Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. None of this work has been published before submission

Signed:..... Date:.....

UNIVERSITY OF SOUTHAMPTON

A Systematic Model for Iterative Deduction of Reverse Engineering

DOI: 10.5258/SOTON/PG/T264

by

Max D. Hayman

ORCiD: 0000-0003-4825-810X

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Physical Sciences and Engineering

School of Electronics and Computer Science

April 2026

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Max D. Hayman

Reverse engineering constitutes a fundamental pillar of cybersecurity, digital preservation, and compatibility engineering, with historical precedents dating to World War II cryptanalysis efforts that precipitated the development of early computational machines including Alan Turing's Bombe. Despite its critical role in malware analysis, vulnerability research, legacy system maintenance, protocol reconstruction, and cultural preservation, reverse engineering has received limited systematic academic treatment. The field is frequently characterised as an intuitive craft requiring "wizardry" rather than a rigorous, teachable discipline , resulting in a significant methodological gap wherein practitioners rely predominantly on tacit knowledge, ad-hoc techniques, tool-specific workflows, and individual expertise rather than generalisable theoritical frameworks. This informal and unregulated practice leads to variability in approaches, inconsistent results, and barriers to knowledge transfer between practitioners.

This thesis addresses this gap by proposing a systematic, domain-independent model for reverse engineering that formalises the iterative process of knowledge acquisition from black-box systems though structured observation of component-environment interactions. We present a formal framework that abstracts reverse engineering into an interactive cycle comprising six core phases: (0) initial model construction based on available knowledge, (1) model instantiation to create observable instances, (2) testing and trace generation through controlled interaction, (3) knowledge synthesis combining new observations with existing understanding, (4) model revision reflecting refined knowledge, and (5) iteration until knowledge reaches the desired completeness threshold. This methodology structures reverse engineering as the systematic observation and analysis of behavioural traces across controlled environmental variations, transcending domain-specific tools and techniques to provide a unified epistemological foundation applicable across diverse reverse engineering contexts.

Contents

List of Figures

List of Tables

Acknowledgements

I am especially grateful to my supervisor, Julian Rathke, for his constant support and guidance throughout this project. His kindness and patience during the global pandemic and through some of the most personally challenging times gave me the space and confidence to keep going. I could not have asked for a more thoughtful or understanding mentor. I am also grateful to Vladimiro Sassone for his support as my second supervisor and as leader of the research group. His help navigating the administrative complexities around the pandemic was deeply appreciated and made a real difference during a difficult time.

I want to thank Evie for her endless patience and support throughout this long process. And to my cat, Boris, whose commitment to sitting on my keyboard at the worst possible times was unmatched, your contributions will not be forgotten.

A heartfelt thank you to my aunt and uncle, Pamela and Luke Hayman, and my grandparents, John and Jane Hayman, for their unwavering encouragement throughout this journey. Their persistent reminders played a bigger role than they probably realise in pushing me to finish this thesis. Their belief in me, even when I doubted myself, has meant the world.

I'd like to thank my best friend, Rachael, for helping me escape on adventures and showing me how to find joy again. Your support has meant a lot during this time.

I would also like to thank everyone at Aspin, Freejam and Kinetic for their friendship, encouragement, and good humour along the way. I'd also like to thank the entire Return of Reckoning team, whom I am proud to call friends. Their support, both within and beyond the game, has been invaluable to me throughout this journey.

A special mention to Liz & Ian Lowry, Madeline & Michael Reilly, Ellen, Tim, Jon, Jude, Dave, Ash, Josh, Chira, Steve, Tom, and Amy, whose kindness and support have made this process easier and more enjoyable.

Finally, I'd like to thank my brother, Dean, his partner, Michelle, and their children, my niece, Charlie, and my nephew, Jamie. I love them all very much. They've been in my thoughts often, and I'm grateful for their presence in my life.

To my mother, Marion, whose love and belief in me never faltered, and whose support helped me face life's hardest moments. She always believed in me, no matter the circumstances, and it was her support that kept me moving forward. This thesis is dedicated to her memory and everything she did for me.

Chapter 1

Introduction

Reverse engineering has existed in some form since before the existence of computers. A famous example of reverse engineering was during World War II, when Enigma machines were employed extensively by Nazi Germany to protect commercial, diplomatic, and military communications. The Enigma machine was considered so secure that it was used to encipher the most top-secret messages [13]. The Polish Cipher Bureau was the first to break the military variant of the Enigma in 1932. In 1938 they built an electro-mechanical machine capable of cracking early encrypted messages called the Bomba, which exploited the fact that the same message indicator was sent twice at the start of each message, a major flaw in the German cryptographic procedures. When the Germans had discovered the weakness, they gave up the double encipherment of the message indicator on most radio networks on 1 May 1940, rendering the Bomba useless. In 1939, British mathematician Alan Turing developed a machine that was capable of recovering the key settings, even after the Germans stopped the double encryption of the message key at the beginning of each communication. It used a completely different approach and was based on the assumption that a known (or guessed) plaintext, a so-called crib, was present at a certain position in the message [14]. Historians estimate that Alan Turing's contributions may have helped shorten the war by at least two years and saved millions of lives. In fact, Winston Churchill was reported to have told King George VI that it was because of these efforts that they won the war [15] [16].

Formally, reverse engineering is defined as "the process of analysing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction" in which the "subject system" is the end product of software development [17]. It allows us to visualise the software's structure, its ways of operation, and the features that drive its behaviour. Today, reverse engineering serves a variety of purposes, from technical innovation to security to preservation.

Reverse engineering is used extensively in cyber security related fields. For example, computer viruses are reverse engineered to investigate how they work and how they operate. It is also used to build detection models for antivirus programs [18]. Reverse engineering is also used by black-hat malicious actors to discover vulnerabilities in systems to bypass security measures, or by white-hat researchers to report these vulnerabilities to fix them and make those systems more secure [19].

Reverse engineering is also used heavily in digital preservation, to restore or archive software, hardware, and media for historical or cultural preservation. For example, extracting data from outdated storage media or emulating retro games [20]. One example of this is the Open 'Tendo project which is an Open Source HardWare (OSHW) recreation of the original 1985 Front-Loading Nintendo Entertainment System (NES) motherboard, which allows people to take the schematics and create their own NES replacement motherboard [21].

Below we list some further examples of reverse engineering.

The Electronic Delay Storage Automatic Calculator (EDSAC) was an early British computer, built at the University of Cambridge in the 1940s [22]. It is generally accepted that the EDSAC was the first practical general purpose stored program electronic computer. Other, earlier machines were either dedicated to a single task (e.g. Colossus and code breaking) or were purely experimental (e.g. the Manchester University "Baby" Small-Scale Experimental Machine) [23]. The Computer Conservation Society announced in 2011 that it planned to build a working replica of EDSAC at the National Museum of Computing (TNMoC) in Bletchley Park and aimed to have it running fully by the end of 2019. It had not been known what happened to the EDSAC parts after they were decommissioned and dismantled in the 1950s, until 2015 when a part, called the Chassis 1A, was donated to the project by someone who had obtained it from a person who had bought several Edsac pieces at auction intending to turn them into bookshelves. However, it is not known who bought all the parts. [24]. The project has required complex reverse engineering to come up with both logical and physical designs to reflect EDSAC as it was in 1951.

Microsoft Word was released in 1983 and was one of the first word processing applications. Unlike most MS-DOS programs at the time, it was designed to be used with a mouse [25]. Word, along with its sister applications Excel and PowerPoint used their own proprietary formats up until 2007. Word used its 'Word Document Format' (.doc) file format, Excel had the Excel Binary File Format (.XLS), and PowerPoint used the .ppt format. The formats evolved through later versions of the product, each binary format different in each version, as functionality was added [1].

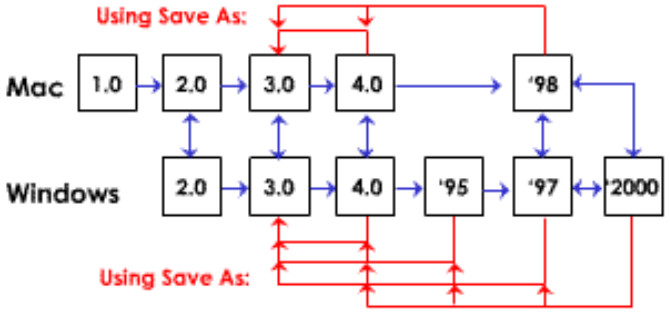


FIGURE 1.1: Showing compatibility of the .xls file format between different versions of Microsoft Excel [1]

The formats were not designed with interoperability in mind. The assumption, and a fairly reasonable one at the time, was that the Word file format only needed to be read and written by Word [26]. The documentation for the formats was included in the Microsoft Excel Developers Handbook in 1997 [27], MSDN CD's, MSDN Web site and Visual C++ [28]. However, none of which were kept up to date, and by 1999 the format documentation was no longer available for download. The specifications of later versions of the Microsoft Office binary file formats were not publicly available. The DOC format specification was available from Microsoft on request[29] under restrictive RAND-Z terms from 2006[30] until February 2008. StarWriter (later known as StarOffice) written by Marco Börries in 1985 was an alternative to Microsoft's Office suite [31]. The company was acquired in 1999 by Sun Microsystems for $73.5 million [32], as "it was cheaper to go buy a company that could make a Solaris and Linux desktop productivity suite than it was to buy forty-two thousand licenses from Microsoft." [33]. Sun Microsystems released the source code the following year as a free and open-source office suite called OpenOffice.org. Microsoft's proprietary formats were reverse engineered by Sun Microsystems and OpenOffice [34]. Microsoft has now switched to 'Office Open XML' formats for all of their office suite of products.

Many companies use reverse engineering to ensure that new systems or products are compatible with existing ones, such as developing drivers [35] or software that seamlessly integrates with proprietary hardware. Additionally, companies use it to reduce expenses associated with replacing licenced technologies, such as recreating parts or software for devices when the original supplier no longer provides support.

Reverse engineering itself is often viewed as an informal and unregulated practice. Although there are legitimate uses in areas like security research, debugging, and software optimisation, the reverse engineering community lacks a clear and universally accepted framework. This informality leads to variability in approaches and tools, making it difficult to guarantee consistent results. Furthermore, legal and ethical boundaries are often unclear or loosely defined, especially when reverse engineering software, hardware, or protocols for systems that are copyrighted or patented. This can lead to complications,

particularly when researchers accidentally or deliberately cross into grey areas of the law, exposing themselves to potential litigation or corporate backlash.

Another key issue is the cultural stigma associated with reverse engineering, particularly in contexts such as hacking. Reverse engineering is often linked to malicious activities such as bypassing Digital Rights Management (DRM) protections [36], cracking software, or cheating in video games [37]. As a result, many in the reverse engineering community are stereotyped as "hackers", a label that brings with it a reputation for being clandestine or subversive. This perception is exacerbated by the fact that many reverse engineers are independent or operate in informal settings, which can lead to the perception of being "hot-headed" or reckless. The use of complex, low-level techniques and the pursuit of hidden knowledge can create a combative or adversarial mindset, which might hinder collaboration with developers or manufacturers whose system is being reverse-engineered.

Manufacturers often have efforts to encourage hackers and security researchers to responsibly disclose vulnerabilities, often through bug bounty programs [38]. These initiatives offer financial rewards in exchange for detailed reports of security flaws and are framed as a collaborative way to improve product security. However, these programs are often rooted in self-interest, aiming primarily at preventing vulnerabilities from falling into the hands of malicious actors who could exploit them for profit or disruption. This allows manufacturers to ensure that patches can be developed and deployed before any public exposure occurs.

These challenges often lead to a lack of trust between reverse engineers and the broader tech industry, especially when reverse engineering is perceived as a tool for exploitation rather than security or performance enhancement. Without formal training or industry oversight, reverse engineers can unintentionally damage systems, misuse information, or breach ethical boundaries. Consequently, the practice can be viewed as inherently disruptive rather than constructive, despite its potential to uncover vulnerabilities and contribute to better cyber security practices.

1.1 Motivation

This section reviews the motivations behind reverse engineering in two key areas: NFC technology and video games. In both fields, reverse engineering is driven by security concerns and the desire to preserve and restore experiences, whether by uncovering vulnerabilities in NFC implementations or by ensuring the longevity and accessibility of video games in the face of evolving technology and server shutdowns.

1.2 Motivation for Reverse Engineering NFC

Almost every large commercial workplace, educational institute, bank or government office uses wireless cards perform physical access control into their buildings. However, it is apparent that these electronic locks are not as secure as they appear to be. The ability to break into one of these places with little effort would be a major security breach. We will investigate issues in these systems and their implementations.

Near Field Communication (NFC) is now widely deployed technology with a variety of different implementation scenarios such as physical access systems, transport ticketing systems, toys, credit / debit card payments, and more. Many implementations use cryptographic techniques to secure the communications and the data on the card. NFC evolved from radio frequency identification (RFID) and chip-based smart card technology. A passive NFC chip inside a card or a fob is activated when in range of another active NFC chip, such as a reader, and then data transferred between them [39]. Operating at small distances, NFC can be a power-efficient method of wireless communication compared to other types of wireless communication.

There are a number of security principles that NFC can bring to applications. In many implementations, it offers a way to authenticate or identify an individual or an object. This is often used in access control or ticketing systems, but can also be used to implement hardware-based digital rights management in applications, such as video games, to ensure consumers have purchased legitimate products [40]. Additionally, systems such as Transport for London's Oyster card expect a level of integrity in the card, as it is required to trust the data which represents the balance on the card. In access control systems, it provides accountability and auditing of a building or campus, which a traditional lock and key cannot effectively do.

The use of NFC technology is still on the rise [41] and is now being deployed in smartphones and used in applications such as ApplePay and AndroidPay. It is also used in toys [42] [40] such as Disney Infinity or Skylanders, which are paired with a gaming console, and IoT applications such as the Yale Smart Lock. By the end of 2018, almost all major capital cities were using NFC technology for public transportation. NFC technology in smart phones is even replacing traditional NFC cards in applications such as public transport networks. NFC technology is also being used in new applications which would not have historically been considered [43] [44] [45].

Although NFC can provide security to a system, it is usually achieved by using one of many extensions to the NFC communication protocol that have been developed, such as the MIFARE range by NXP Semiconductors, FeliCa by Sony [46], Topaz by Innovision and more. Each type of card usually implements part of the ISO/IEC 14443 NFC specification prior to their own proprietary protocol. Most protocols offer a form of cryptographic communication, such as MIFARE Classic with Crypto1 and MIFARE

DESFire [47] and FeliCa [46] with AES and DES/Triple-DES. Unfortunately, some of these, such as Crypto1, have been analysed and proven to be cryptographically insecure, and more modern implementations have decided to go with more well-known cryptographic algorithms. Whether an application uses these cryptographic features or if they have been implemented correctly is usually dependent on the manufacturer of the product. For example, the Yale Smart Lock ships with MIFARE Classic cards, however, uses none of the cryptographic features and authenticates using essentially plain text.

So far, research in NFC technology security has largely focused on proving or disproving the correctness of cryptographic algorithms used in cards [48] [3] [49] [50], such as Crypto1 in MIFARE Classic, which was proven to be insecure and has had many published attacks to defeat security. Other research also focusses on side channel attacks [5] on cards with more proven secure cryptographic attacks such as this side channel attack by monitoring power consumption. However, security can be severely affected by the manufacturers' misuse of the underlying protocols in the implemented systems. For example, the Yale Smart Lock uses MIFARE Classic cards to authenticate users, however, exits the protocol before using cryptographic functions.

There is little academic research which analyses the security of implementations of applications irrespective of whether its underlying protocol is secure. In [3] we see an analysis and partial documentation of the structure and functionality of the Oyster Card system used by Transport for London [3]. Until December 2009, when Transport for London started rolling out MIFARE DESFire cards, the Oyster system was using exclusively MIFARE Classic cards. Due to the vulnerabilities in the MIFARE Classic card, hackers were able to manipulate data on the cards, including the balance on the card. The process in [3] involved creating images of the data on the card before and after interactions with the readers, comparing the changes to the data on the card, and inferring the meaning of the fields. This process is time-consuming, and a lot of the work is manual and is not practical in large applications. The reading fails to identify a serialisation algorithm for the data and only provides the values along with the real world observables. Additionally, this process only deduces actions that result in modifications of the card data and does not reveal any read actions. This leaves us with only a partial understanding of the underlying algorithm used, and therefore it is difficult to assess any security implications of the implementation.

As another example, an unknown hacker released an Android application that allowed users to top up their Tarjeta Bip! NFC cards used in public transportation in Chile [1]. The application has since been reverse engineered and documented [2]. The application highlights a flaw in the implementation of the Tarjeta Bip! which was that every card deployed used the same secret key for accessing the sector which stored the card balance, and therefore once someone discovered the key, they could change the balance on any

[1] https://securelist.com/android-nfc-hack-allow-users-to-have-free-rides-in-public-transportation/67283/

card. The correct implementation would have used a different key for every deployed card, ideally one which is totally random and not dependent on the card identifiers. This should have easily been identified as a flaw when the system was being designed. In [51] there is a proposal of a high level attack on an Automated Fare Collection system by essentially using mobile phones which connect to share a collection of NFC tickets to reduce costs; this paper attacks NFC cards by cloning them and distributing virtual copies between several devices. This can be used to destroy the integrity of the system. Although it describes interactions with readers in a high-level language, it focuses largely on their smart card sharing infrastructure. It mostly depends on the ability to clone NFC cards, which means that it will depend on the underlying protocol used in the targeted system. The design of the application would not expect the cards to be able to be cloned and replicated between multiple devices.

The example of cases such as the Tarjeta Bip! highlights a clear need for tools to assist in the development and implementation of NFC applications. The ideal approach for new or small existing systems should be done using a secure by design approach. Secure by design would allow the whole system to be designed from the ground up with security in mind. A model could be made representing the application logic and a sequence of calls to the high-level protocol. We would then be able to do model checking to ensure that the designed system meets all the required security parameters. The model could then be run through a code generation process to produce deployable code.

However, many legacy NFC systems have already been deployed, so it may be necessary to assess security in a post hoc manner. Traditionally, black-box penetration testing would be the method used here, as with very large existing applications, there may also be an intolerably high cost associated in deploying a whole new application. An alternative approach would be to reverse engineer the legacy system. This would involve collecting low-level traces of interactions between cards and readers. This would take the form of raw opcodes and data that would be passed between them. The low-level information would then need to be identified as a series of higher-level protocol commands. Using reverse engineering techniques, we should be able to identify the actions of the high-level protocol from the corresponding actions of the low-level protocol. We would then be able to map these high-level protocol actions to the actions used in a model. Which would allow us to rebuild the model of the system and then continue to model checking on it. Using a model allows us to meet in the middle and encompass both new and existing implementations. Therefore, in this thesis, it is proposed that we build a model.

1.2.1 Contributions

The main contributions of the work in Chapter 3 can be summarised in the following.

1. Development of a new tool for capturing and scripting NFC interactions, allowing replay and analysis of reader logic using custom specifications.
2. Reverse engineering of real-world NFC systems including the Yale Keyless Smart Lock and University of Southampton ID card system.
3. Discovery of security weaknesses through analysis of UID-derived access controls and static data patterns on MIFARE Classic cards.
4. Demonstration of how NFC-based access systems rely on poorly protected or easily replicable data, with implications for physical security.

1.3 Motivation for Reverse Engineering Video Games

Over 50 years since the release of the first computer game, the industry has developed to be larger than Hollywood and the music industry combined [52]. Since the early days of the internet, online video games have provided countless hours of entertainment for players around the world. As we age into an ever-connected world, these games have increasingly utilized internet capabilities to enhance the player experience and deliver ongoing content to players [53]. Using 'games as a service' model, publishers push regular content updates for their games to keep players engaged in their product. However, it has already become apparent that preserving these experiences for future generations is almost impossible. Now video game publishers are often removing or replacing content and upgrading graphics with these content updates, thereby preventing players from experiencing previous iterations of the game.

Furthermore, once online video games are shut down by their publishers there is very little people can do to be able to play these games again [54] [55]. Ubisoft announced that after almost a decade of support, they would be decommissioning 'The Crew' on March 31, 2024 "due to server infrastructure and licencing constraints". The game was then subsequently removed from customer's digital game libraries, preventing them from downloading or installing it again [56]. In 2024 a petition was created asking the UK government to "require video game publishers to keep games they have sold in a working state", which reached more than 27,000 signatures before being closed early due to the general election [57].

Some video game hardware was not originally designed to last a long time. The Game Boy, Game Boy Colour, and Game Boy Advance were a line of handheld video game consoles from Nintendo. Games for these consoles were distributed on cartridges, called 'Game Boy Game Paks', which were plugged into the console. The cartridges contained read-only memory and other hardware that stored the game and often stored a player's save progress, volatile RAM, and some even had real-time clocks. Due to the availability of re-writeable persistent storage such as flash, the cartridges included a small battery,

which would power the RAM, and the real-time clock. However, the life expectancy of these batteries was between ten and fifteen years, or even as little as five when using the real-time clock. Due to this limitation cartridges would eventually lose all save data and lose the ability to save game progress any more. Reverse engineers have developed ways to replace the soldered battery with replaceable battery holders and developed ways to perform these modifications without losing data. The interface for the cartridges has also been reverse engineered, which has led to the creation of devices such as the GBxCart. This allows people to connect a cartridge to a standard computer and allows the saved data to be read and backed up. Furthermore, the Game Boy consoles themselves have been reverse engineered [58], allowing for the creation of emulators which allow games to be played on other devices such as desktop computers, which can save game progress in non-volatile storage. There have also been reverse engineering efforts into video games for the console itself. For example, games such as Pokémon Red and Blue have been decompiled into an arm assembly [59] (their original programming language), which means the games can be natively ported to other platforms or bugs fixed.

Since the European Union's Restriction of Hazardous Substances (RoHS) directive in 2006 [60], most modern electronics have switched to lead-free solder to reduce the environmental and health risks associated with lead. Consoles released after or in preparation for this directive, such as the Sony PlayStation 3 or Microsoft Xbox 360, used lead-free solder. However, lead-free solder has a higher melting point, which can cause greater thermal stress on components during manufacturing. The Xbox 360 failure rate was estimated to be as high as 60% in 2009 [61] [62]. Due to these manufacturing defects, the lifespan of these consoles is significantly limited compared to those of other generations, making it harder to play games created for these consoles once the hardware fails.

Some console manufactures have incorporated backwards compatibility into their consoles. The Sony PlayStation 2 console also contained hardware from the original PlayStation console and was therefore able to play games from the previous generation. This trend continued and the PlayStation 3 contained the Emotion Engine (CPU) and the Graphics Synthesizer (GPU) from the PlayStation 2 which meant it was also able to play PlayStation 2 games and, using software emulation, original PlayStation games. However, the second revision of the PlayStation 3 removed the PlayStation 2 CPU and replaced it with software emulation of the chip. The third revision then removed the GPU and PlayStation 2 compatibility was removed from the system. The PlayStation 3 CPU was the Cell Broadband Engine™ (Cell/B.E.), developed jointly by Sony, Toshiba, and IBM [63]. The Cell/B.E. processor includes one PowerPC® processor element (PPE) and eight synergistic processor elements (SPEs). The CBEA is designed to be well suited for a wide variety of programming models, and it allows for the partitioning of work between the PPE and the eight SPEs. In particular workloads even a single PS3 can significantly accelerate some computations. Marc Stevens, Arjen K. Lenstra, and Benne de Weger demonstrated an MD5 brute-force attack in a few hours and that "a

single PlayStation 3 performs like a cluster of 30 PCs at the price of only one" [64]. The Air Force Research Laboratory connected together 1,760 consoles with 168 GPUs and 84 coordinating servers in a parallel array capable of 500 trillion floating-point operations per second (500 TFLOPS) called the 'Condor Cluster' [65]. At the time Condor Cluster was the 33rd largest supercomputer in the world and was used to analyse high definition satellite imagery at a cost of only one tenth that of a traditional supercomputer [66]. Due to its architecture the PlayStation 3 console is hard to emulate correctly. Sustained single core performance on an SPU is still higher than almost all other alternatives in the market. Aside from writing code for the SPU and PPE, one must also make sure that parallelism is enabled by default. Thus, emulating them on any modern CPU would be near impossible. In addition, there are a variety of challenges such as loading from memory with cache misses, dealing with branching, not having enough registers, and the complexity of the Direct Memory Access (DMA) [67] [68]. The PlayStation 4 and PlayStation 5 consoles do not contain PlayStation 3 hardware and do not offer support for PlayStation 3 games via emulation. However, Sony have started to offer PlayStation 3 games via their PlayStation Now gaming platform which involves remotely streaming games from a customised PlayStation 3 in a data centre. This, however, means users cannot play games from their own discs and access to this service requires a subscription and can be discontinued at any time. In 2011 DH and Hykem started working on an emulator for the PlayStation 3 called RPCS3. In 2017 it had its first implementation of a PPE thread scheduler, enhancing its emulation of the many-core Cell microprocessor [69]. As of August 2024, nearly 70% of PlayStation 3 games have been classified as 'playable', meaning that a significant portion of the native library can be played from start to finish without any major problems [70]. In 2024 Mark Cerny, who was lead system architect for the PlayStation 4, filed a patent titled 'Backward compatibility testing of software in a mode that attempts to induce skew' [71]. It states that if a new device has higher or even lower performance than a legacy device, backwards compatibility issues will occur and it attempts to overcome those issues. This indicates that there is a strong commercial motivation for reverse engineering, as well as one from a preservation standpoint.

In 1998 Sony released SecuROM, a CD/DVD copy protection and digital rights management (DRM) system [72]. It aimed to prevent unauthorised copying and reverse engineering of software, primarily commercial computer games running on Windows. SecuROM would require the original game disc to be placed in the computer's CD-ROM drive in order to allow the game to start, as well as requiring an initial online activation. In 2008, a class action lawsuit was filed against Electronic Arts for its use of SecuROM in the video game Spore [73]. Legitimate customers who purchased the game first-hand or second-hand were plagued with DRM-related issues. As a consequence, Spore became the most pirated game ever in 2008 [74]. It was common practice that video game developers would have included programming that degrades the gameplay experience when copy protection tampering was detected [75] [76]. Another such game

which used SecuROM was Rockstar's Manhunt, which also deliberately engages many glitches that make the game unwinnable if its SecuROM DRM is missing. However, shortly after its release the copy protection was cracked, and these additional glitches were bypassed by hacking group Razor 1911. Razor's crack was so effective that Rockstar used it for Manhunt's 2008 Steam digital launch without anyone realising it. However, the company's use of unofficial anti-DRM tools in the Steam edition of Max Payne 2 came to light in 2010, after which it released patches to hide the evidence in both titles. Unfortunately, Rockstar's attempt to cover its tracks broke measures it had implemented to make the Razor crack compatible with recent versions of Windows, activating the anti-piracy bugs. As a result, the Steam release of Manhunt has been unplayable without third-party modifications for more than 13 years [77].

Through this research, we will investigate the legal, technical and ethical issues along with the limitations associated with preserving these games. We will explore and overcome some of the technical issues which would allow us to start to preserve these games. Finally, we will create tools we can use to preserve these games and assess their impact.

1.3.1 Contributions

The main contributions of the work in Chapter 4 can be summarised below:

1. Define the core components of online video games and explored the methods and techniques used to analyse and categorise them.
2. Assess the impact of ever-connected online video games, focusing on their influence on gameplay, player behaviour, and game design.
3. Develop novel methods for reverse engineering online video games, enabling the analysis of game systems, protocols, and network structures.
4. Propose techniques for reverse engineering various file formats, such as archive files, enhancing the ability to extract, modify, and analyse game assets.
5. Develop methods for reverse engineering anti-cheat systems, identifying vulnerabilities, and detecting cheats in real-time.
6. Investigate tools and strategies to preserve the integrity of online games, ensuring their long-term accessibility and functionality.

1.4 Defining a Method for Reverse Engineering

Reverse engineering remains a critical yet under-formalised discipline within computer science. Unlike software engineering, which has matured through the adoption of systematic methodologies across diverse programming paradigms, reverse engineering is often

conducted in an ad hoc manner, guided more by a practitioner's intuition than by a structured process. This thesis proposes a generalisable, systematic approach to reverse engineering that formalises the core principles behind the analysis and reconstruction of unknown systems.

At its core, reverse engineering involves the observation and interpretation of the interactions between a target component and its surrounding environment. The objective is to infer a model of the unknown component such that it can be accurately reproduced, simulated, or substituted. We define a reverse engineering process as successful when the artefact produced exhibits sufficiently indistinguishable behaviour from the original.

This framework is applied across several domains where reverse engineering is commonly practised, including file formats, software executables, network protocols, runtime environments, and hardware devices. For example, in the analysis of NFC systems, we alternate between emulating a reader and a card to observe bidirectional communication. In the context of video games, we examine the interactions between executables and proprietary file formats to reconstruct the format sufficiently to produce compatible assets. Across all these examples, the methodology is focused on systematic trace collection, abstraction of observed interactions into models, and iterative refinement of those modules until behavioural equivalence is achieved.

The stages of the proposed systematic approach are as follows:

0. **Initial Model Construction:** Define initial models of the component and its environment based on the initially available knowledge.
1. **Model Instantiation:** Instantiate an instance to observe the behaviour of the system through interaction.
2. **Testing and Trace Generation:** Test the instance against actual components or environments and extract new knowledge.
3. **Knowledge Synthesis:** Combine new observations with existing knowledge to form a refined understanding.
4. **Model Revision:** Update the models to reflect new knowledge.
5. **Iterate Until Complete:** Repeat steps 1–4 until the accumulated knowledge meets the desired threshold.

We have selected two primary domains in which to demonstrate this method, NFC devices and video games. NFC technology, dating back to the 1980s, is now widely adopted and billions of devices are being used. It represents a hardware-focused challenge, ideal for demonstrating reverse engineering at the protocol and communication level. In contrast, video games present a predominantly software-focused challenge driven by both consumer interest and commercial significance. Our chosen case study is *Warhammer Online - Age of Reckoning*, a *Massively Multiplayer Online Role-Playing Game (MMORPG)* set in the *Warhammer Fantasy* universe. Despite a lifespan of just over five years, the game sold over a million copies and reached 800,000 subscribers at its peak. With the original servers no longer operational, this offers a textbook example of a black-box scenario. Reverse engineering efforts have since led to the development of *Return of Reckoning*, an emulated private server that not only has revived the game for a dedicated community, but has also earned praise from the original developers for its technical achievement and its role in expanding the game in line with its original vision [78] [79] [80] [81].

By defining a generic reverse engineering workflow and associated artefacts, such as component to reverse, its environment, system boundaries, and observed traces, we aim to transition reverse engineering from a craft to an engineering discipline supported by reproducible methods.

1.4.1 Contributions

The main contributions of this work can be summarised in the following.

1. Introduction of a consistent terminology and convention for describing reverse engineering processes, aiding clarity and reproducibility.
2. Proposal of a systematic model for reverse engineering, incorporating component and environment models that evolve through iterative knowledge refinement.
3. Development of a method to observe and model unknown artefacts in reverse engineering tasks, facilitating the creation of drop-in replacements for original components.
4. Demonstration of the practical applicability of the model by applying it to real-world reverse engineering tasks, including NFC devices and video games.
5. Formalisation of a multistep reverse engineering process for knowledge acquisition, offering a repeatable and structured framework for reverse engineering projects.

1.5 Definitions

Phrase	Definition
Component	A distinct part or unit of a system, typically representing a software module or hardware element, that is the focus of reverse engineering efforts.
Environment	The context or external conditions in which the component operates, which can influence its behaviour. The environment can include hardware, software, or even simulated systems.
Interaction	The exchange of information, data, or commands between components of a system, or between a system and its environment. These interactions are often observed to understand how a system functions.
Knowledge	The information, insights, and understanding derived from observations, traces, and models. Knowledge serves as the foundation for hypotheses and informs decisions throughout the reverse engineering process.
Observation	The act of monitoring or capturing the interactions, states, or behaviours of a system in real-time or through logs. Observations form the basis for hypotheses and are used to generate new traces or insights.
Trace	A recorded sequence of actions, events, or interactions within a system, typically used for analysis to derive new knowledge or insights about the system's behaviour.
Model	A structured representation of either the component or the environment in the context of reverse engineering.
Hypothesis	A proposed explanation or assumption made based on initial observations, used to guide further testing or analysis. Hypotheses are refined through iterative observation and testing.
Instance	A concrete realisation of a model used for testing or interaction.

1.6 Report Structure

The remainder of this report is structured as follows. Chapter 2 presents a review of the literature covering reverse engineering methodologies, including approaches to specific NFC systems and video games. It also discusses known weaknesses, vulnerabilities, and attacks on these systems. Chapter 3 explores practical methods for reverse engineering NFC systems, focusing on identifying issues of non-compliance and poor implementation in both recent commercial products and large-scale deployments. Chapter 4 examines the reverse engineering of video games, highlighting relevant tools and techniques. In Chapter 5, we propose a structured method for effective reverse engineering. Finally, Chapter 6 concludes the report and outlines potential directions for future work.

Chapter 2

Literature Review

In this chapter, the state of the art in academic reverse engineering is examined in Section 2.2. Section 2.3 provides an overview of NFC technology and its applications. In Section 2.3.1, current research on the security of various NFC protocols and possible attack vectors is discussed. Section 2.3.4 reviews the existing libraries and tools commonly used in NFC and smart card research. Section 2.3.6 explores current approaches to modelling security protocols, as well as relevant certification procedures. Reverse engineering in the context of video games is addressed in Section 2.4, where notable case studies and methodologies are presented. Finally, Section 2.5 investigates the legal landscape of reverse engineering, including relevant legislation and landmark legal cases that have shaped the field.

2.1 Literature Review Methodology

This literature review was conducted systematically to ensure comprehensive coverage of potential reverse engineering methodologies, NFC security, and video game security domains while acknowledging the unique challenges posed by the nature of security research and reverse engineering practices.

2.1.1 Search Strategy and Academic Sources

The primary literature search was conducted across established academic databases including IEEE Xplore, ACM Digital Library, Springer, ScienceDirect, and Google Scholar. Search terms were organised into three groups: (1) general reverse engineering methodologies (e.g. “reverse engineering”, “binary analysis”, “protocol analysis”, “software comprehension”, etc), (2) NFC and contactless security (e.g. “NFC security”, “EMV vulnerabilities”,, “contactless payment”, etc), and (3) video game security (“game hacking”, “anti-cheat”, “game preservation”, “cheat detection”, etc). The searches were

refined and filtered by publication date to prioritise recent work while establishing historical foundations.

