DEPARTMENT OF COMPUTER SCIENCE

A Study of the Coverage and Precision of the CWE When Analysing Unusual Software

Performed Using Mario Games

[redacted]

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Thursday 4th May, 2023

Abstract

The Common Weakness Enumeration (CWE) is a resource that attempts to list and categorise as many common software weaknesses as possible. In addition, the list is intended to serve as a common ground for unified taxonomy to clarify the discussion surrounding these weaknesses and therefore support further progress in the field. In this research project, I have used a qualitative research techniques to identify the weaknesses behind observed erroneous behaviour in the investigated software and compared them to weaknesses from the CWE. The project objective is to discover any missing or ambiguous definitions in the CWE. The main contributions and achievements of the project are as follows:

- Identified and classified over 100 bugs within 3 early Super Mario games, through a combination of firsthand reports and personal verification.
- Used a combination of open and closed coding to describe and categorise underlying weaknesses responsible for the observed behaviours.
- Defined multiple labels for weaknesses for the causes of behaviours that I felt could not be adequately described by labels present in the CWE.
- Validated my results through checking for inter-rater reliability and therefore providing support for the necessity of my created labels.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

[redacted], Thursday 4th May, 2023

Contents

List of Figures

List of Tables

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Joseph Hallett.

Supporting Technologies

- I used the Scikit-Learn library to compute Cohen's Kappa while investigating inter-rater reliability.

Notation and Acronyms

CWE	:	Common Weakness Enumeration
SMB1	:	Super Mario Bros.
SMB3	:	Super Mario Bros. 3
SMW	:	Super Mario World

Chapter 1

Introduction

1.1 Motivation

Software security has always been of great importance. As our lives become more and more inextricably linked with the technology we use, in ways such as digital banking, self driving cars and smart household devices, it has only become more important that software is kept as secure as possible. Exploitable unintended behaviours could have catastrophic consequences if they are not found before distribution, as the products containing them can be quickly distributed to very large audiences in the modern day, all of whom would be at risk.

To properly maintain software security, it is imperative that those involved are aware of the types of weaknesses that could present themselves and the ways in which they manifest. Conversely, weaknesses that are poorly defined present a more significant threat as this lack of clarity can interfere with attempts to rectify the behaviour they cause. To this end, organisations such as the MITRE Corporation undergo projects such as the Common Weakness Enumeration (CWE) to aggregate understanding of these weaknesses and ensure they can be clearly identified and discussed.

The CWE list was created in 2006, and has been iteratively developed since. It contains 1332 unique labels for weaknesses, including 459 labels defining categories, subcategories or specific weaknesses for software development. This forms the basis of my research; these labels constitute the closed codebook I use in my first pass over the behaviours I have documented. It is also important to note that, as a resource under constant development, there are potential gaps or discrepancies in detail contained within. This lays the foundation for the main intended goal of this research; I expect to encounter these absences or discrepancies and make an attempt to resolve them.

Another key to continued software security is the innovation and improvement of state of the art techniques for discovering unintended behaviours. For proprietary software and other such instances where the source code is unavailable, advanced techniques and technologies are necessary for a third party wishing to evaluate the security of the product. As manual testing is slow and vulnerable to human biases in the process of using technology, automated processes such as fuzzing are employed. The forward march of vulnerability detection techniques and tools is responsible for the continued discovery of new bugs as software is written with old bugs in mind.

So far I have explained the importance of software security in a general sense, and acknowledged relevant modern techniques. But more importantly, why video games? As pieces of software like any other, it's clearly apparent that they have software flaws and security weaknesses. Despite this, games are very rarely treated from a security perspective with similar importance as other types of software. Without security conferences investigating these games' weaknesses and a general lack of professional investigation outside of their development, we can't say with any certainty how similar the weaknesses present in games are to other software.

Although video games do not often appear at conferences, there are independent communities surrounding these games that endeavour to find out as much about their software vulnerabilities as possible. The speedrunning community is most notable for this, doing their best to find bugs that can be exploited to improve clear times of various games. As a consequence of the disconnect from the traditional software security industry, these communities have developed terminology of their own to describe weaknesses that may or may not be analogous to various CWEs. This is the other half of my starting point: these communities have created large collections of bugs defined in their own terms, and in this project I am attempting to codify as many of these in more traditional terms as possible.

1.2 Aims & Objectives

The following are the aims and objectives of this research project:

- Identify underlying causes behind observed erroneous behaviour
- Identify unclear or lacking definitions in the CWE
- Create acceptable labels for weaknesses that need them

Chapter 2

Background

2.1 Grounded Theory

Grounded Theory is a research method first codified by Glaser and Strauss [10] for use as a qualitative research strategy in the field of sociology. This involves first collecting sufficient amounts of empirical data, from which the researcher should observe emerging patterns that can be described by theory. This process repeats, with further passes refining the codes until no further changes are made, known as information saturation. More specific detail on this process will be covered in the methodology section.

Although conceived as a strategy for sociologists, with related fields such as nursing being an acknowledged use case, grounded theory has also been extended to computer science [1]. The cited work used Glaserian Grounded Theory, whereas this project is derived from Strauss' work. Nonetheless, the main ideas of all Grounded Theory methods are from the same source, allowing all methods to be equally valid for relevant problems. This is acknowledged in the aforementioned article, advising that the best method to use is most dependent on your access to supporting resources as opposed to the differences in the methods themselves.

Despite the above demonstration of the method's applicability to the field, there are some ways in which this project violated Strauss' grounded theory principles as outlined in [8]. In this project, data collection and analysis were naturally disjoint due to it being based on a set of known observered incorrect behaviours, and thus analysis of early data could not more accurately inform later collection of further data as advised.

2.2 Bug Discovery

Software security is a very important area, so naturally large amounts of varied research are invested in improving our ability to find critical bugs before they can be abused. One such area is the realm of software tools, where these tools and how they are used are evaluated [3] and innovated on to be made more effective [4]. As mentioned in the latter, new vulnerabilities are constantly discovered. In particular, it discusses the necessity of approaches such as Fuzzing for verifying the integrity of software without available source code. Due to not having access to the source code, the only way to discover bugs is by observing behaviours. Similarly, this project attempts to identify and classify bugs from observed behaviours. However, I am collecting these behaviours through a combination of finding firsthand reports and manual confirmation.

Relevant to this difference is another field of bug discovery, that of bug bounty programs (BBPs). BBPs take a crowdsourcing approach to discovering bugs, typically offering compensation for individuals who find bugs. The exact format of the BBP can vary due to many factors[12], but all such formats are fundamentally similar in their motivating work with rewards contingent on successes. The nature of these programs allowed the formation of legitimate software vulnerability markets [2] where discovered bugs can be treated as a commodity, bought and sold due to their redeemable bounty value, intended to discourage these sales on black markets to potentially malicious individuals. While my project is founded on an initial croudsourced dataset of erroneous behaviours, no financial incentives were offerred. This dataset was created through a combination of the large time the software I'm investigating has existed for, allowing plenty of opportunity for bugs to be found, and the difference in cultures surrounding video games and more traditional software.

2.3 Bug Classification

Similar justifications for research into improving methods for discovering bugs can be applied to attempts to improve classifications of bugs, although there is less research to be found here. The Common Weakness Enumeration [9] is a very large and continuously updated resource for the listing and categorisation of software vulnerabilities. This resource is the starting point for the vulnerability classification performed in this project. While the CWE lists very many bugs, the classification hierarchy leaves much to be desired, as the level of specificity can vary wildly depending on the type of bugs being described. For this reason, one of the goals of this project is to provide an equal and sufficient level of description for the categories of classification that I create.

The Bugs Framework (BF) [5] is an attempt to provide more meaningful and precise classifications for bugs. To do this, the BF attempts to break weaknesses down into orthogonal components, from which all manner of bugs can be constructed. Further following work into classification of memory bugs [6] introduced four additional classes exclusively targeting memory bugs and supported the previous conclusions asserting that clearer and more precise definitions allow for better communication and therefore efficacy in designing around known weaknesses. My project also attempts to improve communication similarly with more meaningful definitions, however I am focusing more on also extending the CWE itself and therefore following its convention of descriptive definitions instead of taking an approach like BF that seeks to methodically define bugs by their components.

