, *Combat* and *Dragonfire*. . . . 62

5.3 An example graph of a random access memory (RAM) location cycling between values. 68

5.4 RAM usage heatmaps for *Boxing*. 69

5.5 *Boxing* RAM graphs for locations `$a4` and `$a5`. 70

5.6 *Boxing* RAM graphs for locations `$a6` and `$a7`. 70
5.7 *Boxing* RAM graphs for locations `$a8` and `$a9`. 71
5.8 *Boxing* RAM graphs for locations `$aa` and `$ab`. 71
5.9 Program counter (PC) heatmaps depicting most executed areas in ROM. 77
5.10 Coverage increase as AI trains. 80
5.11 Average reward over session for *Dragonfire*. 82

C.1 RAM Usage Heatmaps for *Combat*. 156
C.2 RAM Usage Heatmaps for *River Raid*. 156
C.3 RAM Usage Heatmaps for *Dragonfire*. 157
C.4 RAM Usage Heatmaps for *Kaboom!* 157
C.5 RAM Usage Heatmaps for *Moonsweeper*. 157
C.6 RAM Usage Heatmaps for *Montezuma's Revenge*. 157
C.7 RAM Usage Heatmaps for *Solaris*. 158

E.1 Average reward over session for *Boxing*. 164
E.2 Average reward over session for *Combat*. 164
E.3 Average reward over session for *River Raid*. 165
E.4 Average reward over session for *Kaboom!* 165
E.5 Average reward over session for *Moonsweeper*. 165
E.6 Average reward over session for *Montezuma's Revenge*. 166
E.7 Average reward over session for *Solaris*. 166

Acronyms

A2C	advantage actor critic
AI	artificial intelligence
ALE	Arcade Learning Environment
CPU	central processing unit
CRT	cathode-ray tube
DDQN	dueling deep q-network
DQN	deep q-network
DSL	domain specific language
MAME	Multiple Arcade Machine Emulator
PC	program counter
PCGML	procedural content generation using machine learning
PIA	peripheral interface adapter
PPO	proximal policy optimization
RAM	random access memory
RIOT	RAM-I/O-Timer
RL	reinforcement learning

ROM	read only memory
TIA	television interface adapter
TV	television

Chapter 1

Introduction

Video games have evolved from entertainment to a global cultural phenomenon, significantly influencing society and technology [1]. Video games, hereafter referred to as games in this thesis, are not only fun but offer a platform for expressing artistic creativity, exploring narratives, and most importantly technological innovation. Games have pushed hardware capabilities to its limits and often influence developments in various industries such as network technology, interaction design and artificial intelligence (AI) [2–6]. Games use innovative user interfaces, including gesture-based controls, motion sensing, and virtual reality to elevate the player's experience. For instance, the Wii remote's motion control was adapted for rehabilitation therapy, allowing patients to perform exercises while playing games [3]. Online multiplayer games have driven advancements in networking, influencing research. Online multiplayer games use efficient algorithms for scalability and load balancing of the servers where millions of players are playing concurrently [2, 4]. This introduces new areas for research such as content distribution networks, network security, and blockchain [4].

The study of video games has emerged as a highly diverse and interdisciplinary field, involving a range of academic disciplines, from the humanities to computer science. In the humanities, the study of video games is often called game studies, involving the study of the cultural and societal context, gender and identity, and narrative in games [7]. In computer science, the study of games involves the technical and computational aspects such as graphics, game design, AI, and development practices. *Archeogaming*, a relatively new multidisciplinary field, explores the intersection of archaeology and video games. Games in this field are treated as digital artefacts which are studied for things such as human-computer interaction in games, design research, cultural and historical impact and the portrayal of actual history in games [8].

Platform studies, another area of study, is an interdisciplinary field that encompasses some elements of computer science but is not a subfield of computer science. It bridges the gap between technical and humanistic perspectives. It includes disciplines such as media studies, cultural studies and history [9]. While computer science primarily focuses on the design, development, and analysis of algorithms, software, and hardware systems, platform studies takes a broader approach. It examines the historical, cultural, and societal aspects of computing platforms and considers how the technical components of platforms influence and are influenced by various human and cultural factors.

Retrogames, that we define as games from the 1980s and are often characterized by pixelated graphics and simple gameplay, form a formative period of time in the history of games. While the games might appear simple, these early games laid the foundation for gaming that we enjoy today

and the hardware limitations of the platform these games were running on spurred creativity and innovation in game programming. The platforms are outdated and not in use, but the development of games for these platforms required programmers to optimize their code and data, a highly relevant skill in modern computer science. As an example of retrogames' relevance in modern computer science, the arcade game *Sea Wolf II* (1978) implemented interpreters, often used to implement programming languages and is an ongoing research area [10].

To study the techniques used in retrogames, we often study the game code. The code does not only show the techniques but also the development culture at the time when information flowed differently and websites such as Stack Overflow and GitHub to share and discuss code did not exist. However, for most games, the only remaining information we have about the game is the binary file. Most retrogames were written in the assembly language of the processor that was used on the platform, which often does not distinguish between code and data unlike most modern platforms. One way to study these techniques is to reverse engineer the binary image; however, as the platforms do not distinguish between code and data, determining whether or not a byte in the binary file was code or data is undecidable [11]. Moreover, games often blurred the line between code and data, where data is executed as code and code is read as data.

Existing techniques to reverse engineer retrogames each have their own advantages and disadvantages. One way to reverse engineer the binary file is by disassembling it without running the game; however, the binary file is nothing but bytes and distinguishing between a byte being code and data is undecidable, as mentioned above. Moreover, obfuscations and opti-

mizations by the programmers make reverse engineering challenging. This method of reverse engineering, called *static analysis*, is faster and scales but does not produce the most accurate results. Another method to reverse engineer and analyze these games is to run the game and collect information on what is executed (code), read and written (data). This method, called *dynamic analysis*, provides the most accurate data but is time-consuming as the game must be run. While this method might be possible for a program without user input dependency, games are the complete opposite. Games depend on user input and logical decisions made by the person playing the game. Hence, this approach often requires a person to play the game and make progress, as some parts of the program might only get executed when a player reaches certain conditions in the game. While we can perform dynamic analysis on a few games, thousands of games may be available for a single platform, and we need tools to help study these games. In this thesis, I present a framework to aid in reverse engineering and analyze many games without any human interaction using AI and without any tuning, treating the AI as a black box.

1.1 Contributions of the Thesis

In this thesis, I have designed and implemented a framework to aid in reverse engineering a binary file of a game and its analysis using AI. The framework can capture information about the execution and memory usage of a program. It is built into an existing emulator and has access to all the devices being emulated for querying additional information. With the data obtained, I introduce a new type of disassembler to disassemble game read-

only memories (ROMs) and detail several analyses. The framework can also train AI agents for several game consoles as a side-effect.

Early versions of this work were presented in Ganesh et al. [12], Michaud et al. [13] and Ganesh, Aycock, and Biittner [14].

1.2 Overview of this Thesis

The remainder of this thesis is organized as follows. Chapter 2 discusses some background needed to understand how and what the framework achieves. It details the Atari 2600 console used to test the framework's efficacy and discusses related work. Chapter 3 and Chapter 4 introduce the emulator that the framework is built into, the modifications made to the emulator, details the design and implementation of the framework and its interface and also discusses some of its limitations and applications. Chapter 4 details explicitly the implementation of the Atari 2600 console in the framework and the analyses performed using the data obtained. It also gives an overview of a new disassembler to disassemble game ROMs from the data obtained using the framework. Chapter 5 describes the test suite of games used to test the framework's efficacy and discusses the results and the analyses performed. It specifically discusses how the data obtained can aid in reverse engineering a game. It also compares different learning methods and discusses their differences and the necessity of a well-playing AI. Finally, Chapter 6 discusses future work and concludes this thesis.

Chapter 2

Background and Related Work

This chapter discusses some of the background needed to understand how the framework I built works and also discusses related work and its pros and cons. While a high-level overview of most of the concepts is sufficient, bank switching has been described in detail, as it is essential to understand how some bank-switching schemes work to understand what the framework achieves and how it achieves it. Before discussing bank switching and related work, we look at the Atari 2600 game console and some of the concepts that build up to code analysis methods.

2.1 Atari 2600

The Atari 2600 (Figure 2.1), initially marketed as the Atari Video Computer System (VCS) until November 1982, is a video game console developed and first released by Atari in 1977 in the US [15]. It was Atari's first cartridge-based console and included two joysticks and two paddle controllers to be plugged into it [16]. The paddle controllers differed from the joystick as they

come with wheels that, when turned, would move the player left or right depending on the direction of rotation, giving more accurate and minute movements than the joysticks, which was useful for games such as *Kaboom!* (1981). Figure 2.2 shows the joystick and paddle. The console was one of the first that was not hardwired to play a fixed set of games and used swappable read only memory (ROM) cartridges, allowing the consumer to play many games. This made it popular for many years until the video game market crashed in the United States during 1983–1985 [17–19]. A newer console version was released in 1986, and the Atari 2600 did not cease production until 1992.

The Atari 2600 is known to have popularized cartridge-based systems and supported ROM cartridges of nearly a thousand games [15, 17, 18, 20]. Even now, games by publishers [21] and several independent game titles produced by hobbyists, commonly referred to as homebrew games, continue to be released [22, p. 87].

2.1.1 Hardware

The Atari 2600 was a straightforward console with limited hardware due to budget constraints and the goal to make it cost-competitive [15]. Instead of the 6502 processor, they opted for the 6507, a cheaper cut-down variant of the 6502 with only 13 address lines compared to the 16 address lines in its parent 6502. However, there were no changes to the assembly language or any of the internal workings, which caused the 6507 to think of itself as a 6502 capable of addressing 64 KiB of memory when in reality, it was only able to directly address 8 KiB [15]. The Atari 2600 only contained the

Figure 2.1: Atari 2600 with a joystick. 1980 version. Credit: Evan Amos, public domain.

(a) Joystick.

(b) Paddle.

Figure 2.2: Atari 2600 joystick and paddle controllers. Credits: Evan Amos, public domain.

following hardware components:

- Processor: MOS 6507
- 6532 RAM-I/O-Timer (RIOT) Chip:
  - Random access memory (RAM): 128 bytes
  - Peripheral interface adapter (PIA) for Input

- Video & Audio Chip: Television interface adapter (TIA)

The processor interfaced with the TIA, which Atari had explicitly developed for the console and only supported cathode-ray tube (CRT) televisions (TVs) [17]. Most systems use a buffer in RAM to store the frame before rendering it on screen. However, Atari took a somewhat different approach, as RAM was expensive at the time. To understand Atari's approach, we must know how a CRT TV works. A CRT TV works by electrically heating a filament in a vacuum inside a glass tube. A "ray" (high velocity electron beam) is generated by an electron gun which strikes the screen coated with phosphor, a substance that lights up when struck by the ray. This results in the screen lighting up. The electron beam travels horizontally, lighting up one line at a time [23, 24]. These lines are also known as scanlines. As there was no framebuffer to store data, programmers had to write code for every scanline in a frame by writing to the TIA in time to display the line on screen as the electron beam swept across the TV [15, 25]. This is also known as "racing the beam" [17].

2.1.2 Overcoming Memory Limitations

As the 6507 could only address a maximum of 8 KiB, the cartridges were limited to a maximum of 4 KiB, because the TIA and RIOT chips used half the address space. The main memory mappings for the console are shown in Table 2.1. However, bank switching changed the game. Bank switching is a method of increasing the addressable memory of a processor to provide more memory [26]. For the Atari 2600, this allowed cartridges to swap one ROM bank for another and allowed cartridges of greater sizes, giving

rise to more sophisticated games. To switch banks, programmers would write code to access specific memory locations, commonly called hotspots, which change banks. The processor, oblivious of this external change in the cartridge, reads from the new bank.

The first Atari 2600 game to use bank switching was *Asteroids* (1979), an arcade game ported by Brad Stewart. The 8 KiB game was published by Atari and used a simple bank-switching technique comprised of two banks which are swapped back and forth when the hotspot is accessed [20]. Most bank-switching games published by Atari used this straightforward method. However, due to Atari owning the rights to these schemes [27, 28], third-party developers had to develop their own bank-switching schemes to avoid legal issues. This gave rise to unique bank-switching schemes. Some cartridges had extra RAM; Atari chose to use a chip marketed as the Super Chip (SC) to include an additional RAM of 128 bytes while third-party developers came up with different techniques to include RAM.

Most bank-switching types change an entire bank of ROM. However, there are a few types which use pages instead. These pages are smaller than banks and can be swapped without swapping the entire bank. For example, the bank-switching type E0 contains eight 1 KiB pages in the cartridge and four 1 KiB banks. The last bank is fixed to the last 1 KiB page of the ROM (8th page); however, the other pages are free to move around in different banks [29]. The 3F type is also similar, but instead of eight 1 KiB pages, there are four 2 KiB pages and two 2 KiB banks. Once again, the last bank is fixed to the last 2 KiB of ROM. The E7 type is one of the more interesting types where we only have one bank, which can either be swapped for 1 KiB of RAM or 2 KiB of ROM and another 1.5 KiB which is fixed to the last 1.5 KiB

of ROM. In between these address ranges lies yet another 1 KiB of RAM. The cartridge has 2 KiB of RAM, of which 1 KiB can be turned off or on [29]. The bank-switching types (excluding homebrew) and relevant information are shown in Table 2.2.[1]

Address	Use
$0000 – $007F	TIA
$0080 – $00FF	RAM
$0200 – $02FF	PIA
$1000 – $1FFF	ROM Cartridge

Table 2.1: Memory Map for the Atari 2600 [30]. Numbers prefixed with a $ are in hexadecimal (base 16).

Type	Developer	Size	RAM	Mapping
F8	Atari	8 KiB	N/A	Two 4 KiB banks
F6	Atari	16 KiB	N/A	Four 4 KiB banks
F4	Atari	32 KiB	N/A	Eight 4 KiB banks
FE	Activision	8 KiB	N/A	Two 4 KiB banks
E0	Parker Brothers	8 KiB	N/A	Four 1 KiB banks, 8 pages
3F	Tigervision	8 KiB	N/A	Two 2 KiB banks, 4 pages
FA	CBS	12 KiB	256 B	Three 4 KiB banks
E7	M-Network	16 KiB	2 KiB	One 2 KiB selectable bank
F0	Megaboy	64 KiB	N/A	Sixteen 4 KiB banks
UA	UA Ltd	8 KiB	N/A	Two 4 KiB banks

Table 2.2: Bank-switching types and information [29].[1]

The framework itself is both an analysis tool and a platform to train artificial intelligence agents. To understand what the framework is capable of and the methods used, it is essential to know about traditional code analysis methods and artificial intelligence playing games.

[1] Several bank switching types and chips used in a few games are not shown in Table 2.2, as the implemented framework currently does not support them.

2.2 Traditional Methods of Code Analysis

Analyzing a program's behaviour is used in various areas of computer science. For example, code analysis is used in security to detect any vulnerabilities that can be fixed before they are exploited and to reverse-engineer programs to find out what a program is doing and how. It is also commonly used to debug code by programmers during development.

From a high level, two methods of analyzing code are static and dynamic. Static analysis is the analysis of a program by examining the code or binary without executing it [31]. A disassembler is a static analysis tool which translates binary code to an assembly language program. This, however, is a problem for systems that do not distinguish between code and data, which is the case for most old platforms. As both instructions (code) and data are in the same address space, it is impossible in general to classify a byte as data or code and is an undecidable problem [11]. Furthermore, unlike most modern systems, data can be executed as code, and code can be read as data. Moreover, code obfuscation makes static analysis challenging. Code obfuscation in retrogames was used for copy protection and could even involve using undocumented instructions to fool a hacker. Moreover, data could also be obfuscated to prevent cheating [10, Ch. 8]. Despite the disadvantages, a static analysis approach considers all possible execution paths and is faster, making it great for large-scale analysis of programs.

Dynamic analysis is the analysis of a program during execution [31]. This approach can determine what is executed, read or written to get accurate results. While this may cover all execution paths for a small, simple program, for large enough programs, it is hard to capture all execution paths.

This is especially true in cases where the program relies on user input to determine the path to take during execution. Using games as an example, part of the game may only get executed once the player reaches a certain level or satisfies certain conditions like power-ups, bonus lives or Easter eggs. In Atari's case, Atari 2600 games did not credit the game's developers which led to developers hiding Easter eggs with their names. One such example is the game *Adventure* (1980), which included a secret room with a hidden message. A screenshot of the message is shown in Figure 2.3. The disadvantage of this dynamic analysis approach is that one needs to interact with the program to cover most possible execution sequences.

Regardless of the methods used to analyze them, most game analyses have been of close reading, a method of analysis focusing on a specific game or a specific aspect of games [32–34]. The analysis techniques often used are manual, time-consuming and do not scale. We need tools to scale; most of them have primarily used static analysis [35], or a combination of both static and dynamic [36, 37]. The dynamic analysis used in previous work [36, 37] is limited and it is only used when a static approach failed to capture information, rather than a primary analysis method, however.

Building on dynamic analysis and how it might be automated, we look at fuzzing, often used to detect bugs in a program.

2.3 Fuzzing

Fuzzing is an automated technique that generates input to observe how the program reacts to it and to check on which inputs the program crashes. It is used to test programs to detect bugs and analyze abnormal behaviour.

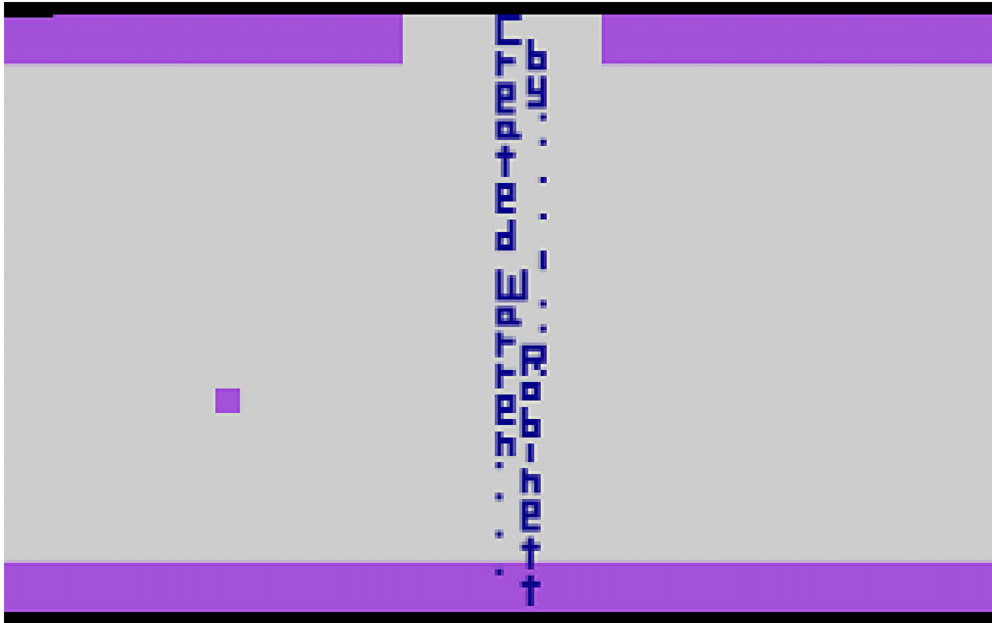


Figure 2.3: Secret room with a hidden message in Atari 2600's Adventure crediting Warren Robinett. Fair dealing.

Overall, there are three types of fuzzing: black, gray and white-box [38]. In black-box fuzzing, the fuzzer does not have access to the source code and generates random inputs without any knowledge of the program. Gray-box fuzzing uses instrumented source code to check how the program reacts and generates input accordingly. This gradually allows the fuzzer to cover more execution paths. Finally, white-box uses analytical methods to systematically cover more execution paths and execute critical code [38–40]. We can leverage black- and white-box fuzzing to capture execution information and use it to reverse-engineer a program. However, it is hard for a fuzzer to cover all execution paths for logical programs such as games. Completely random inputs are rejected at earlier stages of the program and may not execute any critical code [41].

To analyze games dynamically and improve on fuzzing techniques, we

look at artificial intelligence (AI) and research on AI playing games.

2.4 AI in Games and Games in AI Research

Artificial intelligence has extensively been used in games since the 1950s [42]. The traditional use of AI in games was centred around generating intelligent non-player characters (NPCs). However, the use of AI in games is not restricted to this and has several uses such as game levels, data mining and game testing [43]. Work such as Chan et al. [44] has used behaviour testing to train an AI for FIFA-99 that leads to unwanted game states, enabling the testing of large games for undesired behaviour, improving the game's overall experience. The unwanted game states can be any behaviour that the developer finds undesirable, like scoring a goal against a certain player too many times by performing the same actions every time. There also exists work such as Bergdahl et al. [45] that uses deep reinforcement learning to test and find game exploits.

While AI in games and AI research differ, games have also played an important part in AI research. Games have extensively been used as an evaluation method to test reinforcement learning algorithms [5, 6, 46–49], which learn to play a game using different sources such as RAM values and image frame data. Mnih et al. [5] developed AI agents for 57 Atari 2600 games using deep q-network (DQN) and benchmarked whether or not they surpassed human scores. Since then, several AI agents have been developed using different algorithms, such as dueling deep q-network (DDQN), which surpass human scores in most games [50]. In 2020, Agent57, an AI agent for the 57 games developed by DeepMind, outperformed the agent developed

in Mnih et al. [5] in every game [6, 49].

Reinforcement learning (RL) algorithms take actions based on reward. If the action taken is desirable, the agent is rewarded, and thus, games provide a great environment to train RL agents. If the score increases due to an action, the AI is rewarded. The AI makes several random moves at the beginning to determine how the game reacts to specific inputs, similar to black-box fuzzing. Once this is complete, it starts learning to take actions which yield the best results. This is somewhat similar to the guidelines suggested in Bell [41], but the AI does not have access to the source code, nor does it care about the code executed. The main objective of the agent is to get a higher score.

As actions are based on reward, the first few agents, such as the base DQN introduced in Mnih et al. [5], did poorly on strategic games like *Montezuma's Revenge* where the agent did not learn to play the game. However, since then, there have been several studies on developing agents for strategic games [51–53].

2.5 Related Work

There exist frameworks for training AI agents. The most frequently used framework, the Arcade Learning Environment (ALE) [46, 47], is specific to the Atari 2600 and does not contain debugging or analysis features and is not portable to other consoles. Its purpose is to train and benchmark AI agents for the Atari 2600, and it cannot be used to analyze game implementation. Moreover, ALE only supports a single player, although derivative works have built on ALE to allow multiple players [54]. My presented framework,

however, supports any number of players. The only other similar framework uses the Multiple Arcade Machine Emulator (MAME): MAMEToolkit [55]. MAMEToolkit works as a Python wrapper around MAME and is focused on training AI agents. The Python wrapper forks a process and runs MAME in a separate process. It makes use of MAME's Lua console to pass in actions and query data. To obtain frame data and other information required for the AI such as number of lives and terminal conditions, it uses MAME's feature of hooking callback functions on every frame which writes all the requested data such as frame data to a named Linux pipe, from which the Python wrapper reads and stores it in a queue. This makes the communication a bottleneck. In addition, the wrapper does not make any changes to MAME's original source other than adding a function to retrieve frame data. This has both advantages and disadvantages; the advantage is that it is highly portable to newer versions of MAME; however, this also means that if the debugger is enabled, which for getting analysis data is required, MAME opens a debugger window regardless of the no video option and is updated every time a change in the emulation occurs, which impacts performance.

Unlike related frameworks, my presented framework's goal is not to train or benchmark AI agents but to use AI agents to aid in the analysis of game implementation.

There are several works where a single game's implementation has been examined closely and limited works where a large-scale analysis of games from a single console or a creation system has been analyzed [35–37]. However, all large-scale studies conducted so far have been mainly a static analysis approach. These methods are often time-consuming and require significant manual work. Unlike other large-scale analyses, the presented

framework uses reinforcement learning algorithms which surpass humans across all genres of games ([5, 6, 49]) to analyze and extract information from a cycle-accurate emulation system dynamically.

Works such as Kaltman, Osborn, and Aycock [56], Osborn et al. [57], and Osborn, Summerville, and Mateas [58] allow specific aspects of a game to be studied but are limited to one console and are not built for reverse engineering game ROMs. Automated Game Design Learning, a field of research proposed in Osborn, Summerville, and Mateas [59], has a similar pipeline proposed to FrAG; however, there does not seem to be an implementation available.

2.6 Summary

This chapter discussed some of the fundamental knowledge required to understand the framework, what it achieves and how it achieves it. The console that is used to evaluate the framework, its historical significance and bank switching were discussed in the first section. Several bank switching types were developed by game publishers, each unique from one another. While not all the types and chips were discussed, the most used bank switching types were discussed.

The second section discussed traditional code analysis methods and their pros and cons. Static analysis, while fast, does not give good results with old platforms as they do not distinguish between code and data, unlike modern systems. On the other hand, dynamic analysis, while accurate, is time-consuming and does not scale for programs like games where input is logical and user-dependent. Building on program analysis, fuzzing and

AI playing games were discussed, introducing AI agents playing Atari 2600 games.

Finally, related work and some of its shortcomings were discussed. While there exist other frameworks enabling training an AI agent for a game, FrAG is the only one to focus on game implementation analysis rather than benchmarking AI agents, using AI as a tool to provide a platform for analyzing games dynamically.

The next chapter discusses the framework, its implementation, functionality, and limitations.

Chapter 3

Framework

This chapter details the implementation of my FrAG framework and my modifications to the Multiple Arcade Machine Emulator (MAME), an open-source multi-purpose emulation framework into which FrAG is built. FrAG is not bundled with reinforcement learning (RL) algorithms and instead provides a platform for training RL agents and performing game implementation analysis. To help understand why the framework was built into MAME specifically, we briefly look at MAME and some of its capabilities.