The review employed a strategy whereby foundational papers from earlier period which underpinned more modern work was examined first to establish theoretical frameworks and methodological precedents, followed by the state-of-the-art research from more recent years to capture emerging techniques and contemporary challenges. This allowed us to examine how the methodologies of reverse engineering have evolved over time and understand any fundamental processes which are still core to the process.

2.1.2 Inclusion and Exclusion Criteria

Inclusion criteria required sources to be: (1) peer-reviewed publications from academic conferences, journals or blogs or other untraditional means which could be externally verified through source code examples or verified by third parties., (2) focused on methodological contributions, empirical evaluation, or significant case studies in reverse engineering. Exclusion criteria eliminated: (1) non-peer-reviewed sources from the primary academic search (with exceptions noted below), (2) purely legal or policy-focused papers without technical content, (3) duplicate publications of the same work, or similar enough where no new significant relevant contributions were mentioned, and (4) papers lacking empirical validation or clear methodological contributions.

2.1.3 Non-Traditional Sources and Grey Literature

Due to the inherent nature of reverse engineering and security research, a significant portion of valuable knowledge exists outside traditional academic publication venues. Reverse engineering, particularly in domains such as video game anti-cheat systems, proprietary protocols, and exploit development, operated at the intersection of legitimate security research and the "black hat" hacking community. Many practitioners present their findings at non-academic security conferences such as DEF CON, Black Hat or REcon, publish technical write-ups on personal blogs, contribute to underground forums, or release proof-of-concept code on platforms like GitHub without formal peer review. In the hacking community people don't gauge a hackers reputation or prestige though the amount or significance of published academic papers, but more the notoriety and social media following, or even how disrupting their efforts are and how much money they can get from a company either through responsible disclosure or more malicious actions.

These non-traditional sources, while lacking formal academic peer review, often provide several forms of validation that establish their credibility and technical merit such as: (1) executable proof-of-concept code or working exploits that can be independently verified,

(2) detailed technical write-ups with reproducible methodology, (3) validation through replication and discussion within expert hacker communities, (4) subsequent citation or acknowledgement by academic researchers or industry security teams, and (5) vendor responses, patches, or acknowledgements in security advisories. For example, anti-cheat system analyses published on technical blogs may include reverse-engineered source code reconstructions, documented function signatures, and working bypass demonstrations that have been independently replicated by multiple researchers in the community.

Where such non-traditional sources provided unique technical insights, documented novel techniques, or filled gaps in peer-reviewed literature, they were selectively included in this review. Their inclusion was justified based on: (1) technical rigour demonstrated through verifiable evidence, (2) community validation and replication, (3) relevance to gaps in academic coverage (e.g. specific anti-cheat implementations, proprietary game server protocols), and (4) influence on subsequent academic or industry work. This pragmatic approach acknowledges that in reverse engineering, the boundaries between "hacker culture" and academic research are blurred, and excluding non-traditional sources would omit significant methodological innovations and practical techniques that have shaped the field.

2.2 Current State of the Art

To contextualise the work presented in this thesis, it is important to examine existing literature that addresses the tools, techniques, and theoretical underpinnings of reverse engineering. This includes both practical guides aimed at teaching hands-on skills, as well as academic research that proposes systematic methodologies or formal models. In the following sections, we review notable sources that contribute to the current state of the art, beginning with the technical literature focused on low-level binary analysis.

2.2.1 Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation

Practical Reverse Engineering, according to its interior cover, claims to contain "a systematic approach to understanding reverse engineering, with hands-on exercises and real-world examples". It contains five chapters covering a broad basis of reverse engineering; 'x86 and x64', 'ARM', 'The Windows Kernel', 'Debugging and Automation' and 'Obfuscation'.

The book focusses primarily on reverse engineering different executable assembly code. It provides a background on x86 and ARM and explains their CPU architecture and the differences between the platforms. It additionally explores their relation to the Windows Kernel and how applications such as malware work. It provides guides for the

user to setup various debuggers and use automated techniques and gives examples and explanations of different obfuscation techniques.

One significant limitation is that its predominant focus is assembly-level reverse engineering. While this focus is undeniably important for understanding the low-level workings of binaries and executable code, it overlooks broader aspects of reverse engineering, such as the analysis of static files, which are equally crucial in many real-world scenarios.

The book provides a good technical background for someone interested in starting reverse engineering and provides some practical case studies and exercises for the reader to follow. However, it does not provide a broad general system approach to the reverse engineering process, but does provide a "systematic approach to the *understanding* of reverse engineering".

2.2.2 Reverse Engineering And Software Maintenance - A Practical Approach

This book uses a more abstract definition of reverse engineering. It uses the term to refer to software maintenance and the activity of having developers performing certain maintenance tasks on older software without any prior experience with the codebase or any up-to-date documentation.

It describes a model or stages for the software maintenance process:

- Understand the request (in the context of the system and operational environment: how domain aspects map onto application aspects).
- Delimit the sections of code that are involved in the change, and are affected by consequences of the change.
- Determine in detail the code or system changes needed and identify a test set.
- Implement these changes.
- Test (rerun existing tests where relevant, and new tests for the new behaviour).
- Update documentation, and document the changes made and the understanding gained.
- Install the new version.

2.2.3 Reversing: Secrets of Reverse Engineering

This book introduces two separate phases of software reverse engineering. The first, *system-level reversing*, is summarised as "a kind of large-scale observation of the earlier

program". It helps determine the general structure of the program and sometimes even locate areas of interest within it. It describes the process as running various tools on the program and utilising various operating system services to obtain information, inspect program executables, track program input and output, etc. Most of the knowledge is derived from the information coming from the operating system, because by definition every interaction that a program has with the outside world must go through the operating system.

The second phase, *code-level reversing* is described as an art form. It is defined as extracting design concepts and algorithms from a program binary that requires mastery of reverse engineering techniques along with a solid understanding of software development, the CPU, and the operating system. It describes a process for understanding the link between high-level code and compiler artefacts in low-level programming constructs, the assembly language, and the inner workings of compilers. The purpose of the book is to provide the reader with the knowledge, tools, and techniques needed to perform effective code-level reverse engineering.

While a majority of this book covers reverse engineering programs in order to get an insight into their workings, Chapter 6, 'Deciphering File Formats' covers the general process of reverse engineering program data and also network data. It defines a process called *data reverse engineering* as "the process of deciphering such data to the point where it is possible to use it for the creation of programs that can accept and produce compatible data". The author then introduces a program they have written called 'Cryptex' which manages archive files that can contain multiple encrypted files, using its own 'Cryptex File Format'. The author then takes us through a step-by-step process of reverse engineering this file format. They use various code-level reverse engineering techniques to understand how the program runs and how it processes and manipulates files. This chapter is fairly flawed in that the author already knows all the underlying details of the file format and how the program works, and essentially helps us translate assembly code into more human-readable instructions to understand the flow of the program. It acknowledges the limitations of this process and the fact that many real-world file formats may employ a far more complex structure. It additionally describes what it refers to as a "primitive, yet effective, approach" which is to let the program update the file and observe changes using a binary comparison program and that, combined with code-level reverse engineering, can produce extremely accurate results.

The book also covers other reverse engineering scenarios, such as Auditing Program Binaries, which describes the process of auditing a program where source code is unavailable and evaluating assembly code for security vulnerabilities. The chapter Reversing Malware starts with a general discussion on basic malware concepts and different types of malware before going on to demonstrate malware analysis on real-world malware. The next chapter, Cracking, describes common techniques employed by software companies in order to prevent software piracy. The next chapter, Anti-reversing Techniques,

describes the problem state in which software developers want to make their software resistant to reverse engineering and describes various techniques in order to complicate the reverse engineering process.

The book covers a broad range of topics in reverse engineering. However, the book primarily focuses on providing a great technical background on the areas of reverse engineering that it covers and focuses on a curated guided walk-through on specific reverse engineering scenarios. It does manage to highlight very specific patterns that would help the reader reverse engineer similar applications, however, the book makes no attempt to generalise the reverse engineering process.

2.2.4 Model-Driven Reverse Engineering Approaches

Recent academic work has focused on formalising reverse engineering through model driven approaches, primarily in the software maintenance domain. [82] conducted a systematic review of the literature of model-driven reverse engineering (MDRE) approaches, analysing 15 different frameworks that extract architectural models from existing software systems for documentation and modification purposes. These approaches focus on recovering UML diagrams, architecture models, and business processes from legacy source code to support software evolution and migration to new platforms. While MDRE demonstrates that model-based representations can improve automation in specific contexts, these frameworks are domain-specific (primarily targeting enterprise software maintenance) and do not provide a general methodology for reverse engineering unknown systems across, hardware, protocols, and binary artifacts where source code is unavailable.

[83] introduced the reAnalyst framework to analyse and annotate reverse engineering activities in controlled experiments. Their work focuses on capturing user interactions during reverse engineering sessions to evaluate tool effectiveness and measure analyst performance. While their framework models observable activities and tool usage patterns, it does not propose a generalized methodology for conducting reverse engineering itself, nor does it address the fundamental problem of systematic knowledge acquisition from unknown components across diverse domains.

[84] provided a comprehensive analysis of client-side anti-cheat defences in modern video games, reverse engineering four major anti-cheat systems: BattleEye, Easy Anti-Cheat, FACEIT, and Vanguard. Their systematic approach documented specific countermeasures including memory scanning detection, code injection prevention, and kernel-level protections, revealing that even sophisticated systems contain exploitable design weaknesses.

For hardware reverse engineering, recent work focused on automated analysis of firmware and embedded systems. [85] demonstrated neural network-based approaches for cross-platform binary code similarity detection, while [86] developed techniques for cross-architecture bug identification in binary code, deomonstrating that machine learning can complement traditional static analysis methods. However, these approaches assume the availability of training data from known binaries and focus on specific analysis tasks (similarity detection, bug finding) rather than providing a systematic methodology for understanding and reconstructing the behaviour of completely unknown systems through the iterative process of observation, hypothesis formation, testing, and model refinement, regardless of whether analysis is performed manually or with automated tools.

2.3 Near Field Communication

Radio frequency identification (RFID) is a wireless communication method that is used for a wide range of purposes, such as transportation, access control, as an alternative to barcodes, tracking items on assembly lines, or identification of livestock [87]. One of its main advantages over other smart cards/electronic keys is that it does not require line of sight or physical contact. If high privacy or security demands have to be met, such as in electronic passports [88], or more recently contactless payments [89], the ISO 14443 standard (near field communication) [90] is used, as it offers sufficient computational power for cryptographic purposes.

RFID technology introduces new threats compared to physical-based communication methods such as electronic identification cards with contacts [91]. One of the benefits of the technology, the fact that it does not require line of sight, can also be a disadvantage, as a card can be read or modified without the consent or knowledge of the owner.

The cost of deploying such a large infrastructure and technology can be quite expensive and therefore some implementations will try and reduce costs by either using outdated technology with weak cryptographic techniques or don't use any cryptographic techniques at all. The process of migrating from outdated insecure systems can be done in a relatively seamless process. New card readers can be installed that accept new and old cards, then new cards can be deployed and cards can be used in parallel, then once all cards are replaced, readers can be configured not to accept old cards [92].

NFC Tools

NFC systems have been suggested to be particularly vulnerable to relay attacks. In most cases, it is possible to relay NFC communications to a remote card. However, some problems can arise due to the latency of messages being sent, which means that if messages take longer than expected, the desired procedure may not complete [93].

2.3.1 Mifare Classic

These cards have been in circulation for nearly 20 years with over 1 billion ICs and 7 million reader components sold. They use the proprietary Crypto1 stream cipher which was kept secret for more than 10 years. The Mifare Classic was reverse engineered in 2007 [94] [49] [95] and since then many weaknesses have been discovered. Many attacks have been developed from the weaknesses found [3].

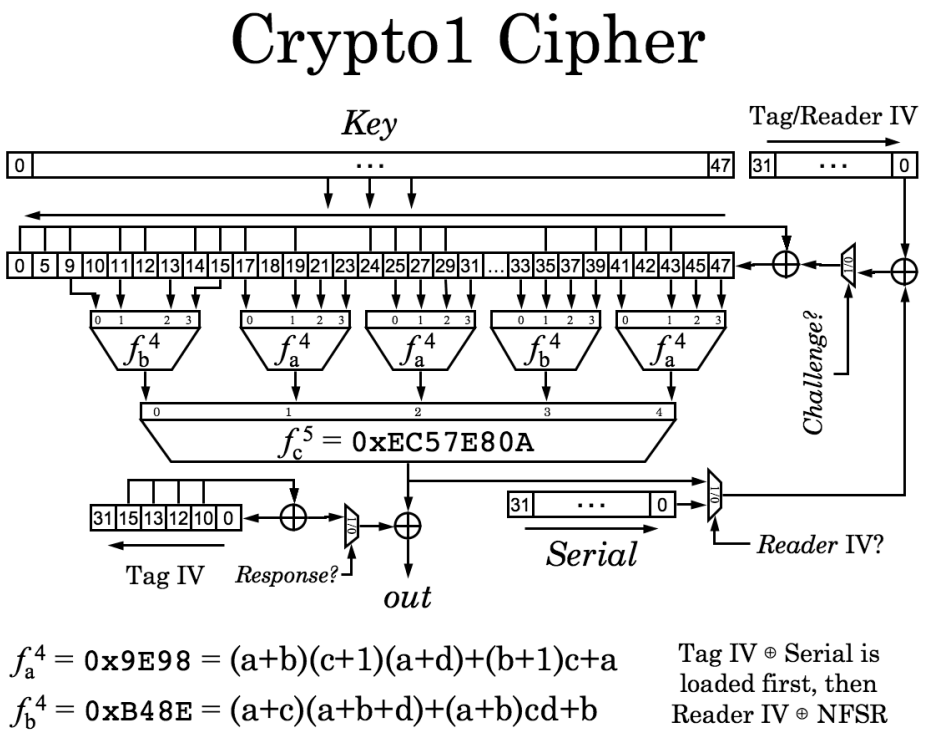


FIGURE 2.1: The CRYPTO1 Cipher [2]

The heart of the Crypto1 cipher is a 48-bit linear feedback shift register (LFSR) 2.1. The authentication protocol used in MIFARE Classic follows a challenge-response mutual authentication mechanism. The stream cipher has to be initialised at both the reader and card during the authentication session before it can be used for subsequent encryption and decryption. Both the tag and the reader must achieve the same cipher state at the end of initialisation for authentication to be achieved.

According to the official data sheet for the Mifare Classic [96], authentication between the reader and the tag is described as a three-pass mutual authentication. The reader first requests to be authenticated for a particular block, then the following three passes occur.

Pass	Action
1	The tag sends a challenge nonce n_t to the reader.
2	From this message onwards, communication is encrypted. The reader replies with an encrypted reader nonce n_r and an encrypted reader response a_r to the challenge in the previous stage.
3	If the reader response is correct, the tag will respond with an encrypted tag answer a_t. If the reader response is incorrect, the tag will not respond.

Once this three-pass mutual authentication is completed successfully, any further operation on the blocks within the same sector can be taken. If operations on blocks in another sector are required, another three-pass mutual authentication will be required.

The challenge nonce is a 32-bit value that is generated by a pseudorandom number generator, and was revealed in [94] to be a 16-bit LFSR which is a separate circuit from that of the 48-bit LFSR used for the cipher. Even though the nonce is 32-bit wide, the LFSR that generates it is only half that width at 16 bit, which is far from secure. The pseudo-random number generator LSFR can be defined as follows, with x_n denoting the bit value and $\oplus$ being the exclusive or logic function.

$$L(x_0 x_1 \dots x_{15}) := x_0 \oplus x_2 \oplus x_3 \oplus x_5$$

Moreover, if we define a 32-bit LFSR sequence $x_0 x_1 ... x_{31}$ consisting of this 16-bit LFSR, then the next 32-bit sequence can be defined using a successor function suc where

$$suc(x_0 x_1 \dots x_{31}) := x_1 x_2 \dots x_{31} L(x_{16} x_{17} \dots x_{31})$$

$$suc^n(x_0 x_1 \dots x_{31}) := suc^{n-1}(suc(x_0 x_1 \dots x_{31}))$$

NXP does not disclose how a_r and a_t are calculated. This is not disclosed in the NXP documentation. However, full details are not given in [49]. The reader and tag responses, a_r and a_t, are derived from the tag nonce using the following relationships.

$$a_r := suc^{64}(n_t)$$

$$a_t := suc^{96}(n_t)$$

The successor function is used to generate the response a_r in the second pass and a_t in the third pass. The card can verify the key the reader has by decrypting and confirming the validity of a_r. If this fails, then the card will not continue to the third pass. The reader can also check the validity of a_t in the third pass.

The Mifare Classic 1k features an 8192 Bit EEPROM which is split into 16 sectors each containing 4 blocks. Each block contains 16 bytes. The first block of the first sector is known as the Manufacturer Code, which includes the unique identifier of the card (UID), as well as a bit count check byte and other data which can determine the manufacturer, this block is not writeable on standard Mifare Classic Cards. You can see the memory layout in Figure 2.4.

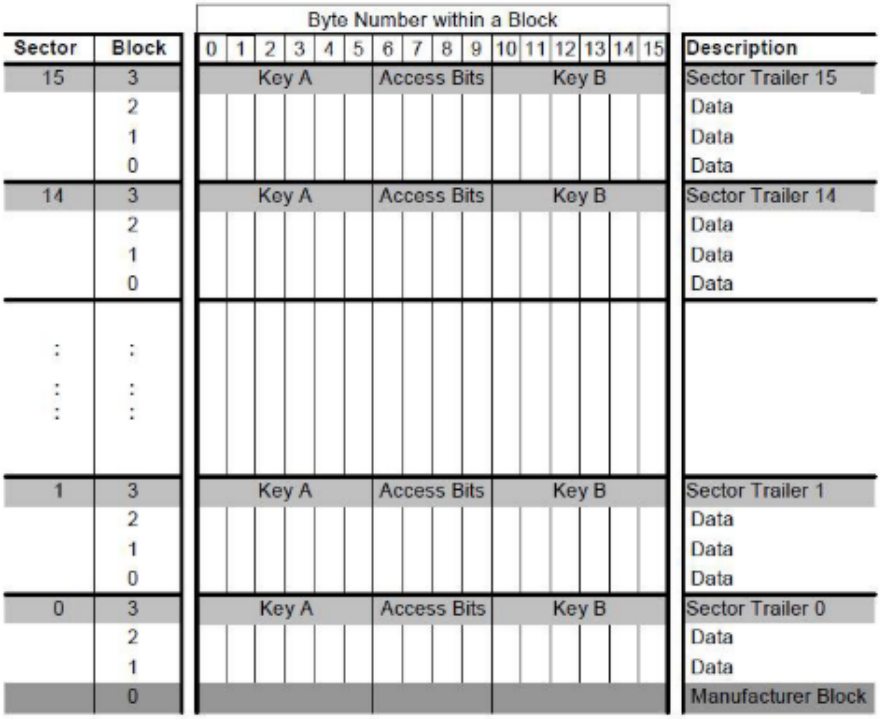


FIGURE 2.2: Mifare Classic 1K Memory [3]

The fourth block of each sector is known as the trailer, which contains at least 1 access key (Key A) and an optional Key B. If a second key is not needed, then the last 6 bytes can be used as data bytes [96].

There are a few operations that can take place on a Mifare Classic chip which are: read, write, increment, decrement, transfer, and restore. The access bits shown in Figure 2.4 define the operations that can be performed with each key. These access bits can define different permissions to different blocks for each key, they also control the permissions to modify the Sector Trailer [3].

Vulnerabilities and Attacks

Since the reverse engineering of the Crypto1 cipher [49], many vulnerabilities have been discovered and many attacks have been released based on them.

The pseudo-random generator on the card, which initiates the algorithm by generating a nonce, is weak. The nonce is generated by a 16 bit LFSR that has an entropy of 2^{16}. The card will return to a known state when it starts to operate. This means that we can wait a fixed amount of time after powering up the card before requesting the nonce. It is possible to get the same nonce every 30 ms [97].

Exploiting the above weakness allows us to recover the keystream used in the Crypto1 cipher. Eavesdropping on the communication between a legitimate card and a legitimate reader and ensuring the card uses the same keystream, we can then modify plain text commands, for example the block number in a read command. The weak pseudo-random generator allows us to reply earlier recorded communications. The attack is based on the principle that we can flip ciphertext bits to try to modify the first command so that it gives another result [48].

There is a weakness in the parity bits when used in the three-pass mutual authentication. During authentication, when the reader sends a_r and n_r the card checks the parity bits before checking the correctness of a_r. If one of the eight parity bits is incorrect, the card will not respond. However, if the eight parity bits are correct but the response a_r is incorrect, it will respond with a 4 bit error code 0x5 (NACK) indicating a transmission error. The error code is sent encrypted even though the the reader is not able to decrypt it as it failed the second pass. Comparing the known plaintext error code 0x5 with the encrypted version, you can recover 4 bits of the keystream [98].

An open source implementation of an attack using this vulnerability is the Mifare Classic DarkSide Key Recovery Tool included in the MiFare Classic Universal toolKit (MFCUK). This attack is card-requires no access to a reader making it a card-only attack. Using this attack, you can generate a key for a sector in less than 300 queries to the card, it also does not require pre-calculation, and it has an instant run time [99].

2.3.2 Mifare DESfire and DESfire EV1

NXP later released newer card systems employing more advanced security protocols and later urged older deployments to upgrade to the newer systems.

Mifare DESfire and DESfire EV1 cards comply with Parts 1-4 of ISO 14443A [90] and offer optional ISO 7816-4 [91]. It has a unique 7 byte UID in a locked section of memory that is programmed at the manufacturing time [47]. NXP have chosen offer open cryptography standards in these newer cards, the DES in the name indicates that they offer 3DES or AES hardware cryptographic engine for communication. DES is a widely used block cipher algorithm. It is designed to encrypt and decrypt data blocks consisting of 64 bits with a 56-bit key. AES has a fixed block size of 128 bits and a key size of either 128, 192, or 256 bits.

With the DESfire EV1 data rates of up to 848 kbits/sec can be achieved, and the card can hold up to 28 different applications and 32 files per application. The files can be one of the following types: Standard data files, Backup data files, Value files with backup, Linear record files with backup, Cyclic record files with backup.

Mifare DESfire uses a mutual three-pass authentication, which has to be completed before reading and manipulating the data. The symmetric key of the card K_C and of the reader K_R should be identical.

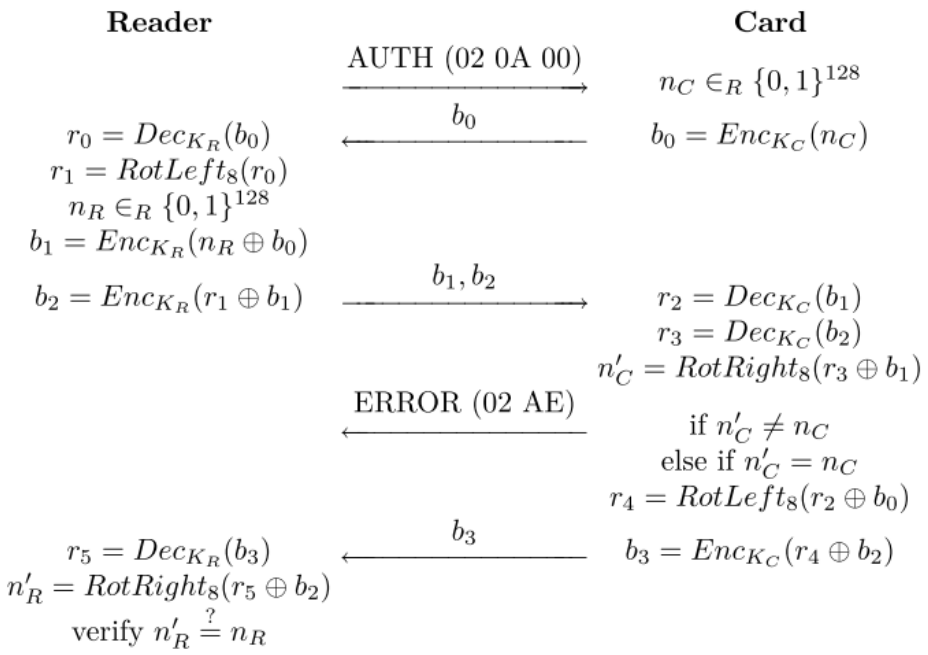


FIGURE 2.3: The Mifare DESFire authentication protocol [4]

Depending on the communication settings of the file/application, the following options of secure communication can occur:

1. Plain data transfer (only possible within the backwards-compatible mode to MF3ICD40)
2. Plain data transfer with cryptographic checksum (MAC): Authentication with backward compatible mode to MF3ICD40: 4 byte MAC, all other authentications based on DES/3DES/AES: 8 byte CMAC
3. Encrypted data transfer (secured by CRC before encryption): Authentication with backwards-compatible mode to MF3ICD40: A 16-bit CRC is calculated over the stream and attached. The resulting stream is encrypted using the chosen cryptographic method. All other authentications-based DES/3DES/AES: A 32-bit CRC is calculated over the stream and attached. The resulting stream is encrypted using the chosen cryptographic method.

Vulnerabilities and Attacks

These newer cards implement AES and 3DES, for which no efficient analytical attack or brute force attack exists, can still be vulnerable to side channel attacks. Researchers have demonstrated practical, non-invasive side-channel attacks on the Mifare DESfire contactless smart card, recovering the complete 112-bit secret key [5].

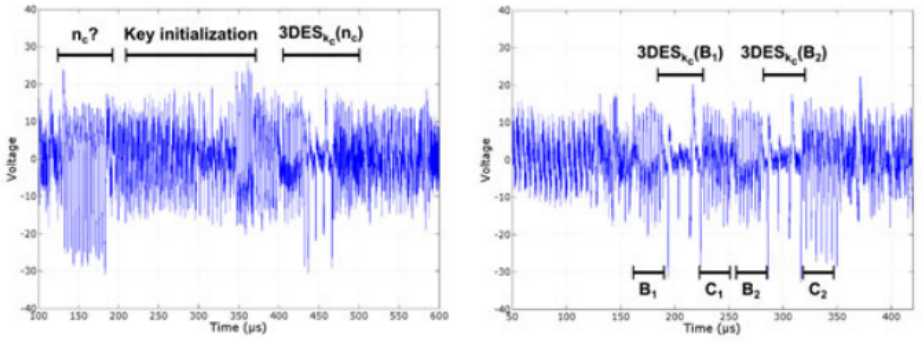


FIGURE 2.4: Annotated traces during the authentication protocol [5]

The attack can be performed within a few hours whilst collecting 250,000 traces and poses a severe threat to the security of DESFire based systems. The Mifare DESFire EV1 is not vulnerable to this attack.

2.3.3 Implementations

NFC cards can be used for a large amount of different applications, such as automated fare collection systems, identification cards, access control, loyalty cards, and many more. More importantly, it can be found in government facilities like the Cabinet Office, nuclear facilities and big banks.

NXP recommended that cards have one sector encrypted with a default key, however, this makes Mifare Classic cards easily vulnerable to the nested attack discussed in Section 2.3.1.

One high profile example of an implementation of Mifare Classic cards is the Transport for London Oyster card system, which stopped distributing Mifare Classic cards in December 2009 and later used DESfire cards. The Oyster card system diversified all keys on their cards making them immune to the Nested attack. The cards store information such as the balance on the card and the previous information from the trip [3]. In other parts of the UK, such as the Unilink bus service in Southampton, the ITSO standard is used. ITSO licenced members stopped issuing Mifare Classic cards after 31st December 2009; however, Mifare Classic Cards were supported until 31st December 2016. The Unilink service previously used University of Southampton ID cards, but in July 2013 they switched to ITSO standard cards, which currently are Mifare DESfire cards.

2.3.3.1 Integrated Circuit Contact Smart Cards

Contact Smart Cards predate NFC Cards and are widely used in applications such as bank payment cards (which are Europay MasterCard Visa (EMV) compliment cards) [100] [101], subscriber identity module (SIM) [102] cards which are used in mobile phones and are also used in high-security identification and access badges. ISO7810 defines physical characteristics such as dimensions, resistance to bending, flames, chemicals, and toxicity [103], while ISO7816 defines the internal characteristics such as command-response pairs, access methods and various secure features [104].

2.3.4 NFC Interfaces and Libraries

There are a number of libraries or APIs (Application Programming Interfaces) that allow application developers to allow their programs to interface with a card through an NFC reader. Many NFC readers implement a number of specific protocols in their firmware. Various APIs exist around software embedded on the chip, most using JavaCard [105].

2.3.4.1 Tools

WinSCard Tools [106] are a set of programs and libraries which aim to provides a new C# wrapper for the underlying winscard.dll [6], a graphical user interface supported by a plugin system to allow extensibility of the main GUI. The GUI is aimed at educational and research projects and allows recording and replay of ADPU 2.3.4.3 commands. The proof of concept was the replay of the Cambridge attack [107].

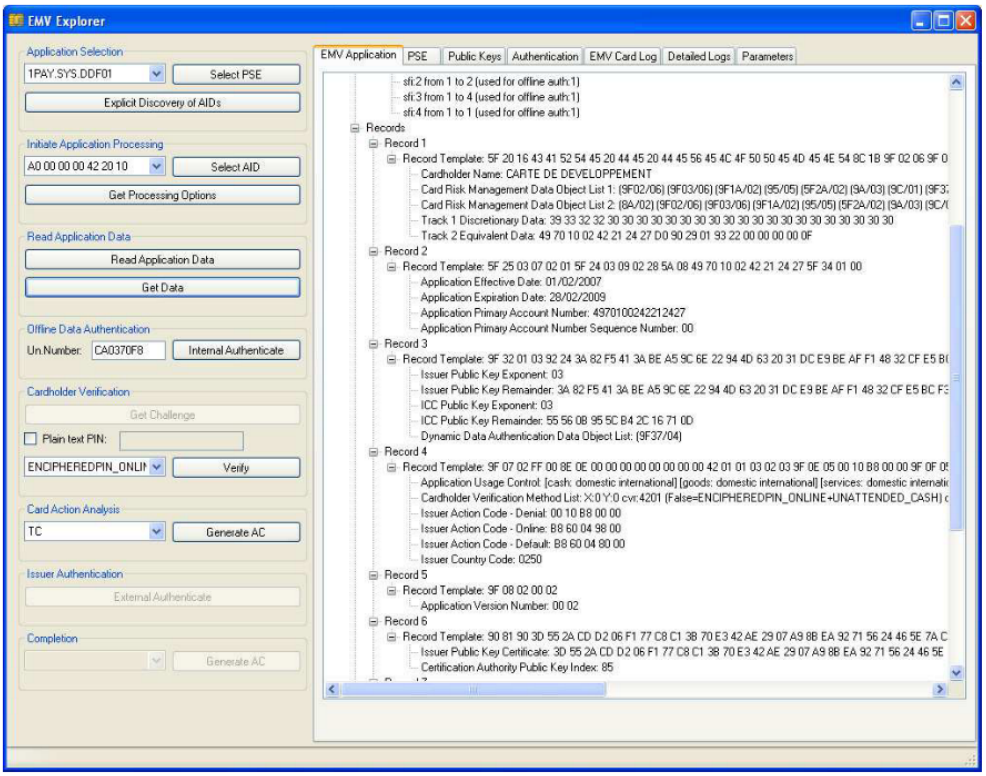


FIGURE 2.5: WinSCard Tool EMV Explorer plugin [6]

2.3.4.2 Vulnerabilities

An area well explored is the EMV protocol used for Chip and Pin payments [107] [108]. Prior to 2009 UK banks were issuing cards with a variant of EMV called static data authentication (SDA) which is not capable of performing RSA operations, and it only presents the terminal with a static certificate. SDA cards are vulnerable to well-known replay attacks in which this certificate is put on a counterfeit card commonly known as "yes cards" [107] because they will accept any PIN that is entered. This exercise can only be performed successfully on terminals without an online connection to the banking network.

In 2010 a protocol design error was discovered that allowed transactions to be authorised with a stolen card [107]. The attack relies on "offline PIN" transactions, as opposed to "online PIN" transactions, where the PIN is sent encrypted to the card issuer for verification and is used in ATMs. Unlike the "yes card" attack, this also works on the newer dynamic data authentication (DDA) cards. This works by launching a man-in-the-middle attack between the card and the point-of-sale machine (POS) and informing the POS machine that the pin was correct. The card will think that the transaction was completed using a signature.

```
        if VERIFY_PRE and command[0:4] == "0020":
                debug(Spoofing VERIFY response)
                return binascii.a2b_hex("9000")
```

Listing 2.1: Pseudocode used for "offline pin" transaction exploit [107]

In 2014 an attack dubbed the "pre-play" attack [108] was documented which relied on weak pseudorandom number generators in Automated Teller Machines (ATM). This allows a card issuer to authorise an ATM to dispense cash when it is given a predicted authorisation request cryptogram (ARQC). It was also discovered that some card issuers will accept replayed ARQC, however, a competent issuer prevents replay by rejecting any transaction whose application transaction counter (ATC) it has already seen.

2.3.4.3 Application Protocol Data Unit

An Application Protocol Data Unit (ADPU) defines the communication between a smart card reader and a smart card, generally with physical contacts, which is defined in ISO/IEC 7816-4 [91]. This specification describes communications as command-and-response pairs. It describes the command header as 4 bytes which contains the class, instruction, and parameters. It also includes a length variable that is encoded in the Basic Encoding Rules (BER) format. When the class byte has bit 8 set to 1 it indicated a proprietary class, except for the value 'FF' which is invalid.

2.3.4.4 PC/SC

Personal Computer/Smart Card (PCSC) is a specification for integrated circuit smart card integration into computing environments. The contributors are Apple, Axalto, Gemplus, Hewlett-Packard, IBM, Infineon, Ingenico, Microsoft, Phillips, Siemens, Sun Microsystems, Toshiba and VeriFone and are part of the PC/SC Workgroup [109]. It was designed as an industry standard API that is meant to provide a card reader standard so that any PC/SC-compliment smart card can work across multiple platforms.

Work has been done to model a transaction mechanism primarily aimed at chip cards. Due to its electronic characterisation and to its physical constraints, data modifications cannot be done atomically. [110] designed a model in which, unless the operation had been completed, a backup state of the card is used. It is possible that a mechanism similar to this could be implemented into NFC based devices; however, this is beyond the scope of this report.

2.3.4.5 NFC Proxy

Using a device such as that described in [111] you can extend the range of NFC communications, which can be a large security risk. This would allow communication between an NFC reader and card from a great distance, possibly without the card or the owner of the card being anywhere near the door.