2.4 Weakness Analysis

Work analysing the presence and distribution of software weaknesses is not a novelty. Work such as trend analysis [7] has been done and show the potential significance of this investigation. The cited work involved covering a very large range of weaknesses (specifically Common Vulnerabilities and Exposures, CVEs) and analysing the trends in frequency and severity of instances of these weaknesses. My work draws clear parallels, however being conducted on a much smaller and less contemporary scale: I am also seeking trends over time in the weaknesses observed.

More similar in scope to my project is [14], as it focuses on a specific set of software as opposed to observing trends over such a large set of weaknesses as the previous one. Both of these papers illustrate a considerably more rigorous and security focused methodology: unlike my intention to just improve CWE coverage of my chosen software, this paper is using existent CWE and CVE vulnerability information to interpret severity through the Common Vulnerability Scoring System (CVSS). Although the scope is more similar here, this paper makes no attempt to evaluate CWE coverage but rather just uses this information to calculate numerical judgements of weakness severity. [11] also does similarly, and involved manual classification of weaknesses like my project. However, this involved access to source code, which is unavailable to me for my purposes.

Chapter 3

Methodology

This chapter will be divided into two sections. First, I will describe the overall process of the project, explaining what I have done, why this method was chosen, and how it is methodologically sound. Second, I will briefly demonstrate the categorisation process using an example.

3.1 Overall Process

3.1.1 Data Collection & Verification

To be able to perform any coding, I had to start with a dataset of bugs sufficiently large for patterns to form. My starting point for this was the mario wiki [13], a community resource containing large but non-exhaustive lists of common bugs in these games. Through a combination of this and following their cited resources (which consists mostly of decentralised youtube videos) I created my dataset. Although many were associated with footage, sometimes multiple recordings, that made them easy to understand and replicate, there were also bugs described exclusively with text that I had to manually verify the existence and behaviour of. Some of these were fairly trivial to replicate, such as SMB1's "Harmless Enemy" glitch having an easily followed and executed description, however others such as the "Disappearing Jumping board" had vague or incomplete descriptions that made replication difficult. Overall, I was unable to replicate 3 bugs in SMB3.

3.1.2 Closed & Open Coding

Once I had created my dataset, I proceeded with a series of coding passes over it. Closed coding refers to describing each entity in the set with a label from a closed codebook: in this context, the closed codebook was the complete CWE reference which I attempted to accurately describe as many bugs as possible with. Open coding, by contrast, involves creating categories to divide your data into. This was only used when I could not find an appropriate CWE. At first I used closed coding, attempting to define as many of these bugs as possible with the codes available to me before creating my own without sacrificing accuracy of description. After this had been done I introduced open coding, allowing myself the flexibility to create labels when I felt none of those available to me were appropriate. I conducted several passes over this dataset, adjusting labels as I saw fit, until at various stages of the project I was happy with the current state and ready to progress.

3.1.3 Inter-Rater Reliability

The final step of the process was to, after creating all of my labels and conducting enough passes over the dataset that I was no longer making adjustments to my labelling, check for inter-rater reliability. This is done to improve the reliability of the results by ensuring they aren't contingent entirely on one individual's perspective. To do this I calculated Cohen's Kappa using Scikit-Learn, a robust method of measuring agreement that factors probability of chance agreement into its evaluation and is therefore more informative than simple percentage agreement. Our resulting low Kappa of 0.13 demonstrated the need for a review of the current state of the codebook, to resolve conflicting labels. After a meeting with this as the main objective, I conducted an additional pass over the dataset to put into place the discussed changes. These changes are described in more detail in the results chapter.

3.1.4 Justification

This method was chosen due to the nature of the software and field being investigated. Although some games do have accessible source code, due to a combination of leaks and reverse engineering, the vast majority do not. The method chosen here is intentionally replicable without access to the source code of the games investigated, allowing it to be generalised to other systems and franchises. Given this restriction, a qualitative research approach seemed most appropriate. In addition to this applicability independent of source code, another crucial aspect of the closed and open coding approach is that it provides a very clear way to evaluate the coverage of the CWE. As investigating this is the primary goal of this project, this approach was chosen.

3.1.5 Limitations

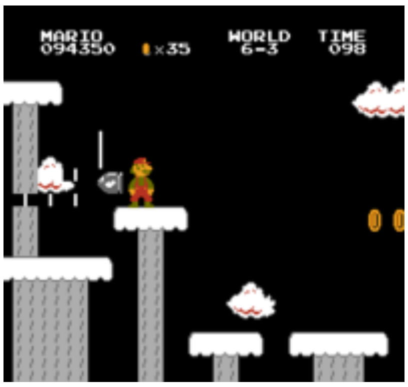
Although I believe this to have been an appropriate justified method to use, some features of this method present a threat to the validity of my results. First and foremost, all of the coding is largely dependent on individual speculation and personal opinion. As a consequence, many of the more complex bugs that cannot be trivially defined with distinct low level errors have labels that cannot be proven to be correct using only this method. This is the purpose of collaboration and comparison with a second coder: agreement between individuals suggests that personal opinion is a small factor, and requiring that conflicts are resolved ensures that these points of personal opinion are addressed.

Another limitation is the potentially incomplete dataset. I used a dataset composed of bugs reported on the internet by players of these games. This is not an exhaustive set created by analysis of the code and application of appropriate tools, but rather just a collection of bugs prominent enough to be common knowledge. While this does suggest that the dataset, and therefore findings, may not be truly representative of the proportion of weaknesses present, it also ensures that the results found are representative of the most common and likely to be encountered bugs, which I believe are important factors in this research.

3.2 Example Coding

Here I will cover a brief example of the coding process, using one of the bugs present in the dataset I used.

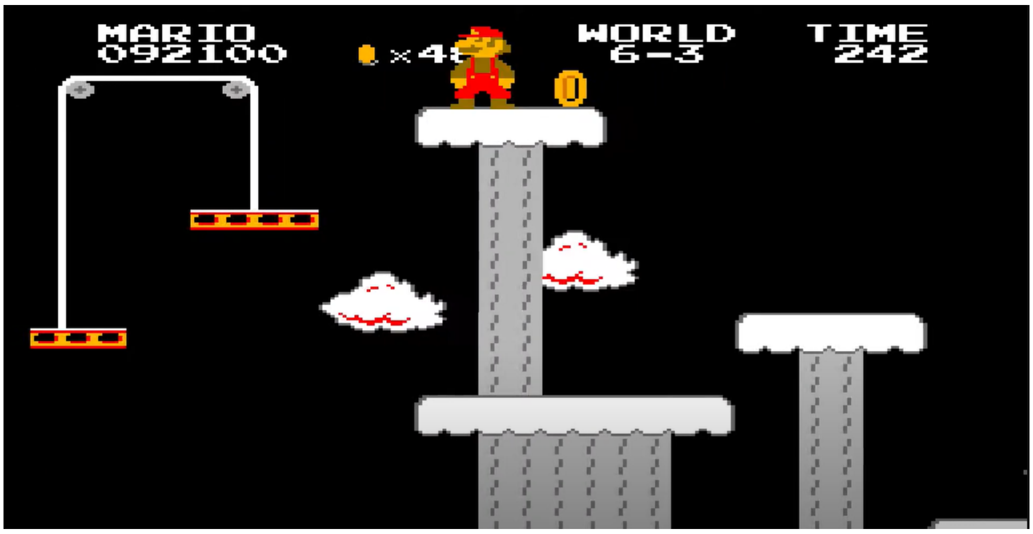
Figure 3.1: The results of the SMB1 Bullet Lift glitch



3.1 above shows the observed behaviour of the bullet lift glitch, where jumping on bullets doesn't knock them down as you'd expect but instead leaves them suspended, with a line being left behind them. Mario reacts normally, still scoring points hitting them for the first time and bouncing upwards on contact. This partially correct behaviour is very unusual, and it doesn't seem to be obviously derived from a common programming flaw.

When confirming the bug for myself, I noticed the lift in 3.2 is located immediately before where one has to wait to perform this glitch. This is the only level in the game that contains both a scale lift and bullet enemies, and these lifts also behave very similarly to how the bugged bullet behaves when Mario jumps on it: descending only while Mario is in contact, and leaving a trail behind. This made me quite confident the bullet was mistakenly reading behaviour from the lift.

Figure 3.2: Lift present to the left of the glitch site



Given this, I searched the CWE for similar types of weakness. While CWE-362 (race condition) discusses accessing a shared resource, this bug had no concurrency component, as the lift was entirely offscreen at this point and not being interacted with. No other CWE closely matches this notion of shared behaviour, so I decided that if I could find other similar types of behaviour I would create a new label for them, which eventually led to me defining the ”Incorrect Share” label.