3.1 Multiple Arcade Machine Emulator

MAME is a software emulator aiming to preserve software history by documenting a system's hardware and its workings. Its initial focus was to emulate arcade video games; however, over time, it has absorbed its sister project, MESS (Multi Emulator Super System), which emulates various vintage computers, video game consoles, calculators and other types of vintage machines [60, 61]. MAME is written primarily in C++, supports most

operating systems, and at the time of writing, the latest version, 0.252, emulates 45,379 systems [62]. Moreover, MAME prefers emulation accuracy over playability which is desirable for reverse engineering and analysis. With emulating over 45,000 systems comes a massive codebase with several layers of inheritance to provide a general interface for all systems. The source code is objectively large and requires knowledge of various languages and concepts to understand the workings of the emulated systems. Table 3.1 shows the output from `cloc`, an open-source tool which counts the number of lines of code.[1] This data excludes third-party libraries used and build scripts.

The emulation accuracy and features of MAME, such as tracking memory usage, execution tracing, and querying information from emulated devices which the framework uses, make MAME desirable as an emulator to build the framework on.

3.2 FrAG

FrAG is a framework written in C++ that allows one to develop artificial intelligence (AI) agents and provides a platform for game implementation analysis thanks to MAME's extensive debugging features and access to emulated devices. The goal of FrAG is not to develop AI agents but to analyze game implementation, solving issues in traditional game analysis methods as described in §2.2 in Chapter 2. The framework is designed to collect data for analysis or perform analyses dynamically while the AI agent trains or plays a game.

To train an AI agent, information on the state of the game and a legal ac-

[1] https://github.com/AlDanial/cloc

Language	Files	Blank	Comment	Code
C++	7985	840370	778954	3407091
XML	1687	28821	10965	2835233
C/C++ Header	5079	141190	101991	544472
Scheme	147	3101	0	11657
Objective-C++	25	1037	344	4741
Python	11	300	157	1831
SVG	1	5	0	1167
C	31	336	301	1085
Text	6	256	0	859
make	3	126	98	480
HTML	2	218	4	429
CSV	2	0	0	277
Markdown	11	81	0	232
GLSL	8	68	87	188
JSON	1	0	0	166
DOS Batch	1	27	0	163
JavaScript	1	24	51	144
YAML	2	16	27	85
Bourne Shell	3	24	15	78
Visual Studio Solution	1	1	1	63
CSS	1	0	0	14
SUM:	15008	1016001	892995	6810455

Table 3.1: Languages used in MAME 0.252 and its corresponding source lines of code.

tion set for a game is required. This game- and system-specific information is used to control the execution of the game. The following list shows the game- and system-specific information that is required at a minimum for the framework to function:

- System-specific: All valid actions and combinations on a controller for a system. For instance: up, down, right, left, fire, up-left, up-fire. Unless available in the system or game, nonsensical combinations such as up-down are not included as they negate the movement.

- Game-specific: All valid actions and combinations on a controller for a game. This is a subset of the valid system-specific actions.
- Game-specific: How scores are calculated and the memory location where the score is stored.
- Game-specific: Terminal conditions and any memory locations used for lives. This is used to check for when the game is over.
- Game-specific: Default values for lives and memory locations used by terminal conditions.
- Game-specific: Actions to start gameplay.

Actions and their combinations are needed to control the game using the API provided by FrAG which helps pass the actions to the emulated system without a controller. Actions such as reset and select are unavailable as the AI does not know what each action does and should not repeatedly reset the game, change game variants, or activate the menu. Game-specific information such as lives, terminal conditions and scores are needed for the AI to help understand whether or not a game is over. The AI agents trained use RL based on rewards, so the score difference between frames is used as a reward. For each frame stepped, the score is calculated, and the terminal conditions are checked for game over, letting the AI know to request a game reset. Actions to start gameplay are necessary as the AI cannot understand the menu screen or know which option to select in a menu. For the Atari 2600, the starting action will always include the game reset switch on the console, which starts the game. Any additional information, such as difficulty levels and game variants, is optional.

While the framework is designed to work with an AI, it is not necessary to have an AI playing the game. It can also be a human playing the game while game data is collected or analyzed. The framework, by default, does not perform any analysis and only provides a platform for it.

We now look at how FrAG is implemented, its interface and its limitations.

3.2.1 Structure & Implementation

Before looking at the framework properly, it helps to understand the execution flow of MAME and how it starts the emulated systems.

A flow diagram of how MAME runs from a high level is shown in Figure 3.1. Functions in the block labelled "Initialization" initialize the UI, parse options passed, and initialize the manager that connects the front and back end. Function H calls I, which initializes all emulated devices. Once initialized, it calls J, which makes the machine available to the front end, allowing the user to interact with the running system using the API provided. This connects the front end and back end of MAME. H then loops, executing all devices sequentially until execution crashes or the user quits. The created machine is destroyed once function G returns, as the machine created is not dynamically allocated.

The overall structure and relationships of classes in the implemented framework are shown in Figure 3.2. In FrAG, the class `MAMESystem` wraps an instance of MAME and relinquishes execution to the caller of the initializer, controlling the framework by stepping through frames. Figure 3.3 shows the high-level execution flow of the framework similar to the one in Figure

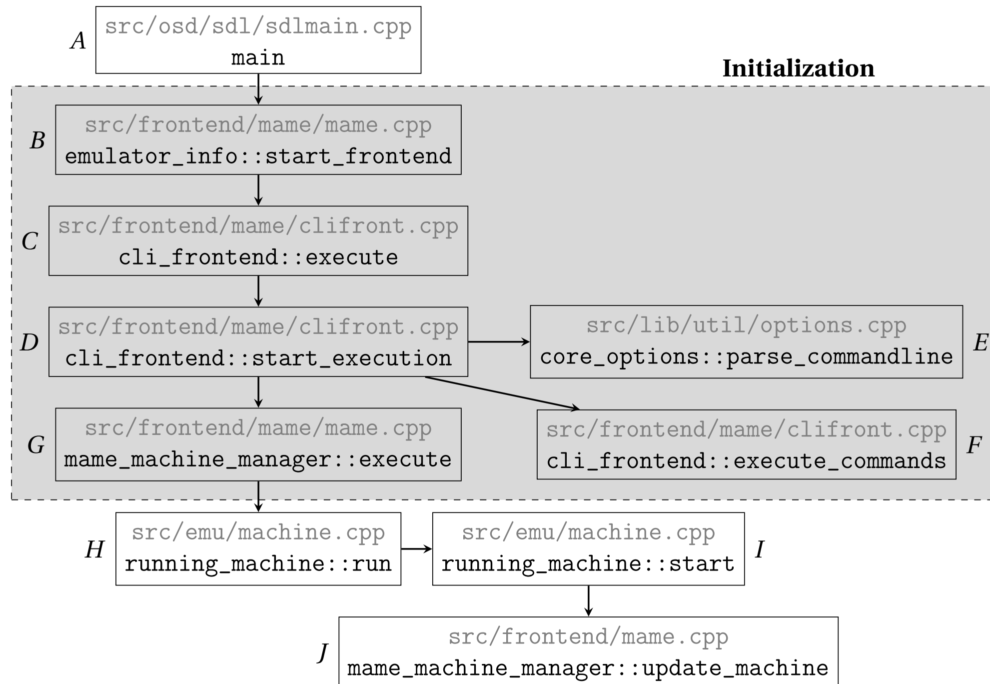


Figure 3.1: High-level overview of MAME's execution flow.

3.1. Functions in the block labelled "MAME Initialization" prefixed with 'frag_' only differ in an argument passed by the framework compared to the same in Figure 3.1. The significant differences are in function J, which calls the initialization function passed from MAMESystem as an argument; G, which creates an instance of running_machine on the heap instead of the stack, which prevents the machine from being destroyed once the function returns; and finally, H, which removes the loop and returns so that the execution flow returns to the program that called A, giving the user access to control MAME using the interface provided. The initialization function of MAMESystem, called from MAME, sets up all the necessary devices in the class for the framework to work.

Most of FrAG's implementation designs are similar to Arcade Learning Environment (ALE)'s and this was done to make the framework compatible

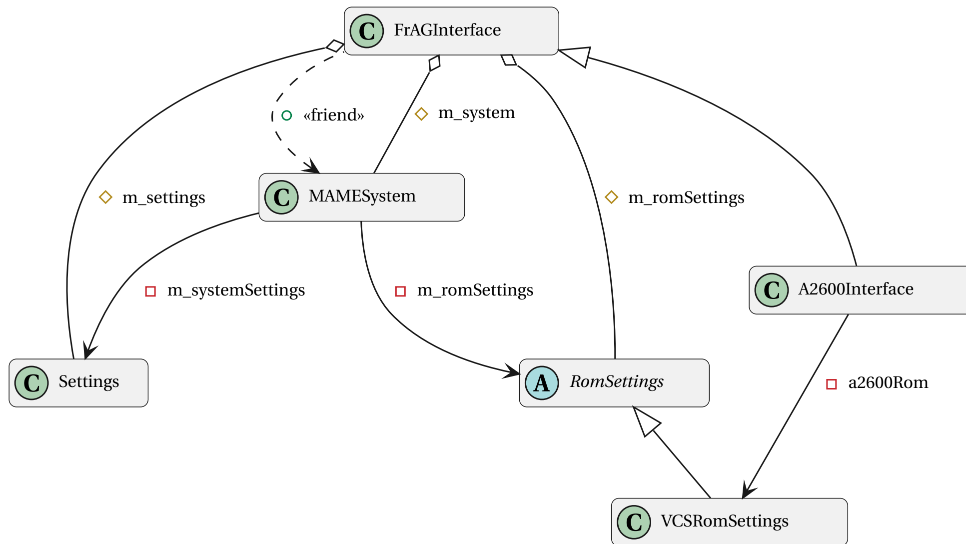


Figure 3.2: High-level UML class diagram of FrAG.

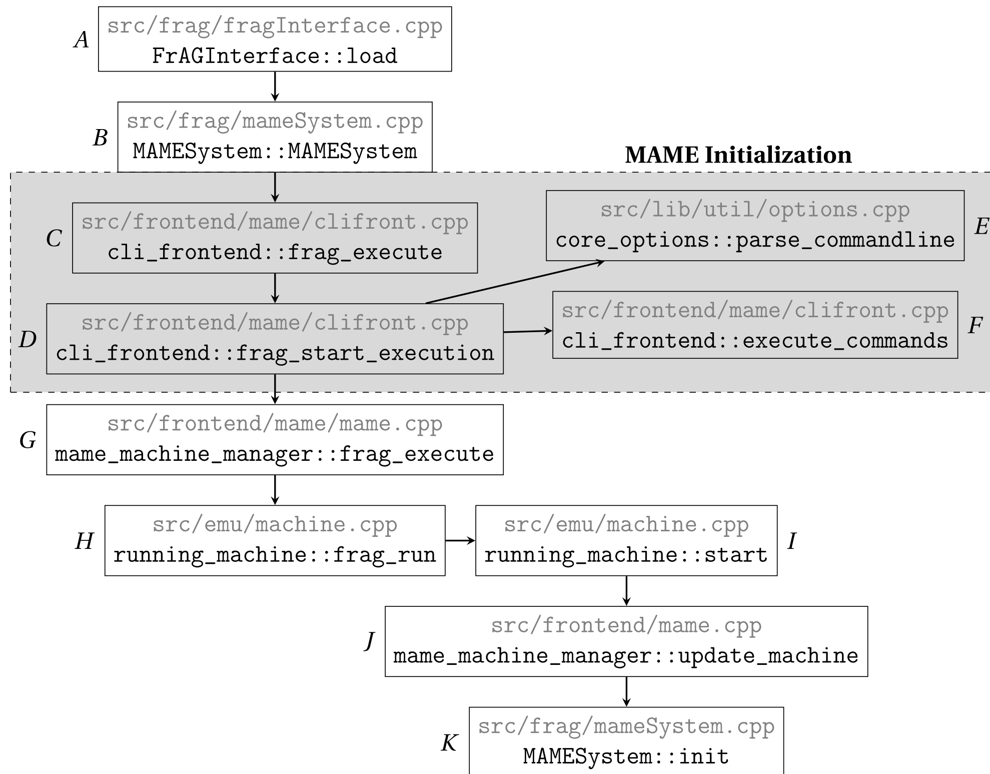


Figure 3.3: High-level overview of FrAG's execution flow.

with existing frameworks and packages that use ALE. The game-specific information required, such as calculating scores, random access memory (RAM) locations of where lives are stored, game-over conditions, etc., is stored in a class for each game derived from the system's general interface for game information, such as `VCSRomSettings`, the general game setting class for the Atari 2600. This design provides all games with a general interface. Moreover, due to each system having different capabilities and features, each system class derived from `RomSettings` can implement any additional functions or feature specific to the system, which can be accessed via the interface specific to the system (such as `A2600Interface`) derived from the general interface class `FrAGInterface`.

In MAME, an emulated controller or input source for a system is represented by an input port and field. Each button or action on a controller has a corresponding port, a field indicating the button on the controller and a value, which is the signal value of the action/button. There are two types of signals: digital and analog. Digital signals are buttons with a discrete value of 0 or 1, while analog signals are continuous. An example of an analog controller is the Atari 2600 paddles, capable of a range of values, allowing minute movements. Figure 3.4 shows a controller and how its port and fields are related. Often, a controller will only have one port; however, when more than one type of signal (digital and analog) is present, each button type will have a different port. For example, the Atari 2600 paddles have both a digital button and an analog dial and therefore have different ports for the digital button and the analog control. An example of this is shown later in Table 4.2 in §4.1 in Chapter 4.

To support multiple systems, each system class derived from `RomSet-`

tings needs to implement its actions using MAME's appropriate ports and fields for the controller emulated. The actions passed to the framework are integers where each integer is mapped to an action where each action has a port, a distinct field and signal value. This map also includes valid combinations of actions as described in §3.2. Figure 3.5 shows an example of how actions can be stored, matched and retrieved for the controller shown in Figure 3.4. A single action here is a 3-tuple containing the port, field and value. However, to allow multiple players and combinations of buttons, actions are returned as a list of 3-tuples. One of the requirements is that zero should always map to no operation. That is, the port, field and value should all be empty, which enables advancing a frame without any action.

The Settings class contains the options passed to MAME and MAMESystem. This includes whether to switch video on or off, frame skip amount, the seed used for a pseudorandom number generator, repeat action probability, throttling and debugging. The repeat action probability and random seed are specific to MAMESystem and are primarily used when training AI agents.

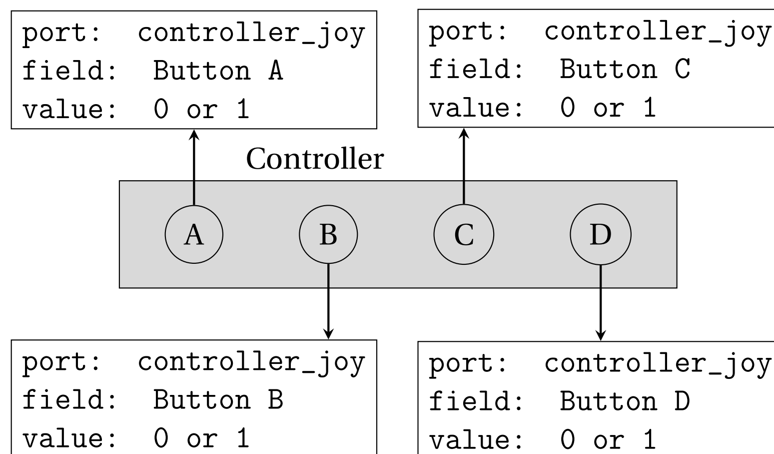


Figure 3.4: Example of a 4-button controller and its ports and fields.

Integer	Action
0	$\{\}$
1	$\{A_T\}$
2	$\{B_T\}$
3	$\{C_T\}$
4	$\{D_T\}$
5	$\{A_T, B_T\}$
6	$\{A_T, C_T\}$
7	$\{A_T, D_T\}$
8	$\{B_T, C_T\}$
9	$\{B_T, D_T\}$
10	$\{C_T, D_T\}$
11	$\{A_T, B_T, C_T\}$
12	$\{A_T, B_T, D_T\}$
$\vdots$	$\cdots$

Figure 3.5: Example mappings from integer to actions for the controller in Figure 3.4. A_T, B_T, C_T and D_T are 3-tuples corresponding to the buttons A, B, C and D, respectively.

3.2.2 Modifications & Compilation

All code written for the framework has been separated into a folder which can be dropped into newer versions of MAME, provided the structure of MAME does not change. MAME has been updated several times during the development of FrAG as the newer version of MAME introduced new features and bug fixes. All modifications made to MAME are enclosed in preprocessor directives to enable conditional compilation. Building MAME and FrAG only differ in a compilation option added to the makefile. When set to 1, it will compile FrAG and produce a shared library instead of the `mame` executable. A shared library is produced to make it easier to use the framework as a library and to create a Python3 package, allowing the use of packages such as gym and Stable Baselines 3 to train AI agents. A target to

install the library has also been added to make the compilation of programs using the framework easier. This target copies the header files needed and the shared library generated to the relevant system directories. A script to add all these modifications to port to newer versions has also been provided, again providing MAME's structure does not change.

To increase performance, MAME's graphical debugger, Lua console, user interface and input managers have been disabled by default when building FrAG. This can be changed by passing in an option when building. However, these elements are updated constantly to reflect any changes in emulation when a game is being run, which considerably slows down the framework.

A *memory tap* in MAME is a callback function called whenever a read or write happens to the memory space, and a *memory handler* is a function responsible for the value read or written. For example, in an emulator, a handler may be a bank or a memory region responsible for a value read or written from/to the memory space. In MAME, both taps and handlers can alter the value read or written. This, however, is a problem, especially for bank-switching games. MAME has no way of retrieving the bank responsible for an address read, written or executed and would return a concatenated string of all the taps or handlers responsible for the address. This would include the user-registered tap as well as the handler that was responsible for the memory region. This is due to how taps and handlers are registered. Both taps and handlers can alter the value of data read or written, making it difficult to ascertain which tap or handler was truly responsible for the value. To fix this, I have added functions to the emulated memory class to return the bank that was responsible for an address.

MAME uses several singleton and static classes and global variables

which prevent two instances of MAME running in one process. I made an attempt to remove all singleton classes and some functionality from MAME to make multiple instances in one process possible. These modifications that were made are not enclosed in preprocessor directives and require several changes which cannot be ported to new versions of MAME. They were made to reduce the overhead of using pipes for inter-process communication when training AI agents, which uses several instances of MAME, making the framework slower. However, after testing, I found that it was slower than running using subprocesses, as multiple instances would execute sequentially and not in parallel. This approach was not used in the testing of the framework and has been abandoned.

A detailed list of all modifications and additions to MAME in a `diff` format can be found in Appendix A.

3.2.3 User Interface

The base class `FrAGInterface` provides a general interface for querying system information to add support for new systems. This general interface cannot be used to launch games and only exists to query information and, as a class, to implement default behaviour which cannot be overridden. For an implemented system, the interface specific to a system such as `A2600Interface` is derived from the general interface and is used to launch and play games. Each system, called a driver in MAME, needs a separate interface. However, this can be avoided if the differences between the two drivers are minimal. For example, Atari 2600 (NTSC) and Atari 2600 (PAL) are different drivers but can be combined into a single interface as they

share the same controllers, central processing unit (CPU), etc.

Much of the framework's interface is similar to ALE to make it user-friendly and a drop-in replacement for packages that use ALE. To start the framework, the constructor of the required system interface with the game name is called, followed by a `load` call, which creates an instance of MAME. Any core settings, such as video output and audio are set through the `setInt`, `setBool` and `setString` methods before calling `load`, as these must be passed to MAME during startup. All MAME options are passed through the `setMAMEArg` function. While the settings passed used using `setInt` and `setBool` are validated, options passed through `setMAMEArg` are not validated due to the large number of options available in MAME. After this, the game can be controlled by using `act`, which accepts either a single input or a list of inputs and returns the difference in score, an integer used as the reward for the AI. If multiple inputs are passed, a list of rewards is returned where the index of the value is the corresponding player's reward. This can be ignored if no AI agent is being trained.

As the framework does not remove any of the features from MAME, an extensive set of debugger commands is available to the user, including interacting with MAME using the Lua console it offers. Debugger commands can be passed using a function provided in the interface to execute a command.

FrAG offers Python bindings for the Atari 2600 and the general interface, including a gym environment for Atari 2600, which can be used with various AI and analysis packages such as Stable Baselines 3, Keras RL, Tensorflow, and networkx.

3.3 Extracting Data

The most important data that one can get from the framework is execution information. To trace execution and memory access, taps can be installed on each memory space and CPU device emulated, which calls the handler functions in `RomSettings` when an address is read, written to or executed. These functions, by default, do not do anything until overridden by the system setting class (such as `VCSRomSettings`). Each system's handler might be different to account for the quirks of the device emulated. These handlers are passed the instance of `MAMESystem`, giving the function access to all of the emulated devices and memory spaces, allowing querying of the current state of a device, such as the registers in the CPU and the current position of the cathode-ray tube (CRT) raster beam.

To extract the pixels of the frame rendered and the entire memory space, functions in FrAG exist that return the current frame or the complete memory space values provided a memory space name. The downside to some debugger commands is that they save data to a file that must be read and parsed for analysis. Instead, some of the functions, such as getting memory spaces are faster and better suited for analysis as MAME returns these memory spaces as an object with functions to query the values of memory locations. It is up to the user on what analysis features to perform in these handlers. Moreover, commands can be run in MAME's debugger, which provides several features such as tracing CPUs, stepping through instructions, and breakpoints.

3.4 Portability

The design of the framework allows it to support any system MAME supports.[2] Adding support for a system is easy as the general interface `FrAGInterface` provides several helper functions which output information about a system aiding in adding support. The system game class must derive from `RomSettings` and implement all the virtual functions. As long as all the virtual functions are implemented and return what is expected, the framework can train an AI agent for a game supported by the system or analyze it.

3.5 Limitations

Due to how taps are implemented, only one tap can be installed on an emulated memory space. This limitation is due to how the taps are installed. The tap installed on the memory space is named, and a memory passthrough handler is registered for each tap, making removing a tap possible. If multiple taps were to be installed, each tap name and memory passthrough handler would need to be remembered, which was unnecessary as all the taps would call the same handler. The framework also only supports RGB and RGBA frames, which is a limitation due to how the function that returns the frame is implemented.

Multiple instances of MAME cannot run in parallel in the same process as several classes in MAME are singletons, preventing more than one class instance in a process and several systems emulated use global variables.

[2] An undergraduate student has already implemented the interface for the Fairchild Channel F console using an older version of FrAG. This work was presented as Michaud et al. [13].

3.6 Summary

In this chapter, the framework and its implementation were discussed along with MAME, the open-source multi-system emulator that the framework is built into, the information needed for the framework to work and its limitations. MAME's terminologies, such as taps, handlers, input ports and fields, were also discussed. Later sections detailed the modifications to MAME, its compilation and how the framework can extract data using the handlers, in-built debugger, etc., and briefly discussed how one might port it to other consoles.

FrAG's portability and scalability allow one to study game implementation of an entire corpus rather than focusing on one game from a particular console. It enables us to reverse engineer read-only memories (ROMs) for which the source has been lost, allowing us to uncover programming practices used without much human interaction. FrAG can also be used as a platform for training AI agents for several systems as a side-effect of the design.

The following two chapters will discuss the implementation of the Atari 2600 interface in the framework and the evaluation of the framework using Atari 2600 games.

Chapter 4

Framework Implementation: Atari 2600

This chapter describes and discusses the implementation of the Atari 2600 interface in the FrAG framework and also details some analyses performed, including a new type of disassembler which disassembles a read only memory (ROM) using data obtained from the framework.

4.1 Atari 2600 Actions

As described previously in §3.2.1 in Chapter 3, actions passed to the framework are integers mapped to a list of 3-tuples containing the port, field and value of the control. This mapping is implemented in the system game interface derived from `RomSettings`. For the Atari 2600, the actions are stored as an enumeration and the enumeration is mapped to a list of 3-tuples. The Atari 2600 has two controllers, a joystick and a paddle; the paddle is an analog controller that requires a value within a range, unlike joysticks

which are digital buttons that can be 0 or 1. All action values are constant in the interface: 1 for digital and 30 for the paddle, an amount of movement which was empirically determined. The constant values for analog buttons or any buttons is required because the artificial intelligence (AI) does not know what value to use for a button and cannot distinguish between analog and digital buttons. The port, field and value returned for each of the actions possible by the joystick and the paddle is shown in Table 4.1 and Table 4.2, respectively. Combinations such as up+fire are returned by adding the tuples to an array. The backend of the framework, `MAMESystem`, which executes the actions, expects a list of tuples from `RomSettings`, making the combinations possible. It executes all the actions and then steps the game for one frame. For example, in a game, if the AI selected the combination up+left+fire, it would pass in the corresponding action integer for the combination. Once the framework receives this integer, it retrieves the list of actions, which here will include the action tuple for up, left, and fire and execute all actions. Following this, the emulation resumes for a frame, simulating a user pressing the buttons on the controller. All possible actions for the joystick and paddles are shown in Figure 4.1. The combinations include all movements with the fire button but are not shown for simplicity. All buttons/actions on the joystick are digital and thus share the same port. However, the paddles have both digital and analog. In total each player has 17 actions including no operation (noop) on a joystick and 6 actions including noop on a paddle. The movement in paddles is analog, allowing fine-grained horizontal control, while the button on the side of a paddle is digital.

Multiplayer actions are also executed similarly. The only difference is

that the actions are passed as a list where the index determines the player number. The Atari 2600 supports up to four players using paddles (as two paddles are connected to one port) and up to two players using joysticks.

Enum	**Action**	**Port**	**Field**	**Value**
0	NOOP	-	-	-
1	FIRE	:joyport1:joy:JOY	P1 Button 1	1
2	UP	:joyport1:joy:JOY	P1 Up	1
3	RIGHT	:joyport1:joy:JOY	P1 Right	1
4	LEFT	:joyport1:joy:JOY	P1 Left	1
5	DOWN	:joyport1:joy:JOY	P1 Down	1
19	FIRE	:joyport2:joy:JOY	P2 Button 1	1
20	UP	:joyport2:joy:JOY	P2 Up	1
21	RIGHT	:joyport2:joy:JOY	P2 Right	1
22	LEFT	:joyport2:joy:JOY	P2 Left	1
23	DOWN	:joyport2:joy:JOY	P2 Down	1
73	RESET	:SWB	Reset Game	1
74	SELECT	:SWB	Select Game	1

Table 4.1: Selected Atari 2600 joystick actions and their respective ports, fields and values.