Similar methods are used in car theft. This takes advantage of the new Passive Keyless Entry and Start (PKES) [112] system that has been implemented in cars in recent years. Experiments have been performed that allow hackers who can reach within 8 metres of a key to unlock a car 50 metres away [113]. A PKES relies on the key giving off an RFID signal which is picked up when in range of the vehicle and allows the doors to be unlocked without pushing any buttons, then it additionally allows you to start the vehicle in the same way.

2.3.5 Other NFC uses

There are a wide range of uses for NFC besides access control, such as Bluetooth Pairing, contactless payments used in bank cards and mobile technologies such as Apple Pay and Android Pay, loyalty cards, sharing data between mobile phones such as contact information, and much more.

2.3.5.1 NFC toys

NFC toys (sometimes referred to as toys-to-life) are figurines with a built-in NFC tag. Games such as Skylanders or Disney Infinity which are available on Playstation, Xbox and Wii consoles use these NFC figurines to unlock characters or extra gameplay features. It is a creative way to sell additional features or characters for a game while providing customers with collectable figurines.

These systems will usually have three parts; the console which will run the game, an NFC reader which is connected to the console via a USB cable, and the NFC figurine. This leaves it open to a number of attacks that attempt to compromise the security of the implementation to unlock the features of the game.

One approach would be to bypass the section of the game code that interfaces with the USB NFC reader and give the responses directly to the game to unlock the characters. However, games consoles are usually locked down systems which require all code to be signed before they will run on the console.

Another approach would be to emulate the NFC reader USB device and send the commands to the console in software to register figurines placed on the reader. However, in

some implementations such as the Disney Infinity 'Portal' they use a custom command scrambling algorithm to obfuscate the USB communications.

The approach in which we are most interested is the data stored on the NFC tag and the communications with the reader. Disney Infinity figurines are identified as MIFARE Classic. As discussed above, Mifare Classic cards are known to be insecure with many known attacks. It is possible that these were chosen because they are cheaper than their more secure and updated counterparts. Game companies usually put a lot of resources into protecting their intellectual property.

The games do not require the console to go online, so unless there is a hardcoded list of sector keys for every figurine released, they must be derived from the UID of the tag or there could be a consistent key for all of them.

2.3.5.2 State of the Art Ticketing

Many large transport networks used in cities have moved or are in the process of moving to fully digital ticketing systems. In 2014 Transport for London made all of the buses on the network cashless, meaning that you were not able to purchase a ticket on the bus and relied on travellers using Oyster Cards, Contactless payment cards, or pre-purchased tickets. There were many motivations behind this, such as decreasing the amount of time buses were stopped to process payments, removing the need for employees to deal with cash directly, and ultimately reducing costs. It costs £24 million a year to accept cash on London's buses [114].

OPTIMOS (acronym for Open PracTical Infrastructure for MObile Services) was a proof-of-concept national e-ticketing system using NFC enabled mobile phones [115]. The German federal government created an NFC initiative that aims to create an open NFC-based ecosystem using secure identities for mobile services. This project aimed to create an interoperable NFC interface connecting NFC mobile devices with existing readers in public transport vehicles that implement ISO/IEC 14443, ISO/IEC 18092 and EMVCo L1.

2.3.6 Modelling Security Protocols

We can choose to model the implementation of a system to verify that it is compliant with a given specification. The verification of security protocols has been around for a long time, and some different tools and modelling techniques are available. It is important to note that proving that an implementation is in compliance with the specification and that the specification is cryptographically secure does not confirm that a system is not vulnerable to side channel attacks such as those described in [5].

Mifare-based cards and readers can be verified by independent test houses such as Arsenal Testhouse, UL, and LSI-TEC. However, it appears that these certifications are given to devices, whether cards or readers, and not to an implemented system. We will also discuss products in Section 3.5.1 that appear to have no certification from any of these companies [116].

2.3.7 Recent NFC Security Research

Research focuses have increasingly shifted toward NFC implementations in smartphones, driven by widespread adoption of mobile payment systems and the unique security challenges introduced by their use in portable computing platforms.

2.3.7.1 Smartphone-Base Analysis Tools

[117] developed NFCGate, an open-source toolkit that transforms Android smartphones into powerful NFC research tools supporting relay, replay, and traffic modification attacks. Their evaluation demonstrated that relay latency remains low enough to bypass many distance-bounding countermeasures, with negligible replay latency making attacks nearly indistinguishable from legitimate transactions. The tool's plugin architecture enables custom protocol analysis, and its integration with Wireshark provides comprehensive traffic inspection capabilities. This work exemplifies how smartphones have become both attack vectors and research platforms for NFC research analysis.

2.3.7.2 Mobile Payment Vulnerabilities

[118] demonstrated practical relay attacks against Google Wallet, exploiting the smartphone's general purpose architecture to intercept and forward NFC communications over network connections, effectively bypassing proximity requirements. A 2024 systematic review [119] documented continued vulnerabilities in mobile NFC implementations, including CVE-2023-35671 [120] which allowed unauthorised NFC devices to access credit card details through Android's Screen Pinning feature. The research demonstrated that while tokenisation and multi-factor authentication provide theoretical protection, implementation weaknesses in mobile operating systems persist across deployed systems. The integration of NFC into general-purpose smartphones introduces attack surfaces absent in dedicated payment terminals and normal smart cards, including malware distribution, OS-level vulnerabilities, and application level exploits.

Mobile payment systems introduce distinct attack surfaces compared to physical contactless cards, particularly when convenience features bypass standard authentication mechanisms. [121] demonstrated critical vulnerabilities in Apple Pay's Express Transit

mode, which allows payments at transport terminals without user authentication (fingerprint, Face ID, or PIN). By replaying specific byte sequences broadcast by Transport for London readers, the researchers bypassed Apple Pay's lock screen and performed relay attacks enabling fraudulent Visa transactions of any amount from locked iPhones to arbitrary EMV readers. The attack exploited the fact that Apple Pay does not enforce value limits in transport mode and Visa's backend fails to verify the Issuer Application Data (IAD) field, which records whether Consumer Device Cardholder Verification Method (CDCVM) was performed. Mastercard transactions were protected against this specific attack because the Merchant Category Code (MCC) is cryptographically authenticated in Mastercard's protocol, preventing relay from transport readers to shop readers. [121] additionally analysed proposed relay countermeasure protocols from both Visa and Mastercard, demonstrating that Visa's Level 1 binding mechanism can be defeated using rooted smartphones to force card identifiers, while Mastercard's Relay Resistance Protocol (RRP) suffers from timing variability at the EMV application layer (Level 3) that makes reliable distance bounding difficult. The research proposed L1RP, a new relay protection protocol operating at the ISO 14443 [90] Level 1, which provides more stable timing measurements and combines timed nonce exchanges with cryptographic binding across protocol layers, verified using the Tamarin prover for formal correctness.

The recent academic literature reveals a consistent pattern, that while cryptographic primitives in NFC systems remain generally sound, implementation vulnerabilities (particularly in smartphone-based systems, side-channel leakage, and protocol-level weaknesses) create exploitable attack surfaces. The shift from dedicated NFC hardware to smartphone implementation has introduced new vulnerabilities, OS level bugs, application security issues, and malware threats, while distance bounding protocols, through theoretically sound, face practical deployment challenges. Relay attacks remain a persistent threat particularly in smartphone implementations where software-based architectures may offer less resistance than hardware isolated secure elements.

2.4 Video Game Reverse Engineering

Reverse engineering has played a crucial role in video game preservation, compatibility enhancement, and modding. One particularly illustrative example is the reverse engineering of legacy game titles to extend their functionality and support modern hardware and software environments.

2.4.1 KeeperFX

The case of *Dungeon Keeper* and its fan-made reimplementation, *KeeperFX*, offers a practical insight into the methods and mechanisms behind such efforts.

In 2007 Bullfrog Productions released Dungeon Keeper, which was a dungeon management 'god game' where the player constructs and manages a dungeon, recruiting and catering for minions to run it and defend it from enemy invaders. The game was originally built for Windows 95 and MS-DOS and consequently had compatibility issues running on Windows NT based operating systems and did not support modern internet protocols. In 2008 a Polish engineer called Tomasz Lis started to reverse engineer the Dungeon Keeper codebase to increase compatibility, support custom modifications, and improve multiplayer capabilities. The project was titled Dungeon Keeper Fan eXpansion (KeeperFX).

Tomasz Lis used a process to transform an ordinary executable file into a dynamically linked library (dll). The requirements of the process are listed here:

- An assembler interpreter and a disassembler
- Knowledge about EXE and DLL file formats
- Knowledge about programing in assembly in Windows
- EXE headers editor
- LOTS of motivation
- Lots of will to solve problems on your own

The process was then broken down into four sections [122]. 'Modifying EXE file header' which defines the process of changing the Portable Executable (PE) file and flagging it as a DLL, creating a new export section and marking it as an Export Directory. Then 'Filling the Export Directories' converts the process of filling the export directory with function relative virtual addresses (RVAs), ordinals, and names. They even created a tool that can update the export table incrementally called the PE/DLL Rebuilder of Export Section (PeRESec) that is designed to be used with IDA Pro. The 'Creating the new entry point function', which describes the process of converting the main exe entry point function into a dll entry point function in order to prevent running the main program function but keeping the libraries initiated. The actual main function can then be called externally from the new executable. The final section 'relocations' describes the differences in how exes and dlls are loaded into memory, and describes the use of relocation tables which are used to help changing addresses when dlls are loaded but the original exe does not have a relocation table.

In 2022 it was announced that the last part of the original Dungeon Keeper code was rewritten so that the dll which contained the original game code was no longer required. In 2024 they announced that they had the game compiling on Linux.

While the process of reverse engineering Dungeon Keeper can be considered a success as they have fully engineered the codebase, the original process is not fully documented and does not contain any fully fleshed out examples. However, the process can be seen as iterative, as the process of converting the original exe file into a dll allows individual functions to be rewritten in C code and allows missing parts not yet rewritten to be called in their original assembly form. The process is quite specific and technical, and the author highlights that the process may not be as simple for other projects.

2.4.2 Anti-Cheat Systems

Video game reverse engineering presents unique challenges due to increasingly sophisticated protection mechanisms, proprietary file formats, and the tensions between security (anti-cheat) and preservation goals.

Modern multiplayer games deploy kernel-level anti-cheat solutions that operate at the deepest levels of the operating system. Recent academic research has systematically analysed these systems.

[123] examined kernel-level anti-cheat systems against rootkit characteristics, analysing BattleEye, Easy Anti-Cheat, FACEIT Anti-Cheat, and Vanguard. Their framework evaluated systems based on privilege level, persistence mechanisms, stealth capabilities, and system integrity impact. They found that Vanguard and FACEIT exhibit rootkit-like behaviour through boot-time initialisation, kernel-mode operation, and deep system hooks, raising significant privacy and security concerns.

[84] reverse engineered BattleEye's architecture though dynamic and static analysis, identifying specific anti-cheat mechanisms including memory protection, driver signature verification, and integrity checking. Their proof-of-concept bypass demonstrated design weaknesses allowing malicious unsigned drivers to execute despite active protection, highlighting the difficulty of securing client side systems against determined adverseries.

In addition to kernel-level protection, game companies deploy sophisticated server-side systems. Blizzard Entertainment's Warden anti-cheat system, first deployed in World of Warcraft, represents an early example of hybrid client-server anti-cheat architecture. Documented reverse enginnering efforts revealed Warden's use of dynamic code injection, memory scanning, and encrypted communication protocols to detect unauthorized game modifications. The system's modular design allowed Blizzard to update detection signatures without client patches, though determined adversaries documented methods to intercept and decode Warden's scan modules [11].

2.4.3 Machine Learning Approaches

Recent work has explored machine learning for cheat detection. [124] proposed vision-based cheat detection that analyzes frame buffer outputs to detecrt illicit overlays without requiring invasive system access. Their approach demonstrates robustness against adversarial methods to evade detection. [125] introduced AntiCheatPT, a transformer-based approach achieving high detection rates on competitive gaming data while maintaining acceptable false positive rates.

2.5 Legality

When reverse engineering is used, there are often significant legal concerns. Many devices and protocols are proprietary, and the companies that develop them have strong commercial interests in protecting their trade secrets. Discovering and disclosing security vulnerabilities without allowing companies a reasonable amount of time to respond can negatively impact their reputation, their products, and, in some cases, their wider business operations.

In response to the growing interest from the security community, many large organisations have introduced bug bounty programs. These programs reward ethical ("white hat") hackers for responsibly disclosing vulnerabilities. Typically, the process gives companies time to develop and release a fix before the details of the vulnerability are made public, thus protecting customers' data while still recognising the researcher's contribution. In contrast, underground criminal organisations are often willing to pay substantial sums for undisclosed vulnerabilities, particularly if they provide access to proprietary information, customer data, or other assets that could be exploited for fraudulent or illegal purposes.

Despite the general restrictions, there are legal protections in many jurisdictions that allow reverse engineering in specific circumstances. These protections often exist to safeguard consumers, provide interoperability, and enable security research. In the following sections, we will explore some of the key laws and rulings that impact the legality and practice of reverse engineering.

2.5.1 Interoperability

One of the most prominent legal justifications for reverse engineering is the ability to enable interoperability between systems or software. This principle has been tested and clarified in several landmark legal cases, particularly in the realm of software and

consumer electronics. The following case illustrates how reverse engineering for interoperability has been challenged in court and the resulting precedent that continues to influence digital rights and software development.

Sega vs Accolade

After the release of the Sega Genesis in 1988, video games publisher Accolade began exploring options to port some of their PC game titles to the console. However, at the time, Sega had a licencing agreement in place for third-party developers, which increased development costs and required Sega to act as the exclusive publisher for any licenced games. This would have prevented Accolade from releasing its games on competing systems. To circumvent the licensing restrictions, Accolade chose an alternative approach. They purchased a Genesis console and reverse engineered it by decompiling the executable code of three Genesis games. This allowed them to program their new cartridges in a way that disabled the security lockouts, preventing the play of unlicensed games. Their efforts led to the successful release of Ishido: The Way of Stones on the Genesis in 1990.

In response to growing concerns over piracy and unlicensed development, Sega incorporated a new technical protection mechanism into a revised edition of the Genesis console released later that year. This mechanism, known as the Trademark Security System (TMSS), checked the presence of the string "SEGS" at a specific location in the cartridge's memory. If the string was detected, the console would allow the game to run and briefly display the message: "Produced by or under license from Sega Enterprises LTD." The TMSS served two purposes, it added an additional barrier against unlicensed software, and it insured that Sega's trademark was displayed, laying the groundwork for legal action based on trademark infringement if necessary.

On October 31, 1991, Sega filed a suit against Accolade in the United States District Court for the Northern District of California, alleging trademark infringement and unfair competition in violation of the Lanham Act. A month later, copyright infringement under the Copyright Act of 1976 was added to the charges. In response, Accolade filed a counterclaim, arguing that the TMSS falsely represented that their games were produced or licenced by Sega, and asserting that their use of Sega's material constituted fair use.

The court initially did not accept Accolade's fair-use argument, noting that Accolade was a commercial competitor whose works directly competed with Sega's, potentially resulting in a loss of sales for Sega. Accolade's defence was further undermined by a Sega engineer's demonstration of two Sega cartridges that were able to bypass the TMSS, offering them to the defence team, but without revealing how the bypass worked. Consequently, the court ruled in favour of Sega and issued an injunction prohibiting Accolade from selling Genesis-compatible games.

Accolade appealed, and the Computer & Communication Industry Association filed an amicus curiae brief, arguing that the district court had erred by failing to consider whether Accolade's games were substantially similar to Sega's copyrighted works. Upon review, the appellate court reversed the preliminary injection, ruling that Accolade's decompilation of Sega's software constituted fair use. The court found that the use was non-exploitative, even though it was undertaken for commercial purposes.

2.5.2 The Digital Millennium Copyright Act

The Digital Millennium Copyright Act (DMCA), passed in 1998, is a United States (US) copyright law, primarily applicable to businesses operating in the US, that addresses copyright issues related to the internet and digital media. It protects copyrighted works online, particularly by giving internet service providers a safe harbour from liability if they remove infringing content after receiving notice. The DMCA also criminalises the creation and distribution of tools or technologies designed to bypass access controls on copyrighted works.

The DMCA has major effects on reverse engineering, especially through its anti-circumvention provisions. Section 1201 makes it illegal to circumvent technological protection measures (TPMs), such as encryption, DRM, or other security mechanisms, even if no actual copyright infringement occurs. This means that even attempting to bypass protections to simply "look inside" a system can be illegal, regardless of intent.

However, there are some important exceptions that make reverse engineering still possible under the DMCA.

Interoperability Exception

If reverse engineering is performed solely to achieve interoperability between programs or systems, for example, to make a third-party program compatible with existing software, Section 1201(f) allows it. This was a critical issue in cases like *Sega v. Accolade*, although that case predated the DMCA, it greatly influenced the way the law was written.

Security Research Exemption

Starting in 2016, the US Copyright Office granted an exemption allowing good-faith security research. This means that researchers can legally circumvent TPMs if they are conducting genuine security testing, following responsible disclosure practices, and not harming users or the integrity of the system.

Despite these exceptions, the DMCA creates several challenges for reverse engineering. It can hinder legitimate research, as researchers may fear being sued even when acting

ethically. Companies can also use TPMs to "lock down" products, not necessarily to protect copyright, but to prevent competition, for example by forcing consumers to use authorised parts or services. As a result, independent security testing becomes riskier unless conducted with extreme care.

2.5.2.1 DMCA Cases

To better understand how the DMCA has been applied in real-world contexts, it is instructive to examine landmark court cases that have tested its boundaries. These cases highlight how companies have leveraged the DMCA to protect their products, and how courts have responded when reverse engineering was used to challenge restrictive practices.

Lexmark v. Static Control Components (2004)

Lexmark International, a printer and printer cartridge manufacturer, initiated legal action against Static Control Components (SCC) in a dispute over the reverse engineering of Lexmark's toner cartridges.

Lexmark used a "lockout chip" in its printer cartridges to prevent refilled cartridges from being used on its printers by third parties. This chip worked with Lexmark printers to verify the authenticity of the cartridges before printing. Lexmark offered specific printer cartridges at a discounted price (up to 20% off) to customers who agreed to "use the cartridge only once and return it exclusively to Lexmark for remanufacturing or recycling" [126]. The terms were disclosed to consumers through notices printed on the toner cartridge packaging, which stated that opening the package would constitute acceptance of the terms, a practice commonly known as "shrinkwrap licencing".

Static Control Components reverse engineered Lexmark's lockout chip and created chips that allowed third-party manufacturers to produce cartridges compatible with Lexmark printers. This chip would duplicate the "handshake" used by the Lexmark chip circumventing Lexmark's authentication system, allowing consumers to use more affordable non-Lexmark cartridges.

Lexmark sued SCC claiming that SCC had "violated copyright law by copying the Toner Loading Program, and violated the DMCA by selling products that bypassed the encrypted authentication sequence between the Lexmark cartridge chip and the printer". The court concluded that SCC's actions fell under the doctrine of fair use, as the copying was limited, transformative, and was done for the purpose of achieving interoperability, which serves a legitimate public interest.

The ruling affirmed that reverse engineering for interoperability is protected, even if it undermines a manufacturer's business strategy. This was a victory for third-party developers and consumers as it promoted competition in markets dominated by proprietary systems. The case clarified that the DMCA anti-circumvention provisions do not extend to technological measures designed primarily to enforce business models rather than to protect access to copyrighted works. The decision reinforced that reverse engineering for legitimate purposes, such as achieving compatibility, can qualify as fair use under copyright law.

Vivendi Universal v. BNETD (2005)

Blizzard Entertainment, a Vivendi subsidiary, released the online game 'StarCraft' on March 31, 1998 which required the multiplayer gaming service Battle.net to enable online multiplayer functionality, the Battle.net service required players to have provided a valid and unique CD key (a code packaged in the physical retail box of the game) in order to be permitted access to the servers. On 28 April 1998 Mark Baysinger, a student at UC San Diago, released a third-party server emulator for the Battle.net service, named 'StarHack' [127]. Soon after Baysinger received a cease and desist letter from the Software Publishers Association (now known as the Software & Information Industry Association) [128]. Baysinger refused to stop the project and no further action was taken [129]. Baysinger later abandoned the project, but as it had been released under the GPL licence, the project continued as the 'bnetd' project [130].

The Internet Service Provider (ISP) hosting the bnetd project, Internet Gateway, Inc., received a cease and desist from Vivendi. Vivendi then sued, claiming that the programmers who wrote bnetd violated the anticircumvention provisions of the Digital Millenium Copyright Act (DMCA) and that the programmers also violated several parts of Blizzard's End User Licence Agreements (EULAs), including a section on reverse engineering.

The court found that the ISP and programmers behind the bnetd project were bound to the terms of Blizzard's EULAs and Battle.net's Terms of Use, and that by reverse engineering Blizzard software, creating servers that emulated Battle.net, and providing matchmaking services for users of Blizzard software, they were in violation of those terms.

The judge also ruled that because the bnetd servers were used to bypass Blizzard's authentication system and therefore it's antipiracy technology, "the defendants' actions constitute a circumvention of copyright under the DMCA" [131].

This set precedence and highlighted how reverse engineering must still respect Digital Rights Management (DRM) systems.

Sony Computer Entertainment America, Inc. v. Hotz (2011)

On 3 January, George Hotz (geohot), a hacker previously involved in the effort to jailbreak the iPhone [132] [133], released the private key to the PlayStation 3 [134], using techniques described by the group fail0verflow at the 2010 Chaos Communication Congress [135].

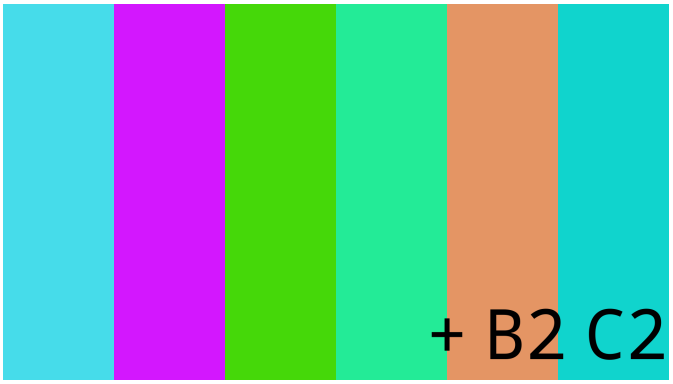


FIGURE 2.6: PlayStation 3 variant of the Free Speech Flag [7]

Sony sued Hotz and members of the hacking group fail0verflow, Hector Martin, and Sven Peter on eight claims, including violation of the Digital Millenium Copyright Act, computer fraud, and copyright infringement on 11 January 2011.

An out-of-court settlement was reached between Sony and Hotz. The agreement included Hotz being "permanently enjoined and restrained from" activies such as "engaging in any unauthorised access to any Sony product under the law" and violating any SCEA licence agreement or terms of use agreement "whether or not Hotz has accepted such agreement or terms of use." [136].

Although there was no court ruling in this case, it highlighted the DMCA's strong protections against bypassing Digital Rights Management (DRM) even for personal use or research.

2.5.3 Licence Agreement Considerations

Licence agreements are legal contracts that define the terms under which a user may access software, hardware, or other digital services. When a product is purchased or accessed, the user typically does not actually "own" the software contained within, instead, they are licenced to use it under specified conditions. These agreements often include clauses that explicitly restrict reverse engineering activities.

The two main types of licence agreements relevant to reverse engineering are:

1. **End User Licence Agreements (EULAs)**: These are the standard agreements that users accept when installing software, accessing a service, or occasionally when using hardware. EULAs frequently contain clauses prohibiting reverse engineering, decompiling, or disassembling the software.

2. **Terms of Service (ToS)**: More common for online platforms and services, ToS agreements similarly often prohibit reverse engineering or analysing the internal operations of a system.

Under contract law, even if copyright statuses such as the DMCA or equivalent legislation allow reverse engineering in certain circumstances, for example, to achieve interoperability or conduct security research, breaching a licence agreement may still constitute a violation of contract. Companies may pursue legal action for breach of contract even where reverse engineering would otherwise be lawful under copyright exceptions. Whether such claims are successful depends heavily on the specific wording of the agreement and the legal jurisdiction involved.

In some jurisdictions, such as the United States, courts may invalidated restrictive license clauses if they conflict with important public policy goals, such as encouraging interoperability or facilitating security research. In other, particularly within the European Union, reverse engineering for purposes of interoperability is protected by law, even where a license agreement attempts to prohibit it [137].

Chapter 3

Near Field Communication (NFC)

This chapter begins with an introduction to NFC authentication procedures and their implementation in Section 3.1. Section 3.2 presents a method for modelling these systems, providing a structured approach to understanding their authentication mechanisms. In Section 3.3, we investigate hardware and software tools related to NFC and define the requirements for a tool to assist in the modelling process. In Section 3.4, we introduce the concept of developing a tool to support the modelling process, outlining key challenges, requirements, and specifications. Finally, in Section 3.5, we examine existing systems and assess their compliance with the protocol specifications.

3.1 Introduction to NFC Access Control

Electronic access systems are widely deployed across institutions and businesses, marketed as secure and traceable entry solutions that allow operators to grant or revoke access with ease. A common implementation of such systems relies on NFC cards, predominantly designed and manufactured by NXP Semiconductors. One of the most widely produced cards in this category is the Mifare Classic [90], introduced in the 1990s. This card has been extensively used in public transit systems, including Transport for London's Oyster card and the Netherlands' OV-chipkaart. However, as demonstrated in previous sections, the Mifare Classic has been proven to be cryptographically insecure. While many large-scale systems have since migrated to more secure alternatives, smaller organisations, such as the University of Southampton's door access ID card readers, have yet to adopt these improved solutions.

Reverse engineering plays a crucial role in understanding the inner workings of electronic access control systems. By analysing how these systems interact with NFC cards at both

the hardware and protocol levels, we can uncover implementation details that may not be publicly documented. This process allows us to verify whether a given system adheres to its intended security protocols or if it deviates in ways that introduce vulnerabilities. In the context of NFC authentication, reverse engineering enables us to assess whether systems currently implement encryption, authentication, and access control measures, providing a means to evaluate their compliance with established security standards.

Many academic articles focus on vulnerabilities within the NFC card protocols themselves, such as MIFARE Classic [49] and Mifare DESfire [5], with the primary goal of addressing and patching cryptographic weaknesses. However, much less attention has been paid to the quality of implementation of these protocols. A protocol may be secure in theory, but its practical implementation may fail to properly enforce the defined standards.

This chapter focuses on identifying and measuring compliance with NFC protocols, rather than discovering new cryptographic flaws. To achieve this, we adopt a reverse engineering approach, not to recover the exact source code of an implementation but to construct a model of how the system behaves. By comparing this behaviour model with the protocol specification, we can evaluate how closely a given implementation adheres to the intended protocol logic.

To identify test cases, we examined several commercially available electronic access systems that represent current industry standards. We used the University of Southampton ID card system due to its accessibility and diverse range of applications, which are explored in more detail in Section 3.5.5.

It is important to note that attempting to extract data from NFC cards, particularly using attacks such as MFOC (MIFARE Classic Offline Cracker) or MFCUK (MIFARE Classic Universal toolKit), may breach the terms and conditions of use set by the service provider. In conducting this research, care was taken to ensure that no terms and conditions of service were breached. All reverse engineering activities were carried out on expired, test, or otherwise non-operational NFC cards obtained legally. No live systems were disrupted, and no unauthorised access to operational services was attempted. Furthermore, the University of Southampton's security department was informed about the nature of the testing conducted, ensuring that the research was carried out with appropriate oversight. The objective was solely to study the implementation practices and protocol compliance, not to exploit or interfere with live infrastructure.

3.2 Modelling NFC Systems

To determine the actual security provided by an implementation, such as the University of Southampton ID access card system, we need to construct a model that accurately

represents its behaviour. This model should encapsulate key components, including the card reader, the card itself, and the communication protocols that govern their interaction. By defining actuation points, expected responses to commands, stored data, and the logic that dictates access control, we can simulate the security policies enforced by the system.

Ideally, a white-box approach, where we have full access to the underlying code and complete visibility into the stored card data, would allow for a thorough assessment. However, in many cases, this level of access is not available. Instead, we may need to gather information empirically through black-box testing, where interactions between the card and the reader are monitored without prior knowledge of the internal workings. Various tools can be used to intercept and analyse these communications, allowing us to infer security mechanisms, identify weaknesses, and refine our model accordingly.

Actuation Points

Actuation points are observable outputs or signals within a system that indicate the outcome of a process or decision. In the context of access control systems, actuation points are critical as they reveal how the system enforces security policies and grants or denies access. These points can take various forms, such as visual indicators, log entries, or physical actions triggered by systems.

For example, in the University of Southampton's ID card system, an actuation point is the status message displayed on the reader, which either reads "OPEN" when access is granted or "ACCESS DENIED" when it is refused. Alternatively, an actuation point could be the electrical signal that activates the magnetic door lock, physically allowing entry. By identifying and analysing these actuation points, we can better understand how the system makes access control decisions and verify whether they align with the expected protocol.

Components

When reverse engineering an NFC system, we examine how individual components interact with their environment, in this case, how an NFC card communicates with an NFC reader. The information gathered about the card can be used to create new, synthesised card instances, effectively allowing us to flip the focus of the reverse engineering process onto the reader itself. The ability to generate custom card data enabled us to test a variety of scenarios which are likely to reveal the underlying logic and decision-making processes within the reader's implementation.

Communications

There is a mutual relationship between the card and the reader, as the output of one serves as the input for the other and vice versa. To accurately model this interaction, we need to construct separate models for both the card and the reader, capturing their individual behaviours and responses.

To build these models, we can simulate an NFC card and record the reader's response. Then, using this recorded response, we can simulate an NFC reader and replay the interaction to observe how the card reacts. By iterating this process in a ping-pong fashion, continuously exchanging and logging responses, we can generate a complete trace of the communications. This allows us to systematically reconstruct both the card and the reader's behaviour, ultimately providing a detailed and functional model of their interactions.

Card Data

The data stored on the cards are a crucial component of the mode, as they directly influence the behaviour of the protocol. Accessing these data typically requires knowledge of the sector keys, which determine the ability to read or write information on the card. In a white-box scenario, these keys would be readily available. However, in a black-box scenario, alternative methods must be used to extract this information.

Legacy NFC systems that rely on older cards, such as the MIFARE Classic, are known to be vulnerable to various attacks. One such attack, implemented in the MIFARE Classic Offline Cracker (MFOC), is the Offline Nested attack. This attack allows an adversary to recover sector keys when at least one key is known. Many real-world implementations leave unused card sectors protected only by publicly known default keys, making the attack even more feasible. By exploiting these weaknesses, an attacker or researcher performing protocol analysis can retrieve a full dump of the data stored on the card, providing valuable information on how the system functions [50].

3.3 Modelling Tool

Gathering the data required to build a model of an NFC card and reader can be both difficult and slow, due to the low-level nature of the communication and the limited access typically available. Creating a tool that simplifies the process of collecting data and analysing traces would be invaluable. Such a tool would allow us to more easily observe interactions, identify protocol behaviours, and ultimately form an accurate model of both the card and the reader. The following section outlines the analysis and

requirements for such a tool, highlighting the key functionalities necessary to facilitate efficient data collection, protocol analysis, and model development.

3.3.1 Challenges

Reducing the NFC Testing Space

When testing the card and reader implementation, it is essential to observe the system's numerous outcomes based on different card inputs. The potential input space is vast due to the flexible nature of data that a card can hold. For example, the Universal Identifier (UID) of the card is a key input variable that is unique to each card. Additionally, cards like the MIFARE Classic can contain varying keys for different sectors and the actual data stored in those sectors can differ greatly between cards.

To make this testing space more manageable, we must identify exactly which parts of the card's data the reader accesses during authentication. For example, a reader might begin by reading the UID, then attempt to authenticate a specific sector using a predefined key, and finally read a particular block within that sector. By narrowing down the protocol's behaviour in this way, we can focus our testing on relevant variables, namely: variations in the UID, the keys used for sector access, and the content of specific data blocks. This targeted approach enables more efficient and meaningful testing and reduces unnecessary permutations. The method by which access keys are defined and used within the system is explored further in Section 3.5.4.

Emulating Cards

One significant challenge in testing NFC systems is the need to change the UID of a card. Standard MIFARE Classic cards do not allow the writing to Block 0, which stores the UID, making this difficult. However, certain variants of MIFARE Classic cards, such as block 0 writeable cards, do permit writing to Block 0 either by default or through undocumented functionality. In the case of "backdoored" cards, it is possible to unlock write access to Block 0 by sending a specific hexadecimal sequence, as shown in Figure 3.1, after which the UID can be modified.

Step	Sender	Hex	Abstract
1	Reader	50 00 57 cd	HLTA + CRC
2	Reader	40	7 bits only
3	Tag	A	4 bits only
4	Reader	43	
5	Tag	A	select(4 bits only)

FIGURE 3.1: Common sequence to allow writing to Block 0 in backdoored cards

Once write access to Block 0 is obtained, these special cards can be used to emulate different UIDs. However, exhaustively testing a door access system with every possible UID would involve up to 65,535 combinations, making manual UID writing highly inefficient. Furthermore, modifying and writing data across various sectors for each test case would further increase the time and complexity. To streamline this process, we can emulate an NFC device and programmatically alter the UID and card data, significantly speeding up testing. A device capable of such an emulation is described in [8].

Different NFC implementations can vary significantly in terms of UID length, proprietary protocols, cryptographic mechanisms, and other design aspects. As a result, any tool developed to model or test these systems must be flexible enough to accommodate a wide range of protocols. For example, while MIFARE Classic uses its own proprietary structure, MIFARE DESfire cards support compatibility with the ISO/IEC 7816-4 [91] command set, introducing a different communication protocol that the tool must be able to handle.

Some NFC systems may involve complex back-end logic that significantly influences the flow of interactions. To conduct effective testing, it would be useful to mock these back-ends in order to simulate their decision-making process. While many implementations rely on relatively simple logic, such as checking UID against an access list, larger and more sophisticated systems, such as Transport for London's Oyster card infrastructure, can involve intricate validation rules and data handling. Being able to replicate or emulate such back-end logic would greatly enhance the accuracy and usefulness of our testing framework.

3.3.2 Requirements

The requirements for the tool are listed below:

1. The ability to gather traces of the system using the ping-pong fashion mentioned in 3.2.
2. Identify the operations performed on that trace in conjunction with the specification for the type of card used in that system, such as [90].
3. Identify the parts of the communications that are compliant with the specification.