Chapter 4

Results

This chapter consists first of a set of tables detailing the state of the codebook at various points, with accompanying descriptions of the process up to the that point, and then a description of the overall meaning and significance of the final state of the codebook including patterns that arise in the CWEs found and descriptions of the non-CWE labels.

4.1 Tables

The following are a series of tables detailing the labels and their distribution over the course of multiple coding passes. Labels created by me to describe bugs poorly defined by the CWE are in **bold.**

Table 4.1: First Pass

Label	SMB1	SMB3	SMW	Total
Off By One 193	2	1	0	3
Early Validation 179	2	0	0	2
Unique 837	1	0	3	4
Cleanup 459	1	4	1	6
Missed Synchronisation 820	3	1	1	5
Overflow 190	1	0	0	1
Wrap-Around Error 128	2	1	0	3
Out of Bounds Read 125	2	0	0	2
Improper Exceptional Handling 755	2	0	2	4
After End of Buffer 788	1	0	0	1
Numeric 189	1	8	3	12
Integer Coercion 192	1	0	0	1
Incorrect Sync 821	0	5	1	6
Improper Validation 20	0	2	0	2
Improper Check 754	0	1	0	1
Unreachable Exit 835	0	0	1	1
Buffer Overflow 120	0	0	1	1
Initialisation 665	0	0	2	2
Double Coded	0	3	0	3

The first pass exclusively focused on bugs that could easily be defined by definitions already present in the CWE. As a result, at this point in the process I had created no labels and only 60 of the final total of 119 bugs had a code.

Table 4.2: Second Pass

Label	SMB1	SMB3	SMW	Total
Off By One 193	2	1	0	3
Early Validation 179	2	0	1	3
Function 628	1	0	0	1
Unique 837	1	0	3	4
Cleanup 459	2	5	3	10
Missed Synchronisation 820	3	1	2	6
Overflow 190	1	0	0	1
Wrap-Around Error 128	2	1	0	3
Out of Bounds Read 125	2	3	0	5
Improper Exceptional Handling 755	4	0	3	7
After End of Buffer 788	1	0	0	1
Numeric 189	1	8	3	12
Integer Coercion 192	1	0	0	1
Incorrect Sync 821	0	5	1	6
Improper Validation 20	3	6	1	10
Improper Check 754	0	2	0	2
Unreachable Exit 835	0	0	1	1
Buffer Overflow 120	0	0	1	1
Initialisation 665	0	0	2	2
Entity Limit	2	1	2	5
Missing Sprite	1	1	0	2
Improper Init	2	0	0	2
Recheck State	0	2	0	2
Area Palette	0	1	0	1
Anim Update	0	0	1	1
Behaviour Persists	0	0	2	2
Concurrent	0	0	1	1
Misplaced Entities	0	0	1	1
CWE	26	32	21	79
NEW	5	5	7	17
Double Coded	0	3	2	5

After the second pass, 19 bugs remained unlabelled due to complexity or other difficulties in understanding the low level issues causing the strange behaviours observed. At this point I had started the process of defining labels that I believed supplemented the CWE.

Table 4.3: Second Pass Review

Label	SMB1	SMB3	SMW	Total
Off By One 193	2	1	0	3
Early Validation 179	2	0	1	3
Function 628	1	0	0	1
Unique 837	1	0	3	4
Cleanup 459	2	5	3	10
Missed Synchronisation 820	3	1	2	6
Overflow 190	1	0	0	1
Wrap-Around Error 128	2	1	0	3
Out of Bounds Read 125	2	3	0	5
Improper Exceptional Handling 755	4	0	3	7
After End of Buffer 788	1	0	0	1
Numeric 189	1	8	3	12
Integer Coercion 192	1	0	0	1
Incorrect Sync 821	0	5	1	6
Improper Validation 20	3	6	1	10
Improper Check 754	0	2	0	2
Unreachable Exit 835	0	0	1	1
Buffer Overflow 120	0	0	1	1
Initialisation 665	0	0	2	2
Entity Limit	2	1	2	5
Missing Sprite	1	1	0	2
Improper Init	2	0	0	2
Recheck State	0	2	0	2
Area Palette	0	1	0	1
Anim Update	0	0	1	1
Behaviour Persists	0	0	2	2
Concurrent	0	0	1	1
Misplaced Entities	0	0	1	1
CWE	26	32	21	79
NEW	5	5	7	17
Double Coded	0	3	2	5

This review was conducted shortly after the second, and largely constituted of confirming the previous set of codes with myself. As no dedicated experimentation had occurred between the two, the number of unlabeled bugs barely changed.

Table 4.4: Third Pass

Label	SMB1	SMB3	SMW	Total
Off By One 193	2	1	0	3
Early Validation 179	2	0	1	3
Function 628	1	0	0	1
Unique 837	1	0	3	4
Cleanup 459	2	5	3	10
Missed Synchronisation 820	3	2	2	7
Overflow 190	1	0	0	1
Wrap-Around Error 128	2	1	0	3
Out of Bounds Read 125	2	4	0	6
Improper Exceptional Handling 755	2	0	3	5
After End of Buffer 788	1	0	0	1
Numeric 189	1	11	3	15
Integer Coercion 192	1	0	0	1
Incorrect Sync 821	0	6	1	7
Improper Validation 20	3	4	1	8
Improper Check 754	0	2	0	2
Unreachable Exit 835	0	0	2	2
Buffer Overflow 120	0	0	1	1
Initialisation 665	0	0	2	2
Entity Limit	2	1	2	5
Missing Sprite	1	1	0	2
Improper Init	2	4	1	7
Recheck State	0	3	1	4
Area Palette	0	0	0	0
Anim Update	0	0	0	0
Behaviour Persists	0	0	2	2
Concurrent	0	0	1	1
Misplaced Entities	0	0	1	1
Incorrect Share	1	2	1	4
Improper Correction	2	0	0	2
Incomplete Definition	0	1	1	2
CWE	24	36	22	82
NEW	8	12	10	30
Double Coded	0	4	4	8

My third pass was the result of further testing of more complicated bugs, and at this point all of the bugs now had a coding. This also marks the point at which I had created all of the newly defined weakness labels that I used, although some of these did not persist until the final version.

Table 4.5: Third Pass Review

Label	SMB1	SMB3	SMW	Total
Off By One 193	2	1	0	3
Early Validation 179	2	0	1	3
Function 628	1	0	0	1
Unique 837	1	0	4	5
Cleanup 459	2	5	3	10
Missed Synchronisation 820	3	2	2	7
Overflow 190	1	0	0	1
Wrap-Around Error 128	2	1	0	3
Out of Bounds Read 125	2	4	0	6
Improper Exceptional Handling 755	2	0	3	5
After End of Buffer 788	1	0	0	1
Numeric 189	1	11	3	15
Integer Coercion 192	1	0	0	1
Incorrect Sync 821	0	6	1	7
Improper Validation 20	3	4	1	8
Improper Check 754	0	5	0	5
Unreachable Exit 835	0	0	2	2
Buffer Overflow 120	0	0	1	1
Initialisation 665	0	0	2	2
Entity Limit	2	1	2	5
Missing Sprite	1	1	0	2
Improper Init	2	5	1	8
Recheck State	0	3	1	4
Behaviour Persists	0	0	3	3
Concurrent	0	0	1	1
Misplaced Entities	0	0	1	1
Incorrect Share	1	2	1	4
Improper Correction	2	0	1	3
Incomplete Definition	0	1	2	3
CWE	24	39	23	86
NEW	8	13	13	34

This pass was similar to the first review, in that it involved reviewing the previous pass ensuring satisfaction before moving on to further steps, in this case before checking for inter-rater reliability. This necessitated committing to one label for certain behaviours I had until this point given double codes. Again, the number of changes is small as a consequence, and since the entire set of bugs has now been labeled no new bugs were labeled.