Enum	**Action**	**Port**	**Field**	**Value**
0	NOOP	-	-	-
1	FIRE	:joyport1:pad:JOY	P1 Button 1	1
2	RIGHT	:joyport1:pad:POTX	Paddle	30
3	LEFT	:joyport1:pad:POTX	Paddle	-30
19	FIRE	:joyport1:pad:JOY	P2 Button 1	1
20	RIGHT	:joyport1:pad:POTY	Paddle 2	30
21	LEFT	:joyport1:pad:POTY	Paddle 2	-30
73	RESET	:SWB	Reset Game	1
74	SELECT	:SWB	Select Game	1

Table 4.2: Selected paddle actions and their respective ports, fields and values.

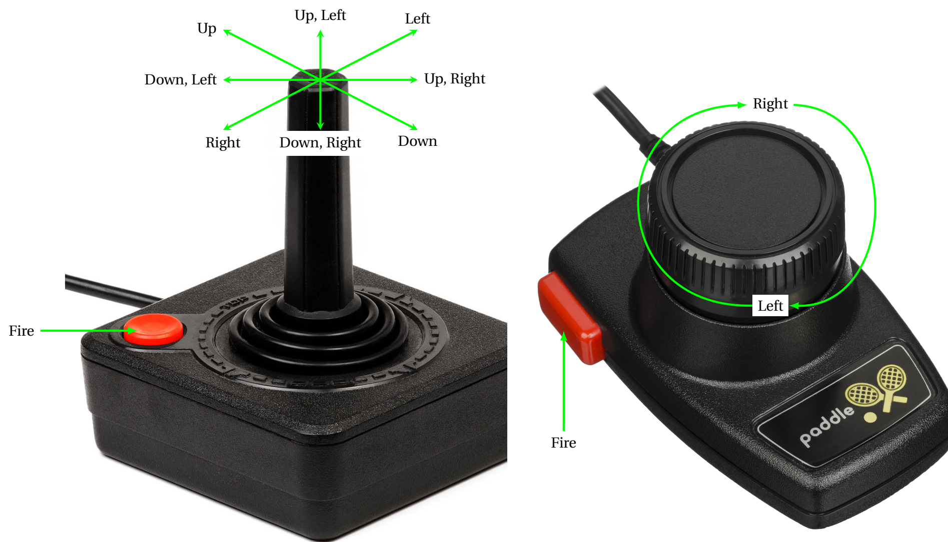


Figure 4.1: All actions possible using the Atari 2600 joystick and paddle. Each single action is mapped to a tuple shown in Tables 4.1 and 4.2. Credits: Evan Amos, public domain.

4.2 Atari 2600 Interface

The Atari 2600 interface includes a few extra functions for the analyses implemented and retrieving the data obtained. As mentioned in Chapter 3, the framework is only a platform capable of performing dynamic analysis but does not perform any itself. Due to the performance of the television interface adapter (TIA) emulation, performing any dynamic analysis for the Atari 2600 slows down the framework considerably. As Multiple Arcade Machine Emulator (MAME) emulates all the devices as accurately as possible, I was reluctant to try optimizing the TIA implementation for performance due to possible accuracy loss.

As the design of FrAG is similar to Arcade Learning Environment (ALE), game settings for a game from ALE can be dropped into FrAG with minimal changes, allowing FrAG to support over 57 games out of the box.

The interface extracts the following data using the read, write and execute handlers:

- Execution Tracking. Tracks the addresses in ROM that are read, written or executed.
- Random Access Memory (RAM) Usage Graph. Builds a directed graph for each RAM location, denoting how values change over a game.
- RAM Usage Count. Tracks how often each RAM location is accessed.
- Execution Graph. Builds a weighted directed graph where the weight denotes how often that execution path was taken.
- Display Kernel Detection. Captures all program counter (PC) executed when the raster beam is actively drawing.

Using the extracted data, the following are some of the analyses that can be performed:

- Analyzing overall memory usage and how often a memory location is read compared to written.
- Detecting self-modifying code by looking at the addresses written to and executed in the execution data.
- Extracting display kernels from the ROM. The display kernel in a ROM is code responsible for displaying video by writing to the TIA registers. It is time sensitive as the writes to the TIA registers must happen as the electron beam passes across the television (TV) screen as described in §2.1.1 in Chapter 2.

- Analyzing how a value held by a memory location changes throughout a game.
- Analyzing bank switching frequency.
- Detecting code/data overlaps.
- Detecting code located in RAM being executed by looking at the addresses of the execution data.
- Reverse engineering ROM using tracked execution information.
- Detecting address space division in ROMs.
- Analyzing ROM code and data separation.
- Analyzing frequency of executed instructions.

Most of the data captured is from the handlers. The read, write and execute handlers for `VCSRomSettings` are shown in Appendix B.

The read handler is particularly long as the emulated 6502 central processing unit (CPU) in MAME does not differentiate between instruction fetches and data reads, which results in many false positives for bytes read as data. The 6507 splits memory into groups of 256-byte pages, and a page boundary refers to the point at which, when one page ends, another begins.

If the processor executes an instruction with a 2-byte operand, the first byte is retrieved from memory, and then the second byte. If the next byte is on a different page than the current address, this is known as a page boundary crossing. When a page boundary is crossed, the processor reads the *wrong* address first. After reading the high byte of the address, the

processor adds the index register to the low byte of the address and reads this. To correct this, the processor fixes the read by incrementing the high-byte and re-reading the address; the first read is invalid. For example, consider a program executing an X indexed instruction (e.g. `lda $10,x`) fetching data from memory, which is on a different page. When the instruction is executed, the CPU reads the value of the X register and adds it to the base address to get the target address. However, if the addition causes a carry to the high byte of the address, the high byte of the address will be temporarily incorrect, and the processor will read from the wrong page of memory at first, using an extra cycle. The problem is that the first address read, which is incorrect, results in FrAG's read handler being called. These wrong reads can result in false positives (addresses marked as data when they are not necessarily data) as they are not read as data. Due to this, the read handler for the Atari 2600 only considers the last read by an instruction as a valid read and marks the address read as data.

Similarly, reads from a branch or returns from subroutine instructions are discarded as they do not read any data and any operand read as data is ignored. For example, consider a branch instruction where the branch is not taken. The processor will read the address to jump to but in the end not jump to it which causes the address to be marked as read, giving us a false positive. To avoid these cases, all reads performed by branch and jump instructions are ignored. This does not affect other reads as branch and jump instructions do not read any memory.

The information on what addresses are executed, read and written to, and the corresponding bank is stored in a 2D array, where the first index is the bank number and the second is the address. Arrays were chosen for

their performance as the maximum number of possible addresses is only 65,536, and the information is stored as a short integer. It uses bit flagging of an 8-bit integer making it memory-efficient. Table 4.3 shows the possible (octal) numbers and their interpretation.

Octal Number	Binary Representation	Interpretation
0	000	Not read, written or executed
1	001	Executed
2	010	Written
3	011	Executed & written
4	100	Read
5	101	Read & executed
6	110	Read & written
7	111	Read, written & executed

Table 4.3: Octal numbers and their interpretation used for tracking addresses.

If the option to analyze RAM usage and track RAM usage count is enabled, the read handlers will track how often each RAM location is accessed and build a directed graph for each RAM location, keeping track of how each RAM value changes. Each memory location has a graph which is updated when a value is written to the location. The graph for a RAM location depicts how the values change. When a RAM location is written to, an edge from the previous value to the current value that is written is added to the graph. For instance, consider the graph shown in Figure 4.2. The graph shows how a RAM location storing lives may change over time in a game. Let us say that the player has four lives. At the beginning of the game, the player has four lives and once the game starts the player has 3 lives remaining. At this point, the value held by the RAM location is decremented by one. This change is

represented by the directional edge from 4 to 3 as depicted in the second graph. When the player dies in the game, a life is lost and therefore, the RAM location is once again decremented which is represented by the edge from 3 to 2. These edges are added until the value held by the RAM location becomes 0, at which point the player is on their last life. At this point, if the player loses, the game is over. The graphs are saved to memory only when explicitly done by a function called by the user. If not, the existing graphs are used, preserving all edges across game resets. Consider the game described above with 4 lives. If a game is reset, at which point a 4 is written to the RAM location and the user did not save the graphs, an edge from 0 to 4 would be added to the graph. The graphs can be exported as graphviz files,[1] which is a tool for drawing graphs specified using the DOT language. The exported graph combines all graphs in one file to avoid having a separate file for each RAM location graph which can then be read into the analysis scripts I developed.

The write handler, on the other hand, is straightforward. As the analyses implemented do not depend on anything written to RAM, much of it is ignored, including in-cartridge RAM. It is only helpful for RAM usage analysis and tracking RAM usage count. However, as a precaution, any write to ROM area is also tracked but highly unlikely as the ROM cartridge cannot be modified. Most RAM analysis graphs are built into the write handler as they track the writes to zero page RAM. The graph's current node is stored in an array using which transitions are added to the graph.

Like the read and write handlers, the execution or PC handler is called before the CPU executes the instruction. The PC handler marks the address

[1] https://graphviz.org

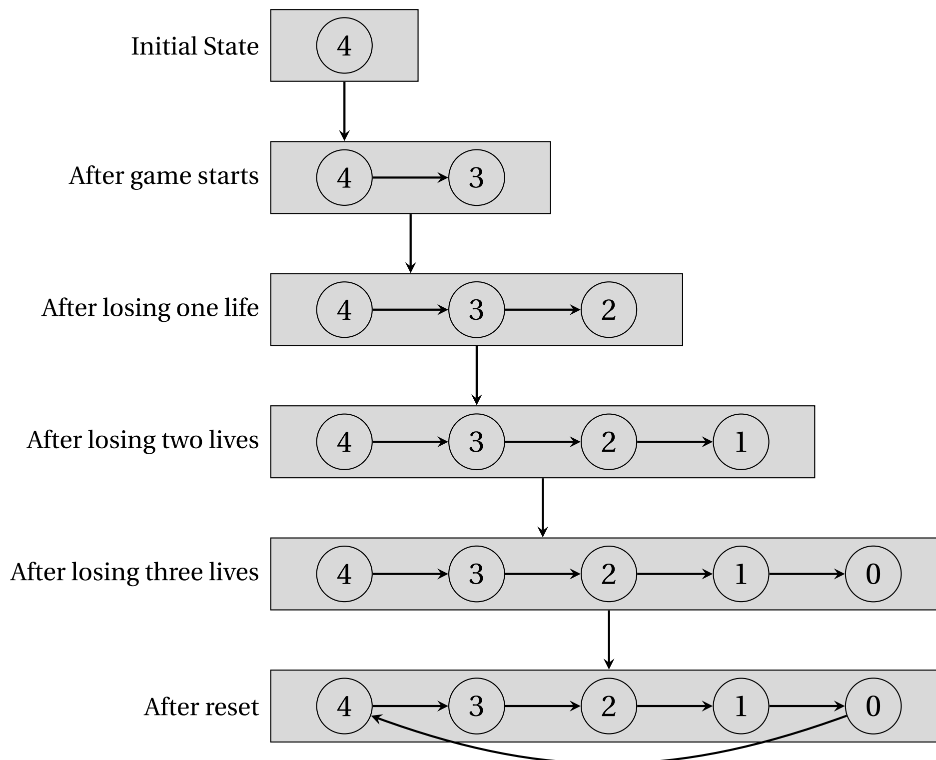


Figure 4.2: An example graph depicting how a RAM location graph is built.

as executed, increments the PC usage counter, tracking how often an address is executed and adds the PC to a map which holds the display kernel if the raster line is between 41 and 232 (if kernel detection is turned on) as this is the visible range for the Atari 2600 [63] and finally also builds a weighted directed execution graph where the nodes are the program counters and the edge weight indicates how often the execution path is taken. Consider the code in Listing 4.1. The execution graph for this code listing is shown in Figure 4.3. The code increments a counter until it reaches 10 after which the instruction after the `jmp` will execute. The weight of the edges represent the number of times that sequence has been executed. In the example here, the code has been executed exactly once from `f000` to `f002` while the line at `f002` has been executed 10 times of which 9 times the execution

sequence was from f007. There exists no edge between f007 and f008 since it is impossible to execute a jump instruction to Loop and execute the instruction at f008.

```
1 f000:       LDY #0    ; Initialize counter to 0
2 f002: Loop  CPY #10   ; Compare counter to 10
3 f004:       BEQ End   ; Branch to End if counter = 10
4 f006:       INY       ; Increment the counter
5 f007:       JMP Loop  ; Jump to Loop
6 f008: End   INX       ; Some instruction
```

Listing 4.1: Example code for execution graph.

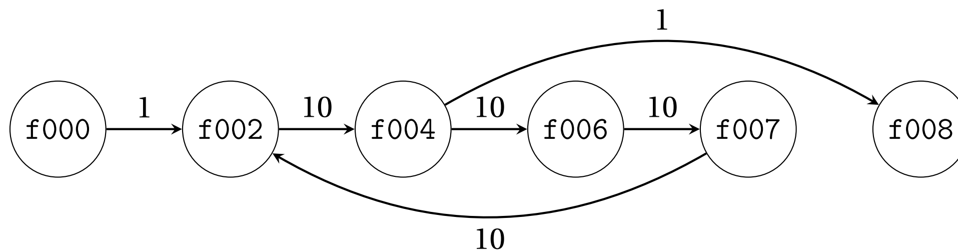


Figure 4.3: Execution graph for the example code in Listing 4.1

4.3 Using Extracted Data

One of the most important analyses possible here is disassembling the source code. As described previously in §2.2, the de-translation of binary to source is an undecidable problem. However, a new disassembly type is possible using the data extracted from the framework.

There are two traditional types of disassemblers: linear and recursive. A linear disassembler iterates over the program binary, disassembling one

instruction after the other, and invalid bytes are treated as data leading to an incorrect disassembly. Recursive disassemblers try to follow the path that the actual program takes. This involves analyzing branches, jumps, sub-routines, etc. [64]. Recursive disassemblers fail to disassemble bank-switching games as both banks share the same address space and fail to follow indirect branch instructions causing bank switches, resulting in an incorrect or incomplete disassembly. Moreover, these disassemblers fail to disassemble relative branches correctly in the case where developers used relative branches as a space optimization to save a byte. However, FrAG gathers data capturing actual code execution.

To take advantage of the execution data extracted, I have developed a new type of disassembler called a *point disassembler*. This reads and parses the execution information dumped by the framework and disassembles only the addresses that have been read, written to or executed in the ROM. This enables us to precisely disassemble an address both read as data and code as either data or code. The evaluation of the framework performs all the analyses noted above.

4.4 Limitations

X-in-1 game cartridges combine several games into one cartridge, making it tricky to have a single `ROMSetting`. While they are supported, only one game can be played and to switch to another game in the cartridge, FrAG would need to restart. Moreover, as an X-in-1 game is a concatenation of small games, it is possible to extract the game or find a single ROM image for the game, which can then be used in the framework. A few bank switching

types and devices supported by the Atari 2600, such as the Display Processor Chip (DPC) used in *Pitfall II: Lost Caverns*, Harmony and Starpath cartridges, etc., are not supported by the interface due to the extensive work that would be required to support them.

As for the limitations to the analyses, the execution graph is not an execution trace as two nodes in the graph cannot have the exact same value as shown in the example earlier in Figure 4.3. While a full execution trace can be implemented, the amount of data generated is enormous, making it hard to store and parse all the data for a large amount of gameplay. However, this would be fine if the analyses were performed dynamically, albeit sacrificing some performance.

4.5 Summary

This chapter discussed the implementation details of the Atari 2600 interface, including all valid actions, the data extracted and the different analyses possible with them. Page boundary, page boundary crossing and a related quirk of the 6507 microprocessor were also discussed to help understand what the read handler does. A new type of disassembler, called a point disassembler, was introduced to help with disassembling a ROM using the execution data obtained from the framework.

Some limitations exist with the support for bank switching types and chips, but the most frequently used bank switching types are supported. The unsupported chips and bank-switching types are supported by MAME but still need to be implemented for the framework. The next chapter evaluates the framework using Atari 2600 games and discusses the analysis results.

Chapter 5

Evaluation

FrAG was evaluated by running it through a suite of games for which we either had original source code or complete human reverse-engineered disassembly. This is useful for comparing the reverse engineered read only memory (ROM) with the ground truth to determine how close to the original source FrAG can get. A summary of the games, type of disassembly and other relevant information is shown in Table 5.1.

5.1 Test suite: Rationale

Games that are diverse in gameplay and bank-switching types were used to test all aspects of the framework and its efficacy. The test suite consisted of the following eight games:

- *Boxing* (1980): *Boxing* by Activision is a 2 KiB game with simple gameplay. The screen is a top-down view of a boxing match between two opponents, which runs for 2 minutes. There are two game variants: single-player against the computer and two-player of which the single-

player variant was used. The player can shuffle around and punch the opponent, pushing the opponent back a little as long as they are away from the ropes. Points are used to determine the game's outcome and are awarded for each punch [65]. This game was chosen because of its simplicity and relatively low number of game variants.

- *Combat* (1977): *Combat* by Atari is a 2 KiB game with 27 game variants and gameplay that gets increasingly difficult as the game variant increases. It is one of Atari's launch titles and was included in the box with the console [17]. It is a multiplayer game in which players control tanks, fighter jets or biplanes (depending on the game variant) and fire missiles at their opponents, gaining points for hits. The points scored determine the winner. The screen for tank and fighter jet games is a top-down view, whereas the biplane games have a side view [66]. *Combat* has a clear distinction between code and data in its ROM: one part of the ROM is code, whereas the other is data.

- *River Raid* (1982): *River Raid* by Activision is a 4 KiB single-player shooter game in which the player controls a fighter jet, and the goal is to destroy enemy equipment and facilities without the jet crashing or going empty on fuel. Fuel stations exist to refuel when needed. Points are awarded for destroying enemy jets or other installations. Each game, the player begins with three jets which are used if the current jet runs out of fuel or crashes; an additional jet is awarded for every 10,000 points scored [67]. The game used procedural content generation to generate the river banks and islands on screen and overlapped code with data to optimize space [10, pp. 130–132, 187]. This code/data

overlap makes it an interesting pick to test the framework.

- *Dragonfire* (1982): *Dragonfire* by Imagic is another 4 KiB game, involving princes and dragons. The premise is that dragons have taken over the player's kingdom, and as a prince, it is the player's duty to get all the kingdom's treasure back [68]. The screen is a side view with two distinct game screens. The player must cross a bridge, while avoiding fireballs and dragonfire, to reach the treasure room guarded by a dragon. Seven chances are allowed to collect all the treasure objects without being hit by a fireball or dragonfire. The game has four variants with increasing levels of difficulty [68]. What is interesting about this game is a code-code overlap: depending upon where in the overlapped sequence the code jumped to, it would consume a different number of cycles. This sequence is stored as data in the ROM [10, pp. 88–89, Figure 8.17]. This allows us to test the framework with complex execution sequences.

- *Kaboom!* (1981): *Kaboom!* by Activision is another 2 KiB game. It is a simple game where the player controls three buckets of water to catch all the bombs thrown by a mad bomber. If a bomb is caught, the player earns several points depending on the speed and type of bomb. If a bomb slips through, all the bombs explode, there is one less bucket, and the mad bomber laughs. The game ends when no bucket is left, or all bombs are caught. There are eight levels of play [69]. The game uses paddles which have a different interface than a joystick, and was included to test the paddles in the framework.

- *Moonsweeper* (1983): *Moonsweeper* is an 8 KiB game with two 4 KiB

banks which employs a frequently used bank switching scheme that switches the bank when a particular address is read or written to. The player is the pilot of the USS Minesweeper, which patrols various moons, rescues stranded miners and destroys enemy facilities while avoiding fire and other natural dangers. After a successful rescue, the USS Minesweeper returns to orbit to continue patrolling and wait for the next rescue operation on another moon [70]. Changing modes in this game affects the moon's colour, indicating the difficulty of the rescue operation on the moon, but there is no change to the gameplay itself. Points are earned under various conditions and the game is a delayed reward game, where actions might not yield points immediately [70]. The game was chosen to test the F8 bank-switching type.

- *Montezuma's Revenge* (1984): *Montezuma's Revenge* is also an 8 KiB game with eight 1 KiB pages that can be swapped into banks. The player must guide Panama Joe (the protagonist) to Montezuma's treasure through a labyrinth of chambers full of creatures and contraptions that can kill him. He must avoid getting killed and pick up several aids to be used in further battles and escape with the stolen treasure [71]. The player must go through mazes, climb ladders and avoid several traps. Points are earned for collecting aid and overcoming dangers [71]. This makes it difficult for the artificial intelligence (AI) to learn strategies, as evident from the results shown in [5]. Several studies have been performed using *Montezuma's Revenge* to solve exploration games using an AI [51, 52]. The performance of the AI makes it interesting to include this in the test suite and gauge how much code is

executed, or data read for a game with little to no progress in terms of gameplay. The game was also chosen as it is one of the very few games of the E0 bank-switching type to have a good disassembly.

- *Solaris* (1986): *Solaris* is a 16 KiB game with four 4 KiB banks, gameplay and a backstory similar to that of *Moonsweeper*. The player's mission is to save people trapped on the lost planet Solaris before the enemies find and destroy it. During the journey of looking for Solaris, many enemies must be destroyed while defending the spaceship and other planets [72]. Loss of all spaceships or arrival at Solaris ends the game. Points are scored at various stages in the game. Similar to *Moonsweeper*, the player controls a spacecraft and can choose a planet to warp to for spaceship fuel or repair [72]. This game was chosen to test the F6 bank-switching type.

Game	**Size**	**Bank-switching**	**Source Code**	**# of Instructions**	**# of Data bytes**
Boxing	2 KiB	N/A	✗	747	647
Combat	2 KiB	N/A	✗	770	571
River Raid	4 KiB	N/A	✓	1588	1066
Dragonfire	4 KiB	N/A	✓	1574	1158
Kaboom!	2 KiB	N/A	✗	847	428
Moonsweeper	8 KiB	F8	✓	2890	2752
Montezuma's Revenge	8 KiB	E0	✗	2914	2388
Solaris	16 KiB	F6	✓	4957	6694

Table 5.1: Test suite summary. ✗ indicates a well-commented disassembly; ✓indicates the original source code was available.

5.2 Results

This section discusses the different analyses performed using FrAG and its results on the eight games described above. The reinforcement learning (RL) method used the deep q-network (DQN) algorithm [5, 73] from Stable Baselines 3 [74] using the CnnPolicy which uses the frame image as input and with the hyperparameters using default values except the learning rate which was changed to match the hyperparameter given in [5]. Some minor changes to the Atari wrapper and environment maker were needed to make it work with FrAG. All agents were trained for 5, 10 and 15 million frames to test whether the amount of time the AI trained for resulted in more coverage, which is the amount of ROM that is accessed.[1] The more the coverage, the better it is as it produces a better disassembly of the ROM, closer to the ground truth. A run for 15 million frames was also conducted using random actions to determine whether the AI was necessary to produce similar results.

Each AI agent played four instances of a game in parallel using subprocesses to train while FrAG collected execution information. This resulted in four sets of execution information for each training session; all were merged where possible to provide greater coverage. Table 5.2 shows the overall coverage for each training session. While some games do not improve, a few games benefit from running longer, such as *Dragonfire* and *River Raid*. Thus, I have decided to discuss the results from the training session of 15 million frames.

[1]These limits were due to resource limitations on the research computing clusters the framework ran on, which only allowed the framework to run for 7 days on central processing unit (CPU) clusters and 24 hours on GPU clusters.

	Ground Truth			# of Frames (M)		
	#Code (Ins.)	# Data (bytes)	Unused bytes	5	10	15
Boxing	747	647	3			
#Code (Ins.)				738	738	738
# Data (Bytes)				644	644	644
Total Bytes				2028	2028	2028
Combat	770	571	8			
#Code (Ins.)				696	696	696
# Data (Bytes)				232	228	232
Total Bytes				1558	1554	1558
River Raid	1588	1066	1			
#Code (Ins.)				1517	1520	1521
# Data (Bytes)				851	863	863
Total Bytes				3745	3762	3764
Dragonfire	1574	1158	8			
#Code (Ins.)				1437	1446	1446
# Data (Bytes)				884	911	922
Total Bytes				3570	3613	3624
Kaboom!	847	428	0			
#Code (Ins.)				815	815	815
# Data (Bytes)				419	420	420
Total Bytes				1978	1979	1979
Moonsweeper	2890	2752	0			
#Code (Ins.)				2546	2587	2587
# Data (Bytes)				2238	2241	2243
Total Bytes				7018	7095	7097
Montezuma's Revenge	2914	2388	3			
#Code (Ins.)				2103	2341	2343
# Data (Bytes)				1278	1398	1408
Total Bytes				5458	6047	6062
Solaris	4957	6694	0			
#Code (Ins.)				4905	4897	4904
# Data (Bytes)				6149	6145	6167
Total Bytes				15733	15713	15748

Table 5.2: ROM coverage for each iteration of DQN model training. "Ins." refers to instructions.

To interpret the results, it is necessary to understand how the point disassembler disassembles the ROM using the execution information dumped by FrAG.

5.2.1 Point Disassembler

The point disassembler built takes as input the tracking data dumped by FrAG and disassembles only the points in ROM which have either been read or executed. It discards all writes because a ROM is read-only memory, and nothing can be written to it. If an address was both executed and read from, it is disassembled as an instruction and a comment is added indicating it is also read as data. However, if an operand was executed (code-code overlap), the program counter (PC) and the operand are disassembled as data with a comment indicating the bytes are executed.