3.3.3 Hardware

The Chameleon is a freely programmable device capable of emulating contactless smart cards that comply with the ISO 14443 standard [90]. It is powered by an ATxmega

microcontroller, which includes hardware acceleration for both DES and AES-128 encryption. It also, it offers an optimised implementation of Crypto1 written in AVR assembly. These features enable the device to emulate a wide range of cards, including MIFARE Classic, MIFARE DESFire, and MIFARE DESFire EV1 [8].

FIGURE 3.2: ChameleonMini RevG a later revision of the device conceived in [8]

The Chameleon can be used to emulate an access card within a system and generate interaction traces with an NFC reader. These traces can then be analysed and compared against the specifications of the corresponding NFC protocol to identify potential vulnerabilities in the implementation of the reader's access control logic. By simulating different conditions and variables defined in the protocol, the Chameleon can be used systematically to verify the reader's behaviour and test for compliance. Crucially, the Chameleon allows us to have a virtual, programmable card that can be dynamically updated, enabling us to easily instantiate new card configurations directly from our model. This makes it an invaluable tool for exploring how the system responds to a wide range of test cases.

Another essential component of our toolchain is the SCL3711 by Identivm [138]. This compact USB-powered device is designed to interface with a wide range of smart cards and NFC-enabled devices, supporting the ISO14443 Type A and B standards. It also offers native compatibility with several common card types, including MIFARE Classic 1K and 4K, MIFARE Plus, DESFire, and Ultralight. Integrated with the open-source library libnfc (see Section 3.3.4), the SCL3711 allows us to emulate card reader functionality within our tool. While devices like the Chameleon enable us to simulate programmable cards, the SCL3711 complements this by giving us the ability to simulate the reader side of the communication, making it a crucial part of our efforts to capture, replay, and analyse interactions from both ends of the NFC protocol.

3.3.4 Software

To interface our modelling tool with various NFC devices, we make use of libnfc, a low-level NFC software development kit (SDK), and a programming API written in C. This library enables us to write programs that can interact directly with NFC cards

and readers, allowing reading, writing, emulation, and trace collection. Libnfc supports all major operating systems and is compatible with a broad range of NFC hardware, including the SCL3711. It includes open-source implementations for communication based on the ISO/IEC 14443 standard [90] and supports proprietary protocols used by various types of MIFARE cards. Libnfc is actively maintained by the nfc-tools project, which also offers companion tools such as mfoc and mfcuk, widely used for analysing and attacking legacy MIFARE Classic systems. Integrating libnfc is essential for building flexible and extensible NFC tooling that can drive our modelling and reverse engineering workflows from both the card and the reader perspectives.

We will make use of these tools later in our workflow to extract data dumps from NFC access cards for further analysis. Where permitted by the terms and conditions of the service, we can apply known attacks, such as mfoc (which performs the offline nested attack) and mfcuk (the MIFARE Classic DarkSide Key Recovery Tool), to recover the contents of a MIFARE Classic card. The output of these tools gives us a full image of the card's memory, including sector keys and stored data. These data can then be loaded into the Chameleon device to emulate a valid access card and perform further inspection of the target system. These outputs are discussed in detail in Section 3.5.3.

This process provides us with the following data: the card's Universal Identifier (UID), the complete contents of each sector, the access keys A and B for each sector, and the access permissions associated with those keys. With this information, we can use libnfc to write scripts that instruct the NFC device to behave as if it were a legitimate card. This enables us to generate detailed communication traces between the emulated card and an NFC reader, which can then be analysed to understand system behaviour and verify protocol compliance.

3.4 New Tool

In this section, we present the design and implementation of a custom tool developed to assist in the modelling and verification of NFC-based access control systems. The tool enables the automated collection of communication traces and assists with the assessment of protocol compliance. It also supports the demonstration of known NFC attacks, helping us to expose implementation flaws in real-world systems.

3.4.1 Specification

In order to develop a tool that meets the objectives of modelling NFC systems and verifying protocol compliance, we define the following functional specification.

The tool allows us to:

1. Load in traces of NFC communications collected from the devices and parse the messages into defined structures.
2. Visually see a representation of the data using a graphical user interface, making it easier to analyse the message exchanges between the card and the reader.
3. Evaluate the responses and communication flow of the emulated version against the actual responses in the given trace.
4. Use the data gathered to build a model of the implementation.
5. Define back-end logic in a simple way for further analysis.
6. Expand functionality to implement additional NFC implementations such as MIFARE DESfire in the future.

The tool we created to provide this functionality is presented below.

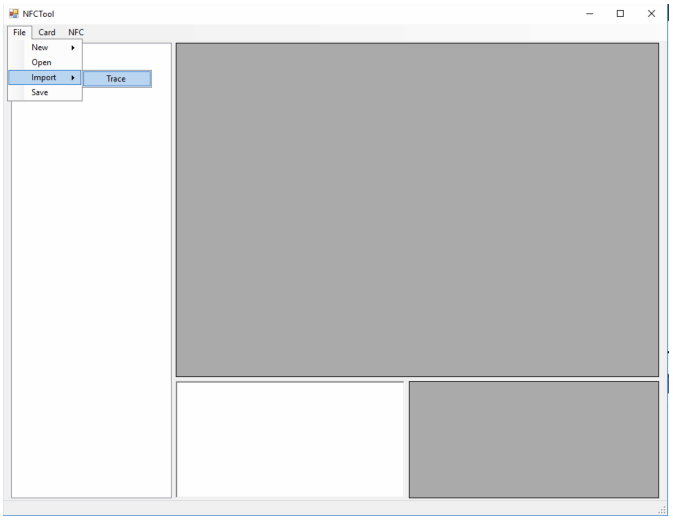


FIGURE 3.3: First screen of the NFC Tool

Importing a trace gathered from a modified version of libnfc will display all the messages captured in that trace. We can then run the trace through our software emulation of the reader to verify its compliance with the specification. If we have a card dump obtained through MFOC, MFCUK, or a similar tool, we can also verify the card's response to the reader's commands and modify the state of the virtual card accordingly.

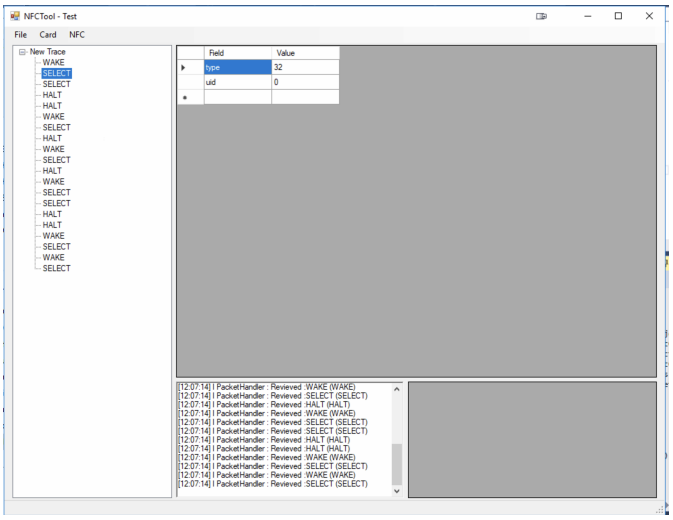


FIGURE 3.4: Imported trace ran through parser and simulator

The values of the data in the message can be modified here and then the simulation can be run again to observe the effect of these changes on the protocol. For instance, if a value is altered and the parity bit becomes incorrect, the simulator will detect the issue and flag it.

One of the key components of this tool is the software emulator, which can emulate either a card, a reader, or both. Currently, it supports the start of most NFC protocols [90] and can emulate a MIFARE Classic card along with the corresponding commands in the protocol.

The messages in a trace are grouped, allowing multiple traces to be viewed and compared simultaneously. The tool can be instructed to run a trace through the emulator for analysis.

Protocols such as Mifare Classic and DESFire can be defined in plugins, making the tool easily extendable to support additional protocols in the future. Each class will specify how the data is stored on the card and expose the commands available to readers.

```
[NFCCard(ATQA.MifareClassic1K, SAK.MifareClassic1K)]
3 references | Max Hayman, 115 days ago | 1 author, 1 change
public class MifareClassic1K : NFCCard
{
    6 references | Max Hayman, 115 days ago | 1 author, 1 change
    public class MifareClassicSector
    {
        public MifareClassicBlock[] blocks = new MifareClassicBlock[4];

        1 reference | Max Hayman, 115 days ago | 1 author, 1 change
        public static MifareClassicSector Load(MemoryStream data)
        {
            MifareClassicSector sector = new MifareClassicSector();

            sector.blocks[0] = MifareClassicTrailer.Load(data);
            sector.blocks[1] = MifareClassicBlock.Load(data);
            sector.blocks[2] = MifareClassicBlock.Load(data);
            sector.blocks[3] = MifareClassicBlock.Load(data);
            return sector;
        }
    }

    4 references | Max Hayman, 115 days ago | 1 author, 1 change
    public class MifareClassicTrailer : MifareClassicBlock
    {
        uint uid;
        1 reference | Max Hayman, 115 days ago | 1 author, 1 change
        public static MifareClassicTrailer Load(MemoryStream data)
        {
            MifareClassicTrailer block = new MifareClassicTrailer();
            data.Read(block.data, 0, 16);
            return block;
        }
    }
```

FIGURE 3.5: Mifare Classic Plugin Code

3.4.2 Reader Logic Scripting

NFC card readers can perform a wide variety of actions, from reading the UID to storing data such as a card's balance, trip information, and cyclic redundancy checks, as seen in the Oyster card system by TfL. Defining this logic enables us to verify that the same actions are executed when analysing a trace captured from a real system.

This logic will need to be defined on a per-application basis. To facilitate rapid prototyping and testing of the application logic, a scripting system has been implemented. LUA has been selected as the scripting language to define this logic.

The functions defined in the LUA scripts will be triggered in response to specific events, such as when the UID of a card is presented. These functions can then control the flow of the application logic, ensuring accurate simulation and testing.

```
function ProcessCard(x)
  if(ValidUUID(x.UUID))
    OpenDoor();
  else
    SendDisplay("Unauthorised Access");
  end
end

function ValidUUID(x)
  if(x == 012345)
    return true;
  else
    return false;
end
```

FIGURE 3.6: Example of a simplified version of door access logic used in the Yale Smart Lock

In 3.6 we can identify the actuation point being the result from the ValidUUID function as that determines if the OpenDoor() function is called. In this example we can see that AuthoriseBlock is not called so no encrypted communications take place.

In Figure 3.6, the actuation point can be identified as the result of the ValidUUID function, which determines whether the OpenDoor() function is called. In this example, we can observe that AuthoriseBlock is not invoked, indicating that no encrypted communication occured during this interaction.

```
function ProcessCard(x)
  if(AuthoriseBlock(DeriveKey(x.uuid, 1))
    staffId = ReadBlock(1, 5); // Block number,
    length
    if(ValidStaffID(staffId))
      OpenDoor();
    else
      SendDisplay("Invalid or expired staffid");
    end
  else
    SendDisplay("Could not authorise block");
  end
end

function DeriveKey(uuid, sector)
  if(sector == 1)
    return uuid ^ 0x45 | 0x12;
end
```

FIGURE 3.7: An example of a more complicated implementation which accesses data on an encrypted block using a key which is derived from the UUID

However, in Figure 3.7, we can see that the control flow requires both AuthoriseBlock and ValidStaffID to return true for OpenDoor() to be called. Since AuthoriseBlock is invoked, we know that the cryptographic communication was invoked in this interaction.

Using a scripting language to define implementations allows us to model highly complex systems, such as the Transport for London's Oyster card.

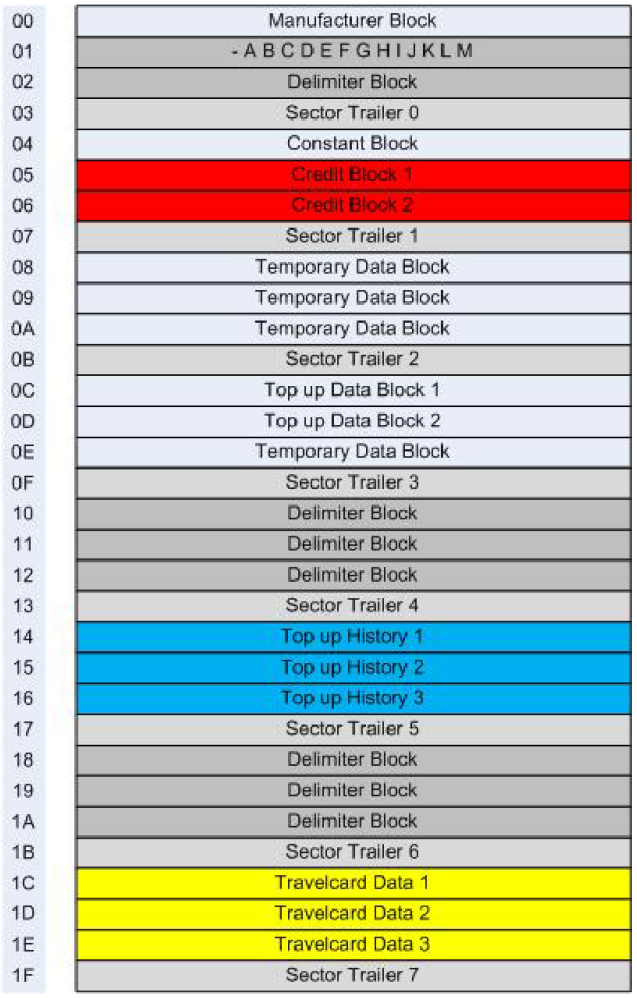


FIGURE 3.8: First eight sectors of Oyster Cards internal structure [3]

The credit balance on an Oyster card is stored across blocks 5 and 6, highlighted in red. These blocks are updated alternately. The first byte in each block acts as a counter, indicating which block holds the most recent credit value. If the counters are equal, block 6 is considered current, otherwise, block 5 is.

Additional blocks store data such as top-up history, travel card information, and temporary transaction records, as described in [3]. This structure suggests that the Oyster card system employs a sophisticated and complex application logic.

```
function GetCredit()
{
  if(AuthoriseSector(2, x.uuid))
  {
    byte[] block5 = GetBlock(5);
    byte[] block6 = GetBlock(6);

    if(block5[0] == block6[0])
    {
      return GetValueFromBlock(block6);
    }
    else
    {
      return GetValueFromBlock(block5);
    }
  }
}
```

FIGURE 3.9: Simplified LUA Pseudo code defining how to read the current balance of an Oyster Card

3.4.3 General Reverse Engineering Support

The newly developed NFC modelling tool directly uses the systematic reverse engineering methodology proposed in this thesis, providing software infrastructure that puts into use the interactive component-environment interaction framework in domain-specific NFC protocol analysis.

The tool's architecture directly maps onto the six step iterative reverse engineering model presented in 5, providing explicit computational support for each phase:

Step 0 (Initial Model Contructuion): The tool supports initial model construction though its plugin-based architecture, which allows researchers to define protocol specifications as structured representations. The MIFARE Classic plugin (Figure 3.9) exemplifies this capability, encoding known protocol elements such as sector structure, authentication commands, and data block organisation as baseline models against which observed implementations can be compared. Researchers can instantiate initial models by loading protocol specifications, card memory dumps, or partial traces, establishing the foundational knowledge from which iterative refinement proceeds.

Step 1 (Model Instantiation): The tool enabled model instantiation through its dual emulation capabilities, allowing researchers to synthesise both card and reader instances from constructed models. Integration with the Chameleon hardware device permits programmatic instantiation of virtual cards with precisely controlled parameters, e.g.

UID values, sector keys, access permissions, and stored data which can be specified programmatically and deployed as testable instances. Similarly, the SCL3711 reader integration, combined with the libnfc bindings, enables instantiation of virtual readers that execute specific command sequences. This bidirectional instantiation capability is critical for the "ping-pong" trace collection method described in Section 3.2, where alternating between card and reader emulation reveals complete protocol behaviour.

Step 2 (Testing and Trace Generation): Systematic trace generation constitutes a core capability of the tool. Modified libnfc libraries capture low-level ISO 14443 communication with accurate timing precision, recording complete message exchanges, including raw hexadecimal bytes, parity bits, CRC values, and protocol timing. The tool's trace import functionality (Figure 3.4) parses these raw communications into structured representations, automatically identifying protocol phases (auto-collision, authentication, data transfer), command types (WUPA, SELECT, AUTH, READ, WRITE), and data payloads. This automated parsing transforms unstructured byte sequences into semantically meaningful protocol events, dramatically reducing the manual analysis overhead.

Step 3 (Knowledge Synthesis): The tool facilitates knowledge synthesis through its integrated simulation engine, which executes protocol specifications alongside observed traces to identify behavioural deviations. When a captured trace is loaded (Figure3.4), the simulator executes the standardised protocol state machine, comparing expected responses against actual responses at each communication step. Discrepancies such as missing authentication phases, unexpected command sequences, or premature HALT commands are flagged automatically, highlighting implementation non-compliance. This differential analysis directly synthesises new knowledge, rather than requiring manual comparison between specification documents and hex dumps, the tool computationally identifies precisely where implementations deviate from standard, focusing analyst attention on security-relevant anomalies.

Step 4 (Model Revision): The tool supports iterative model refinement through its Lua scripting interface (Section 3.4.2), which provides a high-level declarative language for encoding implementation-specific logic. As new knowledge is synthesised through trace analysis, researchers update Lua scripts to reflect discovered behaviours. For example, upon discovering that the Yale Smart Lock performs only UID-based authentication without invoking `AuthoriseBlock` (Figure 3.6), this simplified logic is encoded in the model. Subsequently discovering that the University of Southampton system uses UID-derived keys (Section 3.5.4) prompts model revision to incorporate the key derivation function (Table B.8).

Step 5 (Iteration Until Complete): The tool facilitates iteration through its integrated workflow that connects trace capture, analysis, model revision, and re-testing in a tight feedback loop.

In summary, the NFC modelling tool operationalises the systematic reverse engineering methodology through concrete software infrastructure that automates trace collection, enables controlled testing across vast paramater spaces, computationally verifies protocol compliance, supports iterative model refinement through scripting, and provides visual analysis interfaces that focuses on knowledge synthesis rather than low-level tool manipulation.

3.5 Existing NFC Systems

In this section, we examine real-world NFC implementations using specific reverse engineering systems. We analyse how they operate, identify potential vulnerabilities, and assess how closely they adhere to expected standards. In addition, we evaluate several recent consumer-grade smart locks, which are often marketed as high-security solutions, to determine the actual quality and robustness of their NFC-based access control mechanisms.

3.5.1 Yale Keyless Connected Ready Smart Lock

In 2015, Yale introduced a smart lock designed for integration with home automation systems, using either a proprietary Yale module or a Z-Wave module for communication. Marketed as a keyless entry solution, the lock supports multiple authentication mechanisms, including remote access through modules, PIN code entry, and NFC-enabled cards or fobs. Yale promotes the device as featuring "Enhanced Encrypted Security" to prevent unauthorised access.

Using the Mifare Classic Offline Cracker tool, as referenced in Section 2.3.1, we extracted the data stored on the card and fob provided with the Yale smart lock. The resulting memory dumps, available in the Appendix A, reveal that while the card and fob have distinct UIDs, the access keys for Sectors 0-6 differ. Additionally, the remaining sectors appear to be unused, as they rely on default keys and do not contain meaningful data. In particular, the protected sectors (1-6) are entirely filled with zeros. Sector 1, however, shows a consistent data structure across different cards, with the primary variation occurring in the second block.

This chapter focuses on NFC-related protocol vulnerabilities, specifically analysing the security of NFC-based access mechanisms. After setting up the Yale smart lock and performing a similar assessment, we identified that the card provided with the device is a Mifare Classic 1K card, an insecure standard, as previously discussed. Furthermore, our analysis revealed that the lock relies solely on the card's UID for authentication. This presents a significant security risk, as the UIDs are not encrypted or secret. As a

result, an attacker could wirelessly clone a valid card in less than a second, effectively bypassing the lock's security and gaining unauthorised access to the property.

FIGURE 3.10: Chameleon device unlocking Yale Smart Lock

This represents a significant security flaw, as a well-established and reputable lock manufacturer should not be distributing a product that relies on a widely known insecure card while also failing to implement proper security protocols. Additionally, our findings indicate that the card signal can be intercepted even while it remains inside the packaging of the lock. This means that an attacker could potentially clone the card without ever being near the property where the lock is installed, further compromising the security of the system.

3.5.2 University of Southampton ID Card

The University of Southampton ID card system is based on the Mifare Classic 1K card, which uses the proprietary Crypto1 encryption protocol. However, this protocol is widely recognised as insecure [9]. By analysing interactions between ID cards and door readers at the university, we observed that communication ceases immediately after the NFC UID is broadcast, before the Mifare authentication protocol even begins. This indicates that the system relies solely on the UID for access control and that all transmitted data remains unencrypted, making it highly vulnerable to cloning and unauthorised access.

The initial communication between the ID card and the reader follows the ISO/IEC 14443-1 Type A standard, which defined the NFC protocol that the Mifare Classic cards comply with. After this initial handshake, the card transitions to using the NXP's proprietary Crypto1 protocol for authentication and encryption. However, as previously

discussed, Crypto1 has been extensively broken, making systems relying on it vulnerable to security breaches.

Step	Sender	Hex	Abstract
01	Reader	26	req type A
02	Tag	04 00	answer req
03	Reader	93 20	select
04	Tag	c2 a8 2d f4 b3	uid,bcc
05	Reader	93 70 c2 a8 2d f4 b3 ba a3	select(uid)
06	Tag	08 b6 dd	MIFARE 1k
07	Reader	60 30 76 4a	auth(block 30)
08	Tag	42 97 c0 a4	n_T
09	Reader	7d db 9b 83 67 eb 5d 83	$n_R \oplus \mathsf{ks}_1, a_R \oplus \mathsf{ks}_2$
10	Tag	8b d4 10 08	$a_T \oplus \mathsf{ks}_3$

FIGURE 3.11: Authentication Trace [9]

According to the manufacturer, when the tag enters the electromagnetic field of the reader: it powers up, identifies itself, and starts the anti-collision protocol by sending its UID, shown in Step 1-6 of Figure 3.11. The reader then sends an authentication request for a specific block, which then begins an authentication trace with the reader in Steps 7-10.

However, looking at a trace with a University of Southampton card and reader, you would see that the authentication trace would not proceed further than Step 6 and instead send a halt command.

Although there is hardware support for cryptographic communications to access the information on the card and prove that it is a legitimate card, many of these systems do not implement these cryptographic communications at all. In fact, the assumed access policy of the systems can differ widely from the actual implemented policy.

3.5.3 Data stored on the cards

A common starting point for reverse engineering an NFC system is to extract the data stored on the card. Inspecting the contents can reveal what kind of information the system stores, such as access rights, identifiers, or credit values, which in turn can uncover potential vulnerabilities or avenues for fraud. For instance, if access levels or balances are stored in plaintext or with weak protection, an attacker may be able to modify them to gain unauthorised access or increased privileges.

In practice, these attacks appear ineffective on newer University of Southampton ID cards, possibly because they are MIFARE Plus cards operating in backward compatibility mode with stronger security mechanisms. However, older or expired cards, which may still be accessible, are often vulnerable and can serve as valuable input for constructing an initial model of the system's behaviour.

By analysing data recovered from these cards, we can begin building a working model of the access control protocol and identify how it handles identity, permissions, and other critical logic.

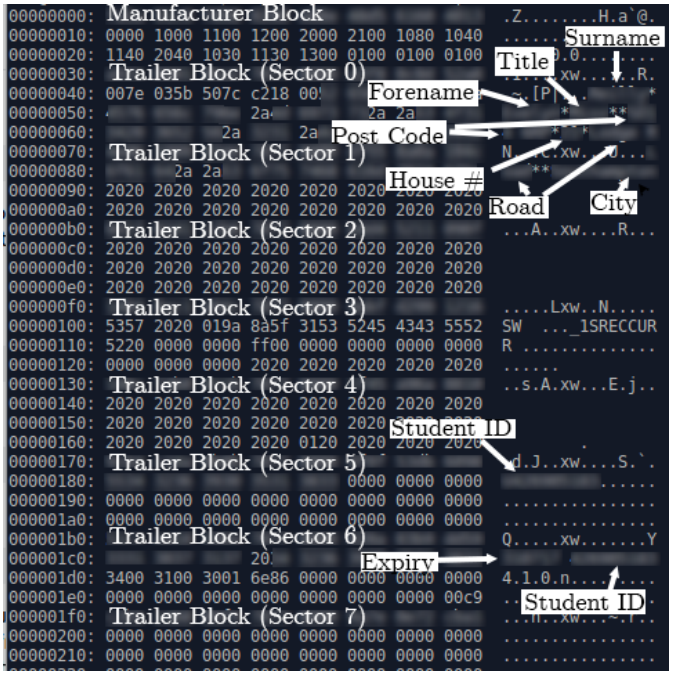


FIGURE 3.12: Dump of University of Southampton ID card

The figure above shows data stored across several sectors of a University of Southampton ID card. Certain fields have been redacted for privacy. Highlighted regions reveal personal information such as the cardholder's name, address, student ID number, and the card's expiry date. This provides insight into the types of data stored on the card and emphasizes the need for secure handling of sensitive information within NFC systems.

3.5.4 Reverse engineering the key function from UID

Some implementations are known to use the same key for the same sectors in all cards in a system [3]. This poses a significant security risk, as obtaining a single sector key would allow an attacker to access that sector on any card in the system. Given that personal information is often stored on these sectors, this vulnerability could lead to serious privacy breaches.

Initial tests using attacks such as MFOC and MFCUK on at least three different University of Southampton cards revealed that while sector keys differ across cards, some bytes within those keys remain constant. Specifically, four out of the six bytes in each key are identical, suggesting that these parts may be hardcoded. It is not yet clear whether these constant bytes are used consistently by the system or are merely remnants of a key derivation system.

One hypothesis is that the access keys are not entirely independent, but are partially derived from the card's UID. This would allow a system to dynamically compute the keys based on the UID rather than having to store all keys individually on the reader side. Such a design could reduce the cost and complexity of the back-end system but may introduce new vulnerabilities if the derivation scheme is predictable or poorly implemented.

This hypothesis guides our initial model and will be further investigated in the subsequent analysis.

Table 3.1 Table showing 3 different Sector 0 A Keys for University of Southampton ID Cards

Key Byte	Card 1	Card 2	Card 3
0	0xA9	0xAB	0x24
1	0x31	0x31	0x31
2	0x00	0x00	0x00
3	0x01	0x01	0x01
4	0xAA	0x86	0xE5
5	0xA2	0xA2	0xA2

To further investigate how the sector keys are generated, an application has been written to perform a brute-force attack on the keys from multiple cards for the same sector. By comparing the keys of different cards, the application can analyse the method used to generate these keys. Specifically, the application takes each byte from the UID of each card and performs a bitwise XOR operation with every possible byte value (ranging from 0x00 to 0xFF). By applying this operation across all three card UIDs and their corresponding sector keys, the application can identify a common pattern or operation that results in the correct key, as demonstrated in Table B.8.

Table 3.2 Calculating Key A for sector 0

Key Byte	A0 Key Formula
0	uid[0] $\oplus$ 0x53
1	0x31
2	0x00
3	0x01
4	uid[3] $\oplus$ 0x52
5	0xA2

Having calculated the function that generates keys based on the UID, we can use a scanner to extract personal information from the card without needing any additional access permissions. Since the UID is easily identifiable through scanning, this allows unauthorised extraction of sensitive data, such as name and address, from the card in under a second. This could pose significant privacy concerns. These functions used to generate keys for sectors 0 through 7 are detailed in the Appendix B.

3.5.5 Different uses of cards around the university

During research on various NFC applications, it was essential to investigate how different NFC readers interact with the same card. This allowed us to create models of different reader types and compare their implementations. The university, for example, uses various readers for different services, each of which appears to operate in slightly different ways. To limit the scope of our analysis, we focused on door readers, but we expect that the vulnerabilities we observed in those readers will be similar across the other types of reader as well.

Door Access is used for almost all university buildings. It appears that the university has lists stored on the back-end systems which restrict access to different buildings.

Table 3.3 Trace from University of Southampton door access reader

Step	Sender	Hex	Abstract
1	Reader	52	WUPA
2	Tag	04 00	answer req
3	Reader	93 20	select
4	Tag	f0 0d 68 70 e8	uid,bcc
5	Reader	93 70 fd 0d 68 70 e8 cc e9	select(uid)
6	Tag	08 b6 dd	Mifare 1k
7	Reader	50 00 57 cd	Halt

The trace above can be compared to the authentication trace in Figure 3.11. The trace is very similar, apart from a different UID. However, we can see that the reader sends a halt command after Step 6 which means it does not authorise using any of the encryption offered by the hardware.

Bike Shed Access uses a different NFC reader, and nothing observable happens during communication with the card if access is denied. However, upon further inspection, the reader uses slightly different commands when interacting with the card. It is assumed that these readers operate under the same policy scheme as the Door Access cards, albeit with different readers. Monitoring the communications on a trace reveals that this system does not seem to be using any MIFARE security features, which could indicate an insecure implementation.

Step	Sender	Hex	Abstract
1	Reader	52	0x01
2	Tag	04 00	0x31
3	Reader	50 00 57 cd	0x00
4	Tag	0x01	0x01
5	Reader	0x01	0x01
6	Tag	0xA2	0xA2

Gym Access seems to use different data on the card, perhaps using data which requires authentication. This system was not investigated further due to limited permissions and to avoid interacting with live operational access systems without explicit authorisation.

Winchester Shuttle Bus has had an ID card scanner on the bus. This system was also not investigated further, as accessing or modifying transport-related systems could present legal and ethical concerns without prior permission.

Printing and Copying at the University is done using ID cards. It appears that they only use the UID to perform user identification and authorisation. No further investigation was conducted to avoid interfering with critical university infrastructure that could impact multiple users.

Chapter 4

Video Games

In this chapter, we use reverse engineering to uncover the inner workings of a video game, analysing its code, data structures, and systems to better understand its design and functionality.

4.1 Archive Files

In this section, we introduce the concept of game archive files, their typical structure, and their significance in the reverse engineering process.

4.1.1 Introduction

It is usually the case that video games are heavily data driven, which allows non-programmers, such as designers and artists, to manipulate the game without needing to write code. These assets can range from data files, in forms such as spreadsheets, to 3D models exported from modelling tools such as 3D Studio Max.

Video games use archive files to store the necessary game assets. In order to be able to analyse these files, we need to be able to access them.

This chapter will focus on providing a review of the literature on the common file formats for video games and the architectural design decisions made by developers and their motives for those decisions. We will also be introducing different techniques to reverse engineer game archive formats in order to be able to identify the structures used and understand how they work, so we can build a model of them. We will then use this to create a tool that will allow us to extract the game data from them.

This research aims to elucidate the importance of extracting game data files from archives and emphasise the potential these files hold in terms of gaining insights into the game's

architecture, data organisation, and the subsequent possibilities this presents for further analysis and modification.

4.1.1.1 Background

Video games use archive files to store the necessary game assets. There are several reasons why game developers may choose to store their game data in this way. For one, archives help in organising and storing assets such as textures, 3D models, sound files, scripts, and more in a structured manner. It allows developers to group these files together, making it easier to manage the content of the game. In addition, these archive files often use compression algorithms to reduce the overall size of game files. Compressed archives save storage space and can also improve loading times by reducing the amount of data that needs to be read from the disc. This was more prominent in games that were stored and read from cartridges or optical media, where storage space was limited, and read times were slow and inconsistent. Accessing data from a single archive is often more efficient than accessing numerous individual files scattered in different locations on a disc. It can also improve loading times and facilitate smoother game play experiences. One such example is Naughty Dog, who organised game files on their optical discs so that the most frequently accessed files were at the start of the optical disc, where seek times between accessing random areas of data were lower, which therefore improved performance.

Archive files can also serve as a form of protection against accidental data manipulation. By bundling files into archives, game developers can make it more difficult for users to modify or access specific game assets, thereby protecting intellectual property and preventing cheating in multiplayer games. Blizzard Entertainment started encrypting specific files such as cut scenes to prevent spoilers from being leaked onto the internet before the content was released. When a user unlocks the cutscene by completing part of the game, the server sends a decryption key to the game client allowing the cutscene to be decrypted, and the player to then watch it. Games often release updates and patches - using archive files can help with the deployment and application of these updates. This is because instead of replacing entire directories or individual files, patches can simply update specific archives or add new ones, simplifying the update process. Archive files are also used to preload future content, allowing players to download content in advance of the release, usually encrypted to prevent leaks, so that on the release day players do not have to wait for the new content to download which can reduce the strain on content servers. Archive files can also help maintain file integrity - by sorting files in a single archive, developers can ensure that all necessary assets are present and speed up the process of calculating checksums to ensure consistency. Developers often use custom proprietary archive file formats as opposed to traditional archiving formats such as PKWARE Zip (.zip) or GNU's gzip (.tar.gz). This allows them to optimise for

their specific use case, for example, fast random read access with compression, where traditional archive formats do not have this requirement.

4.1.2 Technical Background Knowledge

In this section, we are going to explore different archive formats, what features they offer, and the different use cases. We are also going to review existing research into reverse engineering of archive file formats.

4.1.2.1 Archive File Formats

There are an abundance of different archive formats and tools under permissive licences which have different strengths and weaknesses. We are going to investigate and explore the technical details of different archive file formats. We will also explore popular compression algorithms and evaluate their strengths and weaknesses in different scenarios.

PKWARE ZIP PKWARE ZIP (.zip) is an archive format developed by Phil Katz as an open data format with an open standard REF. It stores files and their directory structure where each file can be individually compressed. The files and the directory structure can be optionally encrypted.

tar and gzip The gzip format was developed as a replacement for the Unix compress utility, which used the LZW compression algorithm. At the time, this algorithm was under patent and its free use was in dispute by the patent holders. The gzip utility was designed as a drop-in replacement for compress. gzip only supports compressing single files, therefore it is commonly used with the tar utility which would create an archive of files, their attributes and the directory structure into a single .tar file. When used as such, using gzip in conjunction with tar would create a compressed archive, with the extension .tar.gz. The .tar.gz format compresses far better than .zip since the compression can take advantage of redundancy across all the files within the archive. Unlike .tar, .zip has a central directory at the end, which provides a list of the contents. This, combined with separate compression per file, allows random access to the individual entries in the zip file. A tar file would have to be decompressed and scanned from start to end to build a list of the contents within it. The patent dispute regarding compress also brought into question the free use of the .gif image format, which was very widely used on the early internet. The losslessly compressed PNG image format was created to replace it. The defalte format was also used for compression. The zlip library was created from gzip, which provided the compression and decompression algorithms for PNG.