Table 4.6: Final Pass

Label	SMB1	SMB3	SMW	Total
Off By One 193	2	1	0	3
Early Validation 179	2	0	1	3
Unique 837	1	0	4	5
Cleanup 459	2	7	7	16
Missed Synchronisation 820	3	3	2	8
Overflow 190	1	0	0	1
Wrap-Around Error 128	2	1	0	3
Out of Bounds Read 125	2	4	0	6
Improper Exceptional Handling 755	2	0	4	6
After End of Buffer 788	1	0	0	1
Numeric 189	1	11	3	15
Integer Coercion 192	1	0	0	1
Incorrect Sync 821	0	6	1	7
Improper Validation 20	3	4	1	8
Improper Check 754	0	5	0	5
Unreachable Exit 835	0	0	2	2
Buffer Overflow 120	0	1	1	2
Initialisation 665	0	0	1	1
Entity Limit	2	1	2	5
Improper Init	2	3	1	6
Recheck State	0	2	1	4
Incorrect Game Elements	1	1	1	3
Incorrect Share	1	1	1	3
Improper Correction	2	0	1	3
Incomplete Definition	0	1	2	3
CWE	24	43	26	93
NEW	8	9	10	27

The final pass was conducted after checking for inter-rater reliability. As discussed in the previous chapter, reliability was quite low and therefore I had a meeting with my secondary coder to resolve our conflicts. This led to some small changes, most notably a reduction in the number of unique new labels defined due to our agreement that too many similar options were now being provided.

4.2 Analysis

4.2.1 Categories of CWE

The weaknesses found within these games largely fell within 4 categories: synchronisation and concurrency, numerical interpretation, conditionals and validation, and transitions between states.

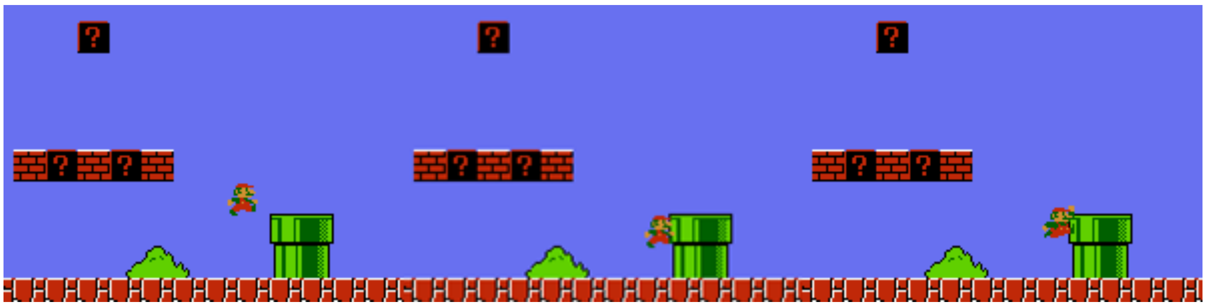
Synchronisation and Concurrency

Synchronisation and Concurrency Weaknesses:
CWE-179: Incorrect Behaviour Order: Early Validation
CWE-820: Missing Synchronisation
CWE-821: Incorrect Synchronisation
CWE-837: Improper Enforcement of a Single, Unique Action

The state of these games is updated at a constant rate, regardless of the events happening. This makes the handling and synchronisation of concurrent events a necessity, as it is very often that multiple interactions happen on the same frame and must be updated simultaneously. The relevant CWEs to this are 179 (early validation), 820 (missing synchronisation), 821 (incorrect synchronisation), and 837 (improper enforcement of unique actions).

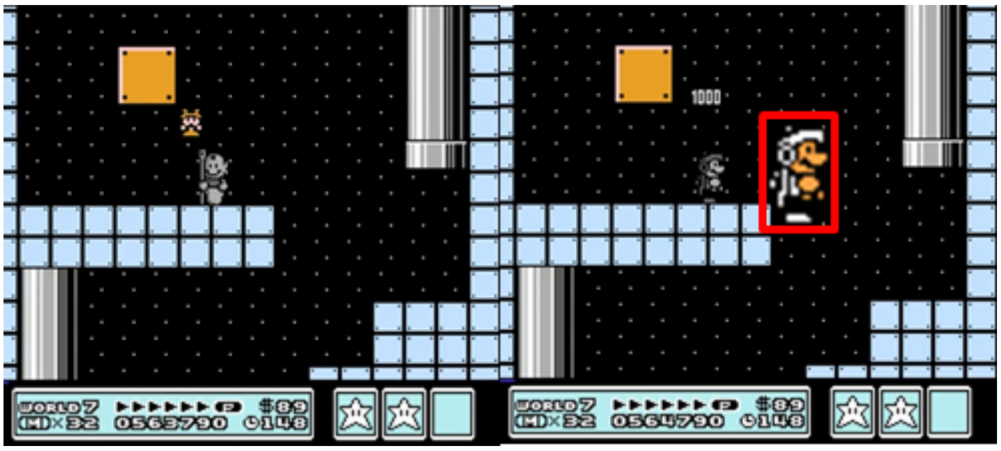
CWE-179 is found when two events happen simultaneously, where the completion of one of these events would affect or prevent the other event's completion. An example of this is the ability to wall jump in SMB1 by jumping towards a wall and pressing the jump button again: the game checks if Mario is standing to be able to jump, and the game also checks if Mario needs to be pushed outside of a solid object. As walls consist of stacked blocks, Mario can be standing on the top of one of these blocks, and therefore satisfy jumping requirements. Due to this, Mario's jump input is validated, even though after resolving the other event of pushing Mario out of the wall he is no longer standing on a surface he can jump from. 4.1 below shows this behaviour in action: although Mario can never stand still on the edge of the pipe, he is able to transition directly from falling to the upwards movement of a new jump. The second and third screenshots are consecutive frames, and Mario's height is already easily seen to increase.

Figure 4.1: Three screenshots from SMB1, showing Mario falling towards a pipe then jumping in midair



CWE-820 and CWE-821 are both fairly common. CWE-821 is completely absent in SMB1, as there appears to be no attempt to synchronise concurrent events; instead what happens is that one event is chosen and the other is missed. Examples of this include attempting to throw a fireball while reaching the bottom of the flagpole, both actions that trigger unique animations, which results in the traditional flag slide animation being skipped in favour of the other animation. CWE-820 continues to appear in SMB3 and SMW, generally when failing to resolve changes to Mario's state. For example, transforming into a Tanuki statue while collecting a different powerup creating a new state for Mario combining the new powerup with some of the statue's behaviours. Most obviously Mario's sprite is the wrong colour, retaining the statue colour palette, but he is also unable to go through pipes due to this also being the case while as statue Mario.

Figure 4.2: Two SMB3 screenshots, showing statue Mario and then a grey Hammer Mario, with the normal palette for reference



CWE-821 is unique to SMB3 and SMW, demonstrating that Nintendo acknowledged after SMB1 the necessity for handling simultaneous events to prevent unintentional behaviour. While this typically ensures that all relevant concurrent events are resolved in some sequence, they do not always work as intended. Two clear examples of this come from SMB3. The first is collecting a 1-Up while using Tanuki Mario's tail: this causes the two sound effects to combine, not playing over one another but rather combining the constituent sounds of the two jingles in alternating sequence. Another example is entering a door while leaving the statue state: Mario will enter the door first, looking as though he has already left the statue state, however the animation for leaving said state will only occur once the next room has loaded.

CWE-837 is slightly different, as instead of exclusively being caused by same-frame events (though this also applies), it also applies to certain events being in process along others. An example of this latter form is the ability to die multiple times in SMB1: after pausing and unpausing while losing a life (which is otherwise noted as a related cleanup issue, the limitations of single coding will be discussed in later chapters), Mario coming into contact with something that would cause him to lose a life will repeat the animation and sound effect. A more straightforward same-frame example can be found in SMW, where two items placed sufficiently close to one another, or overlapping, can both be picked up due to neither one disabling the ability to pick up items until after both have been allowed (again demonstrating the limits of single coding, due to early validation potential). A clear single-coding example would be repeatedly passing below 100 seconds on the clock repeatedly increasing the background music to unintended speeds, as the speedup is stacked each time.

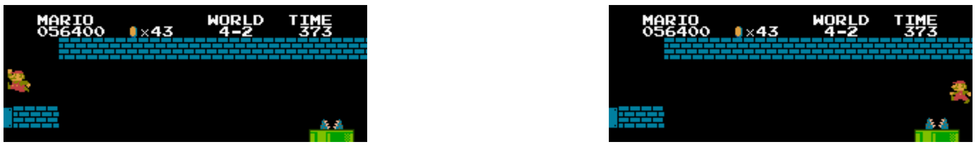
Numerical Logic

Numerical Logic Weaknesses:
CWE-189: Numeric Errors
CWE-193: Off-by-one Error
CWE-128: Wrap-around Error
CWE-192: Integer Coercion Error

One of the largest categories in the codebook is 189 (numeric errors), however it is not the only one in this category. Also included here are 193 (off-by-one errors), 128 (wrap-around error), and 192 (integer coercion), however these all occur in smaller numbers due to their increased specificity. CWE-189 is very general, and as a result covers a considerable range of different behaviours, whereas the other weaknesses mentioned are all quite specific and therefore describe fewer weaknesses each.