A relocatable origin (RORG) is the address to which the code is assembled as opposed to the origin (ORG) where the code is placed physically in ROM. When possible, the disassembler disassembles with the relocatable origin intact. However, due to bugs in Multiple Arcade Machine Emulator (MAME), bank switching games make this tricky. For example, Figure 5.1 shows a screenshot from MAME's debugger executing the instruction `dex` at `$3073` in *Moonsweeper*. However, the instruction does not correspond to the PC shown in the ROM. The instruction corresponds to `$1073` in bank `0`, and MAME detects the bank correctly; however, the address shown corresponds to bank 1. Since the current bank indicated is correct, the PC handler adds the address `$3073` to bank `0`, which gives us a false PC. If the point disassembler cannot detect the relocatable origin, it defaults to the physical location

```
306D  sta $d7          85 D7
306F  sta $2b          85 2B
3071  ldx #$0b         A2 0B
3073  dex              CA
3074  lda $008d, y     B9 8D 00
3077  sty $c3          84 C3
3079  pha              48
307A  and #$f0         29 F0
307C  lsr a            4A
307D  adc #$00         69 00
307F  cmp #$00         C9 00
3081  bne $3087        D0 04
3083  lda $c5          A5 C5
3085  bne $308b        D0 04
3087  ldy #$00         A0 00
3089  sty $c5          84 C5
308B  sta $cc, x       95 CC
308D  dex              CA
```

Figure 5.1: MAME debugger disassembly window.

in ROM. *Moonsweeper* is the only game in the test suite that is affected by this bug.

5.2.2 Coverage & Disassembly

While none of the games produced 100% coverage, all had over 74% coverage, and all but *Montezuma's Revenge* and *Combat* had at least 86.6% coverage. For simple games such as *Boxing* where most of the instructions are executed and data read, investigating why some addresses were not executed or read provides insight into how the framework can be improved for greater coverage. A visual representation of which areas were covered in ROM and a comparison to the source for three games – *Boxing*, *Combat* and *Dragonfire* – is shown in Figure 5.2. These games were selected for their small sizes and separation of code and data, as larger bank switching games make the graph harder to interpret. As mentioned in §5.1, *Combat* has a clear separation in ROM between code and data, and it is interesting to

compare it to *Dragonfire*, which freely mixes code and data. While there are a few games that mix code and data more frequently than *Dragonfire*, when shown on a graph, the separation of code and data can be hard to notice as they are often only a few bytes. Overall, we observe that most code is placed in the first half of the ROM and data in the second half and that most code is executed as the AI agent trains while several data bytes are not used.

To understand what was not executed or read, *Boxing* was investigated in detail as the coverage is very good, with over 95% of the ROM covered. Several addresses are not executed as they are only executed when the difficulty is switched, the grayscale mode is switched on, colours start to cycle after a certain amount of time, or the game variant changes (all games cycled colours when idle to prevent burn-ins on old cathode-ray tube (CRT) televisions (TVs)). How can these oversights be addressed?

Most unknown addresses corresponding to game variants and difficulty switches can be seen by training the AI on all the game variants and difficulties. This is one of the tradeoffs of training the AI on only one game variant. Another simple way to increase the coverage is to cycle between the game variants to make sure the code for it gets executed. This way, the AI does not have to train on multiple game variants. However, some code corresponding to difficulty levels might not get executed as these require the gameplay to be active. The fix for grayscale is also simple, as switching it on for a few frames and then switching it off will execute areas of the code that correspond to grayscale frames. However, solving colour cycling can be tricky. The attract-mode code takes a long time to get executed and might never get executed while an AI is training as the AI often passes a game-playing action; colour cycling only starts when no action is passed,

and the game is idle. In *Boxing*, colour cycling does not begin until a counter reaches `$ff`. Moreover, this counter is reset when a new game starts. *Boxing* is not unusual in this respect as none of the games in the corpus execute or read the code corresponding to features such as grayscale mode, colour cycling, and difficulties as they are never toggled.

Games sometimes will not use some bytes in ROM and thus will never provide 100% coverage using only dynamic analysis. For example, some data bytes in *Boxing* are never read. At this point, it is possible to use this information from FrAG and switch to static analysis to analyze the bytes not covered. This is especially an advantage for reverse engineering as one does not have to investigate if a byte is used in the game or not. Manually analyzing unused bytes can often be time consuming as the bytes may be read by some non-obvious indexed instruction.

Combat is a simple game, but we do not see full coverage as it is a multiplayer game comprised of 27 variants that are never cycled through or played by the AI. This results in poor data coverage as most variants differ from the default game variant. Moreover, the code for the second player is never executed, as the second tank is always stationary. To verify this, *Combat* was run three times. The first run cycled through all game variants, the second run cycled through all game variants and started the game in that variant (did not play), and the third run in which the second player played until timeout while the first player remained stationary. The execution information from each of these runs was combined with the execution data from the AI run and coverage was measured. The results obtained are shown in Table 5.3. Overall, we see that there is not any code for the second player and cycling through the game variants obtains greater coverage. We find

that cycling the game variants is also not enough for a complete coverage as none of the other game variants were played, that is the players did not move which means that the graphics code and data for turning and shooting were not used.

Run	Code (Ins.)	Data (bytes)	Total bytes
Original Run (re-stated from Table 5.2)	696	232	1558
Game variant cycle	716	372	1739
Game variant cycle & game start	716	372	1739
Second player play (1st variant only)	696	232	1558

Table 5.3: Coverage of *Combat* with different runs.

Game	# of Frames		
	5 M	**10 M**	**15 M**
Boxing	2812	5627	8452
Combat	2448	4900	7352
Kaboom!	7786	14553	21068
Dragonfire	5200	5990	12287
River Raid	7161	13057	19544
Moonsweeper	3590	7054	9589
Montezuma's Revenge	12	23	41
Solaris	679	1334	2059

Table 5.4: Number of games played by all four instances using DQN.

The number of games played by all the four instances during training is shown in Table 5.4. We notice that the number of games played varies

dramatically across different types of games. However, this does not mean the AI played more *Kaboom!* than *Montezuma's Revenge* as the AI was trained for 15 M frames regardless of the number of games. The AI only resets the game if the game is over and for games that have short gameplay time, the AI resets more. Perhaps the most surprising result here is of *Montezuma's Revenge* even though it does not have a timer and the game is only reset if the player has lost all lives. Since the AI makes some random actions before training starts, it would be expected that four instances of FrAG running the game would play more games as it is quite easy to lose lives. But surprisingly, this is not the case here as *Montezuma's Revenge* was only played 41 times by the AI.

To test the efficacy of the point disassembler and of FrAG, the disassembled ROM was compared to the source code or well-commented disassembly. Both the disassemblies are compared by their instructions and data. The source code was assembled using a custom version of DASM, a 6502 assembler, to generate source code listings.[2] The generated listing was then passed through a program I wrote to extract the relevant information and split the listing into banks. The point disassembly (AI disassembly) and the source were then compared to compute a confusion matrix. The matrices are shown in Table 5.5.

Most games produce a good disassembly where the disassembled addresses are disassembled similar to the ground truth. However, some interesting differences exist between the point disassembly output and the ground truth listings. One such find is a discrepancy in the source and ROM

[2]The only change to DASM is incrementing the maximum hexadecimal bytes shown in the listing, as the default value of 4 does not include all the bytes if the line has more than 4 bytes of data.

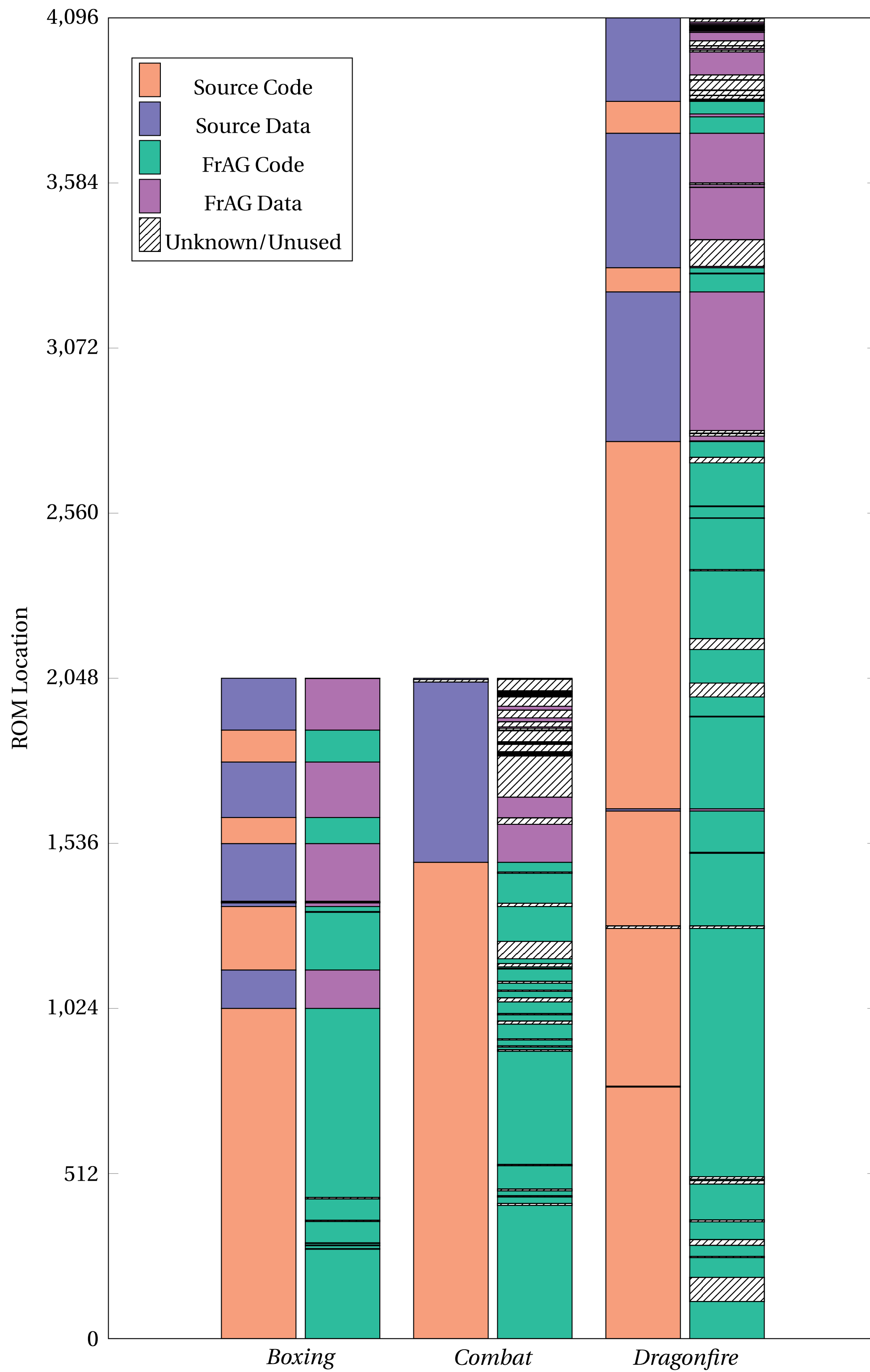


Figure 5.2: ROM location covered for *Boxing*, *Combat* and *Dragonfire*.

for *Combat* as shown in Listing 5.1 and Listing 5.2

```
1  03bb: lda   $3c,x
2  0446: lda   $30,x
3  048b: lda   $34,x
4  04d6: lda   $32,x
5  04da: lda   $37
```

Listing 5.1: Point Disassembler

```
1  03bb: lda   $0c,x
2  0446: lda   $00,x
3  048b: lda   $04,x
4  04d6: lda   $02,x
5  04da: lda   $07
```

Listing 5.2: Listing by DASM

The instructions are the same, but the operands all have a difference of $30. It is necessary to understand the hardware concept of mirroring to understand why the operands are different and why it does not make a difference in the game. Mirroring is when memory accessible by a space appears in different memory locations simultaneously, caused by the incomplete hardware decoding of addresses. Since the higher 3 address pins on a 6507 are missing, it cannot detect these bits of a 16-bit address and they are therefore ignored. For example, any address of the form $1xxx will address the same memory as $3xxx, $5xxx, $7xxx, $9xxx, $Bxxx, $Dxxx, and $Fxxx. Furthermore, both the television interface adapter (TIA) and RAM-I/O-Timer (RIOT) addresses are mirrored as well.

The mirrored addresses are used here as the DASM listing uses $0x for the TIA addresses. $3x is the mirrored address location which the ROM uses, which is surprising as there is no obvious reason to use a mirrored address here.

The point disassembler disassembles code-code overlap as data bytes, as it is often easier to read the different addresses the code can jump to. As

mentioned in §5.1, *Dragonfire* was chosen to test the game with code-code overlap. We notice that the point disassembler does indeed disassemble it correctly. This sequence, shown in Listing 5.3, implements a delay and consumes different cycles depending on where the code jumped to in the sequence. Due to code-code and code-data overlaps, a perfect disassembly similar to the source is rarely possible as there are at least two different ways to disassemble the address, either as data or code. The discrepancies in the confusion matrix are due to a block of instructions read but never executed, leading the point disassembler to disassemble it as data.

```
1  1664:  a9   .byte $a9  ; executed
2  1665:  a9   .byte $a9  ; executed
3  1666:  a9   .byte $a9  ; executed
4  1667:  a9   .byte $a9  ; executed
5  1668:  ad   .byte $ad  ; executed
6  1669:  a5   .byte $a5  ; executed
7  166a:  ea   .byte $ea
```

Listing 5.3: *Dragonfire* code-code overlap sequence disassembled by the point disassembler

As an example of code-data overlap, *River Raid* has several instances of code being read as data. Listing 5.4 shows a section of the disassembled ROM using the point disassembler where several instructions are read as data. What is interesting is why the code is being read as data in this case. All of the reads in the block shown are read by the same instruction. When an explosion occurs in the game, its colours are retrieved from a colour table. However, there is no colour data for explosions in the table, and to

get random explosion colours, bytes after the colour table are read. Since the bytes after the colour table are instructions, these are read as data for random explosion colours.

```
1  ff5c: 84 f9   sty  $f9   ; PC also read as data. ff5d read as data
2  ff5e: 85 f8   sta  $f8   ; PC also read as data. ff5f read as data
3  ff60: a5 8e   lda  $8e
4  ff62: 45 93   eor  $93
5  ff64: 29 04   and  #$04
6  ff66: f0 24   beq  $24   ; PC also read as data. ff67 read as data
7  ff68: a5 8a   lda  $8a   ; PC also read as data. ff69 read as data
8  ff6a: f0 09   beq  $09   ; PC also read as data. ff6b read as data
9  ff6c: 24 93   bit  $93   ; PC also read as data. ff6d read as data
10 ff6e: 10 1c   bpl  $1c   ; PC also read as data. ff6f read as data
11 ff70: 20 fad3 jsr $fad3 ; PC also read as data. ff71 read as data
       ff72 read as data
12 ff73: d0 17   bne  $17   ; PC also read as data. ff74 read as data
13 ff75: 24 93   bit  $93   ; PC also read as data.
```

Listing 5.4: *River Raid* disassembled by the point disassembler

5.2.3 Memory Analysis

Modern games require enormous amounts of memory in gigabytes, but the Atari 2600 only had 128 bytes, and programmers had to utilize it effectively. Data on read and write operations were collected using FrAG to understand how each random access memory (RAM) location was used and how frequently it was used. By analyzing memory usage and how the values change throughout a game, we can gain insights into how these games were

		Point Disassembly		
		C	D	U
Source	C	1384	0	17
	D	0	644	0
	U	0	0	3

(a) *Boxing*

		Point Disassembly		
		C	D	U
Source	C	1326	0	151
	D	0	231	332
	U	0	0	8

(b) *Combat*

		Point Disassembly		
		C	D	U
Source	C	2699	9	230
	D	3	913	234
	U	0	0	8

(c) *Dragonfire*

		Point Disassembly		
		C	D	U
Source	C	2901	0	129
	D	0	863	202
	U	0	0	1

(d) *River Raid*

		Point Disassembly		
		C	D	U
Source	C	1559	0	60
	D	0	419	9
	U	0	0	0

(e) *Kaboom!*

		Point Disassembly		
		C	D	U
Source	C	4051	0	1753
	D	0	1195	1191
	U	0	0	2

(f) *Montezuma's Revenge*

		Point Disassembly		
		C	D	U
Source	C	9581	3	106
	D	0	6164	530
	U	0	0	0

(g) *Solaris*

		Point Disassembly		
		C	D	U
Source	C	4848	0	590
	D	6	2243	503
	U	0	0	0

(h) *Moonsweeper*

C = Code (bytes, including operands), *D* = Data (bytes), *U* = Unused/Unknown bytes

Table 5.5: Confusion Matrices.

designed and provide valuable information for reverse engineering.

A directed graph for each byte in RAM was built to analyze how a RAM location was used and how it changed over a game as described in §4.2 of Chapter 4.[3] This provided over 2000 graphs for a single RAM location. To get the most coverage, the graph with the most edges was considered for each RAM location as it denotes that the RAM location was used more during that particular run of the game. Self-edges or self-loops were removed from these graphs as they are most often initialization edges that provide false positives.

For example, consider the code given in Listing 5.5 and the graph shown in Figure 5.3 for a RAM location cycling between zero to four. The example code zeroes out all RAM locations and initializes the RAM location to four. After this, the game is played until the game is over at which point the graph is saved (by the user as described in §4.2 of Chapter 4).

When MAME starts emulating, the RAM locations are memset to zero (already zeroed out) and when the code zeroes it out again, a self-loop is added to the graph. Regardless of the location being zeroed out by MAME, if a RAM location is updated every frame but there is no change to the value being written, all the nodes in the graph would have self-loops. The write handler that builds this graph (see Appendix B.2 in Appendix B) reads the current value of the RAM location and adds an edge from the current value to the value being written resulting in the self-loop being added to the graph. These self-loops result in false positives as the analysis scripts that I developed using the NetworkX library to perform basic statistics and

[3] Due to a bug, the reset was captured for each game as the game was reset and the RAM graphs were saved after the reset instead of saving them before the reset.

analyses searches for cycles in the graph to check if RAM locations cycle through values in a game. Since the algorithm that NetworkX uses considers the self-loop to be a cycle, this results in false positives.

```
1  f000:       LDA #0     ;
2  f002:       LDY #$80   ; RAM location to zero out
3  f004: Loop  STA $7f,Y  ; Zero out location = $7f + Y
4  f006:       DEY        ; Decrement Y
5  f007:       BEQ End    ; Branch to End if Y == 0
6  f009:       JMP Loop   ; Jump to Loop
7  f00b: End   LDA #$04   ; Load 4
8  f00d:       STA CYCLE  ; CYCLE = RAM cycling between 0-4
```

Listing 5.5: Example code for RAM graph.

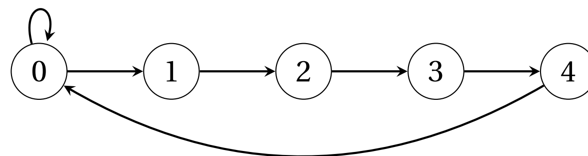
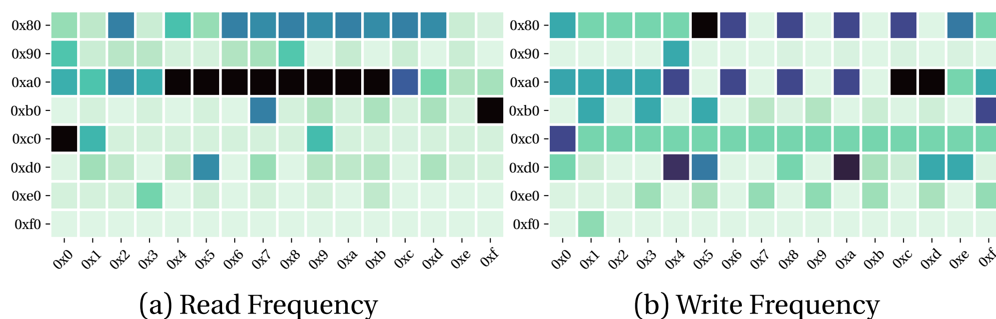


Figure 5.3: An example graph of a RAM location cycling between values.

To analyze the RAM graphs, we look at the least and most used RAM locations. Heatmaps for the frequency of reads and writes for *Boxing* are shown in Figure 5.4 (see Appendix C for memory heatmaps of all games in the suite). To determine how often a RAM location is used, the read and write frequencies have been combined to produce a combined heatmap. The heatmaps are constructed from data using only one game instance and are not merged. These heatmaps combined with the directed graphs can aid in reverse engineering a ROM.



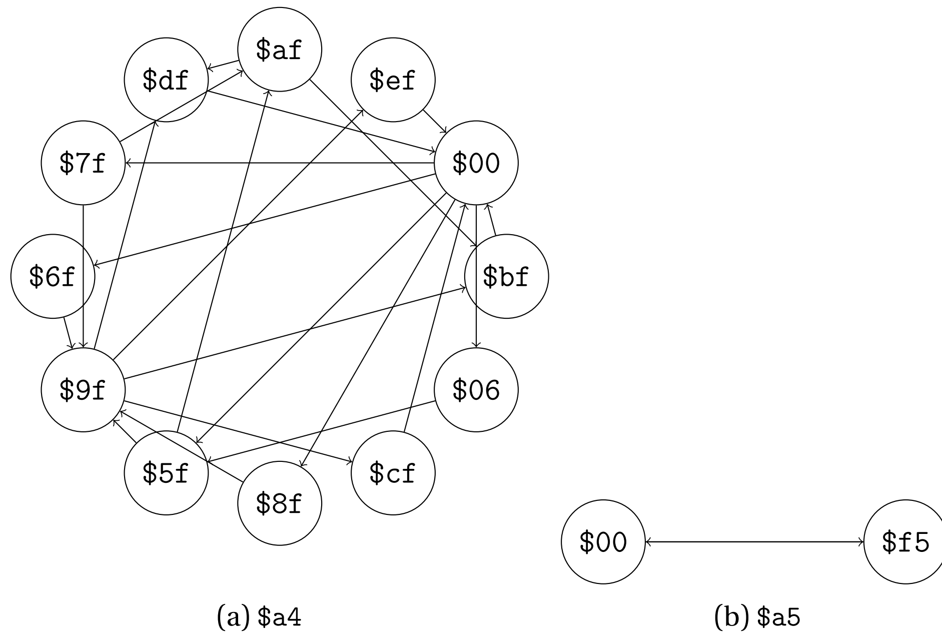
(a) Read Frequency (b) Write Frequency

Figure 5.4: RAM usage heatmaps for *Boxing*.

Looking at the heatmaps for *Boxing*, we notice that the locations `$a5`, `$a7`, `$a9` and `$ab` are read from frequently but rarely written to. And similarly, `$ac` and `$ad` are written to frequently but not read from often. This pattern might indicate that these locations are used as pointers to data in ROM where only the least significant byte of the pointer needs to be updated. Analyzing the RAM graphs for locations `$a4` to `$a5` show that these locations did not hold many values throughout the game. `$a5` only fluctuates between two values, `$f5` and `$00` while the RAM location `$a4` takes on a range of different values. These graphs are shown in Figures 5.5 to 5.8. From these graphs, we see that `$a4`, `$a6`, `$a8` and `$aa` hold a small set of values that are similar and locations `$a5`, `$a7`, `$a9` and `$ab` cycle between two values. As the Atari 2600 is a little-endian machine, the 16-bit address in `$a4` and `$a5` fluctuates between `$f5xx`, where `xx` is the value of `$a4`. The zero here is the value of `$a5` at the beginning of the game as a result of zeroing out memory (see Footnote 3).[4] After initialization, the value held by the RAM location always stays constant until a reset occurs. The 16-bit addresses at `$a4`, `$a6`, `$a8` and `$0a` always fluctuate between addresses pointing to data in ROM. Analyzing the ground truth available confirms that the RAM locations from

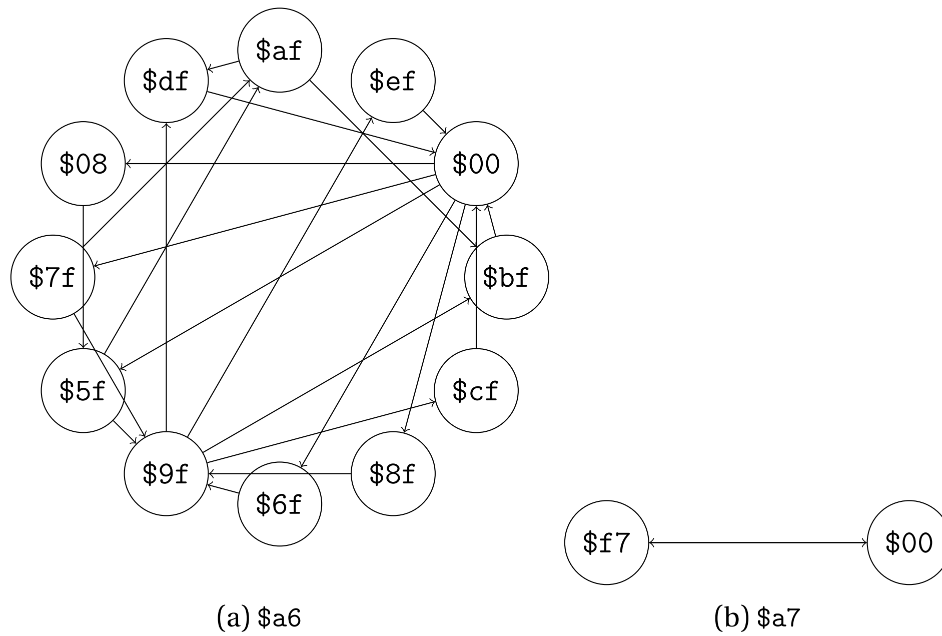
[4]This was manually checked and it indeed is due to initialization.

`$a4`–`$ab` store 16-bit pointers to the boxer's graphics in ROM.