Id	Compression Method
0	The file is stored (no compression)
1	The file is Shrunk
2	The file is Reduced with compression factor 1
3	The file is Reduced with compression factor 2
4	The file is Reduced with compression factor 3
5	The file is Reduced with compression factor 4
6	The file is Imploded
7	Reserved for Tokenizing compression algorithm
8	he file is Deflated
9	Enhanced Deflating using Deflate64(tm)
10	PKWARE Data Compression Library Imploding (old IBM TERSE)
11	Reserved by PKWARE
12	File is compressed using BZIP2 algorithm
13	Reserved by PKWARE
14	LZMA
15	Reserved by PKWARE
16	IBM z/OS CMPSC Compression
17	Reserved by PKWARE
18	File is compressed using IBM TERSE (new)
19	IBM LZ77 z Architecture
20	deprecated (use method 93 for zstd)
93	Zstandard (zstd) Compression
94	MP3 Compression
95	XZ Compression
96	JPEG variant
97	WavPack compressed data
98	PPMd version I, Rev 1
99	AE-x encryption marker

FIGURE 4.1: Compression methods supported in PKWARE ZIP CITE

MPQ (MoPaQ) MPQ (MoPaQ) is an archive format designed by and named after Mike O'Brien (Mike O'Brien PaCK). It was created while O'Brien was at Blizzard Entertainment for the purpose of storing data for their games. The format was subsequently used in all Blizzard Entertainment games beginning with Diablo (1996). However, since 2014 it has been phased out and older games that are still maintained have been updated to a new CASC (Content Addressable Storage Container) format.

CASC (Content Addressable Storage Container) In 2014 Blizzard started using the Content Addressable Storage Container (CASC), which would replace the previously used MoPaQ file format. Blizzard claimed that the MPQ format had become the source of a number of technical limitations in World of Warcraft. Blizzard highlighted some benefits CASC would provide over MPQ:

- File Corruption - The file structure would maintain itself, helping to prevent errors during installation.
- Speed - Real-world game performance would increase, due to its non-redundant file structure.
- Patching - Updating game files would be easier, patch data would be integrated seamlessly, and would no longer require double the installation size on disc.
- Hotfixing - CASC would allow them to hotfix client game files, meaning they could address issues quickly that would previously require a full patch.
- Streaming - The new format would support streaming game data while playing and allow them greater flexibility to define how content updates were delivered and released.
- Expandable - The file format was built to allow easy iteration of new technologies which they have not released yet.

While a lot of the listed benefits *were* actually supported using MPQ the data structure was completely different.

The CASC file format is heavily integrated with Blizzard's Next Generation Data Pipeline (NGDP), their Trusted Application Content Transfer (TACT), and Rabbit. Not much is known about the NGDP, however it appears to be a backend pipeline they use to prepare patch data, upload it to their CDN (Content Delivery Network) and distribute to their TACT and Rabbit repositories.

The TACT system has been partially reverse engineered. It appears to be the communication protocol between Blizzard's servers and Blizzard's game client downloader, which describes the data required for any particular version of the game. It contains patch configuration manifests, archive indexes and unarchived stand-alone files, typically binaries, mp3s, and movie files, mentioned in the configurations.

The TACT protocol exists over HTTP, therefore we are able to easily download files and explore game manifests with off-the-shelf tooling. However, TACT has been phased out for a similar system without using HTTP and instead only using a TCP socket with a custom protocol.

The CASC file format has been mostly reverse engineered, and its structure has been heavily documented.

wharf Itch.io released a new open source toolkit primarily designed for their content delivery platform under the same name. Wharf is a protocol that enabled incremental uploads and downloads to keep software up-to-date. It includes; a diffing algorithm

based on rsync, an open file format specification for patches and signature files based on protobuf, a reference implementation in golang, and a command line tool which contains several commands.

Butler is a command-line tool for generating patches and uploading them for distribution on Itch.io. Although this is not directly linked to archive file formats, the principles and practices may be similar.

4.1.3 Reverse Engineering

To better understand the inner workings of the game and its supporting infrastructure, we have employed reverse engineering techniques.

4.1.3.1 Research Methodology

We begin by examining the file system of Warhammer Online and installation media to identify key executables and configuration files.

4.1.3.2 System-Level Analysis

Here we will briefly analyse all of the game data files on the file system. Although a behavioural analysis may provide better results, it is unfortunately not possible to complete many of the required steps because the original servers are no longer available.

The game was originally distributed on two DVDs. Disc one contains several directories: data1, help, pictures, setup, trailers. It also contains .ico (icon) files, and a Windows PE executable WARLauncher.exe. Disc two contains all of the same files and data except the data1 folder, instead containing a data2 folder.

Running the WARLauncher.exe executable and selecting 'install' then launches setup/war/war.exe. By inspecting war.exe in a hex editor, we can see references to "InnoSetup". Inno Setup is a free installer for Windows programs, first introduced in 1997. Unfortunately, there is no official unpacker - the only method of getting files out of the self-extracting executable is to run it. innounp is an open source Inno Setup unpacker based on the Inno Setup source code and supports Inno Setup versions 2.0.7 through 6.1.2. Using this tool on war.exe extracts an install_script.iss file. It also extracts several embedded asset files that would have been self-extracted during the installation process.

In the installation script, we can see that the entry game executable that is used after installation is warpatcher.exe. Installing the game using the installer seems to move all of the files from data1 and data2 into the install directory and place the correct UserSettings.xml files into place, depending on the language selected during installation.

4.1.3.3 warpatcher.exe - Updater client

The patcher was responsible for checking for updates to the game client, downloading them, and patching the files on the user's system. It was also responsible for authentication and the launch of the main game executable.

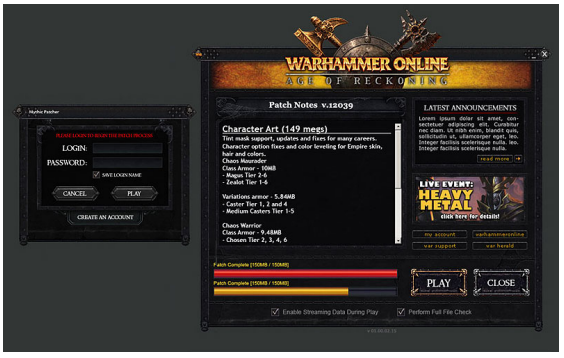


FIGURE 4.2: Design of the Warhammer Online – Patcher by Jason Krieger [10]

We can see the configuration for warpatcher.exe in patch.cfg. Here we can see the names of the products it maintains and the manifest repositories for all of the files. We can see 2 products 'EAMythic Patcher' and 'Warhammer Early Release'.

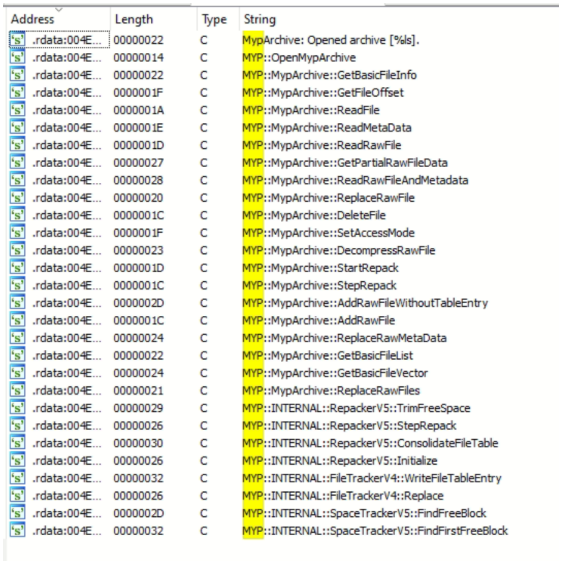
```
<PatchConfig>
 <Config>
         <ResourceModules>
          <Module name="http" />
          <Module name="login">
                <PreAuth />
          </Module>
         </ResourceModules>
  <PatcherProduct>
   <Product name="EAMythic Patcher" description="EAMythic Patcher" prodfile="
    patcher-goa.prod">
    <manifestrepos>
     <repo url="http://patch.war-europe.com/patch/patcher/manifest/" />
    </manifestrepos>
   </Product>
  </PatcherProduct>
  <MainProduct>
   <Product name="Warhammer Early Release" description="Warhammer Early Release
     Client" prodfile="war-goa.prod">
    <manifestrepos>
     <repo url="http://patch.war-europe.com/patch/warhammer/manifest/" />
    </manifestrepos>
   </Product>
  </MainProduct>
 </Config>
</PatchConfig>
```

FIGURE 4.3: patch.cfg file from the game directory after installation

Running warpatcher.exe produces many pop-up windows errors before exiting. It creates some files in the same directory as itself as well as a log file. Examining the contents of the log file, we can see that the application attempts to make a number of HTTP requests to a server. However, these requests fail, which causes the executable to exit.

There are several large files in this directory with the file extension .myp which have names such as art.myp, interface.myp, world.myp, video_english.myp and vo_english.myp etc. These would appear to be archive files that we could assume contain the assets of the game.

Opening warpatcher.exe in IDA Pro we can find many strings which reference “MYP” 4.4.



Address	Length	Type	String
.rdata:004E...	00000022	C	MypArchive: Opened archive [%ls].
.rdata:004E...	00000014	C	MYP::OpenMypArchive
.rdata:004E...	00000022	C	MYP::MypArchive::GetBasicFileInfo
.rdata:004E...	0000001F	C	MYP::MypArchive::GetFileOffset
.rdata:004E...	0000001A	C	MYP::MypArchive::ReadFile
.rdata:004E...	0000001E	C	MYP::MypArchive::ReadMetaData
.rdata:004E...	0000001D	C	MYP::MypArchive::ReadRawFile
.rdata:004E...	00000027	C	MYP::MypArchive::GetPartialRawFileData
.rdata:004E...	00000028	C	MYP::MypArchive::ReadRawFileAndMetadata
.rdata:004E...	00000020	C	MYP::MypArchive::ReplaceRawFile
.rdata:004E...	0000001C	C	MYP::MypArchive::DeleteFile
.rdata:004E...	0000001F	C	MYP::MypArchive::SetAccessMode
.rdata:004E...	00000023	C	MYP::MypArchive::DecompressRawFile
.rdata:004E...	0000001D	C	MYP::MypArchive::StartRepack
.rdata:004E...	0000001C	C	MYP::MypArchive::StepRepack
.rdata:004E...	0000002D	C	MYP::MypArchive::AddRawFileWithoutTableEntry
.rdata:004E...	0000001C	C	MYP::MypArchive::AddRawFile
.rdata:004E...	00000024	C	MYP::MypArchive::ReplaceRawMetaData
.rdata:004E...	00000022	C	MYP::MypArchive::GetBasicFileList
.rdata:004E...	00000024	C	MYP::MypArchive::GetBasicFileVector
.rdata:004E...	00000021	C	MYP::MypArchive::ReplaceRawFiles
.rdata:004E...	00000029	C	MYP::INTERNAL::RepackerV5::TrimFreeSpace
.rdata:004E...	00000026	C	MYP::INTERNAL::RepackerV5::StepRepack
.rdata:004E...	00000030	C	MYP::INTERNAL::RepackerV5::ConsolidateFileTable
.rdata:004E...	00000026	C	MYP::INTERNAL::RepackerV5::Initialize
.rdata:004E...	00000032	C	MYP::INTERNAL::FileTrackerV4::WriteFileTableEntry
.rdata:004E...	00000026	C	MYP::INTERNAL::FileTrackerV4::Replace
.rdata:004E...	0000002D	C	MYP::INTERNAL::SpaceTrackerV5::FindFreeBlock
.rdata:004E...	00000032	C	MYP::INTERNAL::SpaceTrackerV5::FindFirstFreeBlock

FIGURE 4.4: Strings of warpatcher.exe containing the word 'MYP'

These would appear to be methods for interfacing with myp files and accessing their data.

4.1.3.4 Manifest Repository

The manifest repository is a remote file store on a web server which stores information on the current build of the game including lists of files, manifests of data archives, cryptographic signatures, file checksums and either the data itself or a pointer to the data.

Due to the servers no longer being available, we are not able to see the contents of the remote manifest repository. However, we know that the architecture of the patcher was used in another product which still has servers available [139].

From here we can see the URL of the manifest repository and the name of the product file. Accessing this file on the web server gives us information such as the product serial number and launch executable name, as well as URLs to the manifest repository, the file repository and a description of the patching stages and the packages within each stage is illustrated in Figure 4.6. It states only one package, called "base" and references a "pkg.mft". Each file in the manifest repository has a corresponding signature file, which can be located by appending ".sig" to the URL. We can then use this information to download the manifest for the base package.

The "pkg.mft" is a complete summary of build, in this case it contains pointers to two sub-manifests "unpacked.mft" and "patch.myp.mft". These files are all binary files that contain compressed information regarding the lists of files they reference. The former

seems to contain files that should be stored natively on the file system and not in an archive file. In this case, it contains the following:

File
camtest.exe
patchui_win.dll
patcher/splash/english/patch.html
patcher/splash/english/daoc_bright.jpg
camtest.cfg
patcher/splash/english/daoc_dark.jpg

The complete flow of the patching process for this product would be the following.

Action	Input
Download	patcher.prod
Download	patcher.prod.sig
Start Stage	main
Start Package	base
Download Manifest	pkg.mft
Parse Manifest	pkg.mft
Download Manifest	pkg.mft
Parse Manifest	pkg.mft
Download Manifest	unpacked.mft
Parse Manifest	unpacked.mft
Download Manifest	patch.myp.mft
Parse Manifest	patch.myp.mft
Download	camtest.exe
Download	patchui_win.dll
Download	patchersplashenglishpatch.html
Download	patchersplashenglishdaoc_bright.jpg
Download	camtest.cfg
Download	patchersplashenglishdaoc_dark.jpg
Processing	patch.myp

```
<PatchConfig>
  <Config>
    <Log path="." prefix="patcher" level="4" />
    <ResourceModules>
      <Module name="http" />
    </ResourceModules>
    <PatcherProduct>
      <Product name="EAMythic Patcher" description="EAMythic Patcher" prodfile=
    "patcher.prod">
        <manifestrepos>
          <repo url="http://patch.daoc.eamythic.com:1380/daocpatch/pendragon/
    patcher/manifest/" />
        </manifestrepos>
      </Product>
    </PatcherProduct>
    <MainProduct>
      <Product name="DAoC Pendragon" description="DAoC Pendragon Client"
    prodfile="daoc-pendragon.prod">
        <manifestrepos>
          <repo url="http://patch.daoc.eamythic.com:1380/daocpatch/pendragon/
    daoc/manifest/" />
        </manifestrepos>
      </Product>
    </MainProduct>
  </Config>
</PatchConfig>
```

FIGURE 4.5: patch.cfg file from another product using the same architecure

```
<MythicMFT>
    <product serial="1709146253" launchfile="camtest.exe">
        <manifestrepos>
            <repo url="http://patch.daoc.broadsword.com:1380/daocpatch/
    pendragon/patcher/manifest/" />
        </manifestrepos>
        <filerepos>
            <repo url="http://patch.daoc.broadsword.com:1380/daocpatch/
    pendragon/patcher/files/" />
        </filerepos>
        <stages>
            <stage name="main" priority="10">
                <completion />
                <packages>
                    <package name="base" rpath="base" priority="0">
                        <manifest n="pkg.mft" ul="2d8" ct="1" cl="1c0" t="65
    df808d">
                            <sig id="30" v="AIEBLL+
    nigvKCsmAvYM3R1Hvfn7zhXR4awFjd5argnQvyabTqfXG9VtK+73FwKKSm6o+
    agjKMWlNQdBciybmEf9X/r9qPUimc6M1auFvVPg+0
    XQW0P7T1MACmrlUB5ySk5XP1x5G7AZ6yM5CXHyjaFkZX0r9GWukqKIEe7PcfPvQkvM=" />
                        </manifest>
                    </package>
                </packages>
            </stage>
        </stages>
    </product>
</MythicMFT>
```

FIGURE 4.6: patcher.prod from the maniferst repository

4.1.3.5 Static File Analysis

In Figure 4.7 we present the contents of the file dev.myp. It is a data archive that stores software assets. Inspecting the file, we can see that there is some data at the start. This appears to contain padding until position 0x200.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	4D	59	50	00	05	00	00	00	43	EC	23	FD	63	03	00	00	MYP.....Cì#ýc...
00000010	00	00	00	00	E8	03	00	00	E7	01	00	00	00	00	00	00	....è...ç.....
00000020	05	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000200	04	00	84	00	00	03	00	80	05	1D	C0	48	50	77	77	31	..„....€..ÀHPww1
00000210	51	66	E2	EE	71	DA	6B	49	24	9A	4F	5D	C6	E1	A9	21	Qfâîq ÚkI$šO]Æá©!
00000220	AC	7E	3D	7E	21	77	10	F7	C3	89	16	CC	94	D4	7D	D5	¬~=~!w.÷Ã‰.Ì”Ô}Õ
00000230	46	4E	AE	08	A2	40	FC	71	AA	51	B0	39	02	E8	A2	86	FN®.¢@üqªQ°9.è¢†
00000240	55	C5	DB	A3	59	B8	95	D3	33	5E	7A	86	A3	79	EA	A0	UÅÛ£Y¸•Ó3^z†£yê
00000250	6E	7F	42	EA	2E	58	E7	A9	DD	D4	28	EC	43	54	56	70	n.Bê.Xç©ÝÔ(ìCTVp
00000260	11	3F	2A	6F	AB	12	3B	06	45	D3	9F	65	F0	A0	DE	D2	.?*o«.;.EÓŸeð ÞÒ
00000270	CA	DA	5B	73	62	C7	F1	56	0F	10	12	6D	D5	4A	4D	66	ÊÚ[sbÇñV...mÕJMf

FIGURE 4.7: MYP file viewed in a hex editor

Using a tool such as IDA Pro or Ghidra, we can open the executable file, view functions, and decompile them into pseudo code. Inspecting the function in WAR.exe that is responsible for loading the .myp file, sub_819B3D, you can see the following:

```
v8 = *(_DWORD *)(v4 + 96);
if ( !v8 )
  goto LABEL_57;
LODWORD(v9) = (*(int (**)(void))(*(_DWORD *)v8 + 44))();
if ( !v9 )
  goto LABEL_57;
(*(void (__stdcall **)(int, signed int, signed int))(**(_DWORD **)(v4 + 96) + 36))(v4, 40, 1);
v10 = (unsigned __int8 *)(v4 + 8);
if ( *(_DWORD *)(v4 + 8) != 0xFD23EC43 )
{
  sub_68BD14(v10);
  if ( *(_DWORD *)v10 == 0xFD23EC43 )
  {
    *(_BYTE *)(v4 + 88) = 1;
    sub_68BD14((unsigned __int8 *)(v4 + 4));
    sub_819B10();
    sub_68BD14((unsigned __int8 *)(v4 + 20));
    sub_68BD14((unsigned __int8 *)(v4 + 24));
    sub_68BD14((unsigned __int8 *)(v4 + 28));
    sub_819B10();
  }
}
v11 = *(_DWORD *)(v4 + 4);
if ( v11 < 4 || v11 > 5 )
{
LABEL_57:
  v39 = *(void (__stdcall ****)(signed int))(v4 + 96);
  if ( v39 )
    (**v39)(1);
  *(_DWORD *)(v4 + 96) = 0;
  goto LABEL_54;
}
v12 = sub_87FB68(16);
```

As you can see, there is a conditional check that checks that the value at an offset of 8 in the file is not 0xFD23EC43, which aligns with what we see in the file. If the value is not present, it calls sub_68BD14 before doing an additional check to see if the value is equal to 0xFD23EC43.

```
int __thiscall sub_68BD14(unsigned __int8 *this)
{
  unsigned __int16 v1; // ax
  int v2; // edx
  int result; // eax

  LOBYTE(v1) = 0;
  HIBYTE(v1) = *(_DWORD *)this >> 16;
  v2 = v1 | ((*(_DWORD *)this & 0xFF00 | (*(_DWORD *)this << 16)) << 8);
  result = this[3];
  *(_DWORD *)this = result | v2;
  return result;
}
```

This function appears to take a pointer to some data and swap the bytes of the data before storing them in their original location in memory. This function will be referred to as EndianSwap32 and by cleaning up the pseudo-code we can make it more readable.

```
void __cdecl EndianSwap32(unsigned int *startOfData)
{
  unsigned int *v1; // ecx
  unsigned __int16 v2; // ax
  unsigned int startOfDataa; // [esp+8h] [ebp+8h]

  v1 = startOfData;
  startOfDataa = *startOfData;
  LOBYTE(v2) = 0;
  HIBYTE(v2) = BYTE2(startOfDataa);
  *v1 = *(v1 + 3) | v2 | ((startOfDataa & 0xFF00 | (startOfDataa << 16)) << 8);
}
```

We can also note that sub_819B10 appears to call EndianSwap32 twice and perform the same operation but on a 64-bit number. Therefore, the section of code referenced appears to check the value and, if it does not match, swaps the endianness of the value that then matches, it assumes the file endianness is the opposite of what it is expecting and swaps the data in the rest of the header. We will refer to this value as "magicNumber". This also reveals the size of the values in the header of the file. Furthermore, at 0x00419025 we can see that the field at 0x04 of the file does a comparison to check if the value is less than 4, or greater than 5, which then skips all subsequent loadings. Therefore, it appears to be a version check. By changing this value in dev.myp we can see that the loading process is skipped and the file is ignored. We can also see that at 0x00419093 the value at 0x0C is then used to jump to that position in the file, which indicates the start of the table of contents. At 0x00419065 we can see that the value at 0x14 is used to pre-allocate 34 multiplied by the value stored at 0x14 array of memory. We can then see at 0x00419077 that the memory block that is then zeroed out. The other values do not appear to be used, which indicates that they are not used in the reading of the archive file and possibly only in the writing. Currently, the header is made up of the following values and has a total size of 0x28.

Offset	Size	Name	Comment
0x00	4 Bytes	File Tag	Always "MYP\0"
0x04	4 Bytes	Version	Skipped if not 4
0x08	4 Bytes	Magic Number	Determines files endianness
0x0C	8 Bytes	Table Offset	Offset of start of the first file table
0x18	4 Bytes	Number of Files	Number of files in archive
0x1C	4 Bytes	Unknown 1	Unused
0x20	8 Bytes	Unknown 2	Unused

The offset in the table points to the position of the first table in the file. A file table appears to be made up of a header, which we will refer to as the File Table Header, and a list of individual file metadata, which we will refer to as a File Table Entry. The header contains the number of files that will appear after this header and a pointer to the start of the next File Table. The File Table is 12 bytes in size.

Offset	Size	Name	Comment
0x00	4 Bytes	Amount of File Table Entries	
0x04	8 Bytes	Offset to the next File Table	

Each File Table Entry is made of up 34 bytes, which aligns with the pre-allocated buffer the executable calculated from the Myp Header. This contains the following fields.

Offset	Size	Name	Comment
0x00	8 Bytes	Data Offset	Pointer to the file data
0x08	4 Bytes	Header Size	Size of the data header
0x0C	4 Bytes	Compressed Size	Size of the data on disk, after compression if compressed
0x10	4 Bytes	Uncompressed Size	Size of the data before compression
0x14	8 Bytes	Key	A unique key to identify the file
0x1C	4 Bytes	Checksum	Checksum of the file data
0x1D	1 Bytes	Compressed Flag	Boolean to represent if compressed with deflate
0x1F	1 Bytes	Unknown	Possibly padding for alignment

The Mythic File Entry Data is made up of two blocks of data, one immediately stored after the other. First are the header data, which starts at the Data Offset position and is Header Size in length, both of which are specified in the corresponding File Table Entry. Then there are either the compressed data, if the Compressed flag is set, or the uncompressed data, which is located at Data Offset + Header Size position and is of Compressed Size length regardless of compression.

4.1.4 Conclusion

In this task, we successfully reconstructed the proprietary .myp archive file format used by Warhammer Online. Through systematic static file analysis, the structure of headers, file tables, and entries was deduced, including the use of offsets, compression flags, and integrity checks. This process enabled the extraction and interpretation of in-game assets, providing both a practical description of the reverse engineering workflow and a basis for preservation and further analysis. The key takeaway is that the archive formats, even when undocumented, can be reliably reverse-engineered.

4.2 Anti-Cheat

In this section, we provide an overview of the anti-cheat landscape, examining both how cheats are developed and the countermeasures designed to detect and prevent them.

```
typedef struct {
        char    /*0x00*/ file_tag[4];
        uint    /*0x04*/ version;
        uint    /*0x08*/ magic_number;
        ulong   /*0x0C*/ table_offset;
        uint    /*0x18*/ file_count;
        uint    /*0x1C*/ unknown_1;
        uint    /*0x20*/ unknown_2;
} Header;

typedef struct {
        uint    /*0x00*/ file_count;
        uint64  /*0x04*/ next_table_offset;
} FileTableHeader

typedef struct {
        uint64  /*0x00*/ data_offset;
        uint    /*0x08*/ header_size;
        uint    /*0x0C*/ compressed_size;
        uint    /*0x10*/ uncompressed_size;
        uint64  /*0x14*/ key;
        uint    /*0x1C*/ checksum;
        byte    /*0x1D*/ compressed;
        byte    /*0x1F*/ padding;
} FileTableEntry;

typedef struct  {
    FileTableHeader fileTableHeader;
    FileTableEntry fileTable[fileTableHeader.file_count];
} FileTable;

struct MYP
{
    Header header <optimize=false, open=true>;

    local int offset <hidden=true>;

    offset = header.table_offset;

    while (offset > 0) {
        FSeek(offset);
        FileTable fileTable <optimize=false, open=true>;
        offset = fileTable.fileTableHeader.next_table_offset;
    }

} myp;
```

Listing 4.1: A MYP file described in the Binary Template format

4.2.1 Introduction

Cheating has been commonplace in video games since their inception. The first cheat codes were put in place for testing purposes. Play testers had to rigorously test different parts of games and cheat codes were added to make the process easier, so they could skip to the required level for testing, for example. However, with the advent of online video games, cheating increasingly allows one player to have an unfair advantage against other legitimate players using legitimate gameplay restrictions.

Video game developers often resort to implementing anti-cheat technology, either their own or from a third party, in order to detect and suspend cheating players to stop them from accessing the services and tainting the game-play experience of other players.

This section will focus on providing background research on common types of cheats for video games and how they are developed. We will also be focusing on methods to detect certain cheats, and the limitations of certain detection techniques. We will then reverse engineer a cheat for a video game and identify a way to detect the cheat. Subsequently we will look at the process of implementing anti-cheat methods into a video game where the original source code of the game or the cheat is unavailable.

In summary, this chapter will dive into reverse engineering cheats to understand their functionality and develop methods to detect them. It will also outline a structured process for reverse engineering in scenarios where the game's source code or the cheat's original code is unavailable.

4.2.1.1 Background

This background provides the necessary context to understand the motivations and challenges in detecting cheating behaviour, setting the stage for a more detailed technical discussion.

4.2.2 Technical Background Knowledge

In this section, we are going to explore different methods of cheating in video games. We will then look at specific video games, common cheats for those games, and anti-cheat technology used by the game developers to combat those cheats.

4.2.2.1 Anti-Cheat Systems

We now turn our attention to specific anti-cheat systems implemented by game developers, starting with Blizzard Entertainment's Warden, one of the earliest and most well-known examples.

Warden Warden is an anti-cheating tool integrated in Blizzard Entertainment games such as Diablo II (since patch 1.11), StarCraft (patch 1.15), Warcraft III and most notably World of Warcraft. When active, Warden monitors the activity of programs and processes on the host computer. It transmits usage data and some desktop details—such as the titles of open windows—to Blizzard's servers over the Internet. Blizzard claims that the Warden does not collect personally identifiable information, focusing solely on game account data and analysing the collected information for signs of hacking or cheating programs [140].

Warden is split into two parts; a server-side and a client-side part. The client-side code is dynamically loaded from the servers and most of the client-side code is obfuscated

[11]. Warden is described as a passive anti-cheat in that it collects information that is then sent to the server side for evaluation [141].

The scanning process performed by Warden is focused on detecting known cheats, specifically those that manipulate memory addresses commonly targeted by cheats or hacks. Warden follows the following steps in its scanning process:

1. Obtaining the list of relative addresses for scanning.
2. Reading the required number of bytes at each of the addresses.
3. Calculating hashes.
4. Building the packet with hashes and sending it to the server.

This process can happen several times per minute and different checks can occur each time. If the server side detects that the hashes mismatch the reference values, it concludes that an illegal code modification has been performed. Accounts are flagged as violating the rules and then batches of accounts are banned in 'ban waves'. Ban waves occur to reduce the chance that cheat developers can reverse engineer the detections in order to bypass them.

In the process a honeypot technique is often used. Developers are aware of known exploits and intentionally leave these vulnerabilities or detectable behaviours exposed in order to track and monitor those who attempt to exploit them. By observing which players trigger the honeypot checks, developers can build a list of offenders. This strategy helps to gather information about cheats and their methods without altering the attackers directly. Moreover, it allows developers to obfuscate their actual cheat detection methods, preventing hackers from easily reverse engineering and bypassing them.

If the server fails to receive a response to Warden cheat checks, then the server side code will know that the client side element is not responding and can act accordingly.

Therefore, this leaves three main methods for creating cheats for games using Warden:

1. Avoiding addresses Warden is known to scan and verify.
2. Using code outside of the application access, such as DirectX calls.
3. Hide all modifications while scanning takes place.

The first method involves creating a new zero-day hack for a game which, if released publicly, will most likely be added as a detection in a future Warden update. The second method is usually used when the game sends a frame to the GPU for rendering and is usually hooked by applications for legitimate uses, such as voice chat overlays. This

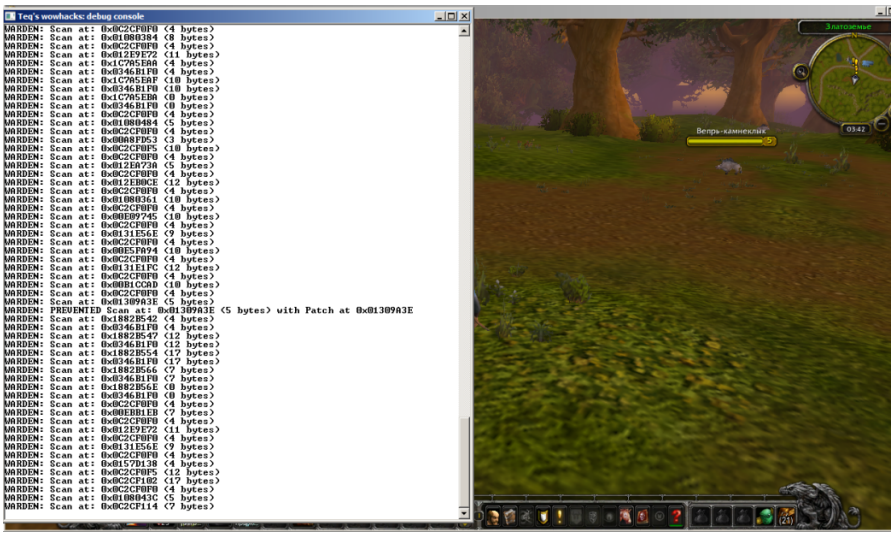


FIGURE 4.8: Prevention of warden cheat detection [11]

method allows limited access and is usually used for overlays such as map hacks, which overlay additional information to a player's screen such as the location of enemy players behind walls. The last method is dynamic and involves identifying the main sections of Warden code, monitoring when the code triggers, and undoing all of the modified code, therefore allowing the memory addresses scanned being the original addresses, and after the scan is completed, reapplying the modified code. This last method is explored in more detail and a proof of concept is implemented in [11] [142].

PunkBuster PunkBuster is an anti-cheat system developed by Even Balance, Inc., designed to prevent cheating in multiplayer online games. It was widely used in games like the Battlefield series, Medal of Honor, and Call of Duty. It is a third-party product that game publishers can use to prevent cheating in online games. Although less prominent in modern games, it was one of the earliest anti-cheat systems to focus on real-time detection and enforcement [143].

A separate client-side application is installed and is required to be running when a game connects to a PunkBuster enabled game server. PunkBuster supports the server side of its application running on a third-party server, often hosted by players. The client and server side of PunkBuster work as modules, both load the main PunkBuster (Dynamic Link Library) DLL, which just exports two functions:

- 'ca' and 'cb' for the client
- 'sa' and 'sb' for the server

Messages between the server-side and client-side modules are usually sent over the normal games network protocol serialised again in the games proprietary protocol.

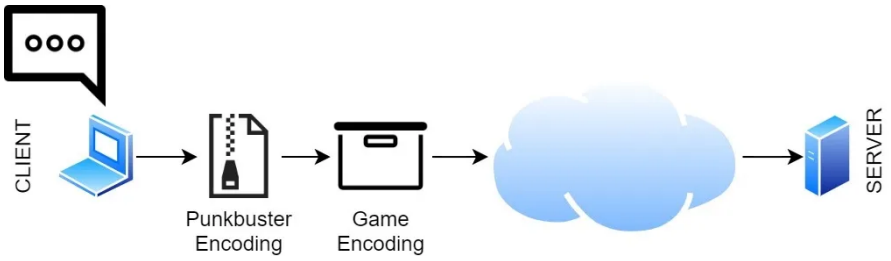


FIGURE 4.9: Diagram demonstrating PunkBuster network packets and serialization [12]

PunkBuster also includes a screen capture facility that allows server admins to request actual screenshots from players' screens while they are playing. The screenshots are transferred over the network and saved by the Server Admins either for private use or for publication to a website. This facility was reverse-engineered and exploited, allowing for directory traversal and remote code execution in [12].

Easy Anti-Cheat (EAC) Easy Anti-Cheat (EAC) is a widely used anti-cheat system developed by Epic Games. EAC is used in popular games like Fortnite, Apex Legends, Rust, and Dead by Daylight. It provides protection against a variety of types of cheat, including aimbots, wallhacks, and other forms of game manipulation. EAC uses a more modern type of anti-cheat system usually referred to as kernel-level which uses a kernel driver with privileged access to the whole of the users' system. Unlike most anti-cheat products, which are either completely closed and in-house or third party but propriety and require a licence agreement for documentation. EAC is free and is included with the Epic Online Services (EOS) package of products for video game developers to integrate into games. That is, the full documentation is freely available online.

EAC comprises of two anti-cheat interfaces, a server interface and a client interface. They provide tools to start your game client in an anti-cheat protected mode which is then validated by either of the following methods:

- A game server using the Anti-Cheat Server Interface
- Another game client using the Anti-Cheat Client Interface

Interestingly, as this is a kernel-level anti-cheat, the client interface does not support virtual machines, whereas the server interface does.