CWE-189 covers a lot of ground. The most notable category for this is various clipping bugs, where due to various numerical edge cases including angled surfaces and corners Mario or other game entities can pass through solid surfaces. Other numeric errors include offscreen values being truncated to instead be located inside the currently visible area, or Mario mistakenly being identified as in the wrong location due to a combination rounding of his pixel positions and generous detection areas. It is important to note that this is a category as oppposed to a specific weakness, and therefore is necessarily vague. As a result, it was only used when no subcategory was appropriate for the behaviour in question.

The remaining numeric errors are fairly straightforward. CWE-193 is most relevant for incorrectly not drawing sprites or neglecting their behaviour, as they are incorrectly calculated as outside of the visible/active area and therefore deemed unnecessary. CWE-128 is involved in mistakenly relocating objects at the edges of the screen, for example being hit by an enemy falling off the bottom of the screen while Mario jumps at the top, or being positioned at the far right while changing position on a vine at the far left. Finally, the lone CWE-192 occurrence involved interpreting a time of 000 being interpreted as a time of 1000 for scoring purposes. These are all easily identified and codified due to the high level of precision in the CWE descriptions.



(a) Mario starts on the left (b) But he appears on the right the following frame

Figure 4.3: An SMB1 Wrap-Around Error caused by a vine just offscreen to the left

As can be seen in 4.3, in a single frame update Mario is placed on the opposite side of the screen, as the x position of the slightly offscreen vine that Mario interacts with is interpreted as being the maximum value.

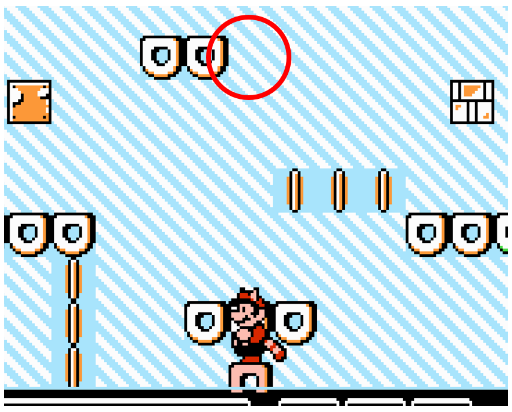
Conditionals and Validation

Validation Weaknesses:
CWE-20: Improper Input Validation
CWE-754: Improper Check for Unusual or Exceptional Conditions
CWE-835: Loop With Unreachable Exit Condition

This is a large category, and often responsible for the most notable bugs. CWE-20 (improper validation) is the broadest descriptor here, applying in general to validation not checking the appropriate conditions. More specifically, CWE-754 (improper check for unusual or exceptional conditions) and CWE-835 (unreachable exit condition) are identifiable as the causes for various behaviours.

Although CWE-20 is not a category like CWE-189, it is still quite vague. This results in it being ascribed a fairly varied set of behaviours. One such behaviour is the harmless enemy glitch from SMB1, where maintaining contact with an enemy after receiving damage. CWE-754 is more specific, representing edge cases that cause unexpected behaviour due to not properly checking conditions. An example of this is falling through donut blocks in SMB3, where while Mario is standing on a falling donut block, other floating donut blocks beneath him will be passed through instead of acting as a new platform.

Figure 4.4: A screenshot of Mario falling through a donut block: circled is where the block he's standing on fell from, and another can be seen behind him



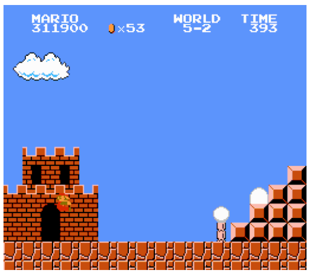
CWE-835 is also more specific, describing the relatively small area of unintended infinite loops where an intended exit condition cannot be triggered to end the loop. The most obvious instance of this is when Yoshi eats a control coin: as the control coin can no longer reach its destination to end its unique music, this music will instead just continue to play until the level is completed or exited.

Transitions Between States

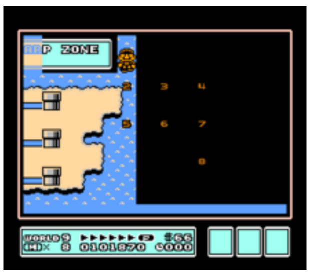
State Transition Weaknesses:
CWE-459: Incomplete Cleanup
CWE-665: Improper Initialisation
CWE-755: Improper Handling of Exceptional Conditions

This category is mostly focused on incorrectly managing screen transitions and Mario's transformations. Most prominent here is CWE-459 (incomplete cleanup), which is possibly large enough to justify a category on its own, as well as CWE-665 (improper initialisation) and CWE-755 (improper handling of exceptional conditions).

CWE-459 is the most represented single weakness in this codebook, being attributed to over 10% of the observed bugs. As a result, there are many varied examples of what it covers. in SMB1, it manifests in creating vines on certain screen spawns after dying on a vine. This is due to the nature of coin heaven, the only area that can be accessed from a vine in the game, always starting with a vine growing from the bottom of the screen. As a result, if the game incorrectly fails to clean up the state of the previous screen upon respawning, being on a vine is remembered. in SMB3 this is the cause of various types of bug, from switching maps while on a boat not removing Mario's ability to move on water, to toad house transitions and certain transformations incorrectly displaying Mario's sprite from before the event in question even if it's no longer legal (running in the toad house and sliding as hammer mario, respectively), to removing the time limit on white block invincibility when performing a screen transition. Both of these patterns return in SMW, in the most consistent bug category between games. Once again screen transitions erroneously remove certain timers and therefore allow intentionally limited behaviours to be maintained indefinitely, such as carrying enemies.



(a) Mario climbing on an invisible vine at the start of a stage, due to dying on a vine that goes to coin heaven



(b) Mario standing in the middle of the ocean, due to previously being on a boat

Figure 4.5: Two different-seeming bugs from different games both caused by failing to correctly clean up the state before a screen transition

CWE-665 only has one noted appearance, however this is partially due to my decision to construct a more specific label of "improper initialisation of a new state". This is also partially due to some level of overlap with cleanup; most bugs due to screen transitions involve loss of what should have been remembered from the previous screen, as opposed to erroneous initialisation of the current screen (suggesting which was considered more important in development). The single bug in question involves P-Switches in SMW re-initialising when brought to a new screen, and as their behaviour is dependent on where they are initialised this can create a P-Switch behaving as though it were the wrong colour. Unfortunately the switch still keeps its original appearance.

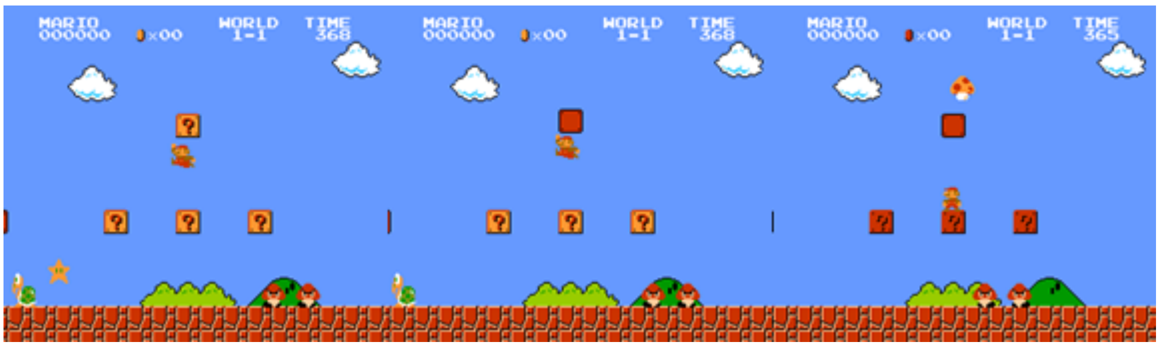
4.2.2 Non-CWE Labels

The non-CWE labels in the final codebook perform one of two functions: either they provide a more specific definition for a behaviour that is a subset of a fairly broadly defined CWE, or they define a uniquely present weakness that is not adequately explained even broadly by the CWE. I will describe each of my final labels here, with a brief acknowledgement of which of these purposes they serve.