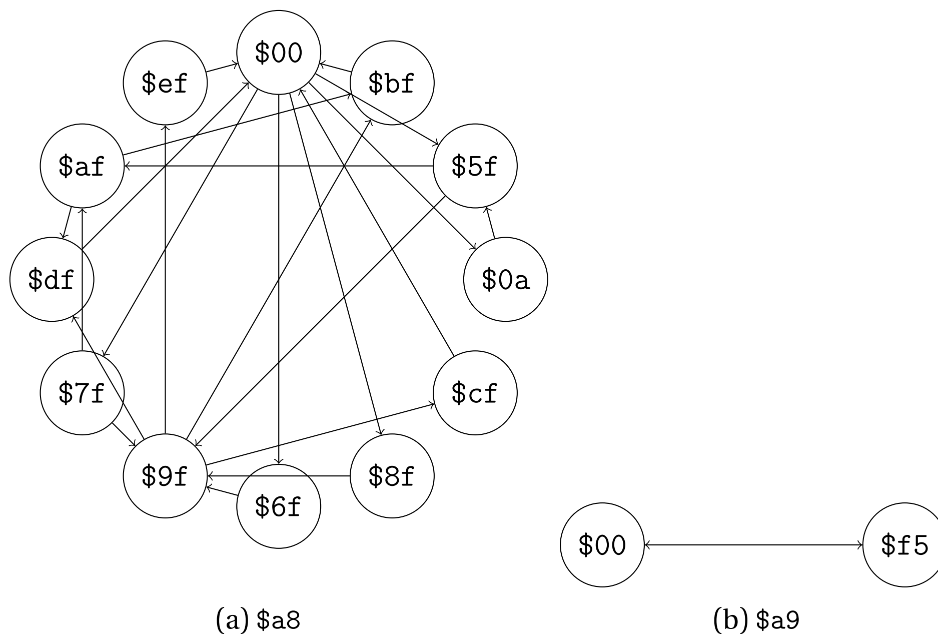
(a) `$a4` (b) `$a5`

Figure 5.5: *Boxing* RAM graphs for locations `$a4` and `$a5`.



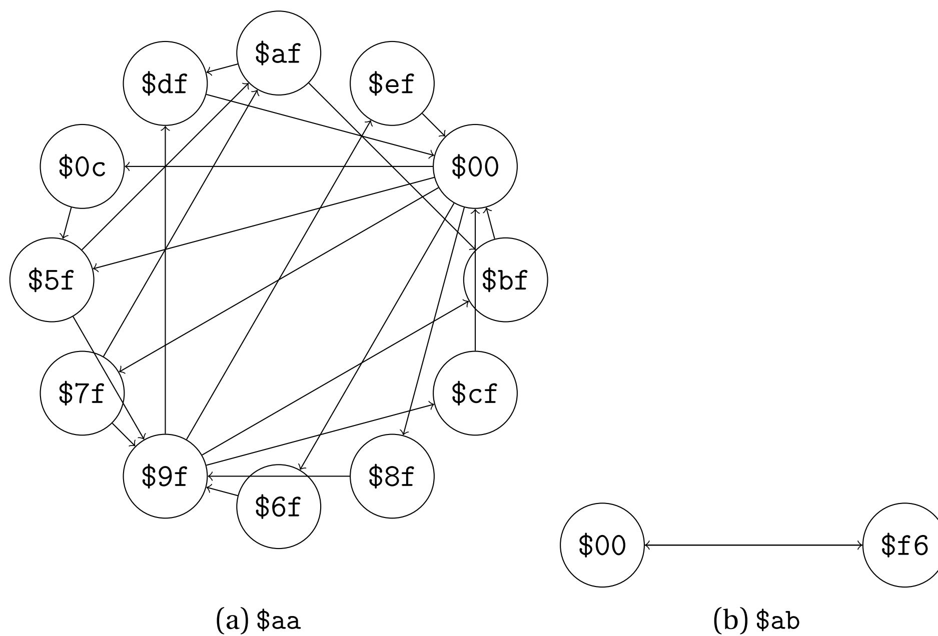
(a) `$a6` (b) `$a7`

Figure 5.6: *Boxing* RAM graphs for locations `$a6` and `$a7`.



(a) `$a8` (b) `$a9`

Figure 5.7: *Boxing* RAM graphs for locations `$a8` and `$a9`.



(a) `$aa` (b) `$ab`

Figure 5.8: *Boxing* RAM graphs for locations `$aa` and `$ab`.

To understand why some locations are written to often but rarely read from, we look at the locations `$ac` and `$ad`. These locations do not have any visible usage pattern, and thus the ground truth was analyzed. They are used as temporary variables to store values. The absence of a pattern might also help in reverse engineering a ROM. A few bytes in RAM are never read from or written to, indicating that those locations are never used. The ground truth was analyzed to confirm whether this is true, and the locations are never used. We also notice locations written to but never read from, which indicates an initialization loop (using the directed graphs). When a new game is started, programmers will often zero out all memory locations. This is observed here as *Boxing* never uses certain locations but will zero them out before starting a new game.

The frequency of bank switching can provide insight into how a game's code and data are organized in ROM. Table 5.6 shows the frequency each bank was switched to in the bank switching games tested. We observe that *Moonsweeper* accesses all banks equally. *Moonsweeper* was manually investigated as it was suspicious for a game to access both banks equally. Upon investigation, all F8 bank switching games produce a similar result leading me to believe there is a bug in MAME. Therefore, the numbers for F8 bank switches reported by MAME might not be correct. *Montezuma's Revenge* and *Solaris* use certain banks more than others. Table 5.7 shows the ROM breakdown for each bank, and we observe that in *Montezuma's Revenge*, the pages that were switched to the most were covered more than the others.

	Access Count
Moonsweeper	
Bank 0	15208793
Bank 1	15208793
Montezuma's Revenge	
Page 0	17870368
Page 1	15073484
Page 2	28417239
Page 3	32943832
Page 4	15073474
Page 5	15073474
Page 6	28417229
Solaris	
Bank 0	39885483
Bank 1	56576039
Bank 2	40095558
Bank 3	41807768

Table 5.6: Number of times a bank was accessed.

5.2.4 Execution Sequence Analyses

Analyzing a program's code execution flow is at the core of all analysis methods as it provides insights into the design of a program, how it is structured, and how similar it is to other versions of a program. Execution flows help us understand the program logic, and how often a path is taken or executed provides insight into how important the execution path might be. For bank switching games, an execution flow might help to understand how the program is structured and how different banks interact.

Moreover, using the execution sequence graph, we can infer whether or not a relative branch is used as a space optimization technique. As discussed in Section 4.3 in Chapter 4, relative branches could be used as space optimization saving bytes rather than using absolute addresses to jump or

	Ground Truth		Point Disassembly	
	# of Instructions	**# of Data (bytes)**	**# of Instructions**	**# of Data (bytes)**
Moonsweeper				
Bank 0	1002	2313	932	1915
Bank 1	1888	439	1655	328
Montezuma's Revenge				
Page 0	494	18	447	14
Page 1	509	0	419	0
Page 2	311	400	287	66
Page 3	388	349	302	213
Page 4	95	844	94	740
Page 5	523	0	290	0
Page 6	492	47	352	6
Page 7	152	730	152	369
Solaris				
Bank 0	1448	1246	1442	1174
Bank 1	1522	1173	1497	940
Bank 2	912	2313	905	2310
Bank 3	1075	1962	1060	1843

Table 5.7: Bank breakdown in bank switching games.

jump and return. Using the execution graph, we notice several branches that are always executed. Table 5.8 shows the number of relative branches in the disassembly and how many are always taken. This information can help reverse engineers understand the control flow of a program, how different sections of code are linked together and to identify subroutines in the code, without access to source code. Moreover, this information can also help reverse engineers deobfuscate code as it reveals the logic and control flow hidden by obfuscation.

Figure 5.9 shows heatmaps of selected games depicting areas of ROM which are executed the most. We can use this data to investigate what these

Game	# of relative branches	# of branches always taken
Boxing	108	9
Combat	89	23
River Raid	278	56
Dragonfire	233	39
Kaboom!	129	15
Moonsweeper	429	58
Montezuma's Revenge	414	90
Solaris	618	22

Table 5.8: Number of relative branches that were always taken.

frequently executed code sections do and why they are executed frequently.

In the case of *River Raid*, the most executed instructions wait for the RIOT timer and execute the main display kernel. The display kernel writes to the TIA registers as the raster beam moves, and executing game logic is usually done during VBLANK when the raster beam travels back to the top of the television. However, *River Raid* spends most of its time waiting for the raster beam. It is not just *River Raid* but all games in the suite that execute these exact two instructions more frequently than any other in its ROM. As the display kernel drives the game and since the timing was essential to display the video, it is unsurprising to see all games syncing with the raster beam.

While the display kernel is captured by the PC handler (see Appendix B.3), it is also identified in the disassembly, as each game needs to implement its display kernel. The kernel was identified by recording all addresses executed

when the raster beam drew in a visible area.[5] However, a game does not necessarily have to write anything to the TIA registers. *River Raid* uses the first few frames to initialize game data while the raster beam is active. This gives us a display kernel detected by FrAG with a superset of addresses that are actually part of the kernel.

5.3 Comparing Learning Methods

All games were run for 15 million frames with random actions and different RL algorithms to examine how the AI choice impacts coverage, if at all. The coverage for random actions and three different learning methods – advantage actor critic (A2C), proximal policy optimization (PPO) and DQN – is shown in Table 5.9. The random results and the DQN algorithm provide similar coverage for a few games. However, the DQN algorithm does provide us with more coverage overall. The coverage might increase if the AI was left to train longer as [5] trains the games for close to 200 million frames. This might indicate that an AI that can learn is beneficial compared to fuzzing, solely based on random inputs.

Examining how coverage increases over time and if the AI performance impacts coverage, all games were run once more using DQN for 15 M frames with the execution information saved for each game as the previous run from which most of the results have been discussed did not save the execution for each game but instead overwrote the saved execution data. The number of games played by all four instances for this run are shown in Table 5.10.

[5] The handler currently only captures PCs between scanlines 41 and 232. These values were obtained from Wright [63] and have only been tested with the NTSC emulation of the Atari 2600.

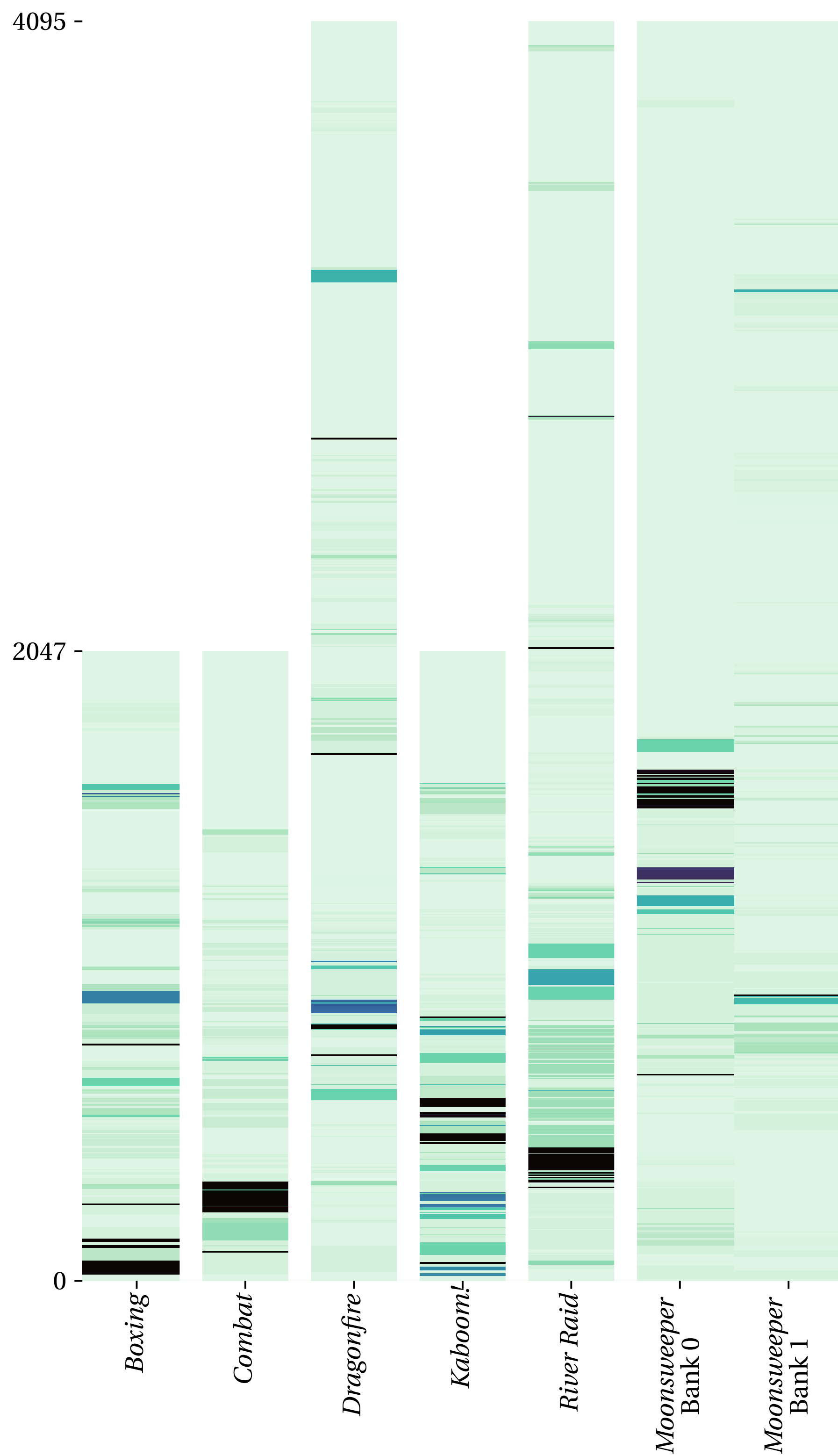


Figure 5.9: PC heatmaps depicting most executed areas in ROM. Darker colours depict higher frequency.

	#Code (Ins.)	# Data (bytes)	Learning Method			
			Random	DQN	A2C	PPO
Boxing	747	647				
#Code (Ins.)			734	**738**	734	734
# Data (Bytes)			637	**644**	637	637
Total Bytes			2013	**2028**	2013	2013
Combat	770	571				
#Code (Ins.)			696	**696**	696	696
# Data (Bytes)			208	**232**	228	232
Total Bytes			1534	**1558**	1554	1558
River Raid	1588	1066				
#Code (Ins.)			1500	**1521**	**1521**	**1521**
# Data (Bytes)			822	**863**	**863**	**863**
Total Bytes			3683	**3764**	**3764**	**3764**
Dragonfire	1574	1158				
#Code (Ins.)			974	**1446**	978	978
# Data (Bytes)			353	**922**	353	353
Total Bytes			2173	**3624**	2180	2180
Kaboom!	847	428				
#Code (Ins.)			800	**815**	805	802
# Data (Bytes)			414	**420**	417	417
Total Bytes			1948	**1979**	1961	1955
Moonsweeper	2890	2752				
#Code (Ins.)			2548	2587	2545	**2569**
# Data (Bytes)			2234	2243	2238	**2294**
Total Bytes			7018	7097	7016	**7117**
Montezuma's Revenge	2914	2388				
#Code (Ins.)			1143	**2343**	1958	1991
# Data (Bytes)			1804	**1408**	1234	1238
Total Bytes			4745	**6062**	5129	5199
Solaris	4957	6694				
#Code (Ins.)			4561	**4904**	4578	4892
# Data (Bytes)			5709	**6167**	5818	6145
Total Bytes			14702	**15748**	14754	15701

Table 5.9: ROM coverage comparing learning methods. Bolded entries correspond to results with the largest number of total bytes covered and “Ins.” refers to instructions.

Since not each instance played the same number of games, the instance that played the most number of games was analyzed to check how coverage increases over time. A graph depicting how coverage increases over time is shown in Figure 5.10 (see Appendix D for all coverage increases by 0.01%). We notice that most games quickly reach the maximum coverage and it is redundant to train the AI any longer. We observe that the games indeed do reach the maximum coverage early and none of the games other than *Solaris*, *Montezuma's Revenge* and *Dragonfire* require extended training.

Game	**# of Games**
Boxing	8452
Combat	7352
Kaboom!	21068
Dragonfire	12287
River Raid	19544
Moonsweeper	9589
Montezuma's Revenge	41
Solaris	2059

Table 5.10: Number of games played by all four instances using DQN (15 M frames).

To determine if the AI performance impacts coverage, we look at the average reward earned by the AI after the first game, the largest coverage jump and the last game after which maximum coverage was obtained. This is shown in Table 5.11. The reward here is the difference between scores for single player games and the difference between the two players' score when multiple players are active. This is the reason why *Boxing* has a negative reward. *Combat* might have a negative reward as well if the second player

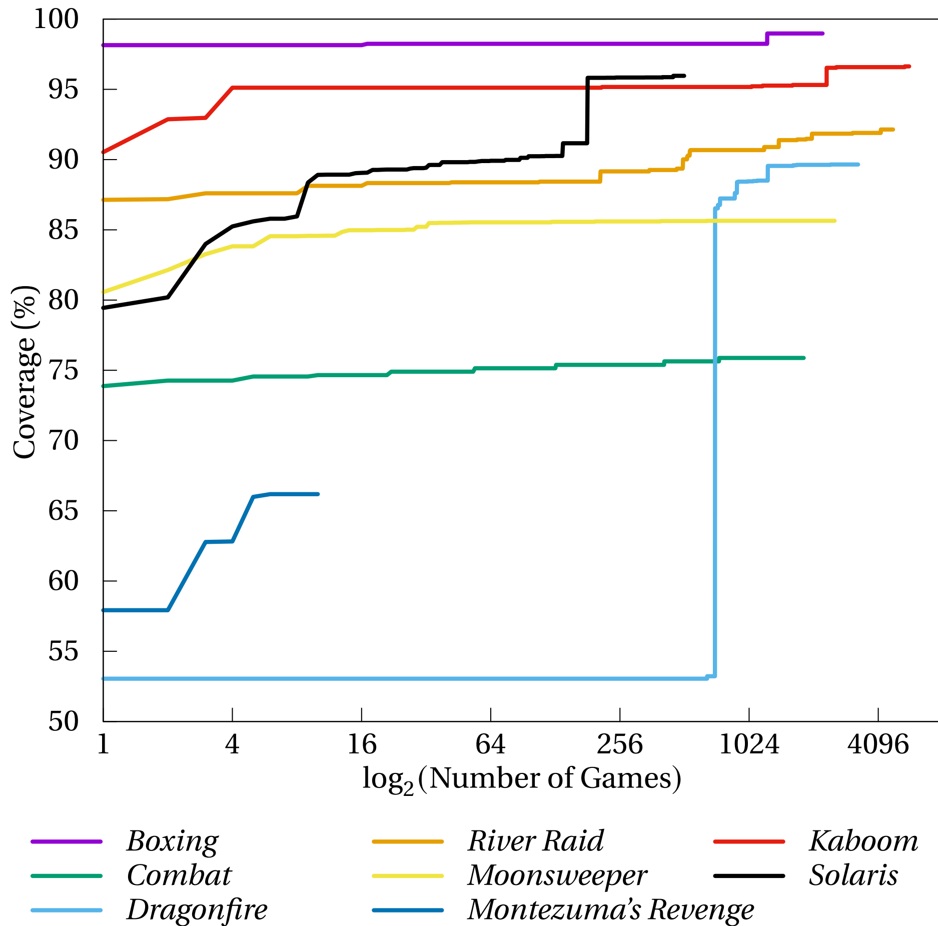


Figure 5.10: Coverage increase as AI trains.

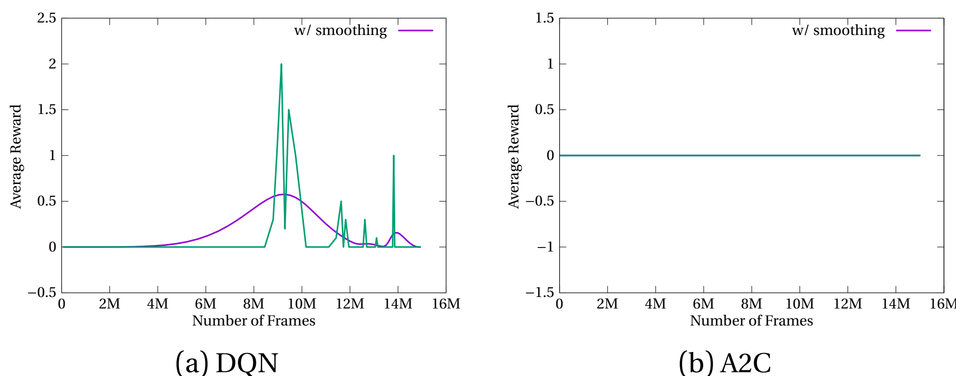
was also actively playing. The reward helps the AI know whether the action taken was desirable. We notice that the AI did not necessarily perform well for most of the jumps observed. However, with *Dragonfire*, we notice that the coverage jumps are related to the reward of the AI. Games such as *Boxing* and *Combat* effectively do not need any AI as they are small 2 KiB games without multiple screens and are highly optimized. As an example, we saw a coverage jump in *Combat* when cycling through game variants. This is because of the multiple screens *Combat* has that do not appear in a single game. However, *Dragonfire* has multiple screens in one gameplay and when

the AI manages to get to the next screen, we see a greater coverage. Thus, with the data obtained, we can say that the AI helps and does not help at the same time and depends on the type of game played.

	# of Games	Reward	Coverage (%)
Boxing (single-player)			
	1	-1.0	98.14
	17	-1.0	98.24
	1244	18.0	98.97
Combat (two-player)			
	1	2.0	73.88
	2	3.0	74.27
	742	8.0	75.88
Kaboom! (single-player)			
	1	3.0	90.53
	2	9.0	92.87
	5443	2431.0	96.63
Dragonfire (single-player)			
	1	0.0	53.05
	711	90.0	86.52
	2458	20.0	89.65
River Raid (single-player)			
	1	1670.0	87.13
	208	2480.0	89.16
	4210	6480.0	92.14
Moonsweeper (single-player)			
	1	90.0	80.57
	2	100.0	82.14
	650	390.0	85.64
Montezuma's Revenge (single-player)			
	1	0.0	57.92
	3	0.0	62.78
	6	0.0	66.19
Solaris (single-player)			
	1	1440.0	79.44
	181	2960.0	95.82
	454	2960.0	95.96

Table 5.11: Reward for coverage jumps shown in Figure 5.10.

As an experiment, the games were also trained using the A2C [75] and PPO [76] learning algorithms. While we do not see any major impact on coverage, *Dragonfire* is the only game where DQN outperforms the other learning algorithms. Looking at the training graphs how the average reward changes over time during training shown in Figure 5.11 (see Appendix E for AI progress over 15 M frames for all games in the suite), we see that A2C does not earn any reward whereas DQN makes some progress, indicating it moved to the next screen.



(a) DQN (b) A2C

Figure 5.11: Average reward over session for *Dragonfire*.

5.4 Summary

This chapter discussed the evaluation of FrAG using the implemented Atari 2600 interface and different RL algorithms. It showed that all analyses implemented work, how one can leverage FrAG's features to analyze game implementation, and how the data obtained can help reverse engineering a game. Some findings and oddities in results were also discussed, which gave insight into how programmers used code and data.

The next chapter discusses some of the future work and concludes this thesis.

Chapter 6

Future Work and Conclusion

In this thesis, I presented FrAG, a framework for analysis of games using artificial intelligence (AI) to study game implementation and assist in reverse engineering of game read-only memories (ROMs) using Multiple Arcade Machine Emulator (MAME) and MAME's features as described in Chapters 3 and 4. This chapter outlines some future work and concludes the thesis.

6.1 Future Work

FrAG has only been tested with the Atari 2600 so far; although a proof of concept for the Fairchild Channel F console exists, it is yet to be extensively tested. It would be interesting to see the ROM coverage obtained for games using different bank switching methods and peripherals available for the console, as currently, the implemented interface only supports a subset of the bank switching methods supported by MAME. It would also be interesting to use an *autoencoder* as described in Sygnowski and Michalewski [77] to identify patterns in random access memory (RAM) usage. Currently,

the Atari 2600 interface dumps the data collected; however, it would be best to build the point disassembler introduced in Chapter 4 into FrAG, which will automatically disassemble with the execution data collected. As MAME has a built-in disassembler and a function to disassemble a specific address internally, this should be possible without any change to the existing code other than a simple function call. The point disassembler can also be combined with a recursive disassembler to test if a better ROM disassembly is possible.

One limitation of this work is the time it takes FrAG to run each game for 15M frames. As a result, the data reported in Chapter 5 are based on single runs as opposed to multiple runs whose results are averaged. As future work, this could be addressed by running each game multiple times and averaging the results produced.

6.1.1 Domain Specific Language

To help with portability and scalability, a domain specific language (DSL), which translates a configuration file for a game to a C++ and Python interface could be implemented. This would make it easier to expand the framework as, currently, basic knowledge of C++ and pybind11 is required to add support for a new console or a game. Moreover, introducing a DSL makes adding support for a game less time-consuming by reducing the manual work necessary. Ella Tomlinson, an undergraduate student has already implemented a miniature DSL to add support for Atari 2600 games. An example DSL configuration file for *Ski Hunt* (1983) is shown in Listing 6.1 which is converted to a C++ class, ready to be dropped in to FrAG.

```
NAME Ski Hunt
PADDLES false
# Player 1 score
P1 0xA8 0xA9 0xAA 0 0 0
# NTSC and PAL names in MAME
MAME skihunt skihuntp
# Starting Actions
START FIRE
#Valid Actions
VALID NOOP FIRE UP DOWN LEFT RIGHT UPRIGHT UPLEFT DOWNRIGHT DOWNLEFT
# Lives location in RAM and initial value
LIVES 0xA7 2
# Terminal condition
TERM 0xA7 == 0
```

Listing 6.1: Example DSL configuration input file for *Ski Hunt*. With permission from Ella Tomlinson.

6.1.2 Increasing Coverage

One of the most critical future improvements is incrementing the coverage of the ROM as much as possible, leading to better results across all areas. As described in Section 5.2, FrAG produces a 99% coverage of *Boxing* with some regions of ROM manually analyzed to check why those sections of code/data were not covered, giving some areas of the framework to improve upon. Cycling through all game variants and game difficulties and running a few frames with no actions in all these changes before handing control over to the AI would ensure that the corresponding code/data sections get

covered.