EAC claims to offer two main kinds of protection, 'cheat prevention' which is to passively prevent common cheating techniques such as reading and writing the games memory using straightforward methods, and 'cheat detection' to identify accounts using more specialised methods to manipulate the game so that sanctions can be applied to them.

Epic Games also offer additional support and features that include:

- Cheat detection work specific to a game by the Epic Games team of analysts
- Tooling for managing recurring cheaters who create new accounts on the same machine
- Training and application of machine learning models for the detection of aim-assisting cheats in shooter games. The models are based on the data on the behaviour of players on game servers

The use of anti-cheat operating at the kernel-level has led to a lot of controversy [144] [145]. Support is typically unavailable for operating systems other than Windows, such as Mac or Linux, preventing some users from participating in online sessions protected by anti-cheat measures. As the kernel runs at a very high privilege level on the operating system, it has full access to all the other lower priority applications, which introduces many privacy concerns [146]. They have also been described as rootkits and [123] attempts to propose a set of criteria to characterise the 'rootkit-ness' of a piece of software. Developers dismiss claims that their kernel-level anti-cheat systems invade privacy, urging users to trust them, as they are committed to ensuring fair play while protecting personal data [147] [148] [149]. However, if vulnerabilities were found in these kernel-level anti-cheats, it could lead to third-party malicious actors and it would not prevent them from doing malicious acts.

In fact, EAC has been reverse engineered [150], leading to exploits. "EasyAntiCheat Exploit to inject unsigned code into protected processes" introduces an exploit which tricks the anti-cheat into protecting your memory as its own and grants access to do things such as the creation of threads, deliberately placed hooks, etc [151].

4.2.2.2 Cheat Detection Methods

There are many ways that cheats can interact with the game, from reading and writing memory, to intercepting network traffic, to interacting with a game's API. Some of these methods are highlighted below.

Memory Check Often a hack on a game will read and write memory. A simplistic example of this is a game which has a value which represents money or health of the player and this value is stored in memory, editing this value in memory will update the game and the player would have a new amount of money in the game. In single player games this level is often tolerated and there are usually commands players can use to access the same functionality. However, in multiplayer games, this can have severe consequences. If the value stored in the memory address is the authoritative version of the players' health or money, then it can have drastic consequences to the fairness of the game or the in-game economy. The best protection against this is having the value

stored in the client being a read-only version of the value, where the true value is stored on the game server and can only be modified through approved API calls. For example, the money is only incremented when a player does an API call which sells an item from their inventory. Then the value in the server is updated, and the new money value is sent to the client, and the read-only value is updated.

However, doing this for all values can lead to some performance overhead. For example, the location of a player in the game is usually stored on the client so that the game client can know where to render the player model. If movement required an API call to be sent to a server and the speed, collision, etc. to all be calculated on the server and a destination coordinate be sent back to the client so the camera can be updated, this can lead to what can be perceived as high latency or poor responsiveness of input. Whereas this can be somewhat mitigated with games where you have globalised servers, where game servers are located in every continent, and use client-side prediction, games which have servers in single points of the world where other players can have hundreds of milliseconds of latency, this approach is not suitable.

One way this can be implemented is that the client has some authority over the players' movement and the server could do periodic updates to verify the legitimacy of movement from the player. This can be done by having a speed variable which is read only and the player input is evaluated on the client, the players position moved accordingly, and the updated location sent to the server. The server then takes the only position and the time since the last position updates and does a simple calculation to see if the position was within the possible limit. The anti-cheat module could periodically ask the client to send the value or a hash of the value in the read only speed variable to see if it matches the value stored on the server. If the value is different than that of the server, then we can be sure the player has modified the speed value in an illegal way and flag the player as a cheater.

This method of changing memory can be used in many different ways to allow for cheating, from removing collision enforcement checks to allowing players to walk through walls. Enabling access to previously restricted API calls through the games modding API allowing map hacks. Allowing the player to change the gravity or allowing the player's character to fly allows the player to get into usually inaccessible places in the game.

File Check Another common method of hacking a game is by modifying the static data stored in the game client. Files such as 3d models, textures, sounds, animations, and text are usually all stored on disk, possibly in archive files. One such cheat could be if there is a small hidden object in the game, the player could edit the file of the small object to make it a lot bigger and easily spot it, therefore giving them an advantage over other players. Similarly, this is done for textures, changing an enemy player's model

texture to something such as a bright pink colour allows them to easily be seen against backgrounds. Map hacks are also common where the player has edited the map to open a door that was not previously openable or to modify the terrain in a way that the player can go into inaccessible areas. It has also been heard that players have modified the audio files that represent footsteps in games to boost the gain so that players can easily hear enemies approaching.

The detection of this type of hack can be done by hashing the file when it is loaded into memory and by checking the hash of the file against the known safe hashes or for hashes of known cheating files. Detection can also be done by having the server ask the client to load and hash files from the disk the game would use. This could have some performance impact if there is a lot of disk activity on the game client trying to load data. However, with modern solid-state disks, this concern is likely to be negligible.

Process Checks Another common cheat detection process is if the cheat requires its own process, such as a bot that automates the process of someone playing the game. The executable would be running on the same computer in a separate process. Simply scanning titles of applications could lead to some cheats being detected. Some anti-cheats even go further as to check the whole memory of every application running in the users system looking for memory signatures of known hacks.

One such anti-cheat program that employs this type of cheat detection is PunkBuster. It uses a memory scanner that aggressively scans for patterns associated with known cheats, both public and private. However, a notable downside to this method was demonstrated by a cheat developer, who sent text-based patterns through certain chat-related systems such as IRC and instant messaging. This caused PunkBuster to mistakenly flag innocent players for violations, highlighting a vulnerability in its detection approach [152] [153].

4.2.3 Reverse Engineering

In order to investigate how cheats are developed and detected, a structured research methodology was used. This involved selecting a specific game target and analysing it using reverse engineering tools and techniques.

4.2.3.1 Research Methodology

The methodology begins with a comprehensive analysis of multiple game builds to identify system-level differences, with a particular focus on anti-cheat components and their implemtation.

4.2.3.2 System-Level Analysis

Here we will briefly analyse all of the game data present on different versions of Warhammer Online. From technical background knowledge, we know that certain anti-cheat systems require artefacts such as Dynamic Link Libraries (DLLs) as part of the anti-cheat software package. Examining file system data will allow us to identify any anti-cheat systems which may have been used as part of the original product.

Here are directory listings of two builds of the game, one was present on the digital distribution platform Steam and represents one of the final builds of the original game; the other is located on an Asian installation disk of the game which had a later release than the western counterpart.

We can see that in the Asian disk of the game that contains the build 1.2.1 of the game there is a directory named 'HackShield'. HackShield is an anti-hack toolkit developed by AhnLab Inc., a Korean cybersecurity company, for use in massively multiplayer online games (MMOs). It has been implemented in games such as MapleStory, Combat Arms, and Mabinogi [154] [155] [156]. First released in 2001 for Korean developers and later expanded to American developers in 2005, it has been widely adopted to detect and mitigate cheating in online games.

We can see that both builds of the game have a directory named 'pb', both of which contain a 'pbclgame.cfg' file which contains ascii text which mentions the phrase 'punkbuster'. This would indicate to us that Punkbuster was used in this game to provide anti-cheat facilities.

```
cl_punkbuster 0
```

LISTING 4.2: pbclgame.cfg present in builds of the game

Opening up the DLL in IDA Pro we can see that it exports two functions ca and cb along with the DllEntryPoint which indicates to us that Punkbuster was used in this product as an anti-cheat.

Looking for strings in the WAR.exe in IDA Pro we can find references to Punkbuster further confirming our hypothesis.

4.2.3.3 Reverse Engineering Client

Searching for 'pbcl.dll' we find a reference to the string in the function sub_6082C2 in which we can see 'LoadLibaryA' (from libloaderapi.h as part of the Win32 API in Windows) which would load the code from the DLL into memory. We can also see the calls to 'GetProcAddress' referencing 'ca' and 'cb' which references are then stored at 229 and 228 respectively.

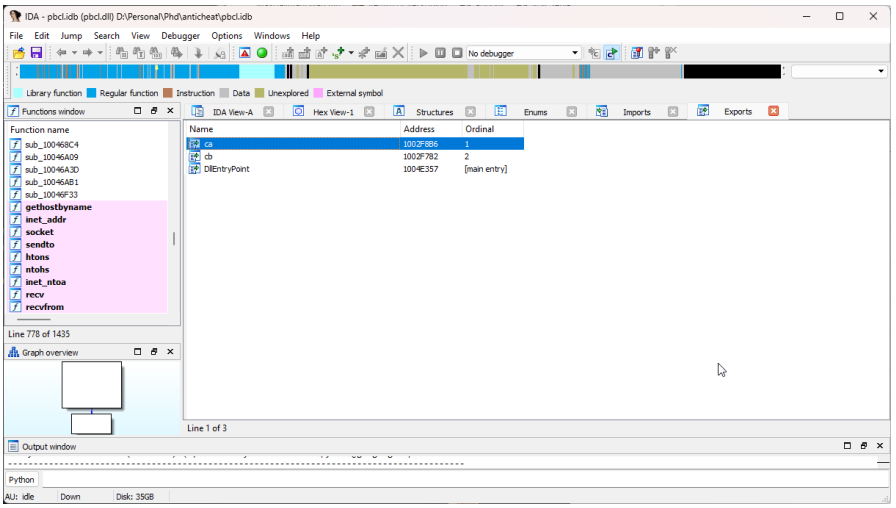


FIGURE 4.10: pbcl.dll opened in IDA Pro

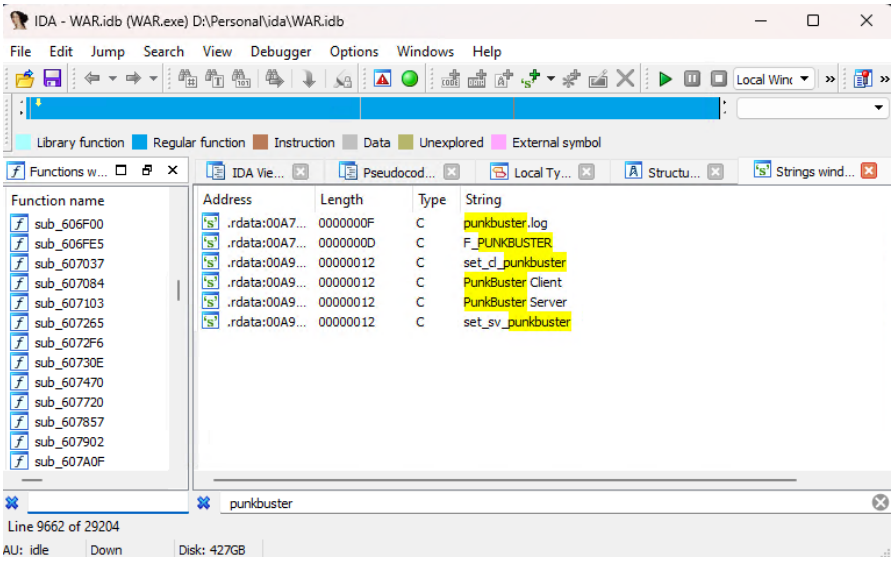


FIGURE 4.11: WAR.exe opened in IDA Pro lookinkg for punkbuster strings

We can then see that sub_6082C2 is called from sub_60843D which after then calls the 'cb' function with five parameters, the first four of which come from its own parameters and the last being '0'. We see that this function is called from three other functions, sub_608A98, sub_608AB6 and sub_60849B. Following sub_608A98 we can see that the first parameter appears to store a state object for the Punkbuster, the second parameter appears to be an opcode, the third appears to be the size of a data payload, and the final appears to be a pointer to the data in memory. Following further up we can see that this function is called from sub_4C9DD9 which is then called from a jump table which appears to be a switch table for packet opcodes, as following further we can see it receives the data from 'recvfrom' which is used to receive a message from a socket.

We can see that loc_4C4C81 the place in the jump table which calls sub_4C9DD9 also references a string 'F_PUNKBUSTER' and following the jump table we can see that

```
const char *__usercall sub_6082C2@<eax>(_DWORD *a1@<eax>)
{
  _DWORD *v1; // esi
  const char *result; // eax
  char *v3; // eax
  FILE *v4; // eax
  char *v5; // eax
  char *v6; // eax
  char *v7; // ST08_4
  char *v8; // eax
  char *v9; // eax
  char *v10; // eax
  char *v11; // ST08_4
  char *v12; // eax
  char *v13; // eax
  HMODULE v14; // eax
  FARPROC v15; // eax
  HMODULE v16; // ST08_4
  FARPROC v17; // eax
  bool v18; // zf
  char v19; // [esp+4h] [ebp-400h]
  char v20; // [esp+204h] [ebp-200h]

  v1 = a1;
  if ( a1[2] )
    return 0;
  sub_60828E(a1);
  v3 = sub_608234(v1, &v20, "pbclnew.dll");
  v4 = fopen(v3, "rb");
  if ( v4 )
  {
    fclose(v4);
    v5 = sub_608234(v1, &v20, "pbclold.dll");
    _chmod(v5, 384);
    v6 = sub_608234(v1, &v20, "pbclold.dll");
    remove(v6);
    v7 = sub_608234(v1, &v19, "pbclold.dll");
    v8 = sub_608234(v1, &v20, "pbcl.dll");
    rename(v8, v7);
    v9 = sub_608234(v1, &v20, "pbcl.dll");
    _chmod(v9, 384);
    v10 = sub_608234(v1, &v20, "pbcl.dll");
    remove(v10);
    v11 = sub_608234(v1, &v19, "pbcl.dll");
    v12 = sub_608234(v1, &v20, "pbclnew.dll");
    rename(v12, v11);
  }
  v13 = sub_608234(v1, &v20, "pbcl.dll");
  v14 = LoadLibraryA(v13);
  v1[2] = v14;
  if ( !v14 )
    return "PB Error: Client DLL Load Failure";
  v15 = GetProcAddress(v14, "ca");
  v16 = v1[2];
  v1[229] = v15;
  v17 = GetProcAddress(v16, "cb");
  v18 = v1[229] == 0;
  v1[228] = v17;
  if ( v18 || !v17 )
  {
    sub_60828E(v1);
    result = "PB Error: Client DLL Get Procedure Failure";
  }
  else
  {
    v1[4] = 0;
    result = 0;
  }
  return result;
}
```

FIGURE 4.12: WAR.exe opened in IDA Pro at sub_6082C2

this code is reached by using opcode 0xD0 in the message sent to the client. Sending a message with this opcode we can see we get into the correct place in the jump table and sub_4C9DD9 is called however it does not call into sub_608A98 as there is a conditional which skips the calling of the function. We can see the conditional checks for the following chars being at the start of the data payload 'ÿÿÿÿPB_' in ASCII which is 'FF FF FF FF 50 42 5F' in hexadecimal.

```
Packet k = new Packet(0xD0);
k.WriteUInt32(0xFFFFFFFF);
k.WriteString("PB_");
```

Listing 4.3: Packet for F_PUNKBUSTER which can reach cb in the DLL

Writing the magic bytes to the start of the message allows the control path to continue down and the cb function of the DLL to be called.

We can then create our own DLL that exports the same function signature as the original and have the WAR.exe call our DLL.

```
#include "PunkbusterPacket.h"
#include "Utils.h"

#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <psapi.h>

#ifdef PUNKBUSTER_EXPORTS
#define PUNKBUSTER_API __declspec(dllexport)
#else
#define PUNKBUSTER_API __declspec(dllimport)
#endif

PUNKBUSTER_API void cb(
    int punkbusterData,
    int punkbusterCommand,
    int packetLength,
    uint8_t *packetData);

PUNKBUSTER_API void ca(
    int a1,
    int a2);
```

Listing 4.4: C Header file for dll which exports ca and cb which matches original signatures

We can see that WAR.exe also has a packet send function which uses the same opcode as 'F_PUNKBUSTER' and a pointer to this function is included in the punkbusterData parameter that is passed into cb of pbcl.dd. Therefore, we can call the function to send data back to the server from our dll.

We can then successfully send data to our dll function and have it send data back to the server. We can then use this to create our own cheat detection system and have the server send requests to the client to perform certain checks. We can then have the client

```
typedef int (*SendPunkbusterPacket)(size_t Size, void *Src);

PUNKBUSTER_API void cb(
    int punkbusterData,
    int punkbusterCommand,
    int packetLength,
    uint8_t *packetData)
{
        // Send data back
        SendPunkbusterPacket SendPacket =
    (SendPunkbusterPacket)(*reinterpret_cast<int *>(punkbusterData + 1908));
        SendPacket(packetLength, packetData);
}

PUNKBUSTER_API void ca(int a1, int a2)
{
}
```

LISTING 4.5: Function to handle data passed to dll and send it back

send back the results of the checks for the server to perform final validation and issue sanctions based on the results.

4.2.3.4 Reverse Engineering Cheats

Now that we have reverse engineers the previous anti-cheat implementation in the game and established a foundation upon which to build our own cheat detection framework, we can shift our focus towards reverse engineering the cheats themselves. By understanding how these cheats function, what memory they modify, what hooks or injections they rely on, and how they avoid detection, we can begin to design targeted detection mechanisms. These mechanisms will then be integrated into our custom framework, enabling us to detect and respond to known cheating behaviours effecively.

4.2.3.5 Implementing Detection for Cheats

In order to detect a cheat effectively, we must first understand how the cheat functions. This requires reverse engineering the cheat itself to examine how it interacts with the game in order to perform its malicious behaviour.

Cheats can interface with games in a number of ways. The most basic forms typically involve modifying easily accessible client-side data such as configuration files, or in some cases, altering game assets. These methods are relatively straightforward and are often limited by the protections and validations implemented by the game developers.

More sophisticated cheat go beyond the surface-level modifications and instead target the game's memory directly. These cheats can be read from or write to specific memory addresses in real time, enabling them to manipulate game states in ways that the developers never intended. For instance, a value stored in memory might represent the

player's movement speed, something that cannot be changed through legitimate gameplay. By locating and modifying this value in memory, the cheat can increase the player's speed to enable entirely new behaviours, such as flying.

In this section, we will reverse engineer a cheat that allows a player to both increase their speed and gain the ability to fly, demonstrating how it achieves these effects and laying the groundwork for detecting similar modifications in the future.

4.2.3.6 RoRPack

RoRPack is a cheat tool developed for Warhammer Online that allowed players to manipulate in-game mechanics such as speed and flight. The origin of this tool is unknown, but its presence demonstrates how client-side manipulation can give players an unfair advantage.

This tool features a minimal graphical user interface (GUI) that allows users to toggle specific cheats on or off with ease. This user-friendliness lowers the barrier for non-technical users to exploit the game. A screenshot of the interface is presented below, showing the simple layout used to control these functionalities.

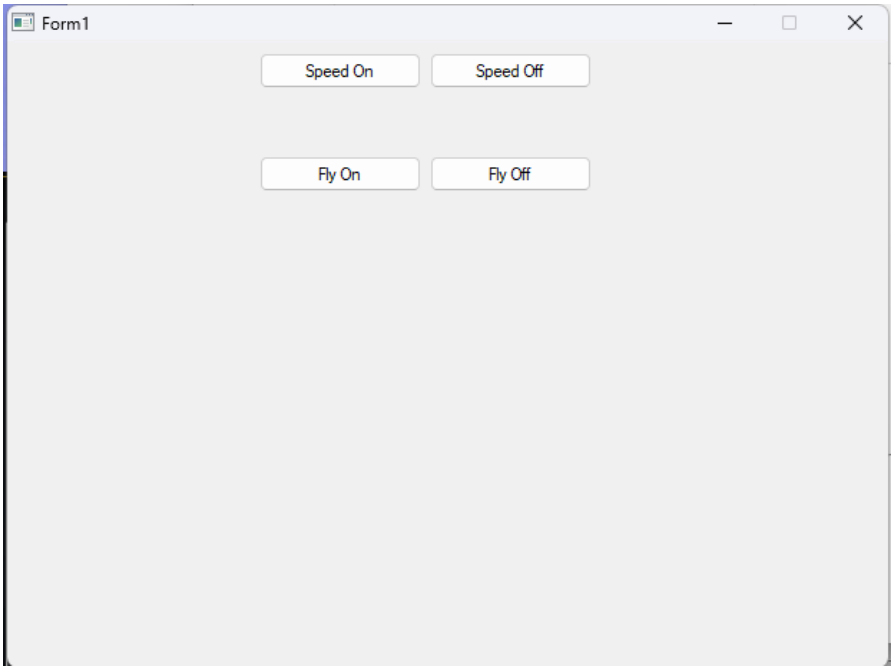


FIGURE 4.13: The interface of RoRPack

The game includes a built-in flying mode intended solely for internal use by authorised personnel, such as developers and game moderators. This feature is not accessible through normal gameplay and is typically hidden from regular players. It is usually activated by sending a specific, privileged packet to the game client, which triggers the client to enable flying functionality.

This packet, once identified and understood, can be exploited by cheat tools such as RoRPack to simulate the same behaviour and grant flight capabilities to regular players. Listing 4.7 illustrates the structure of the packet used to achieve this restricted developer mode.

```
[Client] packet : (0xE4) F_UPDATE_STATE  Size = 14 08:23:43.6803
|-----------------------------------------------|----------------|
|00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F |0123456789ABCDEF|
|-----------------------------------------------|----------------|
|00 0B E4 00 4F 02 01 01 00 00 00 00 00 00       |....O.........  |
-----------------------------------------------------------------
```

LISTING 4.6: Packet which enables the developer features within the game

Opening RoRPack.exe in a hex viewer, we can see references to AutoIt. AutoIt v3 is a freeware BASIC-like scripting language designed to automate the Windows GUI and general scripting. It uses a combination of simulated keystrokes, mouse movement, and window / control manipulation to automate tasks in a way that is not possible or reliable with other languages. As AutoIt scripts are interpreted, the original scripts can be extracted from executable files created with the AutoIt toolchain. A tool that exists to extract resources from AutoIt compiled executables is AutoIT Extractor [157].

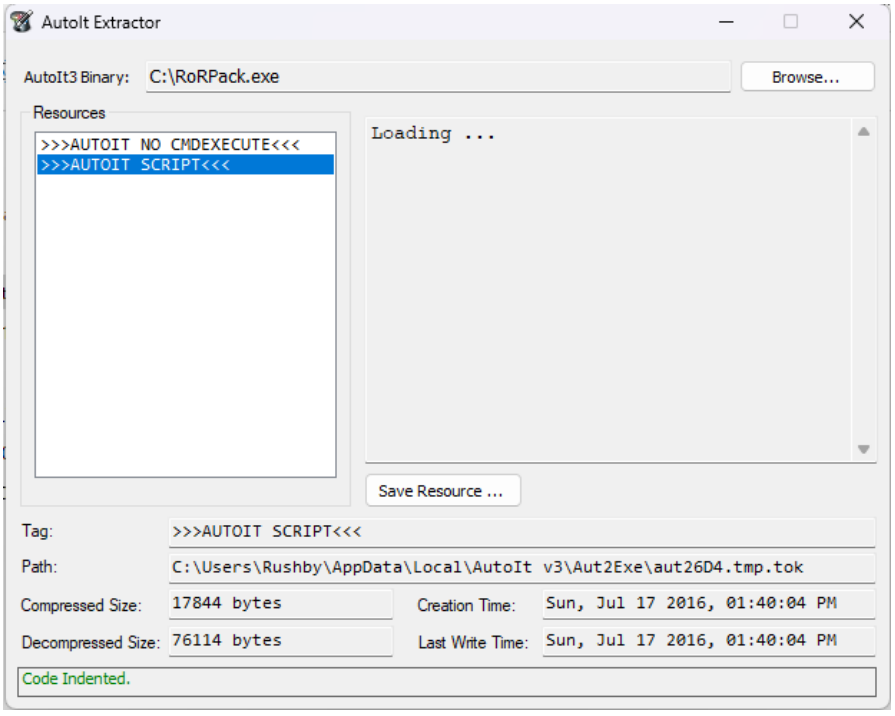


FIGURE 4.14: AutoIT Extractor extracting RoRPack

Using AutoIT Extractor, we successfully unpacked the embedded resources within the RoRPack.exe binary. This process revealed the compiled AutoIt script along with associated files. The extracted script provides valuable information about the original logic

and behaviour of the application. Figure 4.15 shows a portion of the recovered AutoIt script source code.

```
Global $rorprocess = _MEMORYOPEN(ProcessExists("WAR.exe"))
$playerdata = _MEMORYREAD(0xf7611c, $rorprocess, "ptr")
$playerchp = "0x" & Hex($playerdata + 0x188)
$playermhp = "0x" & Hex($playerdata + 0x18c)
$playercap = "0x" & Hex($playerdata + 0x198)
$playermap = "0x" & Hex($playerdata + 0x19c)
$playerspd = "0x" & Hex($playerdata + 0x9c)
$playerflya = "0x" & Hex($playerdata + 0x60)
$playerflyb = "0x" & Hex($playerdata + 0x184)
$playerflyc = "0x" & Hex($playerdata + 0xa0)
$playerflyd = "0x" & Hex($playerdata + 0xa0 + 0x4)
$playerflye = "0x" & Hex($playerdata + 0x94)
$speedon = 0x0
Func SPEEDHACK()
    $speedva = 0x5
    _MEMORYWRITE($playerspd, $rorprocess, $speedva, "float")
EndFunc   ;==>SPEEDHACK
Func FLY()
    _MEMORYWRITE($playerflya, $rorprocess, 0x600, "int")
    _MEMORYWRITE($playerflyb, $rorprocess, 0x2, "int")
EndFunc   ;==>FLY
#Region ### START Koda GUI section ### Form=
$form1 = GUICreate("Form1", 0x267, 0x1b5, 0xc0, 0x7c)
$button1 = GUICtrlCreateButton("Speed On", 0xb0, 0x8, 0x71, 0x19)
$button2 = GUICtrlCreateButton("Speed Off", 0x127, 0x8, 0x71, 0x19)
$button3 = GUICtrlCreateButton("Fly On", 0xb0, 0x50, 0x71, 0x19)
$button4 = GUICtrlCreateButton("Fly Off", 0x127, 0x50, 0x71, 0x19)
GUISetState(@SW_SHOW)
#EndRegion ### END Koda GUI section ###
While 0x1
    $nmsg = GUIGetMsg()
    Switch $nmsg
        Case $button1
            SPEEDHACK()
        Case $button2
            _MEMORYWRITE($playerspd, $rorprocess, 0x1, "float")
        Case $button3
            FLY()
        Case $button4
            _MEMORYWRITE($playerflya, $rorprocess, 0x400, "int")
        Case $gui_event_close
            Exit
    EndSwitch
WEnd
```

FIGURE 4.15: Extracted source code to RoRPack script

The extracted source code reveals the functions that are invoked when each user interface button is pressed. At the beginning of the script, the variable $playerdata is defined as a pointer to the memory address of 0xf7611c. Numerous other variables are then defined as offsets relative to this base pointer, targeting specific regions within the player's in-memory data structure. For instance, the SPEEDHACK() function writes 0x5 to $playerspd, located at the offset of 0x9c from $playerdata. In contrast, the "Speed Off" button restores this value to 0x1. Similarly, the FLY() function manipulates another memory location analogously to alter the player's movement characteristics.

Having identified the memory addresses targeted by the hack, attention can now be turned to the compiled game code to investigate how these are referenced, protected, and used in normal game operation.

Examining the packet handler function sub_4C30CE, previously discussed in Section 4.2.3.3, we can find the handler which deals with the packet illustrated in Listing 4.7, reveals relevant behaviour. When the client receives a packet with an opcode of 0xE4, the execution is directed through a jump table to the function located at 0x004C4EFC which subsequently calls sub_4D7E44, then sub_51DA97, then sub_51DA7D with a parameter value of *10* and finally sub_45A363.

```
void __userpurge sub_51DA97(int a1@<eax>, int a2@<edx>, char a3)
{
  sub_51DA7D(a1, a2, 10, a3);
  if ( a3 )
    sub_5EAC8D(dword_F7828C);
}
_DWORD *__userpurge sub_51DA7D@<eax>(
    int a1@<eax>,
    int a2@<edx>,
    int a3@<ecx>,
    char a4)
{
  _DWORD *result; // eax
  int v5; // [esp-4h] [ebp-4h]

  v5 = a1 + 96;
  if ( a4 )
    result = sub_45A363(a3, a2, v5);
  else
    result = sub_527BE2(a3, a2, v5);
  return result;
}
_DWORD *__fastcall sub_45A363(int a1, int a2, int a3)
{
  _DWORD *result; // eax

  result = (a3 + 4 * (a1 / 32));
  *result |= 1 << a1 % 32;
  return result;
}
```

LISTING 4.7: Functions which are called as a result of the data being sent in Listing 4.7

The function sub_51DA7D conditionally updates a bitfield. It calls sub_51DA7D, passing two input values and a hardcoded bit index of 10, along with a flag a3. Inside sub_51DA7D, the function computes an offset that likely points to the base of a bit field array. Depending on whether the flag a3 is set, it either calls sub_45A363 or sub_527BE2 to modify the bit field. The sub_45A363 function sets the tenth bit within the array, using bitwise operations to locate the correct 32-bit word and update the bit at the appropriate position.

The tenth bit corresponds to the value 0x400, which is explicitly referenced in Figure 4.15. This suggests that the hack operates by directly setting the same value that would normally be triggered via a legitimate packet, thereby simulating the expected network

interaction. By monitoring this specific bit in the process memory, it becomes possible to infer whether the hack has been activated. This insight is valuable for developing a reliable detection mechanism.

To take advantage of this, we can configure our pbcl.dll module to read specific values from the client's memory. By passing the base address 0x00F7611C along with an offset of 0x60, we can retrieve the bytes stored at the corresponding memory location. This enables to remotely verify whether the client has modified the memory to activate the hack. The implementation of this detection logic is shown in Listing 4.8.

```
uint32_t dataLocation;
uint32_t dataOffset;
uint32_t dataLength;

inPack->read(&dataLocation);
inPack->read(&dataOffset);
inPack->read(&dataLength);

int dataLocationPosition = *reinterpret_cast<int *>(dataLocation);
uint32_t response_data =
    *reinterpret_cast<uint32_t *>(dataLocationPosition + dataOffset);

outPacket->writeUint32R(response_data);
```

LISTING 4.8: pbcl.dll client code which can scan the client process for the hack.

When the server receives the value read by pbcl.dll, it can compare this against known flag values associated with specific cheat states. For example, as shown in the table below, the value 0x400 corresponds to the "Debug Mode" flag, while 0x200 indicates that the player is in a "Flying" state, both of which are behaviours typically associated with the use of hacks.

Flag	Comment
0x400	Debug Mode
0x200	Is Flying

By comparing these values with those extracted from the client process memory, the system can reliably determine whether a cheat has been activated. If such a condition is detected, appropriate countermeasures, such as logging, or an active disconnection, etc, can be taken against the offending player. This demonstrates how reverse engineering of client memory and protocol behaviour can be directly leveraged to enhance the effectiveness of anti-cheat mechanisms.

4.2.4 Conclusion

In this task, we examined the operation of anti-cheat systems by reverse engineering cheats and their corresponding detection mechanisms. The analysis demonstrated how

cheats exploit client systems through file modification, memory manipulation, and process injection, and how detection strategies such as integrity checks, memory validation, and client-server verification can be derived through reverse engineering.