Entity Limit

This weakness is somewhat related to issues of resource limitations in traditional software, however I could not find any appropriate weaknesses or categories in the CWE. Bugs described with this label involve behaviours caused by too many entities of a certain type (typically sprites) than there are spaces for them defined. As a consequence, entities may overwrite one another, fail to spawn, display incorrectly or even copy and combine behaviour. This label defines a weakness uniquely present in these games.

Figure 4.6: Three screenshots of SMB1, showing how spawning a new powerup removes a previous one from the screen

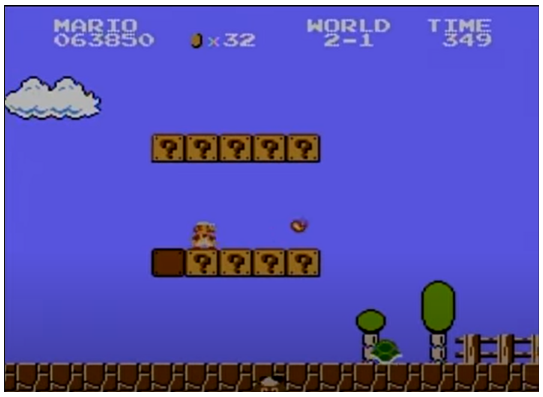


4.6 shows an example of this type of behaviour in action. SMB1 only allows for one powerup to be loaded into the game at a time. As a consequence, loading a new powerup overwrites the previous one immediately, regardless of what else may be happening. The reverse can happen to enemies, where additional enemies are preventing from appearing until space is made for them. This form of the bug is somewhat reminiscent of more traditional memory management bugs, where memory beyond what is available cannot be allocated. However, as it can manifest in other ways, I chose to keep this as its own label instead of making further attempts to describe it using traditional memory management CWEs

Improper Initialisation of a New State

This weakness is very closely tied to CWE-665, however I felt that "Improper Initialisation" was too broad in scope for my usage, as I was often describing the initialisation of small states like Mario's transformation as opposed to large scale initialisations such as screens. I believe that this CWE is well served with increased specification, as large-scale initalisation happens less frequently and in more predictable circumstances than small initialisations of specific states at unclear points based on user interaction.

Figure 4.7: A SMB1 screenshot of small Fire Mario, a transformation state that shouldn't be possible

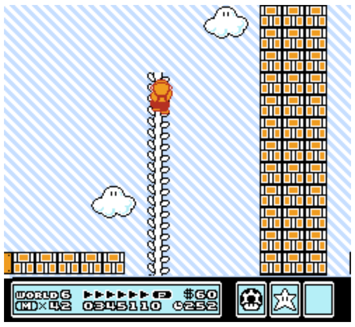


In 4.7 you can see Fire Mario with the proportions of small Mario. This is a consequence of the game failing to correctly initialise Mario's powerup state when completing a castle level at the same time as taking a hit. When the new level loads, Mario will appear to be large but loses a life in one hit as though he were small. This improperly initialised state causes further flawed transformation states, including the above small Fire Mario, as collecting powerups will still swap Mario's size despite him starting at the powered up size.

Failure to Appropriately Recheck Condition

The CWE is largely quiet on the subject of conditions. Properly managed conditionals are crucial to the correct functioning of these games, as many things have to be constantly checked against even at a very basic level, for example collision detection of various sorts. I was very wary to use this, due to its vague nature, and it was only used for persistent behaviours that the game erroneously stopped maintaining.

Figure 4.8: A SMB3 screenshot of Fire Mario climbing a vine that has stopped in midair

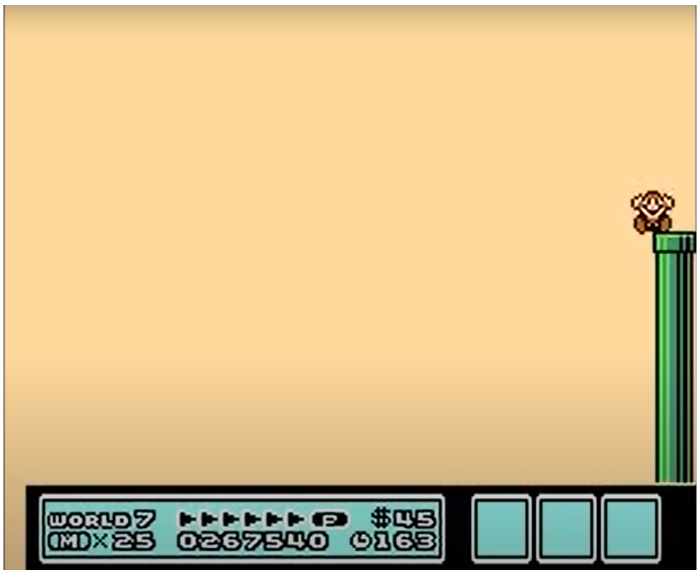


4.8 shows a clear example of this type of behaviour. The game checks if a vine is still on screen while it grows: if this is no longer the case, it stops growing. However, returning it to the screen does not check this condition again, and instead since this was not constantly checked the vine is permanently stuck at less than its maximum height.

Incorrect Game Elements

This was understandably absent from the CWE, due to it being intrinstically exclusive to the subject matter of games (although potentially generalisable to UI elements in general, the security application of this is questionable). The bugs defined here stem from the game simply being set up incorrectly: two enemies with no sprite set to spawn, and stage walls mistakenly defined to kill Mario.

Figure 4.9: A SMB3 screenshot of Mario dying on a pipe for no apparent reason

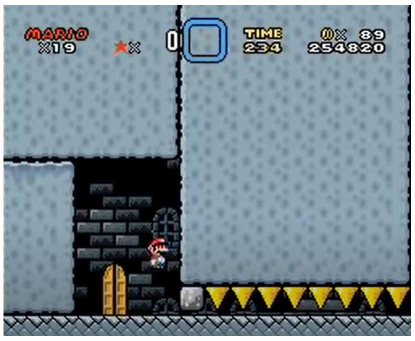


As seen in 4.9, Mario's death clearly indicates the presence of an enemy here (which we know to be an invisible muncher) despite the complete lack of a sprite. All three of the bugs coded with this label were fixed in rereleases of the relevant games, both clarifying that these are unintended and that the missing or incorrect elements are indeed the fundamental error.

Incorrect Sharing

This weakness involves unrelated entities mistakenly using behaviours attributed to one another, and was responsible for the most confusing bug present (the bullet lift). Although there are some similarities to extant weaknesses, most notably the access of a shared resource in race conditions, the absence of editing the shared resource but instead simply using the shared resource as given was enough of a distinction in my opinion. As a result, this is not a uniquely present type of weakness but rather a specification.

Figure 4.10: A SMW screenshot of Mario spin jumping in Ludwig's Castle

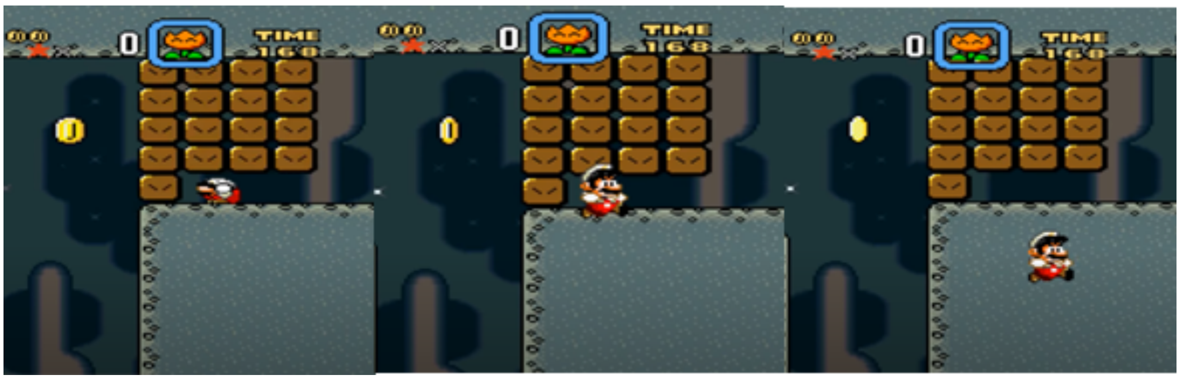


As mentioned before, the bullet lift glitch is a consequence of this weakness. While Mario still recognises the bullet as a bullet due to the correct sprite being present, and therefore bounces up and gains points, the bullet is incorrectly copying the behaviour from the lift slightly offscreen and conducts itself accordingly. Relatively complex behaviours for entities are not the only behaviours affected by this weakness however: the room displayed above has a peculiar instance of this bug. After waiting for the spiked wall to reach the ground, performing a spin jump will use the sound effect for the wall hitting the ground, instead of the usual spin jump sound effect. This variety in ways that this bug appears is part of why I decided to create this label.