6.1.3 Analysis

Several papers have introduced analysis tools such as S^{4}LVE and *Mappyland* [56–58]. The tool *Mappyland* introduced in Osborn et al. [57] produces a map of the rooms in an NES game which can then be used for procedural content generation using machine learning (PCGML) to generate more content. Integrating Mappyland's core into FrAG could enable one to map a game's level for any retrogame console. Moreover, building the proposed pipeline for Automated Game Design Learning (AGDL), a field of research proposed in Osborn, Summerville, and Mateas [59], could be used for code studies, game design and PCGML.

6.2 Conclusion

In this thesis, I presented FrAG, a framework for automated dynamic analysis of games AI built into MAME, an emulator capable of emulating over 40,000 platforms. The design of FrAG allows it to be portable to different platforms and provides a platform for training AI agents. The evaluation of the framework's efficacy has been focused on the Atari 2600 console and its games. The goal of this work is to build a framework capable of analyzing a plethora of games at scale for different platforms using AI, treating the AI as a black box, eliminating human interaction with gameplay, and aiding in reverse engineering game ROMs for which the source has been lost.

As described in Chapter 5, we obtain good coverage of the ROM for most games in the test suite and perform analysis using the gathered execution

data using an AI agent playing the game without tuning any hyperparameters of the reinforcement learning (RL) algorithm. From the data obtained, it was shown how one could uncover programming practices and how the data can aid in reverse engineering a ROM. FrAG's analysis features and the point disassembler are helpful in reverse engineering and analyzing game implementation by eliminating any human interaction needed with the gameplay. Moreover, integrating existing research and tools with FrAG can make it an analysis framework for games and other programs, as MAME also emulates computers and not just game consoles. While some manual work and basic knowledge of C++ is required to port FrAG to a new system, implementing a DSL, as described above in the previous section, can remove this requirement.

Since FrAG provides a platform for training AI agents, it can also be used as a test bed for RL methods with not just Atari 2600 games but any system that MAME supports. However, the primary focus of this work is to aid in reverse engineering game ROMs and game implementation studies.

Working on this project was equally fun and not fun. Enormous amount of time went into studying MAME's code and trying to make the framework as user friendly and portable as possible took much effort. However, through all this, I learnt new concepts which facinated me and also discovered many new features of C++. There were several earlier versions of the framework which I did not feel confident in and as such kept redoing it until I was satisfied with what was produced. While there are several changes I would still like to make, these are minor and is not worth spending too much time on. Perhaps the most frustrating time during the project was when the read handler was built and tested. Since the emulated central processing

unit (CPU) does not differentiate between instruction fetches and memory reads, countless days of testing went into the read handler accounting for every possible edge case. The other part of the project included the Python wrapper. Since the AI scripts were written in Python, several runs during testing failed as the error would not surface until the framework was about to exit. Overall, I enjoyed the project and I am also satisfied with the produced framework and its results.

Bibliography

[1] Vít Šisler, Jaroslav Švelch, and Josef Šlerka. "Global Digital Culture| Video Games and the Asymmetry of Global Cultural Flows: The Game Industry and Game Culture in Iran and the Czech Republic". In: *International Journal of Communication* 11.0 (2017). ISSN: 1932-8036. URL: https://ijoc.org/index.php/ijoc/article/view/6200.

[2] Philipp Moll, Mathias Lux, Sebastian Theuermann, and Hermann Hellwagner. "A Network Traffic and Player Movement Model to Improve Networking for Competitive Online Games". In: *Proceedings of the 16th Annual Workshop on Network and Systems Support for Games*. NetGames '18. Amsterdam, The Netherlands: IEEE Press, 2018. ISBN: 9781538660980.

[3] Emmanuel Tsekleves, Alyson Warland, Cherry Kilbride, Ioannis Paraskevopoulos, and Dionysios Skordoulis. "The Use of the Nintendo Wii in Motor Rehabilitation for Virtual Reality Interventions: A Literature Review". In: *Virtual, Augmented Reality and Serious Games for Healthcare 1*. Ed. by Minhua Ma, Lakhmi C. Jain, and Paul Anderson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 321–344. ISBN: 978-3-642-54816-1. DOI: 10.1007/978-3-642-54816-1_17.

[4] Mansoor Ali, Faisal Naeem, Georges Kaddoum, and Ekram Hossain. *Metaverse Communications, Networking, Security, and Applications: Research Issues, State-of-the-Art, and Future Directions.* 2023. arXiv: 2212.13993 [cs.CR].

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. DOI: 10.1038/nature14236.

[6] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. "Agent57: Outperforming the Atari Human Benchmark". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event.* Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 507–517. URL: http://proceedings.mlr.press/v119/badia20a.html.

[7] Gundolf S. Freyermuth. *Games, game design, game studies : an introduction.* Media studies. Bielefeld: Transcript, 2015. ISBN: 3839429838.

[8] Andrew Reinhard. *Archaeogaming: an introduction to archaeology in and of video games.* 1st ed. New York, NY: Berghahn Books, 2018. ISBN: 1785338749.

[9] Ian Bogost and Nick Montfort. "Platform Studies: Frequently Questioned Answers". In: (2009).

[10] John Aycock. *Retrogame Archeology - Exploring Old Computer Games*. Springer, 2016. ISBN: 978-3-319-30002-3. DOI: 10.1007/978-3-319-30004-7.

[11] R. N. Horspool and N. Marovac. “An Approach to the Problem of Detranslation of Computer Programs”. In: *The Computer Journal* 23.3 (Aug. 1980), pp. 223–229. ISSN: 0010-4620. DOI: 10.1093/comjnl/23.3.223.

[12] Shankar Ganesh, Ana Bindiu, Rachel Ralph, and John Aycock. “FrAG: AI-Driven Analysis of Retrogames”. Conference Presentation. History of Games. 2022.

[13] Matthew Michaud, Shankar Ganesh, John Aycock, Katie Biittner, and Carl Therrien. “Replaying Early Videogame History: The Channel F”. Conference Presentation. The Interactive Pasts Conference 3. 2023.

[14] Shankar Ganesh, John Aycock, and Katie Biittner. “FrAG: A Framework for the Analysis of Games”. In: *2023 IEEE Conference on Games (CoG)*. 2023, pp. 1–4. DOI: 10.1109/CoG57401.2023.10333209.

[15] Peter Corcoran and Joe Decuir. “Champions in Our Midst: Game on!” In: *IEEE Consumer Electronics Magazine* 4.3 (2015), pp. 59–66. DOI: 10.1109/MCE.2015.2421572.

[16] Helen Sheumaker. *Artifacts from Modern America*. Santa Barbara, UNITED STATES: ABC-CLIO, LLC, 2017. ISBN: 978-1-4408-4683-0.

[17] Nick Montfort and Ian Bogost. *Racing the Beam: The Atari Video Computer System*. MIT Press, 2009. ISBN: 9780262012577.

[18] Tristan Donovan. *Replay : The History of Video Games*. East Sussex, England: Yellow Ant, 2010. ISBN: 9780956507204.

[19] Albert Yarusso Joe Grand. *Game Console Hacking: Xbox, PlayStation, Nintendo, Game Boy, Atari and Sega*. 1st ed. Rockland: Elsevier Science, 2004, pp. 335–397. ISBN: 9780080532318.

[20] Michael Thomasson. “Video Game Cartridges: The History of Durable, Removable, and Portable Software”. In: *The Routledge Companion to Media Technology and Obsolescence*. 1st ed. Routledge, 2019, pp. 311–321. ISBN: 9781138216266.

[21] Robert N. Adams. *Circus Convoy is a New Atari 2600 Game from Audacity Games*. 2021. URL: https://techraptor.net/gaming/news/circus-convoy-is-new-atari-2600-game-from-audacity-games (visited on 10/24/2022).

[22] Mark J. P. Wolf. *Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming*. Santa Barbara: ABC-CLIO, LLC, 2021. ISBN: 1440870195.

[23] Ann Holms. *An Introduction to the Cathode Ray Tube*. Tech. rep. University of California, 2005. URL: http://holms.faculty.writing.ucsb.edu/2E_CRT_report_2.pdf (visited on 10/26/2022).

[24] University of Oxford Department of Physics. *Cathode ray tube*. URL: https://www2.physics.ox.ac.uk/accelerate/resources/demonstrations/cathode-ray-tube (visited on 10/26/2022).

[25] Tekla S. Perry and Paul Wallich. “Microprocessors: Design case history: The Atari Video Computer System: By omitting lots of hard-

ware, designers added flexibility and gave video-game programmers room to be creative”. In: *IEEE Spectrum* 20.3 (1983), pp. 45–51. DOI: 10.1109/MSPEC.1983.6369841.

[26] E. L. Dagless. “Processing Elements – PMS”. In: *The Microprocessor and its Application: An advanced course.* Ed. by D. Aspinall. Cambridge University Press, 1978, pp. 41–58. ISBN: 9780521222419.

[27] Atari Inc. “Bank Switchable Memory System”. U.S. pat. 4368515. May 7, 1981.

[28] Atari Inc. “Memory Cartridge For Video Game System”. U.S. pat. 4432067. Aug. 9, 1982.

[29] Kevin Horton. *Atari 2600 Mappers.* URL: http://blog.kevtris.org/blogfiles/Atari%202600%20Mappers.txt (visited on 10/26/2022).

[30] Andrew Davie. *Atari 2600 Programming for Newbies.* URL: https://www.randomterrain.com/atari-2600-memories-tutorial-andrew-davie-05.html (visited on 10/26/2022).

[31] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. “A Survey on Automated Dynamic Malware-Analysis Techniques and Tools”. In: *ACM Comput. Surv.* 44.2 (Mar. 2008). ISSN: 0360-0300. DOI: 10.1145/2089125.2089126.

[32] John Aycock and Katie Biittner. “Inspecting the Foundation of Mystery House”. In: *Journal of Contemporary Archaeology* 6.2 (Mar. 2020), pp. 183–205. DOI: 10.1558/jca.36745.

[33] John Aycock and Tara Copplestone. “Entombed: An archaeological examination of an Atari 2600 game”. In: *The Art, Science, and Engineering of Programming* 3.2 (Nov. 2018). DOI: 10.22152/programming-journal.org/2019/3/4.

[34] Paul Allen Newell, John Aycock, and Katie M. Biittner. “Still Entombed After All These Years: The continuing twists and turns of a maze game”. In: *Internet Archaeology* 59 (2022). ISSN: 1363-5387.

[35] John Aycock, Shankar Ganesh, Katie Biittner, Paul Allen Newell, and Carl Therrien. “The Sincerest Form of Flattery: Large-Scale Analysis of Code Re-Use in Atari 2600 Games”. In: *Proceedings of the 17th International Conference on the Foundations of Digital Games*. FDG ’22. Athens, Greece: Association for Computing Machinery, 2022. ISBN: 9781450397957. DOI: 10.1145/3555858.3555948.

[36] John Aycock. “Game Studies at Scale: Towards Facilitating Exploration of Game Corpora”. In: *Loading...* 10.17 (2017). URL: https://journals.sfu.ca/loading/index.php/loading/article/view/198.

[37] John Aycock and Katie Biittner. “LeGACy Code: Studying How (Amateur) Game Developers Used Graphic Adventure Creator”. In: *Proceedings of the 15th International Conference on the Foundations of Digital Games*. FDG ’20. Bugibba, Malta: Association for Computing Machinery, 2020. ISBN: 9781450388078. DOI: 10.1145/3402942.3402988.

[38] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. “Fuzzing: Challenges and Reflections”. In: *IEEE Software* 38.3 (2021), pp. 79–86. DOI: 10.1109/MS.2020.3016773.

[39] Vijay Ganesh, Tim Leek, and Martin Rinard. "Taint-based directed Whitebox Fuzzing". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[40] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "Model-Based Whitebox Fuzzing for Program Binaries". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE '16. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 543–553. ISBN: 9781450338455. DOI: 10.1145/2970276.2970316.

[41] R. Charles Bell. "Monte Carlo Debugging: A Brief Tutorial". In: *Commun. ACM* 26.2 (Feb. 1983), pp. 126–127. ISSN: 0001-0782. DOI: 10.1145/358024.358056.

[42] Eugene F. Grant and Rex Lardner. "The Talk of the Town: It". In: *The New Yorker* (Aug. 2, 1952), pp. 18–19. URL: https://www.newyorker.com/magazine/1952/08/02/it (visited on 10/10/2023).

[43] Geogios N. Yannakakis. "Game AI Revisited". In: *Proceedings of the 9th Conference on Computing Frontiers*. CF '12. Cagliari, Italy: Association for Computing Machinery, 2012, pp. 285–292. ISBN: 9781450312158. DOI: 10.1145/2212908.2212954.

[44] Ben Chan, Jörg Denzinger, Darryl Gates, Kevin Loose, and John Buchanan. "Evolutionary behavior testing of commercial computer games". In: *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*. Vol. 1. 2004, 125–132 Vol.1. DOI: 10.1109/CEC.2004.1330847.

[45] Joakim Bergdahl, Camilo Gordillo, Konrad Tollmar, and Linus Gisslén. "Augmenting Automated Game Testing with Deep Reinforcement Learning". In: *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020*. IEEE, 2020, pp. 600–603. DOI: 10.1109/COG47356.2020.9231552.

[46] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. DOI: 10.1613/jair.3912.

[47] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. "Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents". In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 523–562.

[48] Vanessa Volz and Boris Naujoks. "Towards Game-Playing AI Benchmarks via Performance Reporting Standards". In: *2020 IEEE Conference on Games (CoG)*. 2020, pp. 764–771. DOI: 10.1109/CoG47356.2020.9231705.

[49] Steven Kapturowski, Victor Campos, Ray Jiang, Nemanja Rakicevic, Hado van Hasselt, Charles Blundell, and Adrià Puigdomènech Badia. "Human-level Atari 200x faster". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/pdf?id=JtC6yOHRoJJ.

[50] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. "Dueling Network Architectures for Deep Reinforcement Learning". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 1995–2003. URL: http://proceedings.mlr.press/v48/wangf16.html.

[51] Adrià Garriga-Alonso. "Solving Montezuma's Revenge with Planning and Reinforcement Learning". BSc thesis. Universitat Pompeu Fabra, 2016. URL: http://hdl.handle.net/10230/30867.

[52] Tim Salimans and Richard Chen. *Learning Montezuma's Revenge from a Single Demonstration*. 2018. DOI: 10.48550/ARXIV.1812.03381.

[53] Viktor Voss, Liudmyla Nechepurenko, Rudi Schaefer, and Steffen Bauer. "Playing a Strategy Game with Knowledge-Based Reinforcement Learning". In: *SN computer science* 1.2 (2020). ISSN: 2662-995X.

[54] J. K. Terry, Benjamin Black, and Luis Santos. *Multiplayer Support for the Arcade Learning Environment*. 2021. arXiv: 2009.09341 [cs.LG].

[55] Michael Murray. *MAMEToolkit*. URL: https://github.com/M-J-Murray/MAMEToolkit (visited on 01/31/2023).

[56] Eric Kaltman, Joseph C. Osborn, and John Aycock. "S4LVE: Shareable Videogame Analysis and Visualization". In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. FDG '19. San Luis Obispo, California, USA: Association for Computing Machinery, 2019. ISBN: 9781450372176. DOI: 10.1145/3337722.3341826.

[57] Joseph C. Osborn, Adam Summerville, Nathan Dailey, and Soksamnang Lim. "MappyLand: Fast, Accurate Mapping for Console Games". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 17.1 (Oct. 2021), pp. 66–73. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/18892.

[58] Joseph Osborn, Adam Summerville, and Michael Mateas. "Automatic Mapping of NES Games with Mappy". In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*. FDG '17. Hyannis, Massachusetts: Association for Computing Machinery, 2017. ISBN: 9781450353199. URL: https://doi.org/10.1145/3102071.3110576.

[59] Joseph C. Osborn, Adam Summerville, and Michael Mateas. "Automated Game Design Learning". In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 2017, pp. 240–247. DOI: 10.1109/CIG.2017.8080442.

[60] *MAMEdev.org | Home of The MAME Project*. URL: https://www.mamedev.org/ (visited on 03/14/2023).

[61] *MAME | About MAME*. URL: https://www.mamedev.org/about.html (visited on 03/14/2023).

[62] *MAME*. Mar. 14, 2023. URL: https://web.archive.org/web/20230314135125/http://adb.arcadeitalia.net/mame.php (visited on 10/01/2023).

[63] Steve Wright. "Stella Programmer's Guide". In: (1979). URL: https://www.digitpress.com/library/techdocs/2600_Stella_Guide_Rev_A_12-3-79.pdf (visited on 07/02/2023).

[64] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. 1st. Addison-Wesley Professional, 2009. ISBN: 0321549252.

[65] Atari. *Boxing Manual*. Activision, 1980. URL: https://atariage.com/manual_thumbs.php?SystemID=2600&SoftwareLabelID=45.

[66] Atari. *Combat Manual*. Atari, 1977. URL: https://atariage.com/manual_thumbs.php?SystemID=2600&SoftwareLabelID=94.

[67] Activision. *River Raid Manual*. Activision, 1982. URL: https://atariage.com/manual_thumbs.php?SoftwareLabelID=409.

[68] Imagic. *Dragonfire Manual*. Imagic, 1982. URL: https://atariage.com/manual_html_page.php?SoftwareLabelID=154.

[69] Activision. *Kaboom! Instructions*. Activision, 1981. URL: https://atariage.com/manual_thumbs.php?SoftwareLabelID=257.

[70] Activision. *Moonsweeper Instructions*. Activision, 1983. URL: https://atariage.com/manual_html_page.php?SoftwareLabelID=313.

[71] Parker Brothers. *Montezuma's Revenge Manual*. Parker Brothers, 1984. URL: https://atariage.com/manual_html_page.php?SoftwareLabelID=310.

[72] Atari. *Solaris Game Manual*. Atari, 1986. URL: https://atariage.com/manual_thumbs.php?SoftwareLabelID=450.

[73] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[74] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[75] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous Methods for Deep Reinforcement Learning”. In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 1928–1937. URL: http://proceedings.mlr.press/v48/mniha16.html.

[76] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[77] Jakub Sygnowski and Henryk Michalewski. “Learning from the Memory of Atari 2600”. In: *Computer Games - 5th Workshop on Computer Games, CGW 2016, and 5th Workshop on General Intelligence in Game-Playing Agents, GIGA 2016, Held in Conjunction with the 25th International Conference on Artificial Intelligence, IJCAI 2016, New York City, NY, USA, July 9-10, 2016, Revised Selected Papers*. Ed. by Tristan Cazenave, Mark H. M. Winands, Stefan Edelkamp, Stephan Schiffel, Michael Thielscher, and Julian Togelius. Vol. 705. Communica-

tions in Computer and Information Science. 2016, pp. 71–85. DOI: 10.1007/978-3-319-57969-6_6.

Ludography

Only some of the games listed have MD5 hashes as the binary image of the games were used in this work.

Midway Manufacturing Co. *Sea Wolf II*. Arcade. Midway Manufacturing Co., 1978.

David Crane. *Pitfall II: Lost Caverns*. Atari 2600. Activision, 1984.

Wickstead Design. *Montezuma's Revenge*. Atari 2600. Parker Brothers, 1984. md5: 3347a6dd59049b15a38394aa2dafa585.

Larry Kaplan and David Crane. *Kaboom!* Atari 2600. Activision, 1981. MD5: 5428cdfada281c569c74c7308c7f2c26.

Doug Neubauer. *Solaris*. Atari 2600. Atari, 1986. MD5: e72eb8d4410152bdcb69e7fba327b420.

Warren Robinett. *Adventure*. Atari 2600. Atari, 1980.

Carol Shaw. *River Raid*. Atari 2600. Activision, 1982. MD5: 393948436d1f4cc3192410bb918f9724.

Ski Hunt. Atari 2600. Homevision, 1983.

Bob Smith. *Dragonfire*. Atari 2600. Imagic, 1982. MD5: 41810dd94bd0de1110bedc5092bef5b0.

Bob Smith. *Moonsweeper*. Atari 2600. Imagic, 1983. MD5: 64a649dce9774e75780785a999ad0705.

Brad Stewart. *Asteroids*. Atari 2600. Atari, 1979.

Larry Wagner and Joe Decuir. *Combat*. Atari 2600. Atari, 1977. MD5: 4c8832ed387bbafc055320c05205bc08.

Bob Whitehead. *Boxing*. Atari 2600. Activision, 1980. MD5: c3ef5c4653212088eda54dc91d787870.

Appendix A

Modifications & Additions

- src/frontend/mame/clifront.h

```
@@ -38,2 +38,7 @@

+#ifdef FRAG
+ int frag_execute(std::vector<std::string> &args,
    mame_machine_manager *&manager, std::function<void ()>
    fragInit);
+ void frag_exit();
+#endif
+
 private:
@@ -77,2 +82,5 @@
  void start_execution(mame_machine_manager *manager, const
      std::vector<std::string> &args);
+#ifdef FRAG
+ void frag_start_execution(mame_machine_manager *manager,
    const std::vector<std::string> &args);
```

```
+#endif
  static const info_command_struct *find_command(const std::
     string &s);
```

- src/frontend/mame/clifront.cpp

```
@@ -1860 +1860,158 @@
 }
+
+
+#ifdef FRAG
+
+//-------------------------------------
+// frag_start_execution - for FRAG.
+//-------------------------------------
+
+void cli_frontend::frag_start_execution(mame_machine_manager
     *manager, const std::vector<std::string> &args)
+{
+ std::ostringstream option_errors;
+
+ // because softlist evaluation relies on hashpath being
    populated, we are going to go through
+ // a special step to force it to be evaluated
+ mame_options::populate_hashpath_from_args_and_inis(
    m_options, args);
+
+ // parse the command line, adding any system-specific
```

```
options
+ try
+ {
+  m_options.parse_command_line(args, OPTION_PRIORITY_CMDLINE
   );
+ }
+ catch (options_warning_exception &ex)
+ {
+  osd_printf_error("%s", ex.message());
+ }
+ catch (options_exception &ex)
+ {
+  // if we failed, check for no command and a system name
   first; in that case error on the name
+  if (m_options.command().empty() && mame_options::system(
   m_options) == nullptr && !m_options.attempted_system_name
   ().empty())
+   throw emu_fatalerror(EMU_ERR_NO_SUCH_SYSTEM, "Unknown
   system '%s'", m_options.attempted_system_name());
+
+  // otherwise, error on the options
+  throw emu_fatalerror(EMU_ERR_INVALID_CONFIG, "%s", ex.
   message());
+ }
+ m_osd.set_verbose(m_options.verbose());
+
+ // determine the base name of the EXE
```

```
+ std::string_view exename = core_filename_extract_base(args
    [0], true);
+
+ // if we have a command, execute that
+ if (!m_options.command().empty())
+ {
+   execute_commands(exename);
+   return;
+ }
+
+ // read INI's, if appropriate
+ if (m_options.read_config())
+ {
+   mame_options::parse_standard_inis(m_options, option_errors
    );
+   m_osd.set_verbose(m_options.verbose());
+ }
+
+ // otherwise, check for a valid system
+ load_translation(m_options);
+
+ manager->start_http_server();
+
+ manager->start_luaengine();
+
+ if (option_errors.tellp() > 0)
+   osd_printf_error("Error in command line:\n%s\n",
```

```
    strtrimspace(option_errors.str()));
+
+ // if we can't find it, give an appropriate error
+ const game_driver *system = mame_options::system(m_options)
    ;
+ if (system == nullptr && *(m_options.system_name()) != 0)
+  throw emu_fatalerror(EMU_ERR_NO_SUCH_SYSTEM, "Unknown
    system '%s'", m_options.system_name());
+
+ // otherwise just run the game
+ m_result = manager->frag_execute();
+}
+
+
+//-------------------------------------------------
+// frag_execute - execute a game via the standard
+// command line interface. for FrAG
+//-------------------------------------------------
+
+int cli_frontend::frag_execute(std::vector<std::string> &
    args, mame_machine_manager *&manager,
+ std::function<void ()> fragInit)
+{
+ // wrap the core execution in a try/catch to field all
    fatal errors
+ m_result = EMU_ERR_NONE;
+ manager = mame_machine_manager::instance(m_options, m_osd,
```

```
fragInit);
+
+ try
+ {
+  frag_start_execution(manager, args);
+ }
+ // handle exceptions of various types
+ catch (emu_fatalerror &fatal)
+ {
+  osd_printf_error("%s\n", strtrimspace(fatal.what()));
+  m_result = (fatal.exitcode() != 0) ? fatal.exitcode() :
   EMU_ERR_FATALERROR;
+
+  // if a game was specified, wasn't a wildcard, and our
   error indicates this was the
+  // reason for failure, offer some suggestions
+  if (m_result == EMU_ERR_NO_SUCH_SYSTEM
+   && !m_options.attempted_system_name().empty()
+   && !core_iswildstr(m_options.attempted_system_name())
+   && mame_options::system(m_options) == nullptr)
+  {
+   // get the top 16 approximate matches
+   driver_enumerator drivlist(m_options);
+   int matches[16];
+   drivlist.find_approximate_matches(m_options.
   attempted_system_name(), std::size(matches), matches);
+
```

```
+   // work out how wide the titles need to be
+   int titlelen(0);
+   for (int match : matches)
+    if (0 <= match)
+     titlelen = (std::max)(titlelen, int(strlen(drivlist.
   driver(match).type.fullname())));
+
+   // print them out
+   osd_printf_error("\n\"%s\" approximately matches the
   following\n"
+     "supported machines (best match first):\n\n", m_options
   .attempted_system_name());
+   for (int match : matches)
+   {
+    if (0 <= match)
+    {
+     game_driver const &drv(drivlist.driver(match));
+     osd_printf_error("%-18s%-*s(%s, %s)\n", drv.name,
   titlelen + 2, drv.type.fullname(), drv.manufacturer, drv.
   year);
+    }
+   }
+  }
+ }
+ catch (emu_exception &)
+ {
+  osd_printf_error("Caught unhandled emulator exception\n");
```

```
+  m_result = EMU_ERR_FATALERROR;
+ }
+ catch (tag_add_exception &aex)
+ {
+  osd_printf_error("Tag '%s' already exists in tagged map\n
   ", aex.tag());
+  m_result = EMU_ERR_FATALERROR;
+ }
+ catch (std::exception &ex)
+ {
+  osd_printf_error("Caught unhandled %s exception: %s\n",
   typeid(ex).name(), ex.what());
+  m_result = EMU_ERR_FATALERROR;
+ }
+ catch (...)
+ {
+  osd_printf_error("Caught unhandled exception\n");
+  m_result = EMU_ERR_FATALERROR;
+ }
+
+ return m_result;
+}
+
+void cli_frontend::frag_exit()
+{
+ util::archive_file::cache_clear();
+}
```

```
+
+
+#endif // FRAG
```