The distinction between the reviewed cheat detection methods Section 4.2.2.1 and the new mechanism introduced in Section 4.2.2.1 lies primarily in the level of access required to the game client. Traditional approaches, such as those employed by proprietary anti-cheat systems like PunkBuster or Easy Anti-Cheat typically rely on memory, file, and process checks, but their integration generally requires access to the original source code or close collaboration with the vendor. As a result, these systems are not suitable for legacy or reverse-engineered games where such access is unavailable. In contrast, the mechanism described in Section 4.2.2.1 enables the implementation of cheat detection directly through reverse engineering. Instead of requiring source-level integration, we can design the anti-cheat system around the existing game binary. This approach allows developers to retrofit effective anti-cheat measures into closed or unsupported systems, thereby overcoming one of the central limitations of existing proprietary solutions.

```
├── WAR.exe
├── art.myp
├── art2.myp
├── art3.myp
├── audio.myp
├── binkw32.dll
├── data.myp
├── interface.myp
├── mft.myp
├── miles
│   ├── mssdolby.flt
│   ├── mssds3d.flt
│   ├── mssdsp.flt
│   ├── msseax.flt
│   ├── mssmp3.asi
│   ├── msssrs.flt
│   └── mssvoice.asi
├── mss32.dll
├── notes
│   ├── english
│   │   ├── button.jpg
│   │   ├── button_read-more.gif
│   │   ├── noteBanner.jpg
│   │   ├── notes.css
│   │   ├── patcher.css
│   │   ├── patcher_ad.jpg
│   │   ├── patcher_bg-main.jpg
│   │   ├── patcher_bg-tile.jpg
│   │   ├── patchnotes.xsl
│   │   ├── readme.html
│   │   ├── readme.xml
│   │   └── util.js
│   ├── french
│   │   ├── button.jpg
│   │   ├── button_read-more.gif
│   │   ├── noteBanner.jpg
│   │   ├── notes.css
│   │   ├── patcher.css
│   │   ├── patcher_ad.jpg
│   │   ├── patcher_bg-main.jpg
│   │   ├── patcher_bg-tile.jpg
│   │   ├── patchnotes.xsl
│   │   ├── readme.html
│   │   ├── readme.xml
│   │   └── util.js
```

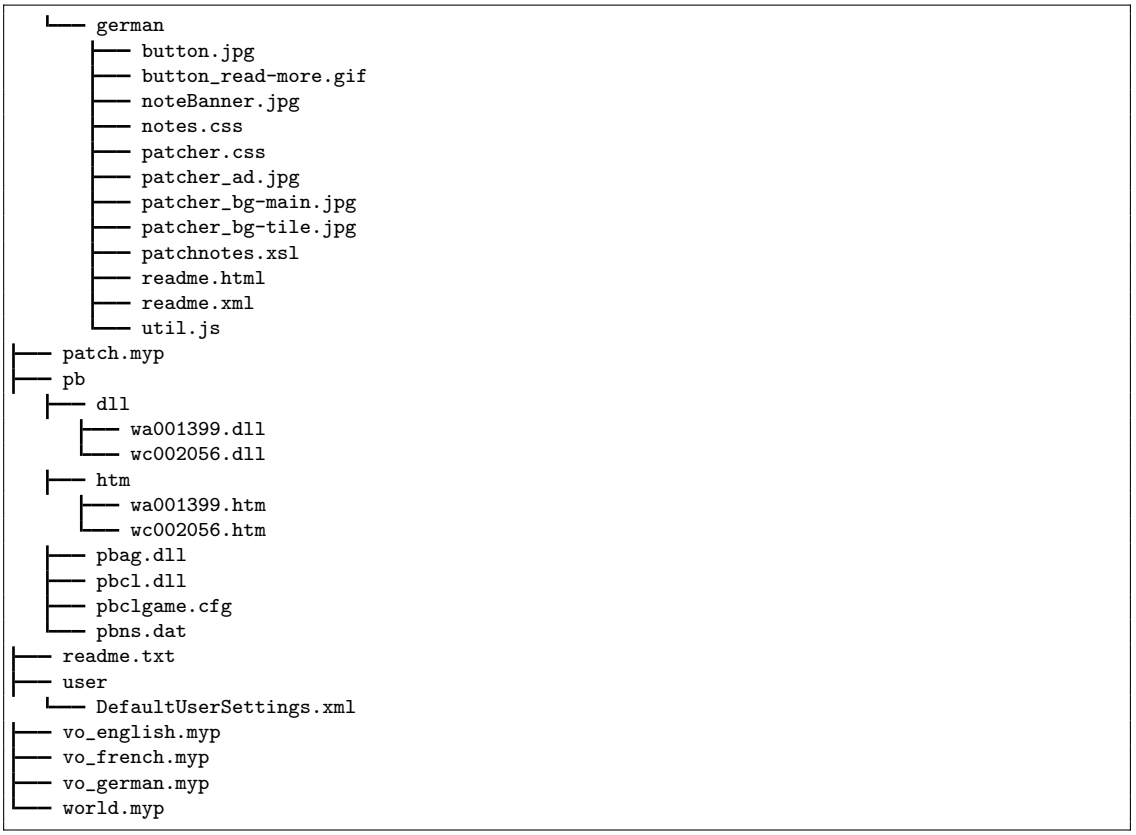


FIGURE 4.16: North American / European build 1.3.5 directory listing

```
├── 3N.mhe
├── BldInfo.ini
├── DirectX90c
│   ├── APR2007_XACT_x64.cab
│   ├── APR2007_XACT_x86.cab
│   ├── APR2007_d3dx10_33_x64.cab
│   ├── APR2007_d3dx10_33_x86.cab
│   ├── APR2007_d3dx9_33_x64.cab
│   ├── APR2007_d3dx9_33_x86.cab
│   ├── APR2007_xinput_x64.cab
│   ├── APR2007_xinput_x86.cab
│   ├── AUG2006_XACT_x64.cab
│   ├── AUG2006_XACT_x86.cab
│   ├── AUG2006_xinput_x64.cab
│   ├── AUG2006_xinput_x86.cab
│   ├── AUG2007_XACT_x64.cab
│   ├── AUG2007_XACT_x86.cab
│   ├── AUG2007_d3dx10_35_x64.cab
│   ├── AUG2007_d3dx10_35_x86.cab
│   ├── AUG2007_d3dx9_35_x64.cab
│   ├── AUG2007_d3dx9_35_x86.cab
│   ├── Apr2005_d3dx9_25_x64.cab
│   ├── Apr2005_d3dx9_25_x86.cab
│   ├── Apr2006_MDX1_x86.cab
│   ├── Apr2006_MDX1_x86_Archive.cab
│   ├── Apr2006_XACT_x64.cab
│   ├── Apr2006_XACT_x86.cab
│   ├── Apr2006_d3dx9_30_x64.cab
│   ├── Apr2006_d3dx9_30_x86.cab
│   ├── Apr2006_xinput_x64.cab
│   ├── Apr2006_xinput_x86.cab
│   ├── Aug2005_d3dx9_27_x64.cab
│   ├── Aug2005_d3dx9_27_x86.cab
│   ├── BDANT.cab
│   ├── BDAXP.cab
│   ├── DEC2006_XACT_x64.cab
│   ├── DEC2006_XACT_x86.cab
│   ├── DEC2006_d3dx10_00_x64.cab
│   ├── DEC2006_d3dx10_00_x86.cab
│   ├── DEC2006_d3dx9_32_x64.cab
│   ├── DEC2006_d3dx9_32_x86.cab
│   ├── DSETUP.dll
│   ├── DXSETUP.exe
│   ├── Dec2005_d3dx9_28_x64.cab
│   ├── Dec2005_d3dx9_28_x86.cab
│   ├── FEB2007_XACT_x64.cab
│   ├── FEB2007_XACT_x86.cab
│   ├── Feb2005_d3dx9_24_x64.cab
│   ├── Feb2005_d3dx9_24_x86.cab
│   ├── Feb2006_XACT_x64.cab
│   ├── Feb2006_XACT_x86.cab
│   ├── Feb2006_d3dx9_29_x64.cab
│   ├── Feb2006_d3dx9_29_x86.cab
│   ├── JUN2006_XACT_x64.cab
│   ├── JUN2006_XACT_x86.cab
│   ├── JUN2007_XACT_x64.cab
│   ├── JUN2007_XACT_x86.cab
│   ├── JUN2007_d3dx10_34_x64.cab
│   ├── JUN2007_d3dx10_34_x86.cab
│   ├── JUN2007_d3dx9_34_x64.cab
│   ├── JUN2007_d3dx9_34_x86.cab
│   ├── Jun2005_d3dx9_26_x64.cab
│   ├── Jun2005_d3dx9_26_x86.cab
│   ├── NOV2007_X3DAudio_x64.cab
│   ├── NOV2007_X3DAudio_x86.cab
│   ├── NOV2007_XACT_x64.cab
│   ├── NOV2007_XACT_x86.cab
```

```
    ├── NOV2007_d3dx10_36_x64.cab
    ├── NOV2007_d3dx10_36_x86.cab
    ├── NOV2007_d3dx9_36_x64.cab
    ├── NOV2007_d3dx9_36_x86.cab
    ├── OCT2006_XACT_x64.cab
    ├── OCT2006_XACT_x86.cab
    ├── OCT2006_d3dx9_31_x64.cab
    ├── OCT2006_d3dx9_31_x86.cab
    ├── Oct2005_xinput_x64.cab
    ├── Oct2005_xinput_x86.cab
    ├── Readme.txt
    ├── dsetup32.dll
    ├── dxdllreg_x86.cab
    ├── dxnt.cab
    └── dxupdate.cab
├── EHSvc.dll
├── HShield.dat
├── HackShield
    ├── 3N.mhe
    ├── BldInfo.ini
    ├── EHSvc.dll
    ├── HShield.dat
    ├── psapi.dll
    ├── v3pro32s.dll
    ├── v3warpds.v3d
    └── v3warpns.v3d
├── WAR.exe
├── art.myp
├── art2.myp
├── audio.myp
├── binkw32.dll
├── data.myp
├── interface
    ├── AddOns
    └── InterfaceCore
        └── Fonts
            ├── Uhei00m.ttf
            └── t_chinese
                ├── CoreFonts.xml
                └── Fonts.xml
├── interface.myp
├── libpatchui.dll
├── mft.myp
├── miles
    ├── mssdolby.flt
    ├── mssds3d.flt
    ├── mssdsp.flt
    ├── msseax.flt
    ├── mssmp3.asi
    ├── msssrs.flt
    └── mssvoice.asi
├── mss32.dll
├── patch.cfg
├── patch.myp
├── patch_user.xml
├── patcher
    └── splash
        ├── english
            ├── patch.html
            ├── war_bright.jpg
            └── war_dark.jpg
        └── t_chinese
            ├── Thumbs.db
            ├── patch.html
            ├── war_bg.jpg
            ├── war_bright.jpg
            └── war_dark.jpg
```

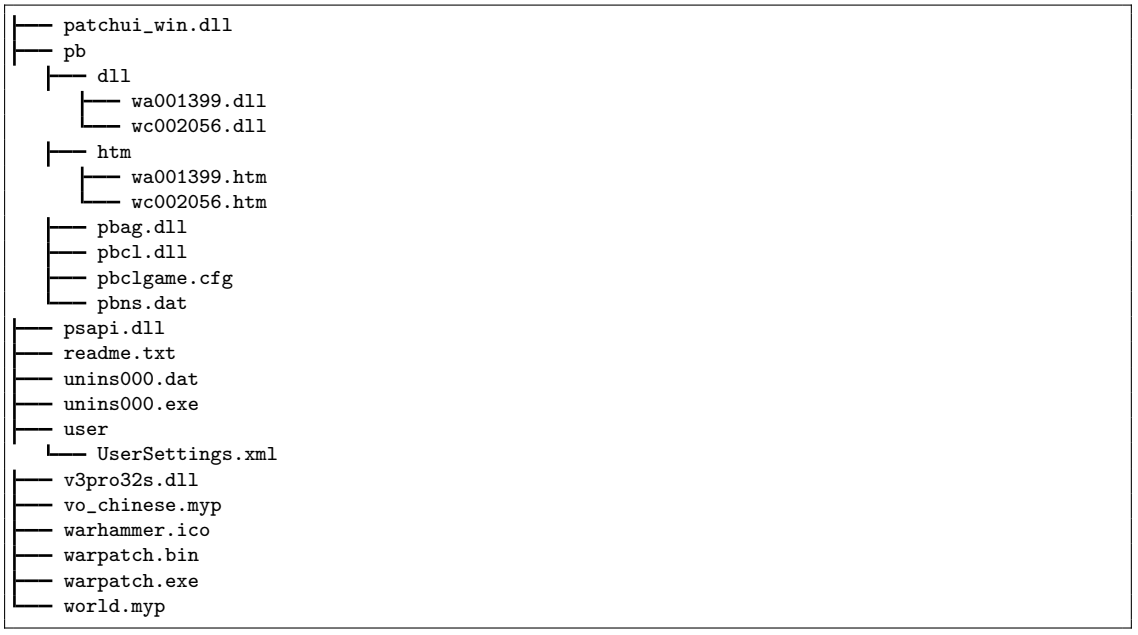


FIGURE 4.17: Asian build 1.2.1 directory listing

Chapter 5

Defining a Method for Reverse Engineering

5.1 Introduction

Reverse engineering is an important discipline in various domains, including software security, hardware analysis, software, and hardware preservation. The term has even been used to describe the process of working on existing code that the developer has not worked on for a while. However, unlike traditional engineering methodologies, which have well established processes, reverse engineering lacks a standardised approach.

Drawing an analogy to software engineering, where structured methods such as the Waterfall Model and Agile Development provide a framework for systematic development and maintenance, we can argue that reverse engineering would also benefit from a methodology and an analysis of potential methods.

The Waterfall Model is a linear and sequential approach to software development, where each phase, such as requirement gathering, design, implementation, testing, and maintenance, must be completed before moving on to the next. This structured methodology ensures complete documentation and planning, which makes it particularly useful for projects with well-defined requirements and minimal expected changes.

In the context of reverse engineering, a waterfall-like approach could involve systematically analysing a system in predefined stages, ensuring that each phase (such as initial reconnaissance, static analysis, dynamic analysis, and documentation) is fully completed before progressing. This structure could be beneficial in cases where the component to reverse is minimal in scope and the associated system is well documented and relatively static.

Agile development, on the contrary, emphasises flexibility, iterative progress, and continuous feedback. Instead of following a strict sequential process, Agile development cycles, often referred to as sprints, allow teams to make incremental changes, rapidly test hypotheses, and adapt based on new findings or changes to requirements. Some methods based on Agile Development methodologies, such as Scrum and Kanban, encourage frequent reassessment and collaboration, which helps manage uncertainty and evolving requirements.

Applying agile principles to reverse engineering could involve an interactive and adaptive approach, where the knowledge gained in one stage is fed back to earlier stages for refinement. This approach is particularly useful when dealing with complex, undocumented, or obfuscated systems, where initial assumptions may need to be revised as new information is uncovered.

In 2.4.1 we talked about the reverse engineering process undertaken for Dungeon Keeper and talked about how the process involved iterative reverse engineering. The process is a perfect example of an agile approach to reverse engineering in which the results of every sprint were released and evaluated, in which the results could be reflected in the next iteration of the process.

However, in 4.1 we perform an approach which could be considered more waterfall-like. First, we did a reconnaissance process where we looked at the state-of-the-art in archive file formats and performed a System-Level Analysis where we inspected files on the file system and remote resources before doing a Static File Analysis on the file format and finally documenting our findings.

A comparison could also be drawn to Unit Testing. Unit testing is the practice in software engineering where individual components of a system are tested in isolation to ensure that they function as expected. By systematically verifying small units of code, it allows developers to detect errors early, improve maintainability, and reduces software complexity. Similarly, in reverse engineering, breaking down a system into smaller components allows for more structured and manageable analysis. Just as unit testing helps developers test the behaviour of individual functions without needing to run the entire program, we can analyse isolated components, such as functions, network packets, or subsets of file formats, to deduce their functionality without requiring a complete analysis of the whole system. Just as unit tests provide reassurance that each individual component remains correct after modifications to a system, each reverse-engineered component contributes to a more comprehensive understanding of the system.

Therefore, we are proposing a method that provides a structured process for someone to follow when performing the reverse engineering process. Like software engineering, there will be many other possible methods for reverse engineering.

5.2 A Systematic Model for Iterative Deduction of Reverse Engineering

In order to propose a method for reverse engineering, we must first identify the different stages that make up this process. These stages represent key steps that systematically break down a target system, analyse its functionality, and build a model of its underlying logic. Identifying these stages allows us to define a structured method that can be applied consistently.

5.2.1 Acceptance Criteria

In order to know when the reverse engineering process is complete, we need to define an end point, the acceptance criteria. This acceptance criteria is when we have gained enough knowledge about the component to reverse to be able to achieve the desired outcome.

In the Introduction, we talk about the reverse engineering process of Microsoft Office formats. In this case, the acceptance criteria may be that a file created by Microsoft Office can be opened in the OpenOffice editor and that the text is readable and is formatted as it appears in the Microsoft Office software. Another desired acceptance criteria may be that the OpenOffice editor can create a file that, when opened in the Microsoft Office suite, can also be read and viewed as intended in the OpenOffice editor. However, there may be additional parts of the format, such as comments or editor history, which would not have been required to have been reverse engineered to achieve this acceptance criteria.

In Section 4.1 we reverse engineer archive files, and the acceptance criteria is that we were able to extract the files to the state in which they were before archiving. We could have had a further acceptance criteria in which we could place a new file into the archive, which could be read by the system as if it were placed in the archive by the original archiver. Another acceptance criteria may have been that we could fully unpack and repack the archive and it be sufficiently indistinguishable to the system. We may want to carry out this repacking process in order to defragment the archive or pack the most frequently used files at the beginning of the archive, where, if using a disc-based storage medium, we would have lower seek times.

In Section 4.2 we build a cheat detection system. It requires us to reverse engineer the game client so that we can run custom code to perform our cheat detection process. We do not need to reverse engineer the entire game client. The acceptance criteria is that we need enough knowledge to be able to call custom code in the game client from the network stack. We also reverse engineer individual cheats so that we can understand how they work so that we can create individual detection patterns for them. When we

reverse engineer a cheat in order to create a detection pattern, the acceptance criteria is simply that we understand how the cheat interacts with the game client to a sufficient level that we can effectively detect it running.

Therefore, the acceptance criteria do not require us to have full knowledge of the original system. Rather, the acceptance criteria only ask that we can use the knowledge gained to create a new replacement component, whether that be a data file, a system which reads a data file, or a system which interacts with other systems, which has a sufficient level of indistinguishability so that we can achieve our desired outcome.

A consequence of this is that recovering the original source code may not always be necessary or desirable. When reverse engineering, such as translating assembly code back into source code, the acceptance criteria do not always require an exact match to be the original instructions. This is because the source code may have been obfuscated or rearranged by a compiler. Instead, as long as the reverse engineered code produces the same behaviour, such as compiling to the same assembly instructions, it remains functionally identical. In some cases, it may even be preferable to obtain a more readable or natural representation of the code rather than its original obfuscated form.

Much like software development methods which include iterative development, such as Agile software development, reverse engineering can be done in a similar approach. Where the component to reverse is incrementally reverse engineered with difference acceptance criteria milestones.

We can then define the acceptance criteria as some level of knowledge about the system specified at the beginning of the process.

5.2.2 Boundaries

When reverse engineering a piece of software, hardware, or an entire system, it can be overwhelming, and one may want to reverse engineer a smaller component first. This is either because knowledge of the whole system is not important for reaching the acceptance criteria or because we are systematically breaking down the process into manageable chunks. Therefore, to systematically analyse a system, it is helpful to split the **component to reverse** from its **environment**.

The component to reverse is the subject of the reverse engineering process. This can be any self-contained system, subsystem, section of code, library, or file that operates within a broader environment. It possesses well-defined, sometimes obfuscated, functionality, logic, and dependencies that determine its behaviour.

We refer to the *component to reverse* as the specific component we are targeting and to the *environment* as everything external to the component that interacts with it. Examples of things which might form part of a component's environment include the

operating system in which a software component runs, the network and communication protocols that a program interacts with, any physical system connected to a hardware component, and user input which may influence behaviour within an application.

We need to be able to identify where a component ends and where the environment begins, which we refer to as the boundary. A boundary is a separation between the component and a part of the environment in which interactions occur. These interactions define how the component communicates with the external elements, and understanding these boundaries is essential in the reverse engineering process. Boundaries can be identified by examining where data passes between the component and its environment through system calls, network datagrams, and dependencies.

In Section 3.5.2 we reverse engineer the University of Southampton door access system to identify the process by which access to buildings is verified. We identify that the boundary exists across the wireless communication for which the protocol is defined in ISO/IEC 14443-1 Type A. When observing these interactions, we are able to see what requests the readers send to the card including requesting identifiable information from it or accessing secure sectors. We are able to compare this trace to other systems that used the card around the university, as well as other systems, as described in [9].

In Section 4.1 we reverse engineer archive files, and identify that the target for the reverse engineering process and, therefore, the component to reverse is the '.MYP' files located in the directory. As this is a data file, we deduce that the immediate interactions come from system calls reading and writing the file. Through disassembly of the executables, we identify that warpatcher.exe and WAR.exe perform file read and write operations to the file. We then examine the operations that the executables perform on the data read from the file. These interactions in the data are identified as the boundary where we separate the component to reverse and the environment. Observing these interactions allows us to build up a structure of the file which is required to be able to reach the acceptance criteria of being able to extract files from the archive or be able to place a new file into the archive as if it were performed by the original archiver. In this case, additional operations are performed by the warpatcher.exe; however, this boundary is not investigated as it was not required in order to achieve our acceptance criteria.

In Section 4.2 we reverse engineer part of the game executable to be able to call custom code inside a DLL from data we send over the network. Here, identifying the component to reverse was a little more complicated. In this process, we identify a boundary that we want to take advantage of, which is the calling of the external functions presented by pbcl.dll, which are **ca** and **cb**. We also find another boundary, which is the network layer in which we send data, and we find a control flow that connects the two boundaries inside the executable. The component to reverse in this case are the functions inside the executables which connect these two boundaries. Reverse engineering these functions allows us to construct a model of the datagram and instantiate one which follows the

correct control flow to reach the other side, which is the other boundary. It is important to note that in this case, although we reverse engineer part of the executable, we isolate a subsystem of it which we identify as our component to reverse.

By clearly identifying the component to reverse and separating it from its environment, we can systematically approach the reverse engineering process. The boundary between these two defines where the interactions occur, and analysing these interactions allows us to understand the component's functionality without needing to reverse engineer the entire system. In both examples, isolating the component to reverse enabled a more focused analysis, ensuring only the necessary parts were investigated to achieve the acceptance criteria.

5.2.3 Traces & Model

In this section, we introduce the concept of traces that are used to construct models. Since interactions define how a component communicates with its environment, traces are fundamentally a collection of observed interactions. These interactions occur at the boundary between the component to reverse and its environment and can be recorded as sequences of observations.

A model is a structured representation of these traces, typically organised by grouping interactions related to high-level calls. A model is a representation of our component to reverse as best we understand it. By focusing on the external interactions of a component represented in a model, this allows us to analyse the behaviour of the component without necessarily reconstructing its entire internal logic. It enables us to make predictions about how the component might react to new inputs or changes to its environment.

Note that the interactions between a component and its environment are symmetric in nature. Similarly, as we build a model of the component by observing its interactions, we are simultaneously constructing a model of the environment. The environment of a component consists of other components, and these roles can be reversed. What we consider an environment in one context may be a component in another. By analysing the same traces, we can gain insights into both the component and its surrounding system, effectively modelling the entire interaction space.

Interactions may include system calls such as file I/O, network messages, library function calls, or hardware signals. The more interactions we capture, the more complete the model becomes. By analysing these traces, we gain insight into how the component behaves within its environment, allowing us to document its functionality without direct access to its source code or internal design.

To construct a model, we first need to observe and collect traces. Since these interactions occur at boundaries, capturing the traces requires monitoring the points where the data,

control flow, or signals pass between the two. Traces can be collected using various techniques, depending on the type of component to be reversed. If the component to reverse is a file, we might want to look at system calls, memory allocations, and file operations using tools like strace, procmon, or various other debugging suites. If the boundary is in a networking layer, we can use tools such as Wireshark to sniff the packets sent over the network for analysis. If a boundary is on a call to a function in a DLL we would need to be able to intercept these calls which can be done with a debugger or hooking onto the functions using a detour.

There may be challenges in collecting the traces. If the boundary is over a network connection it would not be uncommon for the communication to be through another protocol such as HTTP in a rest-style request, if this is the case the payload may be encoded in JSON (JavaScript Object Notation) notation. It is also common for the communication stream to be encrypted. If using HTTP, the communication may be done over an SSL connection as a HTTPS request, which means the traffic cannot be inspected using a tool such as Wireshark. However, tools which provide a man-in-the-middle attack can perform an SSL termination and relay the traffic allowing the unencrypted traffic to be viewed. If the network communication is a custom protocol over TCP or UDP further reverse engineering may need to be done to identify the encryption method used. It may be possible to create a detour hook which happens before encryption for outgoing traffic or after decryption for incoming traffic so the actual payload can be inspected.

Once the traces have been collected, the next step is to organise and interpret them within a structured model. A model is not simply a raw collection of traces. It is an abstract representation that captures the key behaviours and relationships observed in the interactions. First, we must identify sequences of traces that correspond to higher-level behaviours, such as authentication, message transfer, or file access. We then recognise common sequences or dependencies between traces that define how the component interacts with its environment. Finally, we map the raw interactions to meaningful information, such as fields in a request or parts of a file in data retrieval. We can then compare the model with additional encoded traces to validate our model. By iteratively refining the model with more traces, we improve its accuracy and completeness. This makes it a powerful tool for analysis.

Once an initial model has been constructed, it is possible to instantiate an instance of the model, effectively creating a mock version of the component to reverse. This allows us to test assumptions, verify behaviours, and even manipulate the system without needing the original component. For example, if the reversed component is a proprietary archive format, such as in 4.1, the model derived from the observed traces can be used to approximate this format, we can then create a new archive file using that approximation format. Similarly, if we are analysing an archive update mechanism, we can use our model instance to inject new data into an existing archive and observe how the system

reacts. By iterating on this process, we can refine the model further, creating new instances of the model, ensuring that our understanding is accurate and complete.

In Section 4.2 we reverse engineer a game cheat to analyse how it interacts with the game executable to perform a *speed hack*. In this case, we observe that the cheat takes advantage of the Win32 API, specifically the base services kernel32.dll, which provides functionality for processes and threads. The trace consists of the cheat reading a section of memory that contains a pointer to some memory responsible for the player's attributes and then writing some data to the structure in memory. This structure contains a value for controlling the speed of the player. Since the cheat performs only a few distinct actions, the resulting model is relatively simple, consisting of only a handful of traces. This highlights that the complexity of a model is directly tied to the number and variety of interactions observed. Some systems will require extensive tracing, while others can be sufficiently understood with just a few key interactions.

We emphasise the importance of traces, which are the sequences of interactions that occur at the boundary between the component to reverse and its environment. A model is then built from these traces, providing a structured representation that allows us to analyse the component's behaviour without fully reconstructing its internal logic.

5.2.4 Acquiring Knowledge

Understanding a system without direct access to its internal logic requires a structured approach to acquire knowledge. Reverse engineering is fundamentally a knowledge acquisition process. The goal is to extract meaningful insights from an unknown system, incrementally building an understanding that allows us to explain or manipulate its behaviour.

Acquiring knowledge is an iterative process: **Run**, **Observe**, **Synthesise**, **Repeat**. This methodology allows us to systematically explore an unknown system, extract meaningful interactions, and build an abstract representation of its behaviour. The first step is to provoke knowledge from both the component and its environment by actively interacting with the system. This interaction can take various forms, such as executing a program, triggering specific functionality, or injecting input to elicit a response. Each action serves to extract knowledge, revealing how different parts of the system behave in isolation and in relation to one another. We then monitor and collect traces of these interactions, logging system calls, capturing network packets, or tracing function execution, to build a clearer picture of the systems operation. Finally, we analyse the collected data, identify patterns and construct a model that abstracts unnecessary details while preserving the fundamental logic required to understand and predict system behaviour. This cycle is repeated iteratively, refining our understanding with each pass as we continuously update our knowledge of both the environment and the component.

A comparison could be drawn to unit testing, where we run test cases on individual components in isolation to evaluate specific functionality. In reverse engineering, we execute or interact with the component to provoke meaningful responses. In unit testing, we use test assertions to define expected behaviour, effectively creating a specification for how the component should function. In reverse engineering, we construct a model that generalises the observed traces, allowing us to infer hidden logic and predict system behaviour under different conditions.

Acquiring knowledge is not a one-directional process. It involves iteratively flipping the roles of the component and its environment to refine our understanding. By running the instance of the component with different environments or creating controlled instances of the environment to interact with a fixed component, we generate new traces that reveal additional insights. Similarly, using different instances of the component or instantiating more controlled instances of the component to reverse can also reveal more knowledge. This two-way exploration ensures that our model accurately represents both the component and the broader system with which it interacts. The ability to instantiate new versions of the component or environment is crucial, as fixed tests with a limited set of interactions constrain the scope of the knowledge that can be extracted. Expanding the range of instances and interactions allows us to develop a more comprehensive and accurate model of the system.

In Section 3.5.4 we reverse engineer the function that generates the decryption key for sectors in the University of Southampton ID card. By analysing sector keys from multiple cards, the study reveals that, although the keys differ, there are consistent patterns in certain bytes. Instead of treating the keys as purely random, the analysis identifies that a pattern is constant, and others are derived from the UID. The model allows for the calculation of unknown keys from the UID without needing access to the card issuer's system. This shows that building a model on a system's behaviour allows us to reconstruct the function without knowledge of its internal logic.

The acquisition of knowledge in reverse engineering is a structured iterative process that involved running the system, observing its interactions, and synthesising insights into a coherent model. By running the component under controlled conditions and capturing traces, we can gradually reveal how it behaves in response to different inputs. These traces are then analysed to identify patterns and construct an abstract representation that explains the system's operation.

5.3 The Process

We are now in a position to explain the iterative process.

Step 0. Initial Model $M_C(K), M_E(K)$

The process begins with an initial model of both the component to reverse (M_C) and its environment (M_E), each based on an initial set of knowledge (K). At this stage, the models provide a structured but incomplete understanding of the system, derived from prior observation, documentation, or assumptions. These models serve as a starting point for further refinement, guiding how we test and expand our knowledge of the system's behaviour. This prior observation could be for example from running the actual component to reverse in its actual environment.

Step 1. Instantiate Model $I \in M_C, M_E$

To generate new traces, we instantiate an instance (I) from either the component or the environment model. This could involve creating a real or simulated version of the system under investigation. Running an instance allows us to observe how it interacts with its counterparts, whether that be the actual environment surrounding a component or a mocked environment that replicates expected behaviour. In doing so, we prepare the system for controlled testing, ensuring that meaningful interactions occur.

Step 2. Test Instances against $I \leftrightarrow A_E$ or $I \leftrightarrow A_C$ - Learn K'

The instantiated instance is then tested by running it against an actual component (A_C) or an actual environment (A_E). This interaction may generate new traces that capture how the system responds to various inputs and conditions. By analysing these traces, we can extract new knowledge (K') about how the system behaves in different scenarios. The flexibility of this step allows for two-way validation, meaning that we can test the component against the actual environment or vice versa, expanding our understanding from different perspectives.

Step 3. Synthesise knowledge $K, K' \Rightarrow K''$

The newly acquired knowledge (K') is then synthesised with the existing knowledge (K) to produce a refined, more comprehensive understanding (K''). This involved identifying patterns, generalising behaviours and incorporating new insights into the model while filtering out unnecessary details. The goal is to construct an abstract representation that accurately reflects the system's operation, allowing us to infer hidden logic and predict future behaviour.

Step 4. Revise Models $M_C(K'')$ and $M_E(K'')$

Once the knowledge has been synthesised, the models of the component (M_C) and environment (M_E) are updated to reflect the improved understanding (K''). This revision ensures that our models become increasingly accurate over successive iterations. By continuously refining both the component and environment models, we bridge gaps in our knowledge and create a more reliable representation of the system.

Step 5. Repeat Process Until Knowledge Threshold Is Reached

The process repeats iteratively, each cycle expanding the knowledge base through new traces and refinements. More instances can be instantiated to create additional traces, helping to validate findings and uncover hidden behaviours. However, the ability to generate new instances is crucial, if the system is only tested with fixed examples and no new instances are created, the knowledge gained will be inherently limited. The iterations continue until the accumulated knowledge (K'') reaches a predefined threshold (K_{ac}), at which point the process is considered complete, and the models provide a sufficiently detailed understanding of the system.

We illustrate the above discussion with a flow diagram.

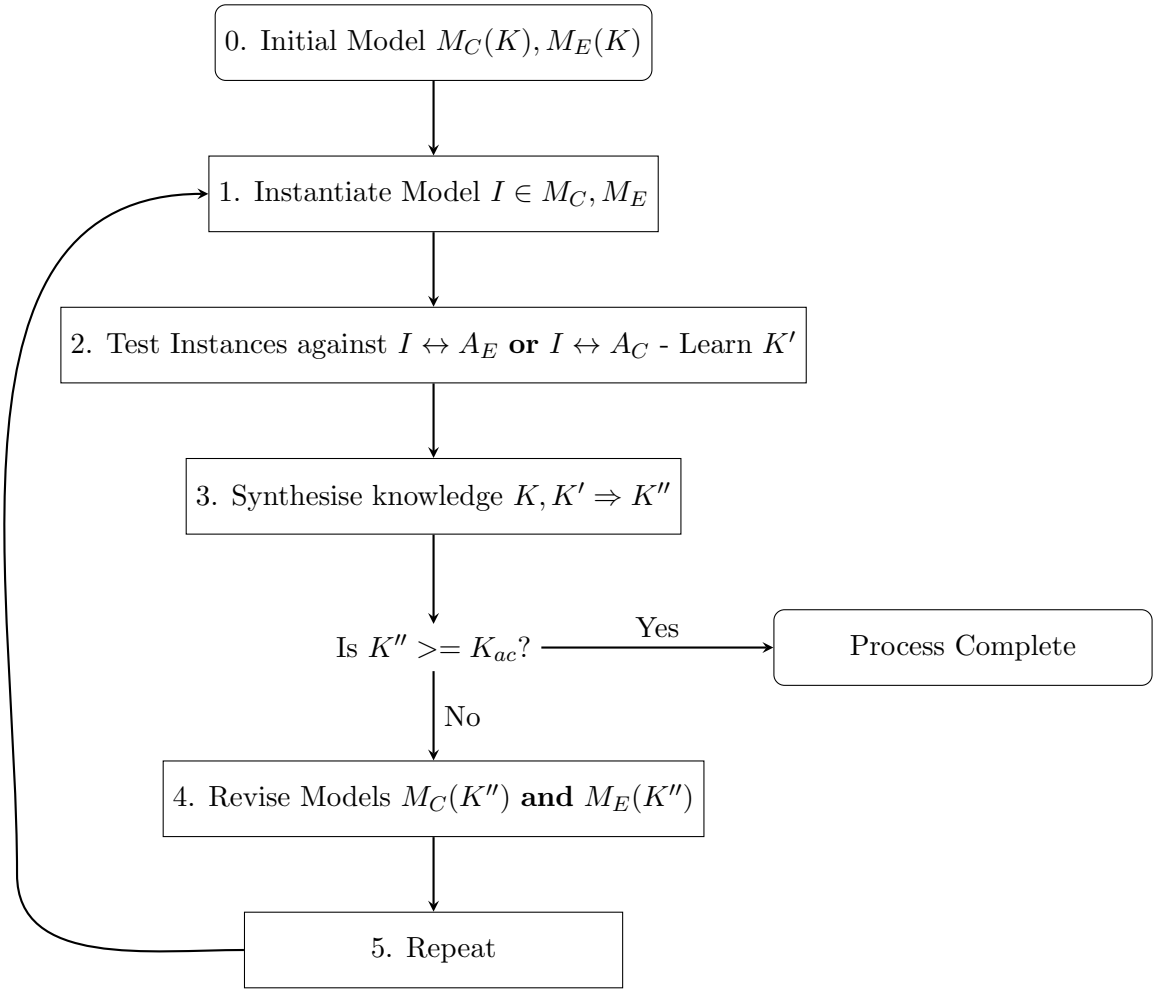


Table 5.1 Definition of Symbols Used in the Flowchart

Symbol	**Definition**
K	Initial knowledge set
K'	Learned knowledge from testing
K''	Synthesized knowledge after processing K and K'
K_{ac}	Acceptance threshold for knowledge synthesis
M_C, M_E	Models of component or environment
A_C, A_E	Actual component or environment
I	Instance of a model

5.4 Examples

We conclude this chapter with a worked examples showing the process.

5.4.1 Network Datagram

In this example, we reverse engineer part of a particular video game and how it interacts with the servers.

In this video game, clients receive packets of information from game servers, effectively making the game client the consumer of network packets and the server the producer. Since we are reverse engineering these packets, our focus is on understanding how the client interacts with them. In this context, the consumer (the game client) serves as the environment, while our goal is to develop a method for producing network packets that mimic the server's output. However, we are not intended to reverse engineer the entire game server, only the portion of its code responsible for generating these packets.

In Figure 5.4 we present a network packet viewed in a hex viewer. An initial inspection reveals that it contains 1,884 bytes. We can send this initial packet to reverse without modification to the game client to see what it does and observe what happens.

Acceptance Criteria

To ensure the accuracy and functionality of our reverse-engineered packet model, we define a set of acceptance criteria. First, the structural integrity of the packet must be maintained, which means that all fields adhere to the expected encoding rules and are correctly positioned within the payload. One way to verify structural integrity is to insert the same data as our original component to reverse into our serializer and compare its output to the original packet. If the generated binary matches exactly, this confirms that our model correctly reconstructs the packet format.