Improper Correction

This bug label involves how games resolve illegal states. This is relatively unique as most programs are free to throw exceptions in unintended situations, but due to the importance of a fluid user experience games are encouraged to solve mild errors. This covers situations such as attempting to remove Mario from a wall but instead pushing him the other way, where the game's in-built error handling is abused to further manipulate the state of the game.

Figure 4.11: Three SMW screenshots of Mario being pushed through the floor

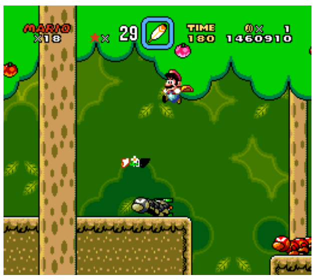


In the situation depicted in 4.11, Mario is crouching in a small space obstructed on 3 sides. When Mario attempts to stand up, the game immediately recognises that a standing Mario cannot fit in his current position, and attempts to push him to a nearby area that he can fit in. Due to the specific setup of the situation, however, this results in Mario passing through the small collision surface of the floor and having his position mistakenly corrected into inside the wall. This particular form of flawed correction appears in all 3 games, in slightly different ways, where one can pass into or through walls by manipulating the game's attempts to get Mario out. Due to this consistent appearence even in slightly different forms, I believe this deserved a unique label.

Incomplete Definition of Nonstandard Behaviour

For when normal behaviours are approached in an unusual way, and the game responds in strange ways. Vaguely titled, but most notable is "incomplete", in that it attempts to perform a behaviour in a standard way but is unable to do so for various reasons. This is typically due to a unique situational circumstance which wasn't considered in development and therefore did not have behaviour prepared for that situation.

Figure 4.12: A SMW screenshot showing a strange sprite where the combo score should be



4.12 is an example of behaviour not being correctly defined. The normal behaviour in question is the jump combo: Mario gains an increasing number of points for each consecutive enemy jumped on. The unusual and mishandled unique circumstance in this context is the wiggler enemy, which normally cannot be hit more than once as they become invincible to jumps after the first hit. However, after hitting them, the ability to hit the same wiggler can be refreshed by moving them far enough off screen. By doing this and hitting the same two wigglers repeatedly, strange sprites will display with unusual point values rewarded. This is rather complex, but mainly serves to illustrate the state of this label as describing corner cases where normal behaviours are not fully defined and therefore function unusually.

Chapter 5

Discussion

5.1 Goals

There were three main goals for this project: to accurately label the bugs found, create more accurate labels for those that couldn't be well defined with CWEs, and to note the points of the process in which potentially appropriate CWE definitions were found unsuitable. Of the 120 bugs labeled, 23% received one of 7 personally created labels from outside of the CWE. This demonstrates that, although the CWE still maintains good coverage of this software, there is clearly a blind spot here.

5.2 Significance

As mentioned, 23% of the bugs found I determined to not be adequately described by CWEs that currently exist. Considering the CWE has been in a state of constant development for years supporting by a large number of professionals, this level of missing coverage seems extreme.

5.2.1 Why don't relevant CWEs already exist?

There are two reasonable suggestions that immediately leap forth to answer this question: either these weaknesses simply aren't present at all in conventional software, or they are known but too niche to be "common" or too benign to be weaknesses and therefore not worth cataloguing. Based on my results, I believe that both answers have merit, and that the space lacking coverage is sufficiently diverse for both to be true. I mentioned previously in the results section that my labels fell into two somewhat distinct categories, and I believe these two suggestions fall upon similar dividing lines.

There is plenty of evidence for the first hypothesis. Most obviously among these are two labels I constructed: the "Entity Limit" weaknesses and the "Incorrect Game Elements" weaknesses. Both of these are intrinsically connected to the format of a game. While entity limits share some similarities with other resource-management weaknesses, the relatively unique inability to halt progress to wait for resources to be freed (perhaps leading to more traditional concerns such as deadlock) leads to situations where instead the software will simply run to the best of its ability while either waiting for the resources in question to be available or immediately removing them from current use. The latter weakness is also very obviously linked to games: while I did consider that any UI elements could theoretically be misplaced, misleading or absent, the degree of freedom a player has to interact with the environment presented to them is considerably higher than in nearly all other software. In addition, this weakness in particular is hard to describe and provide advice for solving, as it essentially boils down to simply setting up the user's environment "correctly" (or perhaps more accurately "in a fully consistent way") which is very context-dependent.

Alongside those two labels, there is another that is less obviously connected to the medium of games. That is the notion of an "Improper Correction". In a similar way to how entity limits manifest differently due to not being able to wait, these weaknesses run on a similar principle of maintaining as consistent and uninterrupted a user experience as possible. While traditional software, particularly if it carries important data or other significance, is likely to interpret any unintended behaviours as fail states and halt or crash, the same cannot be said of video games. A combination of their low importance and the aforementioned requisite smoothness enables this tolerance of illegal states. As a consequence, correcting behaviour beyond simply failing exists for these states that is unlikely to be present in other software, and can be exploited if implemented poorly, enabling the existence of this unique weakness.

The second suggestion approaching the question from the criteria of the CWE concerns the remaining labels. All of the remaining labels bear at least some resemblance to existing weaknesses, as noted in their descriptions. Most obviously was the expansion on improper initiation to specify new states; in the context of a game where this type of initialisation happens very frequently it makes sense to be more precise, however this is not necessarily widely applicable and deserving of the description "common". Similar comparisons, though perhaps not quite as direct can be made for each of the following; "Incorrect Sharing" and various limited-resource weaknesses, which while not analogous are perhaps similar enough that a niche variant need not be enumerated; "Failure to Appropriately Recheck Condition" is just an extension of any notion of flawed conditional checking, a special case specifically notable in games due to their continuous nature; and "Incomplete Definition of Nonstandard Behaviour" being a broad descriptor tailored to a common cause for mistakes in games.

Overall, I believe the relatively low coverage of the CWE on the subject matter covered is due to these weaknesses being either too alien to standard software to manifest, or considered too specific to be differentiated from more general descriptors. This raises a related question - if games are so different, should they have their own equivalent resource to the CWE? Personally, I am of the opinion that games are sufficiently relevant software and their bugs sufficiently generalisable to be treated as any other software. However, having just detailed the ways in which games are treated differently, and the justifications for not assessing their weaknesses equally, I also see the merit in a separate resource if these bugs are indeed considered too specific.

5.2.2 How did these games develop?

Throughout this paper I have been talking about the three investigated games more or less as a single monolithic unit, occasionally distinguishing them for the purpose of discussing their examples. This isn't a fair representation, however: these games released over the course of 5 years across two different pieces of hardware, and are a series of sequels and thus each was clearly built with the successes and failures of the previous iterations in mind. With all that said, I am going to discuss how I believe the bugs present in each iteration affected development of subsequent titles, and how patterns found in these bugs changed over time.

To begin with, we have to look at SMB1. SMB1's list of bugs is most notable for what it does not contain, as opposed to what it does. This is somewhat to be expected: it's the first iteration of the series, and one of the earliest games of its genre in general. As a consequence, it's considerably less complex than both later installments, with fewer complicated interactions to track and just generally fewer possibilities overall. The clearest demonstration of this is the absence of CWE-821 (Incorrect Synchronisation) despite the presence of CWE-820 (Missing Synchronisation). The presence of CWE-820 makes it clear that concurrent events were already very important from the start, as expected given how the game updates all events on the screen each timestep. Despite this, there appears to be no attempt to organise simultaneous events, and therefore no incorrect synchronisation borne from mistaken organisational procedures intended to correct synchronisation problems. Instead, all synchronisation problems are simply a matter of events not occurring and being forgotten, as they could not happen when they were supposed to and there was no handling for this situation. Besides this, SMB1 also has the most straightforward bugs that one might expect from an early installment. The only overflow and integer coercion bugs found throughout all three games were found in SMB1, and it also leads for most off by one errors and wrap around errors despite the lowest bug total. While the latter bugs still appear in later games, their reduced frequency even in considerably more complex games clearly shows a lesson learned from the errors in the original.