- src/frontend/mame/mame.h

```
@@ -42,2 +42,3 @@
  static mame_machine_manager *instance();
+
  ~mame_machine_manager();
@@ -73,2 +74,7 @@

+#ifdef FRAG
+ static mame_machine_manager *instance(emu_options &options,
     osd_interface &osd, std::function<void ()> fragInit);
+ int frag_execute();
+#endif
+
 private:
@@ -84,2 +90,7 @@

+#ifdef FRAG
+ mame_machine_manager(emu_options &options, osd_interface &
    osd, std::function<void ()> fragInit);
+ std::function<void ()> m_fragInitCall;
+#endif
+
  const game_driver * m_new_driver_pending; // pointer to the
```

```
next pending driver
```

- src/frontend/mame/mame.cpp

```
@@ -108,2 +108,7 @@
  m_lua->attach_notifiers();
+
+#ifdef FRAG
+ m_fragInitCall();
+ m_fragInitCall = nullptr;
+#endif
 }
@@ -500 +505,81 @@
 bool emulator_info::standalone() { return false; }
+
+
+#ifdef FRAG
+
+mame_machine_manager* mame_machine_manager::instance(
    emu_options &options, osd_interface &osd,
+ std::function<void ()> fragInit)
+{
+ if (!s_manager)
+ s_manager = new mame_machine_manager(options, osd, fragInit
    );
+
+ return s_manager;
+}
```

```
+
+mame_machine_manager::mame_machine_manager(emu_options &
    options,osd_interface &osd, std::function<void ()>
    fragInit) :
+ machine_manager(options, osd),
+ m_plugins(std::make_unique<plugin_options>()),
+ m_lua(std::make_unique<lua_engine>()),
+  m_fragInitCall(fragInit),
+ m_new_driver_pending(nullptr),
+ m_firstrun(true),
+ m_autoboot_timer(nullptr)
+{
+}
+
+//--------------------------------------------------
+// frag_execute - run the core emulation for FrAG
+//--------------------------------------------------
+
+int mame_machine_manager::frag_execute()
+{
+ int error = EMU_ERR_NONE;
+
+ m_new_driver_pending = nullptr;
+
+ // if no driver, use the internal empty driver
+ const game_driver *system = mame_options::system(m_options)
    ;
```

```
+ if (system == nullptr)
+ {
+  system = &GAME_NAME(___empty);
+ }
+
+ // parse any INI files as the first thing
+ if (m_options.read_config())
+ {
+  // but first, revert out any potential game-specific INI
    settings from previous runs via the internal UI
+  m_options.revert(OPTION_PRIORITY_INI);
+
+  std::ostringstream errors;
+  mame_options::parse_standard_inis(m_options, errors);
+ }
+
+ // otherwise, perform validity checks before anything else
+ bool is_empty = (system == &GAME_NAME(___empty));
+ if (!is_empty)
+ {
+  validity_checker valid(m_options, true);
+  valid.set_verbose(false);
+  valid.check_shared_source(*system);
+ }
+
+ // create the machine configuration
+ machine_config *config = new machine_config(*system,
```

```
    m_options);
+
+ // create the machine structure and driver
+ // WE DO NOT WANT THE MACHINE TO GET DESTROYED WHEN
    RETURNING
+ // AS THE ALE WILL USE THE MANAGER TO GET THE MACHINE AND
+ // USE IT TO CONTROL MAME.
+ running_machine *machine = new running_machine(*config, *
    this);
+
+ set_machine(machine);
+
+ // run the machine
+ error = machine->frag_run(is_empty);
+
+ // return an error
+ return error;
+}
+
+#endif // FRAG
+
```

- src/osd/sdl/sdlmain.cpp

```
@@ -113 +113,53 @@
 }
+
+
```

```
+#ifdef FRAG
+
+sdl_osd_interface* fragOSDInit(sdl_options* osdOptions)
+{
+ // disable I/O buffering
+ setvbuf(stdout, (char *) nullptr, _IONBF, 0);
+ setvbuf(stderr, (char *) nullptr, _IONBF, 0);
+
+ // Initialize crash diagnostics
+ diagnostics_module::get_instance()->init_crash_diagnostics
    ();
+
+#if defined(SDLMAME_ANDROID)
+ /* Enable standard application logging */
+ SDL_LogSetPriority(SDL_LOG_CATEGORY_APPLICATION,
    SDL_LOG_PRIORITY_VERBOSE);
+#endif
+
+ // FIXME: this should be done differently
+
+#ifdef SDLMAME_UNIX
+ sdl_entered_debugger = 0;
+#if (!defined(SDLMAME_MACOSX)) && (!defined(SDLMAME_HAIKU))
    && (!defined(SDLMAME_EMSCRIPTEN)) && (!defined(
    SDLMAME_ANDROID))
+ FcInit();
+#endif
```

```
+#endif
+ sdl_osd_interface* osd = new sdl_osd_interface(*osdOptions)
    ;
+ osd->register_options();
+
+ return osd;
+}
+
+sdl_options* fragOSDOptionsInit()
+{
+ return new sdl_options();
+}
+
+
+void fragOSDExit()
+{
+#ifdef SDLMAME_UNIX
+#if (!defined(SDLMAME_MACOSX)) && (!defined(SDLMAME_HAIKU))
    && (!defined(SDLMAME_EMSCRIPTEN)) && (!defined(
    SDLMAME_ANDROID))
+ if (!sdl_entered_debugger)
+ {
+  FcFini();
+ }
+#endif
+#endif
+}
```

```
+
+#endif
+
```

- src/emu/ioport.cpp

```
@@ -3535,3 +3535,5 @@
     m_absolute = true;
+#ifndef FRAG
     m_autocenter = true;
+#endif
     m_interpolate = !field.analog_reset();
```

- src/emu/diexec.h

```
@@ -195,2 +195,7 @@

+#ifdef FRAG
+ void set_frag_ins_hook(delegate<void (offs_t)> callback) {
   m_frag_ins_call = callback; m_frag_hook = true; }
+ void remove_frag_ins_hook() { m_frag_ins_call = nullptr;
   m_frag_hook = false; }
+#endif
+
 protected:
@@ -230,2 +235,7 @@
  {
+#ifdef FRAG
+ if (m_frag_hook)
```

```
+ m_frag_ins_call(curpc);
+#endif
+
    if (device().machine().debug_flags & DEBUG_FLAG_CALL_HOOK
        )
@@ -340,2 +350,7 @@

+#ifdef FRAG
+ delegate<void (offs_t)> m_frag_ins_call;
+ bool m_frag_hook{false};
+#endif
+
 public:
```

- src/emu/video.h

```
@@ -89,2 +89,15 @@

+#ifdef FRAG
+ // Return frame
+ void get_screen(screen_device *screen, std::vector<u8>& dst
    );
+ void get_screen(screen_device *screen, u8 *buff);
+
+ // Frame dimensions
+ s32 getSnapHeight(screen_device *screen);
+ s32 getSnapWidth(screen_device *screen);
+
```

```
+ // Bitmap Format
+ bitmap_format get_bitmap_format(screen_device *screen);
+#endif
+
 private:
```

- `src/emu/video.cpp`

```
@@ -1281 +1281,73 @@
 }
+
+
+
+#ifdef FRAG
+
+//-----------------------------------------------------
+// get_screen - Fills current frame to dst vector
+//-----------------------------------------------------
+
+void video_manager::get_screen(screen_device *screen, std::
    vector<u8>& dst)
+{
+ assert(!m_snap_native || screen != nullptr);
+
+ create_snapshot_bitmap(nullptr);
+
+ get_screen(screen, dst.data());
+
```

```
+}
+
+//-----------------------------------------------------
+// get_screen - Fills current frame to provided buffer
+//-----------------------------------------------------
+void video_manager::get_screen(screen_device *screen, u8 *
    buff)
+{
+ assert(!m_snap_native || screen != nullptr);
+
+ create_snapshot_bitmap(nullptr);
+
+ for (int y = 0; y < m_snap_bitmap.height(); y++)
+ {
+ const u32 *src = &m_snap_bitmap.pix(y, 0);
+ for (int x = 0; x < m_snap_bitmap.width(); x++)
+ {
+ rgb_t pixel= *src++;
+ *buff++ = pixel.r();
+ *buff++ = pixel.g();
+ *buff++ = pixel.b();
+ if (screen->format() == BITMAP_FORMAT_ARGB32)
+ *buff++ = pixel.a();
+ }
+ }
+}
+
```

```
+//--------------------------------------------------
+// get_bitmap_format -- Returns bitmap format
+//--------------------------------------------------
+
+bitmap_format video_manager::get_bitmap_format(screen_device
    *screen)
+{
+ assert(screen != nullptr);
+
+ return screen->format();
+}
+
+
+s32 video_manager::getSnapHeight(screen_device *screen)
+{
+ create_snapshot_bitmap(nullptr);
+
+ return m_snap_bitmap.height();
+}
+
+
+s32 video_manager::getSnapWidth(screen_device *screen)
+{
+ create_snapshot_bitmap(nullptr);
+
+ return m_snap_bitmap.width();
+}
```

```
+
+#endif
+
```

- src/emu/machine.h

```
@@ -46,2 +46,5 @@
  MACHINE_NOTIFY_EXIT,
+#ifdef FRAG
+ MACHINE_NOTIFY_RUNNING,
+#endif
  MACHINE_NOTIFY_COUNT
@@ -178,2 +181,7 @@

+#ifdef FRAG
+ int frag_run(bool quiet);
+ int frag_exit();
+#endif
+
  // TODO: Do saves and loads still require scheduling?
```

- src/emu/machine.cpp

```
@@ -1379 +1379,192 @@
 #endif /* defined(__EMSCRIPTEN__) */
+
+
+#ifdef FRAG
+
```

```
+
+//-------------------------------------------------
+// frag_run - execute the machine. For FrAG.
+// Called from MAMESystem
+//-------------------------------------------------
+
+int running_machine::frag_run(bool quiet)
+{
+ int error = EMU_ERR_NONE;
+
+ // use try/catch for deep error recovery
+ try
+ {
+  m_manager.http()->clear();
+
+  // move to the init phase
+  m_current_phase = machine_phase::INIT;
+
+  // if we have a logfile, set up the callback
+  if (options().log() && !quiet)
+  {
+   m_logfile = std::make_unique<emu_file>(OPEN_FLAG_WRITE |
   OPEN_FLAG_CREATE | OPEN_FLAG_CREATE_PATHS);
+   std::error_condition const filerr = m_logfile->open("
   error.log");
+   if (filerr)
+    throw emu_fatalerror("running_machine::run: unable to
```

```
    open error.log file");
+
+     using namespace std::placeholders;
+     add_logerror_callback(std::bind(&running_machine::
    logfile_callback, this, _1));
+   }
+
+   if (options().debug() && options().debuglog())
+   {
+     m_debuglogfile = std::make_unique<emu_file>(
    OPEN_FLAG_WRITE | OPEN_FLAG_CREATE |
    OPEN_FLAG_CREATE_PATHS);
+     std::error_condition const filerr = m_debuglogfile->open
    ("debug.log");
+     if (filerr)
+       throw emu_fatalerror("running_machine::run: unable to
    open debug.log file");
+   }
+
+   // then finish setting up our local machine
+   start();
+
+   // load the configuration settings
+   manager().before_load_settings(*this);
+   m_configuration->load_settings();
+
+   // disallow save state registrations starting here.
```

```
+   // Don't do it earlier, config load can create network
+   // devices with timers.
+   m_save.allow_registration(false);
+
+   // load the NVRAM
+   nvram_load();
+
+   // set the time on RTCs (this may overwrite parts of NVRAM
    )
+   set_rtc_datetime(system_time(m_base_time));
+
+   sound().ui_mute(false);
+   if (!quiet)
+    sound().start_recording();
+
+   m_hard_reset_pending = false;
+
+   // initialize ui lists
+   // display the startup screens
+   manager().ui_initialize(*this);
+
+   // perform a soft reset -- this takes us to the running
    phase
+   soft_reset();
+
+   // handle initial load
+   if (m_saveload_schedule != saveload_schedule::NONE)
```

```
+   handle_saveload();
+
+  export_http_api();
+
+#if defined(__EMSCRIPTEN__)
+  // break out to our async javascript loop and halt
+  emscripten_set_running_machine(this);
+#endif
+
+ }
+ catch (emu_fatalerror &fatal)
+ {
+  osd_printf_error("Fatal error: %s\n", fatal.what());
+  error = EMU_ERR_FATALERROR;
+  if (fatal.exitcode() != 0)
+   error = fatal.exitcode();
+ }
+ catch (emu_exception &)
+ {
+  osd_printf_error("Caught unhandled emulator exception\n");
+  error = EMU_ERR_FATALERROR;
+ }
+ catch (binding_type_exception &btex)
+ {
+  osd_printf_error("Error performing a late bind of function
     expecting type %s to instance of type %s\n", btex.
    target_type().name(), btex.actual_type().name());
```

```
+   error = EMU_ERR_FATALERROR;
+ }
+ catch (tag_add_exception &aex)
+ {
+   osd_printf_error("Tag '%s' already exists in tagged map\n
    ", aex.tag());
+   error = EMU_ERR_FATALERROR;
+ }
+ catch (std::exception &ex)
+ {
+   osd_printf_error("Caught unhandled %s exception: %s\n",
    typeid(ex).name(), ex.what());
+   error = EMU_ERR_FATALERROR;
+ }
+ catch (...)
+ {
+   osd_printf_error("Caught unhandled exception\n");
+   error = EMU_ERR_FATALERROR;
+ }
+
+ return error;
+}
+
+
+//---------------------------------------
+// frag_exit - Stop machine and destroy
+// MAME instance
```

```
+//---------------------------------------
+
+int running_machine::frag_exit()
+{
+ schedule_exit();
+ int error = EMU_ERR_NONE;
+
+ try{
+  m_manager.http()->clear();
+
+  // and out via the exit phase
+  m_current_phase = machine_phase::EXIT;
+
+  // save the NVRAM and configuration
+  sound().ui_mute(true);
+  if (options().nvram_save())
+   nvram_save();
+  m_configuration->save_settings();
+ }
+ catch (emu_fatalerror &fatal)
+ {
+  osd_printf_error("Fatal error: %s\n", fatal.what());
+  error = EMU_ERR_FATALERROR;
+  if (fatal.exitcode() != 0)
+   error = fatal.exitcode();
+ }
+ catch (emu_exception &)
```

```
+ {
+  osd_printf_error("Caught unhandled emulator exception\n");
+  error = EMU_ERR_FATALERROR;
+ }
+ catch (binding_type_exception &btex)
+ {
+  osd_printf_error("Error performing a late bind of function
     expecting type %s to instance of type %s\n", btex.
    target_type().name(), btex.actual_type().name());
+  error = EMU_ERR_FATALERROR;
+ }
+ catch (tag_add_exception &aex)
+ {
+  osd_printf_error("Tag '%s' already exists in tagged map\n
    ", aex.tag());
+  error = EMU_ERR_FATALERROR;
+ }
+ catch (std::exception &ex)
+ {
+  osd_printf_error("Caught unhandled %s exception: %s\n",
    typeid(ex).name(), ex.what());
+  error = EMU_ERR_FATALERROR;
+ }
+ catch (...)
+ {
+  osd_printf_error("Caught unhandled exception\n");
+  error = EMU_ERR_FATALERROR;
```

```
+ }
+
+ // make sure our phase is set properly before cleaning up,
+ // in case we got here via exception
+ m_current_phase = machine_phase::EXIT;
+
+ // call all exit callbacks registered
+ call_notifiers(MACHINE_NOTIFY_EXIT);
+ util::archive_file::cache_clear();
+
+ // close the logfile
+ m_logfile.reset();
+ return error;
+}
+
+#endif // FRAG
```

- src/emu/emumem.h

```
@@ -618,2 +618,6 @@

+#ifdef FRAG
+ virtual memory_bank *get_bank(offs_t addr) const;
+#endif
+
  inline void populate(offs_t start, offs_t end, offs_t
     mirror, handler_entry_read<Width, AddrShift> *handler)
     {
```

```
@@ -692,2 +696,6 @@

+#ifdef FRAG
+ virtual memory_bank *get_bank(offs_t addr) const;
+#endif
+
  inline void populate(offs_t start, offs_t end, offs_t
      mirror, handler_entry_write<Width, AddrShift> *handler)
       {
@@ -2374,2 +2382,7 @@

+#ifdef FRAG
+ virtual memory_bank *get_read_bank(offs_t address) const =
     0;
+ virtual memory_bank *get_write_bank(offs_t address) const =
     0;
+#endif
+
  // read accessors
@@ -2462,2 +2475,6 @@

+#ifdef FRAG
+ int entries() const { return m_entries.size(); }
+#endif
+
  // set the base explicitly
```

- src/emu/emumem.cpp

```
@@ -1059 +1059,18 @@
 }
+
+
+#ifdef FRAG
+
+template<int Width, int AddrShift> memory_bank *
    handler_entry_read<Width, AddrShift>::get_bank(offs_t
    addr) const
+{
+ return nullptr;
+}
+
+template<int Width, int AddrShift> memory_bank *
    handler_entry_write<Width, AddrShift>::get_bank(offs_t
    addr) const
+{
+ return nullptr;
+}
+
+#endif
+
+
```

- src/emu/emumem_aspace.cpp

```
@@ -429,2 +429,22 @@
```

```
+#ifdef FRAG
+ // The above get_ptr method returns the base to which the
    bank points to
+ // It does not return the bank itself, which is why this
    method is added
+ // return a pointer to the read bank, or nullptr if none
+ // nullptr indicates RAM
+ virtual memory_bank *get_read_bank(offs_t address) const
    override
+ {
+  return m_root_read->get_bank(address);
+ }
+
+ // return a pointer to the write bank, or nullptr if none
+ // nullptr indicates RAM
+ virtual memory_bank *get_write_bank(offs_t address) const
    override
+ {
+  return m_root_write->get_bank(address);
+ }
+#endif
+
+
+
  // native read
@@ -1251 +1271,2 @@
```

```
 }
+
```

- src/emu/emumem_hem.h

```
@@ -24,2 +24,6 @@

+#ifdef FRAG
+ memory_bank *get_bank(offs_t addr) const override;
+#endif
+
  std::string name() const override;
@@ -43,2 +47,6 @@

+#ifdef FRAG
+ memory_bank *get_bank(offs_t addr) const override;
+#endif
+
  std::string name() const override;
@@ -67,2 +75,6 @@

+#ifdef FRAG
+ memory_bank *get_bank(offs_t addr) const override;
+#endif
+
  std::string name() const override;
@@ -86,2 +98,6 @@
```

```
+#ifdef FRAG
+ memory_bank *get_bank(offs_t addr) const override;
+#endif
+
  std::string name() const override;
```

- src/emu/emumem_hem.cpp

```
@@ -199 +199,26 @@
 template class handler_entry_write_memory_bank<3, -3>;
+
+#ifdef FRAG
+
+template<int Width, int AddrShift> memory_bank *
    handler_entry_read_memory<Width, AddrShift>::get_bank(
    offs_t addr) const
+{
+ return nullptr;
+}
+
+template<int Width, int AddrShift> memory_bank *
    handler_entry_write_memory<Width, AddrShift>::get_bank(
    offs_t addr) const
+{
+ return nullptr;
+}
+
+template<int Width, int AddrShift> memory_bank *
```

```
    handler_entry_read_memory_bank<Width, AddrShift>::
    get_bank(offs_t addr) const
+{
+ return &m_bank;
+}
+
+template<int Width, int AddrShift> memory_bank *
    handler_entry_write_memory_bank<Width, AddrShift>::
    get_bank(offs_t addr) const
+{
+ return &m_bank;
+}
+
+#endif
+
```

- src/emu/emumem_hedw.h

```
@@ -29,2 +29,6 @@

+#ifdef FRAG
+ memory_bank* get_bank(offs_t addr) const override;
+#endif
+
  offs_t dispatch_entry(offs_t address) const override;
```

- src/emu/emumem_hedw.ipp

```
@@ -711,2 +711,10 @@
```

```
+#ifdef FRAG
+template<int HighBits, int Width, int AddrShift> memory_bank
     *handler_entry_write_dispatch<HighBits, Width, AddrShift
    >::get_bank(offs_t addr) const
+{
+ return m_a_dispatch[(addr & HIGHMASK) >> LowBits]->get_bank
    (addr);
+}
+#endif
+
+
 #endif // MAME_EMU_EMUMEM_HEDW_IPP
```

- src/emu/emumem_hedr.h

```
@@ -29,2 +29,6 @@

+#ifdef FRAG
+ memory_bank *get_bank(offs_t addr) const override;
+#endif
+
  offs_t dispatch_entry(offs_t address) const override;
```

- src/emu/emumem_hedr.ipp

```
@@ -712,2 +712,9 @@

+#ifdef FRAG
```

```
+template<int HighBits, int Width, int AddrShift> memory_bank
     *handler_entry_read_dispatch<HighBits, Width, AddrShift
    >::get_bank(offs_t addr) const
+{
+ return m_a_dispatch[(addr & HIGHMASK) >> LowBits]->get_bank
    (addr);
+}
+#endif
+
 #endif // MAME_EMU_EMUMEM_HEDR_IPP
```

- src/emu/emumem_het.h

```
@@ -23,2 +23,6 @@

+#ifdef FRAG
+ memory_bank *get_bank(offs_t addr) const override;
+#endif
+
  std::string name() const override;
@@ -46,2 +50,6 @@

+#ifdef FRAG
+ memory_bank *get_bank(offs_t addr) const override;
+#endif
+
  std::string name() const override;
```

- src/emu/emumem_het.cpp

```
@@ -111 +111,16 @@
 template class handler_entry_write_tap<3, -3>;
+
+#ifdef FRAG
+
+template<int Width, int AddrShift> memory_bank *
    handler_entry_read_tap<Width, AddrShift>::get_bank(offs_t
     addr) const
+{
+ return this->m_next->get_bank(addr);
+}
+
+template<int Width, int AddrShift> memory_bank *
    handler_entry_write_tap<Width, AddrShift>::get_bank(
    offs_t addr) const
+{
+ return this->m_next->get_bank(addr);
+}
+
+#endif
+
```

Appendix B

Handlers

B.1 Read Handler

```
1  // Handle all reads. Called by memory space before giving read value
       to system
2  void VCSRomSettings::handleRead(const MAMESystem &system,
       address_space &space, const offs_t addr, const u64 data)
3  {
4    // ZP RAM
5    if (isBetween(addr, 0x80, 0xff))
6    {
7      if (m_ramusage)
8      {
9        m_ram_read_usage_count[addr ^ 0x80]++;
10       m_ramusage = false ? m_ram_read_usage_count[addr ^ 0x80] ==
             std::numeric_limits<uint64_t>::max() : m_ramusage;
11     }
12   }
13
14   if (m_bswitch_count)
```

```
15  {
16    switch (m_type)
17    {
18      case a26_f8:
19        if (isBetween(addr&0x1fff, 0x1ff8, 0x1ff9))
20          m_bankSwitch_freq[addr&0x1fff]++;
21        break;
22      case a26_f6:
23        if (isBetween(addr&0x1fff, 0x1ff6, 0x1ff9))
24          m_bankSwitch_freq[addr&0x1fff]++;
25        break;
26      case a26_f4:
27        if (isBetween(addr&0x1fff, 0x1ff4, 0x1ffb))
28          m_bankSwitch_freq[addr&0x1fff]++;
29        break;
30      case a26_e0:
31        if (isBetween(addr&0x1fff, 0x1fe0, 0x1ff7))
32          m_bankSwitch_freq[addr&0x1fff]++;
33        break;
34      case a26_fa:
35        if (isBetween(addr&0x1fff, 0x1ff8, 0x1ffa))
36          m_bankSwitch_freq[addr&0x1fff]++;
37        break;
38      case a26_e7:
39        if (isBetween(addr&0x1fff, 0x1fe0, 0x1fe6))
40          m_bankSwitch_freq[addr&0x1fff]++;
41        break;
42      case a26_fv:
43        if ((addr&0x1fff) == 0x1ff0)
44          m_bankSwitch_freq[addr&0x1fff]++;
45        break;
```

```
46        case a26_ua:
47          if ((addr&0x1fff) == 0x220 || (addr&0x1fff) == 0x0240)
48            m_bankSwitch_freq[addr&0x1fff]++;
49          break;
50        default:
51          break;
52      }
53    }
54
55    // If state is not set return as it is essential
56    if (!m_state) return;
57
58    // Get current pc
59    // The mask is needed as the pc is relative to the relative origin
60    m_cpc = m_state->pcbase() & 0x1fff;
61
62    // If pc changed, try to add the last read if valid
63    if (m_last_pc != m_state->pcbase())
64    {
65      // If the previously read value is not the current pc
66      // and is not RAM
67      if (m_last_read != 0 && m_last_read != m_cpc && m_last_bank !=
            -1)
68      {
69        m_trackData[m_last_bank][m_last_read] |= TR_READ;
70      }
71      else if (m_2last_read != 0 && m_2last_read != m_cpc &&
            m_2last_bank != -1)
72      {
73        m_trackData[m_2last_bank][m_2last_read] |= TR_READ;
74      }
```

```
75
76      m_last_read = 0;
77      m_2last_read = 0;
78      m_last_bank = -1;
79      m_2last_bank = -1;
80    }
81
82    m_last_pc = m_state->pcbase();
83
84    // Ignore branch instruction and jump reads as they don't really
          read data
85    // but addresses to jump or branch to.
86    // JSR is here as it reads one extra byte past its operand but
          doesn't use it.
87    // This causes the opcode to be marked as read which is true but
          not true.
88      // Also, JSR is still a jump!
89    u8 currentIns = m_ir_state->value();
90    if (is(currentIns, 0x90, 0xB0, 0xF0, 0x30, 0xD0, 0x10, 0x50, 0x70,
           0x20, 0x4C, 0x6C, 0x60))
91      return;
92
93    // If the address read is not the current pc or its operand then
          consider the read
94    // Limitation: If an instruction were to read its own opcode or
          operand as data
95    // the read handler discards it. This is unlikely though.
96    // Well, even if it did, if it's executed then the disassambler
          will break it down as
97    // an instrucion and not data. Unless the operand is also executed
           in which case we have overlapping
```

```
98      // instructions
99      if (addr >= 0x1000 && addr != m_cpc && (addr < m_cpc || addr >
            m_cpc+m_opcodeBytes[currentIns]))
100     {
101       m_2last_read = m_last_read;
102       m_last_read = addr;
103       m_2last_bank = m_last_bank;
104       m_last_bank = getBank(space, addr, read_or_write::READ);
105     }
106   }
```