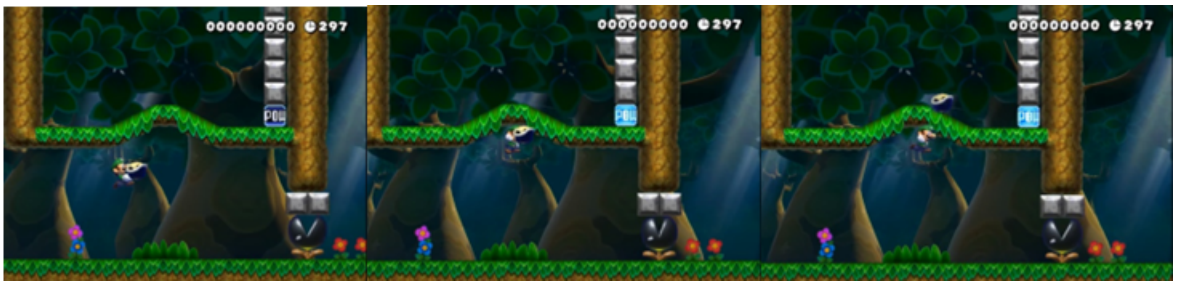
Next is SMB3. This game has the longest total list of bugs, partially justified by the greatest number of levels out of any of the three games. The step up in complexity is very apparent, as are the lessons learned from its predecessor. Easily defined mathematical bugs like mentioned before are almost completely eliminated, although a couple still remain, and to demonstrate the increased complexity have been replaced by the vast majority of the series' numeric errors that couldn't be otherwise categorised. These include clipping through walls, bosses repositioning themselves outside of the screen, and misplaced visual effects, among others. This game also introduced synchronisation handling, and as a consequence incorrect synchronisation for various simultaneous events such as collecting a powerup and taking a hit at the same time or mishandling simultaneous music triggers. It is clear here that this was a first attempt however, as this handling was far less error prone in the subsequent release. A similar story can be told about improper condition checking, where the increased complexity necessitated handling for unusual circumstances that hadn't previously been attempted. Finally, and less uniquely to SMB3, cleanup bugs became significantly more common, a pattern we will see continued.

Finally, SMW. The complexity difference between SMW and SMB3 is far lesser than the previous gap, so the focus here is instead on how this game improved upon what can be observed in the previous. As mentioned, the two key areas for this are in synchronisation and conditionals. SMW is the most consistent at keeping concurrent events synchronised without errors, having improved upon the initial synchronisation handling from SMB3 very successfully. Less successfully, however, was the continued prevalence of cleanup bugs: though the number remained constant, given that SMB3 had nearly 50% more bugs than SMW it's surprising to see a category largely unmoved. Despite this, it's clear that once again Nintendo was aware of the previous installment's shortcomings and prepared to tackle them.

Overall, I believe there is a clear trend of consciously motivated improvement throughout the three games. Each of the three has notable weaknesses that their successor improves upon, while new weaknesses are created due to the increasing complexity and freedom of options available in the games. More interestingly than the progress of the CWEs, however, is how little the non-CWE bugs fluctuated. I believe this demonstrates a relevance for these findings: even in the context they are created for, these bugs seem to have been under the radar or at least hard to address. With that in mind, I believe codifying these bugs may prove beneficial for relevant fields.

Moving beyond the three games in question, how have game bugs changed in the time between then and now? In 2015 and 2019, Nintendo released "Super Mario Maker" (SMM) and "Super Mario Maker 2" (SMM2), games designed to allow players to create levels based on various previous Mario games, including those investigated here. Here, I found two bugs quite reminiscent of those covered in this project. One, pictured in 5.1, is buggy slope behaviour that directly mirrors the ability to throw shells through certain angles of slope in SMB3 and SMW.

Figure 5.1: Three screenshots from SMM2, showing a shell clipping through a slope



The second bug is slightly more involved, showing the progress that has been made but also a clear demonstrating of these bugs having meaningful effects on the running of the software. Creative stage developers use the game's behaviour concerning entity limits - a weakness label I created for similar bugs - to create behaviours that the player doesn't expect that depend on the player removing or creating entities. This shows that although some bugs are quickly stamped out between iterations, even over 35 years later the traditional 2D Super Mario game style suffers from bugs that have been present since the beginning.

Chapter 6

Conclusion

This project served as an investigation into the applicability of the CWE to the software area of video games, asking questions about the extent of the CWE's universal applicability and the potential for considerable improvement in our understanding of the patterns in and types of bugs present in these games. At this point, I have successfully created a codebook that, after several revision passes, accurately describes the bugs present in these games. Using this, I have answered these questions to a satisfactory degree.

As briefly mentioned in the discussion, I had 3 goals for this project:

- Identify the weaknesses responsible for bugs
- Show that the CWE does not sufficiently represent this area of software vulnerabilities
- Personally produce meaningful labels for weaknesses that could not be adequately described by the CWE

The first goal is satisfied by my final, second-coder verified codebook. Although I cannot guarantee the exact form of the software weakness present due to the limitations of my method, I can say with confidence that the codes used accurately represent the types of erroneous behaviour causing these bugs. The second is also easily answered by my final code: with over 20% of the final codebook being inhabited by non-CWE labels I believe it is very clear that there is a notable lack of coverage for defining software vulnerabilities in these games. Finally, the fact that non-CWE labelled weaknesses remained a reasonable consistent presence between the three games, and at least one of the labels I produced can be identified at a glance in a game from 2019, I believe the labels I have constructed are indeed sufficiently meaningful and descriptive for the weaknesses that they target.

Further work can be taken in a few different directions. One clear source of uncertainty in this project is the qualitative method used, so one potential avenue forward is to attempt to replicate and verify these results by acquiring and inspecting the source code. This could lead to more concrete definitions and examples of the constructed labels used, and would contain less conjecture due to having access to ground truth.

Another direction is a more comprehensive breakdown of the bugs themselves. I found many of the behaviours described to be quite complex due to the nature of many interacting parts in any given state of a video game. Given this, some bugs could easily be described with multiple codes. A larger scope project could perhaps focus on this aspect, encouraging the use of double coding and investigating the interactions between various CWEs as opposed to attempting to strictly categorise each instance of erroneous behaviour with one label.

Bibliography

[1] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16:487–513, 2011.

[2] Abdullah M Algarni and Yashwant K Malaiya. Software vulnerability markets: Discoverers and buyers. *International Journal of Computer and Information Engineering*, 8(3):480–490, 2014.

[3] Richard Amankwah, Patrick Kwaku Kudjo, and Samuel Yeboah Antwi. Evaluation of software vulnerability detection methods and tools: a review. *International Journal of Computer Applications*, 169(8):22–27, 2017.

[4] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 427–430. IEEE, 2011.

[5] Irena Bojanova, Paul E. Black, Yaacov Yesha, and Yan Wu. The bugs framework (bf): A structured approach to express bugs. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 175–182, 2016.

[6] Irena Bojanova and Carlos Eduardo Galhardo. Classifying memory bugs using bugs framework approach. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1157–1164, 2021.

[7] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. Trend analysis of the cve for software vulnerability management. In *2011 IEEE third international conference on privacy, security, risk and trust and 2011 IEEE third international conference on social computing*, pages 1290–1293. IEEE, 2011.

[8] Juliet M Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, 13(1):3–21, 1990.

[9] The MITRE Corporation. Common weakness enumeration. https://cwe.mitre.org/index.html.

[10] Barney G Glaser and Anselm L Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Transaction Publishers, 2009.

[11] Katerina Goseva-Popstojanova, Jacob P Tyo, and Brian Sizemore. Security vulnerability profiles of nasa mission software: Empirical analysis of security related bug reports. Technical report, 2017.

[12] Suresh S Malladi and Hemang C Subramanian. Bug bounty programs for cybersecurity: Practices, issues, and recommendations. *IEEE Software*, 37(1):31–39, 2019.

[13] Super Mario Wiki. Mario glitches. https://www.mariowiki.com/Category:Glitches.

[14] Wenjun Xiong, Melek Gülsever, Koray Mustafa Kaya, and Robert Lagerström. A study of security vulnerabilities and software weaknesses in vehicles. In *Secure IT Systems: 24th Nordic Conference, NordSec 2019, Aalborg, Denmark, November 18–20, 2019, Proceedings 24*, pages 204–218. Springer, 2019.