B.2 Write Handler

```
1   void VCSRomSettings::handleWrite(const MAMESystem &system,
        address_space &space, const offs_t addr, const u64 data)
2   {
3     // ZP RAM
4     if (isBetween(addr, 0x80, 0xff))
5     {
6       if (m_ramusage)
7       {
8         m_ram_write_usage_count[addr ^ 0x80]++;
9         m_ramusage = false ? m_ram_write_usage_count[addr ^ 0x80] ==
              std::numeric_limits<uint64_t>::max() : m_ramusage;
10      }
11
12      if (m_ram_analyze)
13      {
14        m_ram_graph[addr ^ 0x80]->addEdge(readRAM(&system, addr), int(
              data));
```

```
15      }
16      return;
17    }
18
19    if (m_bswitch_count)
20    {
21      switch (m_type)
22      {
23        case a26_f8:
24          if (isBetween(addr&0x1fff, 0x1ff8, 0x1ff9))
25            m_bankSwitch_freq[addr&0x1fff]++;
26          break;
27        case a26_f6:
28          if (isBetween(addr&0x1fff, 0x1ff6, 0x1ff9))
29            m_bankSwitch_freq[addr&0x1fff]++;
30          break;
31        case a26_f4:
32          if (isBetween(addr&0x1fff, 0x1ff4, 0x1ffb))
33            m_bankSwitch_freq[addr&0x1fff]++;
34          break;
35        case a26_e0:
36          if (isBetween(addr&0x1fff, 0x1fe0, 0x1ff7))
37            m_bankSwitch_freq[addr&0x1fff]++;
38          break;
39        case a26_fa:
40          if (isBetween(addr&0x1fff, 0x1ff8, 0x1ffa))
41            m_bankSwitch_freq[addr&0x1fff]++;
42          break;
43        case a26_e7:
44          if (isBetween(addr&0x1fff, 0x1fe0, 0x1fe6))
45            m_bankSwitch_freq[addr&0x1fff]++;
```

```
46          break;
47        case a26_fv:
48          if ((addr&0x1fff) == 0x1ff0)
49            m_bankSwitch_freq[addr&0x1fff]++;
50          break;
51        case a26_ua:
52          if ((addr&0x1fff) == 0x220 || (addr&0x1fff) == 0x0240)
53            m_bankSwitch_freq[addr&0x1fff]++;
54          break;
55        default:
56          break;
57      }
58    }
59
60    // If not ROM space
61    if (addr < 0x1000) return;
62
63    auto bank = getBank(space, addr, read_or_write::WRITE);
64    if (bank != -1)
65      m_trackData[bank][addr] |= TR_WRITE;
66  }
```

B.3 PC Handler

```
1  // PC tracking handler. Called by the CPU before executing the
       instruction
2  void VCSRomSettings::handlePC(const MAMESystem &system,
       address_space &space, const offs_t pc)
3  {
4    m_prevPC = m_currentPC;
```

```
5     m_currentPC = pc;
6
7     auto bank = getBank(space, pc, read_or_write::READ);
8
9     m_trackData[bank][pc] |= TR_EXEC;
10
11    if (m_track_pc_count)
12    {
13      m_pc_usage[bank][pc]++;
14      m_track_pc_count = false ? m_pc_usage[bank][pc] == std::
          numeric_limits<uint64_t>::max() : m_track_pc_count;
15    }
16
17    if (m_catchKernel)
18    {
19      // Adds to kernel if the raster beam is in visible area (NTSC)
20      // Values from the stella programming manual
21      if (isBetween(m_screen->vpos(), 41, 232))
22        m_kernel[bank].insert(pc);
23    }
24
25    if (m_execTrace)
26    {
27      m_execSeq_graph->addEdge(m_prevPC, m_currentPC);
28    }
29  }
```

B.4 Bank Retrieval

```
1  [[nodiscard]]
```

```
2  int VCSRomSettings::getBank(address_space &space, const offs_t& addr
       , read_or_write mode)
3  {
4    // Discard RAM reads as this is not very useful and if
5    // a cart uses RAM, it is sufficient enough to disassemble the ROM
6    // as doing so would show how the RAM is used.
7    // But if needed, one can store it in the read/write handler as
8    // the int returned here only indicates that it's RAM.
9
10   // Only for address between 0x1000 and 0x1fff, rom space.
11   // This is for accurate tracking of banks.
12   memory_bank* m_bank = nullptr;
13   auto address = addr & 0x1fff;
14
15   switch (m_type)
16   {
17   case a26_2k_4k:
18     return 0;  // only one bank
19
20   // Simple bankswitching mechanisms with no RAM
21   case a26_ua:
22   case a26_x07:
23   case a26_fv:
24   case a26_f4:
25   case a26_f6:
26   case a26_f8:
27   case a26_dc:
28   case a26_jvp:
29   case a26_f8sw:
30   case a26_fe:
31     if (mode == read_or_write::READ)
```

```
32        m_bank = space.get_read_bank(address);
33      else
34        m_bank = space.get_write_bank(address);
35
36      if (!m_bank)
37        return -1;
38
39      return m_bank->entry();
40
41    case a26_fa:
42      if (address <= 0x11ff) // No point in checking if it's greater
          than 0x1000
43        return -1;
44      else
45      {
46        if (mode == read_or_write::READ)
47          m_bank = space.get_read_bank(address);
48        else
49          m_bank = space.get_write_bank(address);
50
51        if (!m_bank)
52          return -1;
53
54        return m_bank->entry();
55      }
56
57    case a26_e7:
58      if (mode == read_or_write::READ)
59        m_bank = space.get_read_bank(address);
60      else
61        m_bank = space.get_write_bank(address);
```

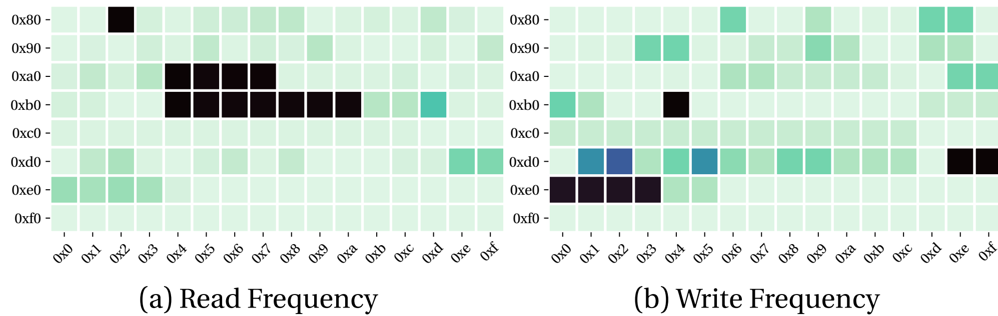
```
62
63      if (!m_bank)
64        return -1;
65
66      if (address >= 0x1a00)
67        return m_bank->entries() - 1; // Last page
68      else if (isBetween(address, 0x1800, 0x19ff)
69        || (m_bank->tag().substr(m_bank->tag().length()-3, 3) == "ram"
            ))
70        return -1;
71      else
72        return m_bank->entry();
73
74    case a26_e0:
75      if (address >= 0x1c00)
76        return 7;  // Last page
77      else
78      {
79        if (mode == read_or_write::READ)
80          m_bank = space.get_read_bank(address);
81        else
82          m_bank = space.get_write_bank(address);
83
84        if (!m_bank)
85          return -1;
86
87        return m_bank->entry();
88      }
89
90    case a26_3e:
91      if (mode == read_or_write::READ)
```

```
92        m_bank = space.get_read_bank(address);
93      else
94        m_bank = space.get_write_bank(address);
95
96      if (!m_bank)
97        return -1;
98
99      if (address >= 0x1800)
100       return m_bank->entries() - 1;
101     else if (m_bank->tag().substr(m_bank->tag().length()-3, 3) == "
          ram")
102       return -1;  // Last page
103     else
104       return m_bank->entry();
105
106   case a26_3f:
107     if (mode == read_or_write::READ)
108       m_bank = space.get_read_bank(address);
109     else
110       m_bank = space.get_write_bank(address);
111
112     if (!m_bank)
113       return -1;
114
115     if (address >= 0x1800)
116       return m_bank->entries() - 1; // Last page.
117     else
118       return m_bank->entry();
119
120   case a26_cv:
121     if (address <= 0x17ff) return -1;
```

```
122       else return 0;
123
124     default:
125       osd_printf_warning("Warning: Could not detect bank type.
              Defaulting to bank 0.\n");
126       return 0;
127     }
128   }
```

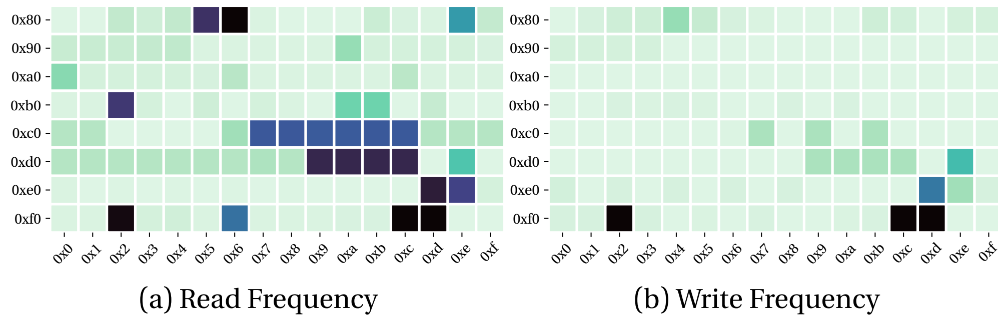
Appendix C

RAM Heatmaps



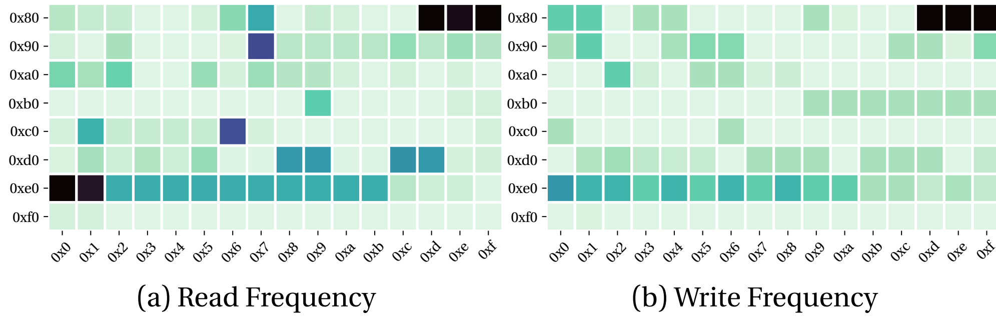
(a) Read Frequency (b) Write Frequency

Figure C.1: RAM Usage Heatmaps for *Combat*.



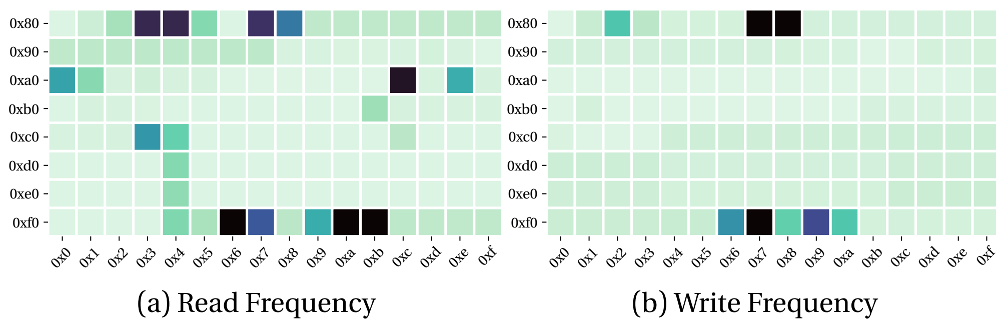
(a) Read Frequency (b) Write Frequency

Figure C.2: RAM Usage Heatmaps for *River Raid*.



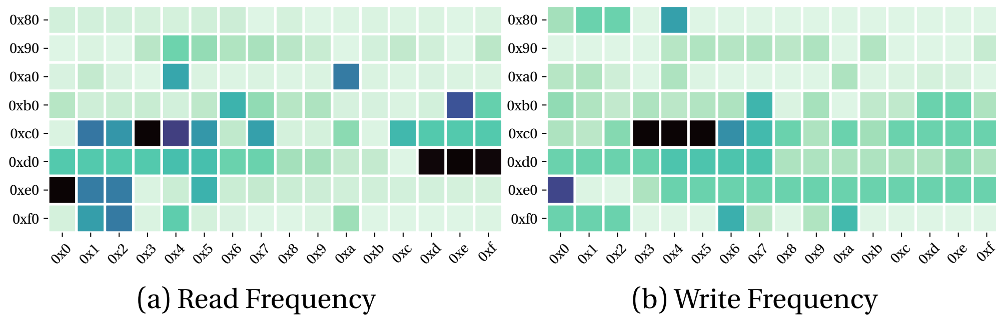
(a) Read Frequency (b) Write Frequency

Figure C.3: RAM Usage Heatmaps for *Dragonfire*.



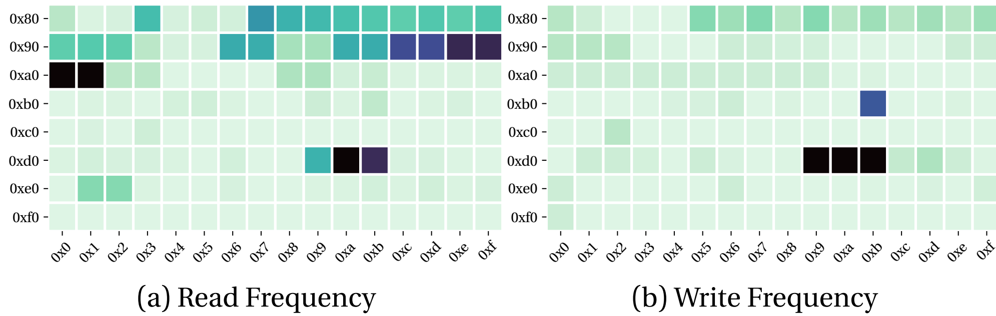
(a) Read Frequency (b) Write Frequency

Figure C.4: RAM Usage Heatmaps for *Kaboom!*



(a) Read Frequency (b) Write Frequency

Figure C.5: RAM Usage Heatmaps for *Moonsweeper*.



(a) Read Frequency (b) Write Frequency

Figure C.6: RAM Usage Heatmaps for *Montezuma's Revenge*.

(a) Read Frequency

(b) Write Frequency

Figure C.7: RAM Usage Heatmaps for *Solaris*.

Appendix D

Coverage Jumps

	Total # of Games	# of Games	Coverage (%)
Boxing	2253		
		1	98.14
		17	98.24
		1244	98.97
Combat	1839		
		1	73.88
		2	74.27
		5	74.56
		10	74.66
		22	74.90
		54	75.15
		129	75.39
		412	75.63
		742	75.88
Kaboom!	5703		
		1	90.53
		2	92.87

(*table continues*)

	Total # of Games	# of Games	Coverage (%)
		3	92.97
		4	95.12
		211	95.17
		1052	95.21
		1183	95.26
		1634	95.31
		2354	96.53
		2611	96.58
		5443	96.63
Dragonfire	3298		
		1	53.05
		653	53.22
		711	86.52
		733	86.77
		750	87.23
		879	87.62
		899	88.40
		913	88.43
		1007	88.45
		1071	88.48
		1117	88.50
		1252	89.55
		1644	89.60
		1704	89.62
		2458	89.65
River Raid	4800		
		1	87.13
		2	87.18
		3	87.60
		9	88.13
		17	88.33

(*table continues*)

	Total # of Games	# of Games	Coverage (%)
		42	88.38
		108	88.43
		208	89.16
		351	89.26
		471	89.36
		503	89.50
		504	90.01
		527	90.21
		531	90.28
		542	90.60
		545	90.67
		1204	90.89
		1409	91.38
		1713	91.43
		1888	91.48
		2010	91.85
		3106	91.89
		4210	92.14
Moonsweeper	2563		
		1	80.57
		2	82.14
		3	83.26
		4	83.83
		6	84.55
		9	84.57
		11	84.58
		13	84.85
		14	84.97
		19	84.99
		26	85.01
		29	85.22

(*table continues*)

	Total # of Games	# of Games	Coverage (%)
		33	85.49
		37	85.50
		43	85.51
		47	85.52
		51	85.53
		123	85.57
		196	85.60
		397	85.62
		650	85.64
Montezuma's Revenge	10		
		1	57.92
		3	62.78
		4	62.82
		5	65.99
		6	66.19
Solaris	510		
		1	79.44
		2	80.19
		3	84.00
		4	85.24
		5	85.60
		6	85.79
		8	85.96
		9	88.37
		10	88.91
		11	88.93
		15	89.03
		16	89.05
		17	89.07
		20	89.28
		21	89.29

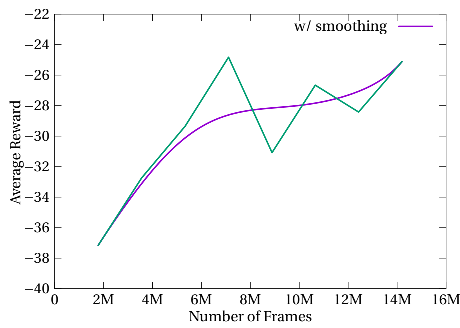
(*table continues*)

Total # of Games	# of Games	Coverage (%)
	27	89.36
	28	89.39
	32	89.42
	33	89.61
	34	89.62
	38	89.81
	51	89.84
	55	89.85
	56	89.87
	57	89.88
	58	89.90
	64	89.92
	75	89.98
	88	90.12
	96	90.23
	110	90.24
	115	90.25
	128	90.26
	139	91.16
	181	95.82
	221	95.83
	245	95.84
	335	95.84
	405	95.87
	454	95.96

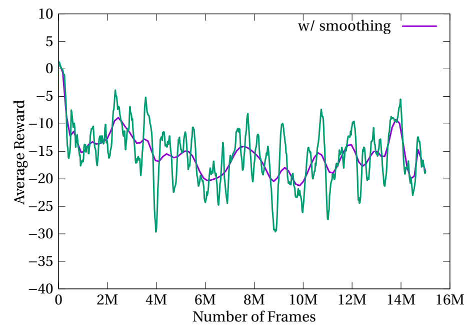
Table D.1: Coverage changes for deep q-network (DQN) 15 M frames. Only one environment which played the most number of games was included as the number of games played varied among environments.

Appendix E

Training Graphs

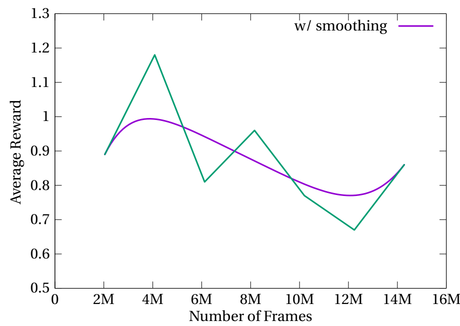


(a) DQN

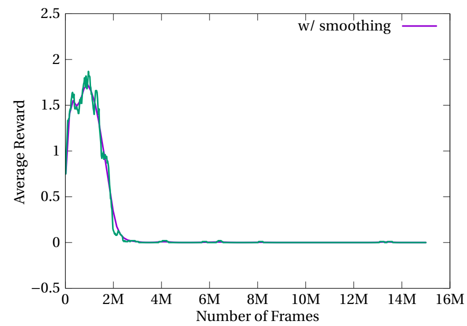


(b) A2C

Figure E.1: Average reward over session for *Boxing*.

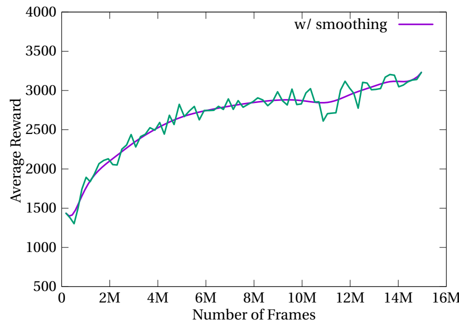


(a) DQN

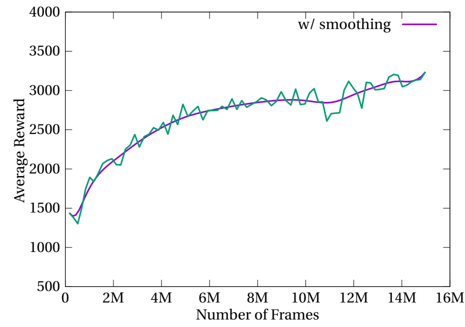


(b) A2C

Figure E.2: Average reward over session for *Combat*.

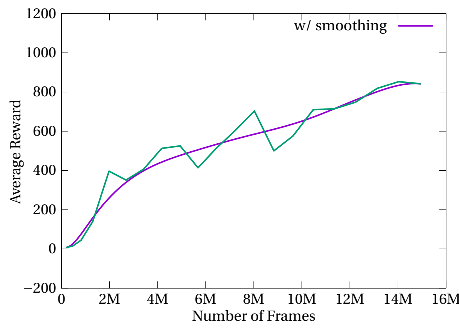


(a) DQN

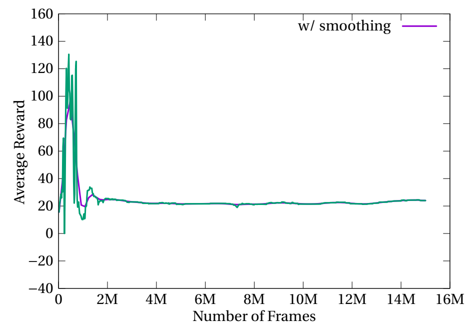


(b) A2C

Figure E.3: Average reward over session for *River Raid*.

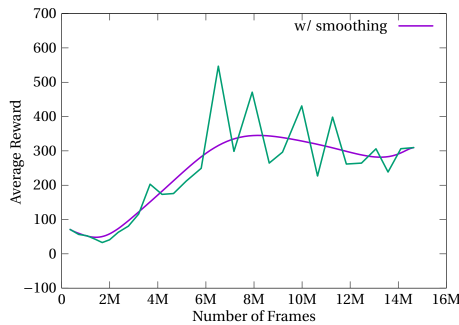


(a) DQN

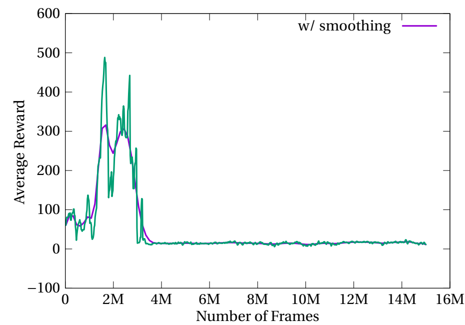


(b) A2C

Figure E.4: Average reward over session for *Kaboom!*

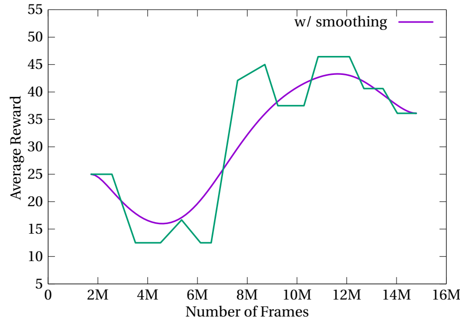


(a) DQN

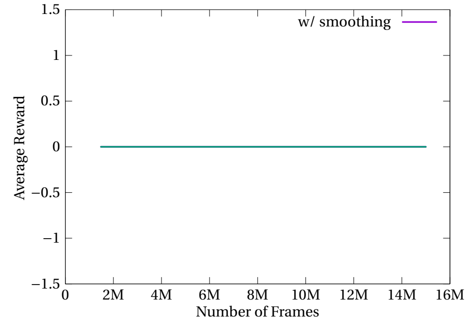


(b) A2C

Figure E.5: Average reward over session for *Moonsweeper*.

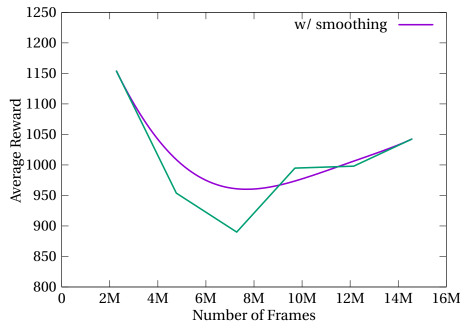


(a) DQN

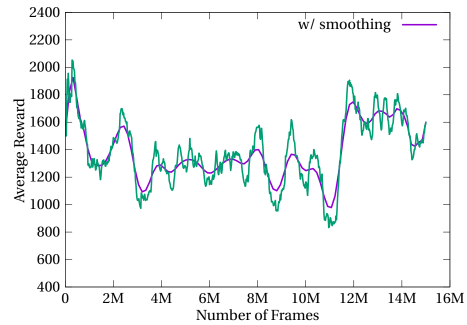


(b) A2C

Figure E.6: Average reward over session for *Montezuma's Revenge*.



(a) DQN



(b) A2C

Figure E.7: Average reward over session for *Solaris*.