Moreover, the model must exhibit functional correctness by ensuring that modified or synthesised packets do not introduce unwanted behaviour. This game client should process our changes without crashing or showing corrupted information. Reproducibility is also essential, and given the same input, the packer serializer must generate identical binary output, and the game client should consistently process these packets without variation.

In this case, our primary goal is to modify the text fields, title, subtitle, left-hand text, and right-hand text, and observe that these changes are correctly displayed in the game client. Successfully achieving this confirms that our model aligns with the actual packet format, meeting our acceptance criteria and allowing for further refinement as needed.

Initial Model

FIGURE 5.1: Result of sending initial packet to reverse to game client

The data presented contains a lot of structured information, including a significant amount of textual data. At the top, the title "Stonetroll Slaughter" is prominently displayed, followed by the subtitle "Weekend Warfront". Below the title, an image is accompanied by a paragraph of descriptive text. Further down, a reward section is visible, organised into three categories: Basic, Advanced, and Elite, each containing one or more associated items. On the right-hand side, an additional body of text provides further details. Finally, at the bottom, four tasks are listed, each accompanied by a brief description and a counter, which likely tracks the progress or completion status of each task.

The first two bytes of the packet are 0x0759, which corresponds to 1,881 in decimal notation, exactly three bytes less than the total size of the packet. This suggests that the first three bytes form a header that describes the rest of the payload. Given that the first two bytes specify the payload size, the remaining byte is likely an identifier that provides additional context about the payload's content or purpose.

The first identifiable string, "Stonetroll Slaughter", is located at an offset of nine bytes from the start of the packet and is 20 bytes in length. Given that each character is represented by a single byte, we can infer that the text is encoded using a single byte character encoding such as ASCII or extended ASCII, or potentially a variable-length encoding like UTF-8. Furthermore, the byte at offset eight has a hexadecimal value of 0x14, which corresponds to 20 in decimal notation. From this we can reasonably hypothesise that this byte serves as a length indicator, specifying the number of bytes as part of

the string. This type of length-prefixed encoding for a string is usually referred to as a Pascal string [158] [159].

From this we can establish an initial model where the first two bytes represent the packet size, followed by an identifier marker in the header. Furthermore, at the eighth position, we observe a Pascal string containing the title message. Although we could analyse each field in detail, these initial observations provide sufficient information to test our initial hypothesis.

Table 5.2 Initial Model of Packet Structure

Field	Size (bytes)	Description	Initial Value
Size	2	Size of payload	0x759
Opcode	1	Payload identifier	0x95
Unknown	5	Unknown data	0x0A15000000
TitleLength	1	Length of title	0x14
Title	Variable	Title text	“Stonetroll Slaughter”
...	Variable	Rest of data	...

Instantiate Model

With our initial model established, we can now create a instances based on it. However, manually constructing a binary packet is complex and time-consuming. To streamline this process, we developed a tool that allowed the synthesis of new instances of components using our model. This tool functions as a serializer, allowing us to iteratively update the packet structure as we reverse engineer additional fields, ensuring greater flexibility in our analysis process.

```
Packet packetOut = new Packet();

// Size of payload
packetOut.WriteUInt16(0x0759); // 0x00

// Payload identifier
packetOut.WriteByte(0x95); // 0x02

// Payload Start
packetOut.WriteByte(0x0A); // 0x03
packetOut.WriteByte(0x15); // 0x04
packetOut.WriteByte(0x00); // 0x05
packetOut.WriteByte(0x00); // 0x06
packetOut.WriteByte(0x00); // 0x07
packetOut.WritePascalString("Stonetroll Slaughter"); // 0x08

// Rest of packet data
```

Listing 5.1: Initial iteration of component synthesiser

In this instance, we generate multiple instances from our model, varying the string length to observe potential patterns or limitations. We specifically chose powers of two and the value immediately before it, 31, 32, 63, 64, 127, 128, 255, 256, 511, and 512, as these

values often represent boundaries or constraints in data structures. We began with 32 arbitrarily and update the Pascal string writer accordingly, modifying the value at offset 8 to reflect the new string length and overwriting the existing title. The remaining payload remains unchanged and is appended after each modified string. Additionally, the first two bytes are adjusted to account for the varying sizes introduced by different string lengths. A visualisation of one of these newly created instances, generated by our packet synthesiser, is presented below.

[Server] packet : Size = 1896

```
|-------------------------------------------------|----------------|
|00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F |0123456789ABCDEF|
|-------------------------------------------------|----------------|
|00 00 95 0A 15 00 00 00 20 41 42 43 44 45 46 47 |........ ABCDEFG|
|48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 |HIJKLMNOPQRSTUVW|
|58 59 5A 31 32 33 34 35 36 10 57 65 65 6B 65 6E |XYZ123456.Weeken|
|64 20 57 61 72 66 72 6F 6E 74 F6 01 53 6D 61 6C |d Warfront..Smal|
|6C 20 63 6F 6E 74 69 6E 67 65 6E 74 73 20 6F 66 |l contingents of|

...
-------------------------------------------------------------------------
```

FIGURE 5.2: Packet generated using packet synthesiser with title of length 32

Test Instances

Now that we have instantiated our model, we need to test its validity. Sending the synthesised packets to the game client will help determine whether our hypothesis that the string follows the Pascal string format is correct. With our first instance, we expect the title of the pop-up to reflect our new value, "ABCDEFGHIJKLMNOPQRSTU-VWXYZ123456". If the change is successfully reflected, it confirms our assumption. However, if the game client crashes or displays unexpected information, we still gain valuable insights, indicated that our model requires further refinement.

FIGURE 5.3: Result of sending synthesised packet with title of length 32 to game client

Here are the results of our various synthesised packets.

Length	Result	Comment
31	Success	-
32	Success	-
63	Success	-
64	Success	-
127	Success	-
128	Fail	Window does not show
255	Fail	Window does not show
256	Fail	Window shows all fields incorrect

FIGURE 5.4: Results of testing synthesisted packets in first iteration

Synthesise Knowledge

Since the title has been updated for multiple test cases, our hypothesis holds for certain string lengths. Testing values of 31, 32, 63, 64, 127, 128, 255 and 256, we observed that lengths up to 127 functioned as expected, while 256 failed, likely due to the unsigned byte limit. However, results for 128 and 255 are unprecedented and require further analysis to determine if any unexpected behaviour occurs. This suggests that our understanding of the string format is generally correct, but additional testing is needed to refine our model. Furthermore, as we have only reverse engineered one field, we have not yet met our acceptance criteria, so we must continue iterating and expanding our model.

When applying the same approach to the left-hand text, we observe a key difference. Instead of a single preceding byte, there are two, specifically 0xF601. In particular, the length of this text is 246 bytes. A similar pattern appears in the right-hand text, which also has two preceding bytes and a length of 254 bytes. This suggests that longer text fields may use a different length encoding scheme, potentially allowing for larger values than a single byte can represent. The length of the left-hand text is 246 bytes which in hexadecimal notation is 0xF6 and similarly for the right-hand text 0xFE. These sizes as they are below 256 (the maximum integer value stored in an unsigned byte) we expect them to only to be needed to be stored in a single byte but as we can see they are stored in two.

From our synthesised packets, we observe that when the most significant bit 0x80 is set in the title length field, the window does not display, suggesting that the game client is unable to parse the packet correctly. Examining other existing fields with lengths exceeding 128 characters, we notice a pattern that when the most significant bit is set, an additional byte follows. Interestingly, while the first byte has its most significant bit set, the subsequent byte has its least significant bit set. This suggests that the length field may use a variable-length encoding scheme, where each byte contributes 7 bits of the total length, while a control bit signals whether another byte should be read. Based on this pattern, we can hypothesise that the original packet producer encodes the length using a continuation bit mechanism, allowing for flexible field sizes while maintaining effective encoding.

To gain a deeper understanding of how the game client interprets and processes our packet, we need to trace the interactions it makes with the received payload. Since network calls act as the boundary between external data and internal processes, we can use them as entry points for analysis. By monitoring how the client reads, parses and manipulates the packet data, we can infer structural patterns and decoding mechanisms. Additionally, by analysing traces of execution and identifying meta-interactions, such as function calls, conditional branches, and memory modifications, we can refine our model of the packet format and the client's expected input structure.

By opening the executable in an assembly debugger such as The Interactive Disassembler (IDA), we can examine the x86 assembly code of the compiled game client. Additionally, by attaching the debugger to the running program, we can set breakpoints and step through individual lines of assembly code. Within IDA, we can easily identify calls to recv and recvfrom, which are functions of Windows Networking API of winsock.h, located in sub_4AF801. Tracing these calls further leads us to sub_4C30CE, which contains a jump table that switches execution based on the opcode byte. When the program processes our specific payload identifier 0x95, it directs execution to 0x004C43A8, which then quickly calls sub_4C00D6. This function appears to be the main handler responsible for processing this specific payload. Inside this function, we identify another jump table that operates on the byte immediately following the opcode at an offset of 4. This reveals

```
char __thiscall sub_992AA0(_DWORD *this, _DWORD *a2)
{
  _DWORD *v2; // eax
  unsigned int v3; // ecx
  unsigned int v4; // ebx
  char v5; // di
  int v6; // esi
  int v7; // edx
  unsigned int v8; // edx
  char result; // al

  v2 = this;
  *a2 = 0;
  v3 = this[2];
  v4 = v3 + 5;
  if ( v3 + 5 > v2[3] )
    v4 = v2[3];
  v5 = 0;
  if ( v3 >= v4 )
  {
LABEL_7:
    *(v2 + 24) = 1;
    result = 0;
  }
  else
  {
    v6 = v2[5];
    while ( 1 )
    {
      *a2 |= (*(v6 + v2[2]) & 0x7F) << v5;
      v6 = v2[5];
      v7 = v2[2];
      if ( *(v6 + v7) >= 0 )
        break;
      v8 = v7 + 1;
      v5 += 7;
      v2[2] = v8;
      if ( v8 >= v4 )
        goto LABEL_7;
    }
    ++v2[2];
    result = 1;
  }
  return result;
}
```

FIGURE 5.5: WAR.exe opened in IDA Pro at sub_992AA0

that the field at this position is a single byte in size and can be classified as a sub-opcode. Additionally, we observe that the subsequent four bytes represent an unsigned integer whose purpose remains unknown, but we can update our model to reflect its field size. Finally, we notice that the function that processes next section of the payload. This code invokes the function sub_992AA0 four times, corresponding to the four expected string fields. This function appears to return the length of each string before reading its contents.

Analysing the code in Figure 5.5, it appears to decode data stored in a variable length encoding method to an integer. In this function, we take the lower 7 bits which hold part of the data, if the value is greater than 128 or, in other words, the most significant bit is set, then we know that there is another byte to follow. We are able to write a function that encodes a value in the same format so that it can be unpacked into our expected value. The WriteVarUInt32 function efficiently encodes a 32-bit unsigned integer using a variable length format by processing it in 7-bit chunks. If the value is 0, it writes a single byte 0 and exits. Otherwise, it extracts the lowest 7 bits, sets a continuation flag if more data remain, and writes the byte. The value is then right-shifted by 7 bits, and the process repeats until all significant bits are written. This method reduces the storage space for smaller numbers while ensuring that larger numbers are still fully represented.

```
procedure WriteVarUInt32(val)
    if val = 0 then
        WriteByte(0)
        return
    end if
    while val > 0 do
        WriteByte((val AND 0x7F) XOR (val > 0x7F ? 0x80 : 0x00))
        val ← val ≫ 7
    end while
end procedure
```

Figure 5.6: Pseudocode for WriteVarUInt32

```
procedure WriteVarUIntString(str)
    if str = null or Length(str) ≤ 0 then
        WriteByte(0)
        return
    end if
    bytes ← UTF8_Encode(str)
    WriteVarUInt32((Length(bytes))
    Write(bytes, 0, Length(bytes))
end procedure
```

Figure 5.7: Pseudocode for WriteVarUIntString

Revise Model

With our analysis confirming the presence of variable-length encoded Pascal-like strings, specifically the VarUIntString format defined in Figures 5.6 and 5.7, we can refine our packet model accordingly. This structure has been validated across multiple string fields, including title, subtitle, left-hand text, and right-hand text, demonstrating consistency in how the client processes these values. As a result, we can now update our model to accurately represent this encoding scheme and incorporate these insights into future iterations of our reverse engineering process.

Table 5.3 Revised Model of Packet Structure

Field	Type	Size (bytes)	Description	Initial Value
Size	ushort	2	Size of payload	0x759
Opcode	byte	1	Payload identifier	0x95
Sub-opcode	byte	1	Payload identifier	0x0A
Unknown	uint	4	Unknown data	0x15000000
Title	VarUIntString	Variable	Title text	“Stonetroll Slaughter”
SubTitle	VarUIntString	Variable	Subtitle text	“Weekend Warfront”
LeftText	VarUIntString	Variable	Left text	“Small contingents of Order...”
RightText	VarUIntString	Variable	Right text	“The Stonetroll Crossing sce...”
...	Variable	Rest of data	...	

We can also update our packet synthesiser to reflect our new hypothesis.

```
Packet packetOut = new Packet();

// Size of payload
packetOut.WriteUInt16(0x0759); // 0x00

// Payload identifier
packetOut.WriteByte(0x95); // 0x02

// Payload identifier
packetOut.WriteByte(0x0A);

// Unknown
packetOut.WriteUInt(0x15000000);

// Title
packetOut.WriteVarUintString("Stonetroll Slaughter");

// Subtitle
packetOut.WriteVarUintString("Weekend Warfront");

// Left text
packetOut.WriteVarUintString("Small contingents of Order..");

// Right text
packetOut.WriteVarUintString("The Stonetroll Crossing sce...");

// Rest of packet data
```

Listing 5.2: Second iteration of component synthesiser

Instantiate Model

Now that we have completed one iteration of the method, we can leverage our revised model to repeat the process. With our updated packet synthesiser, we can regenerate the same test cases while ensuring that they conform to our refined understanding of the packet structure. In the following, we visualise one of our newly synthesised packets featuring a title that exceeds 128 characters in length. Notably, the length field now consists of two bytes, indicating the use of a variable-length encoding scheme to accurately represent the string length.

Test Instances

Now that we have generated instances of our revised model, we can send them to the client to evaluate our hypothesis. Specifically, we expect that packets containing strings lengths of 128, 255, and 256 will now be processed correctly, as well as those that passed in the previous iteration. By observing the client's response, we can confirm whether our updates to the length encoding mechanism align with the expected behaviour, further validating our model.

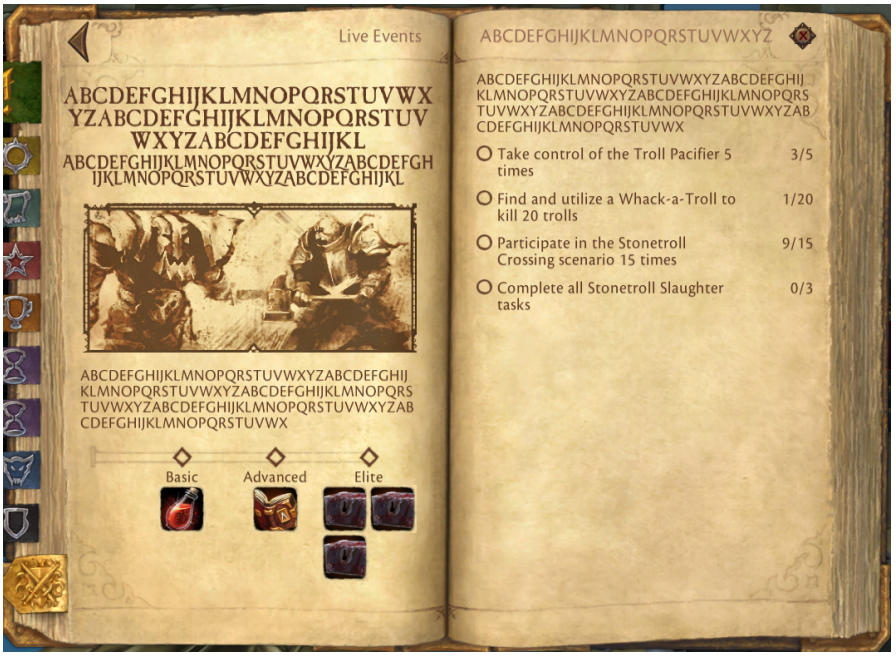


FIGURE 5.8: Result of sending synthesised packet with title of length 128 to game client

Here are the results of our various synthesised packets.

Length	Result	Comment
31	Success	-
32	Success	-
63	Success	-
64	Success	-
127	Success	-
128	Success	-
255	Success	-
256	Success	-

FIGURE 5.9: Results of testing synthesised packets our second iteration

Synthesise Knowledge

The results presented in Figure 5.9 indicate that all test cases for our newly generated instances were successful. This confirms that our hypothesis regarding the updated length encoding mechanism holds, further validating the accuracy of our revised model. As we have now been able to freely modify title, subtitle, left-hand text, and right-hand text, and the structural integrity has been preserved and the results are functionally correct, we have therefore met our acceptance criteria. As we have now achieved our acceptance criteria, we have completed the reverse engineering process. However, we will continue the process and try and reverse engineer the tasks on the right-hand side.

Revise Model

Examining the raw packet data, we identify that the text for the first task begins at offset 1,320. The observed text, "Take control of the Troll Pacifier 5 times", is immediately followed by a single byte and then the extended text, "Take control of the Troll Pacifier 5 times during the Stonetroll Crossing scenario". These strings are followed by eight bytes set to zero. Furthermore, we observe the values 0x03000000 and 0x05000000, which correspond to the tasks completed fields. At offset 1,319, we identify a variable-length encoding byte associated with the first text. Preceding this we find four bytes with the value 0x08520000, which translates to 21,000 in decimal notation. A similar pattern is found before the second task, where the corresponding value is 0x09520000 (21,001 in decimal), suggesting that this field may represent a unique identifier for each task. At offset 13,14 we also observe the byte 0x04 which corresponds to the number of tasks we have. As a result, we can now update our model to account for these newly discovered task fields.

Table 5.4 Model of a task contained within the component to reverse

Field	Type	Size (bytes)	Description	Initial Value
Id	uint	2	Task identifier	21,000
Text	VarUIntString	Variable	Task Text	"Take control of the..."
ExtendedText	VarUIntString	Variable	Extended Task Text	"Take control of the...'
Unknown	bytes	8	Unknown data	0x0000000000000000
Current	uint	4	Current amount	3
Total	uint	4	Required amount	5
Unknown	bytes	6	Unknown data	0x010000000000

Instantiate Model

With our updated model in place, we can begin the third iteration of our reverse engineering process. This iteration will allow us to refine our understanding further and validate our previous findings. Figure 5.3 illustrates the updated packet synthesiser,

which has been modified to align with our revised model. By incorporating our newly discovered structures, we can generate more accurate test cases and continue the iterative process of verification and refinement.

```
Packet packetOut = new Packet();

// Size of payload
packetOut.WriteUInt16(0x0759); // 0x00

// Payload identifier
packetOut.WriteByte(0x95); // 0x02

// Payload identifier
packetOut.WriteByte(0x0A);

// Unknown
packetOut.WriteUInt(0x15000000);

// Title
packetOut.WriteVarUintString("Stonetroll Slaughter");

// Subtitle
packetOut.WriteVarUintString("Weekend Warfront");

// Left text
packetOut.WriteVarUintString("Small contingents of Order..");

// Right text
packetOut.WriteVarUintString("The Stonetroll Crossing sce...");

// Other packet data

// Task Count
packetOut.WriteByte(0x04);

// Task 1

// Id
packetOut.WriteUInt32R(21000);

// Text
packetOut.WriteVarUIntString("Take control of the...");

// Extended Text
packetOut.WriteVarUIntString("Take control of the...");

// Unknown
packetOut.WriteUInt64(0);

// Current Count
packetOut.WriteUInt32R(3);

// Total Count
packetOut.WriteUInt32R(5);

// Unknown
packetOut.WriteUInt32R(1);
packetOut.WriteUInt16(0);
```

Listing 5.3: Third iteration of component synthesiser

We will be creating an instance of our model with five tasks with identifiable task texts and controlled values for the current amount and total amount of objective to complete.

In this iteration, we will instantiate an instance of our model with five tasks, each containing a uniquely identifiable task text. In addition, we will assign controlled values

to the current progress and the total completion requirement for each objective. This will allow us to systematically observe how the game client processes and displays task-related data.

Test Instances

Sending our newly synthesised packet to the client produces the expected results, and the game correctly displays the five tasks with the specified text and the corresponding objective counts. Both the current progress values and the total completion requirements appear as intended, further validating our updated model. This successful outcome confirms that our understanding of the task structure and the whole packet is accurate, allowing us to refine and extend our model for future iterations.

FIGURE 5.10: Result of sending synthesised packet with new task information

Synthesise Knowledge

Since we have successfully met our revised acceptance criteria, our current knowledge aligns with the requirements necessary to model the packet structure. As a result, we can conclude our reverse engineering process, having accurately reconstructed the format and behaviour of the packet based on our observations and iterative refinements.

```
|-----------------------------------------------|----------------|
|00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F |0123456789ABCDEF|
|-----------------------------------------------|----------------|
|07 59 95 0A 15 00 00 00 14 53 74 6F 6E 65 74 72 |.Y.......Stonetr|
|6F 6C 6C 20 53 6C 61 75 67 68 74 65 72 10 57 65 |oll Slaughter.We|
|65 6B 65 6E 64 20 57 61 72 66 72 6F 6E 74 F6 01 |ekend Warfront..|
|53 6D 61 6C 6C 20 63 6F 6E 74 69 6E 67 65 6E 74 |Small contingent|
|73 20 6F 66 20 4F 72 64 65 72 20 61 6E 64 20 44 |s of Order and D|
|65 73 74 72 75 63 74 69 6F 6E 20 66 6F 72 63 65 |estruction force|
|73 20 63 6F 6E 76 65 72 67 65 20 75 70 6F 6E 20 |s converge upon |
|74 68 65 20 6D 61 69 6E 20 6E 6F 72 74 68 2D 73 |the main north-s|
|6F 75 74 68 20 72 6F 75 74 65 20 74 68 72 6F 75 |outh route throu|
|67 68 20 54 72 6F 6C 6C 20 43 6F 75 6E 74 72 79 |gh Troll Country|
|20 69 6E 20 61 6E 20 65 66 66 6F 72 74 20 74 6F | in an effort to|
|20 73 65 63 75 72 65 20 74 68 65 20 73 74 72 61 | secure the stra|
|74 65 67 69 63 20 74 72 61 64 65 20 72 6F 75 74 |tegic trade rout|
|65 20 66 6F 72 20 74 68 65 69 72 20 72 65 73 70 |e for their resp|
|65 63 74 69 76 65 20 73 69 64 65 73 20 E2 80 A6 |ective sides ...|
|20 77 69 74 68 6F 75 74 20 61 74 74 72 61 63 74 | without attract|
|69 6E 67 20 74 68 65 20 61 74 74 65 6E 74 69 6F |ing the attentio|
|6E 20 6F 66 20 77 61 6E 64 65 72 69 6E 67 20 74 |n of wandering t|
|72 6F 6C 6C 73 2E FE 01 54 68 65 20 53 74 6F 6E |rolls...The Ston|
|65 74 72 6F 6C 6C 20 43 72 6F 73 73 69 6E 67 20 |etroll Crossing |
|73 63 65 6E 61 72 69 6F 20 69 73 20 6F 70 65 6E |scenario is open|
|20 61 67 61 69 6E 20 66 6F 72 20 63 68 61 72 61 | again for chara|
|63 74 65 72 73 20 6F 66 20 61 6C 6C 20 6C 65 76 |cters of all lev|
|65 6C 73 21 20 54 61 6B 65 20 63 6F 6E 74 72 6F |els! Take contro|
|6C 20 6F 66 20 74 68 65 20 54 72 6F 6C 6C 20 50 |l of the Troll P|
|61 63 69 66 69 65 72 20 66 6F 72 20 79 6F 75 72 |acifier for your|
|20 72 65 61 6C 6D 21 20 55 73 65 20 74 68 65 20 | realm! Use the |
|57 68 61 63 6B 2D 61 2D 54 72 6F 6C 6C 20 74 6F |Whack-a-Troll to|
|20 6D 6F 77 20 64 6F 77 6E 20 61 73 20 6D 61 6E | mow down as man|
|79 20 74 72 6F 6C 6C 73 20 61 73 20 70 6F 73 73 |y trolls as poss|
|69 62 6C 65 21 20 43 68 61 72 67 65 20 66 6F 72 |ible! Charge for|
|74 68 2C 20 61 6E 64 20 72 65 6D 65 6D 62 65 72 |th, and remember|
|3A 20 74 68 65 20 6F 6E 6C 79 20 67 6F 6F 64 20 |: the only good |
|74 72 6F 6C 6C 20 69 73 20 61 20 64 65 61 64 20 |troll is a dead |
|74 72 6F 6C 6C 2E 0F 00 00 00 00 00 00 00 01 00 |troll...........|
|00 00 01 00 00 00 00 03 01 00 00 00 7E 00 00 00 |............~...|
|00 00 00 00 00 00 00 00 01 00 01 00 00 03 0D 47 |...............G|
|12 49 00 00 00 00 00 00 00 00 00 1F 01 01 00 00 |.I..............|
|00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 0F |................|
|00 0A 00 01 00 00 00 00 00 00 00 00 00 00 00 00 |................|
|13 48 61 72 74 73 68 6F 72 6E 65 20 54 69 6E 63 |.Hartshorne Tinc|
|74 75 72 65 00 00 01 00 01 2A 87 03 84 00 30 00 |ture.....*....0.|
|00 00 00 01 01 00 01 03 04 20 00 00 00 00 00 00 |......... ......|
|00 00 00 00 00 00 07 00 00 09 3A 80 00 00 02 00 |..........:.....|
|00 00 7C 00 00 00 00 00 00 00 00 00 00 00 01 00 |..|.............|
|01 00 00 01 51 80 11 C0 00 00 00 00 00 00 00 00 |....Q...........|
|00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 |................|
|00 00 00 00 00 00 00 14 00 01 00 00 00 00 00 00 |................|
|00 00 00 00 00 00 11 56 61 6E 71 75 69 73 68 65 |.......Vanquishe|
|72 27 73 20 42 6F 6F 6E 00 00 01 00 00 60 34 00 |r's Boon.....`4.|
|0A 00 00 00 00 00 00 01 01 00 00 03 04 00 00 00 |................|
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 A3 00 |................|
|00 00 03 00 00 00 D8 01 00 00 00 00 00 00 00 00 |................|
|00 00 01 00 03 00 00 03 72 09 06 1F 00 00 00 00 |........r.......|
|00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 75 |...............u|
|3B 33 00 00 00 00 00 00 00 00 00 01 00 01 00 00 |;3..............|
|00 00 00 00 00 00 00 00 00 00 0E 4D 65 64 61 6C |...........Medal|
|20 6F 66 20 42 72 61 77 6E 00 00 00 00 00 00 30 | of Brawn......0|
|52 69 67 68 74 2D 43 6C 69 63 6B 20 74 6F 20 72 |Right-Click to r|
|65 63 65 69 76 65 20 61 20 53 74 72 65 6E 67 74 |eceive a Strengt|
|68 20 66 6F 63 75 73 65 64 20 4D 65 64 61 6C 2E |h focused Medal.|
|01 01 00 00 03 04 00 00 00 00 00 00 00 00 00 00 |................|
|00 00 00 00 00 00 02 A3 00 00 00 00 00 00 03 72 0B |...............r.|
|06 1F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
|00 02 00 00 00 48 48 44 00 00 00 00 00 00 00 00 |.....HHD........|
```

```
|00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00 |................|
|13 4D 65 64 61 6C 20 6F 66 20 44 69 73 63 69 70 |.Medal of Discip|
|6C 69 6E 65 00 00 00 00 00 00 31 52 69 67 68 74 |line......1Right|
|2D 43 6C 69 63 6B 20 74 6F 20 72 65 63 65 69 76 |-Click to receiv|
|65 20 61 20 57 69 6C 6C 70 6F 77 65 72 20 66 6F |e a Willpower fo|
|63 75 73 65 64 20 4D 65 64 61 6C 2E 01 01 00 00 |cused Medal.....|
|03 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
|00 00 02 A3 00 00 00 00 00 03 72 0D 06 1F 00 00 |..........r.....|
|00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 |................|
|00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 |................|
|00 00 00 00 00 00 00 00 00 00 00 00 15 4D 65 64 |.............Med|
|61 6C 20 6F 66 20 50 65 72 73 65 76 65 72 61 6E |al of Perseveran|
|63 65 00 00 00 00 00 00 2E 52 69 67 68 74 2D 43 |ce.......Right-C|
|6C 69 63 6B 20 74 6F 20 72 65 63 65 69 76 65 20 |lick to receive |
|61 20 57 6F 75 6E 64 73 20 66 6F 63 75 73 65 64 |a Wounds focused|
|20 4D 65 64 61 6C 2E 01 01 00 00 03 04 00 00 00 | Medal..........|
|00 00 00 00 00 00 00 00 00 00 00 00 00 02 A3 00 |................|
|00 00 04 08 52 00 00 2A 54 61 6B 65 20 63 6F 6E |....R..*Take con|
|74 72 6F 6C 20 6F 66 20 74 68 65 20 54 72 6F 6C |trol of the Trol|
|6C 20 50 61 63 69 66 69 65 72 20 35 20 74 69 6D |l Pacifier 5 tim|
|65 73 53 54 61 6B 65 20 63 6F 6E 74 72 6F 6C 20 |esSTake control |
|6F 66 20 74 68 65 20 54 72 6F 6C 6C 20 50 61 63 |of the Troll Pac|
|69 66 69 65 72 20 35 20 74 69 6D 65 73 20 64 75 |ifier 5 times du|
|72 69 6E 67 20 74 68 65 20 53 74 6F 6E 65 74 72 |ring the Stonetr|
|6F 6C 6C 20 43 72 6F 73 73 69 6E 67 20 73 63 65 |oll Crossing sce|
|6E 61 72 69 6F 2E 00 00 00 00 00 00 00 00 03 00 |nario...........|
|00 00 05 00 00 00 01 00 00 00 00 00 09 52 00 00 |.............R..|
|32 46 69 6E 64 20 61 6E 64 20 75 74 69 6C 69 7A |2Find and utiliz|
|65 20 61 20 57 68 61 63 6B 2D 61 2D 54 72 6F 6C |e a Whack-a-Trol|
|6C 20 74 6F 20 6B 69 6C 6C 20 32 30 20 74 72 6F |l to kill 20 tro|
|6C 6C 73 56 46 69 6E 64 20 61 20 57 68 61 63 6B |llsVFind a Whack|
|2D 61 2D 54 72 6F 6C 6C 20 69 6E 20 74 68 65 20 |-a-Troll in the |
|53 74 6F 6E 65 74 72 6F 6C 6C 20 43 72 6F 73 73 |Stonetroll Cross|
|69 6E 67 20 73 63 65 6E 61 72 69 6F 20 61 6E 64 |ing scenario and|
|20 75 73 65 20 69 74 20 74 6F 20 6B 69 6C 6C 20 | use it to kill |
|32 30 20 74 72 6F 6C 6C 73 2E 00 00 00 00 00 00 |20 trolls.......|
|00 00 01 00 00 00 14 00 00 00 01 00 00 00 00 00 |................|
|0A 52 00 00 38 50 61 72 74 69 63 69 70 61 74 65 |.R..8Participate|
|20 69 6E 20 74 68 65 20 53 74 6F 6E 65 74 72 6F | in the Stonetro|
|6C 6C 20 43 72 6F 73 73 69 6E 67 20 73 63 65 6E |ll Crossing scen|
|61 72 69 6F 20 31 35 20 74 69 6D 65 73 39 50 61 |ario 15 times9Pa|
|72 74 69 63 69 70 61 74 65 20 69 6E 20 74 68 65 |rticipate in the|
|20 53 74 6F 6E 65 74 72 6F 6C 6C 20 43 72 6F 73 | Stonetroll Cros|
|73 69 6E 67 20 73 63 65 6E 61 72 69 6F 20 31 35 |sing scenario 15|
|20 74 69 6D 65 73 2E 00 00 00 00 00 00 00 00 09 | times..........|
|00 00 00 0F 00 00 00 01 00 00 00 00 00 00 00 00 |................|
|00 27 43 6F 6D 70 6C 65 74 65 20 61 6C 6C 20 53 |.'Complete all S|
|74 6F 6E 65 74 72 6F 6C 6C 20 53 6C 61 75 67 68 |tonetroll Slaugh|
|74 65 72 20 74 61 73 6B 73 28 43 6F 6D 70 6C 65 |ter tasks(Comple|
|74 65 20 61 6C 6C 20 53 74 6F 6E 65 74 72 6F 6C |te all Stonetrol|
|6C 20 53 6C 61 75 67 68 74 65 72 20 74 61 73 6B |l Slaughter task|
|73 2E 00 00 00 00 00 00 00 00 00 00 00 00 03 00 |s...............|
|00 00 00 00 00 00 00 03 00 00 00             |............    |
-----------------------------------------------------------------
```

Listing 5.4: The component to reverse in this example

5.4.2 Validation Example: University of Southampton Key Derivation Function

In Section 3.5.2, we presented the discovery of a UID-based key derivation function used in the University of Southampton's access control system. This section revisits that analysis, explicitly mapping the reverse engineering process to the systematic iterative methodology proposed in this chapter. By doing so, we demonstrate how the structured approach can enable the discovery of security-critical implementation flaws.

Context and Initial Observation

The University of Southampton deploys MIFARE Classic 1K cards for building access control. MIFARE Classic cards support cryptographic authentication using sector-specific keys (Key A and Key B for each of 16 sectors). Initial analysis using tools such as MFOC and MFCUK successfully recovered sector keys from multiple cards, revealing an unexpected pattern: while keys differed across cards, certain byte positions remained constant.

Observed Pattern (Sector 0, Key A)

Key Byte	Card 1	Card 2	Card 3
0	0xA9	0xAB	0x24
1	0x31	0x31	0x31
2	0x00	0x00	0x00
3	0x01	0x01	0x01
4	0xAA	0x86	0xE5
5	0xA2	0xA2	0xA2

This regularity suggested that keys were not randomly generated per card but followed a deterministic derivation scheme. Security best practice dictates that cryptographic keys should be unique and unpredictable for each card, yet four out of six bytes remained identical across all tested cards. The remaining two bytes varied in a way that suggested a relationship with the card's publicly readable UID. The systematic methodology will allow us to test this hypothesis rigorously and ultimately recover the complete key derivation function.

Initial Model

Our initial knowledge consisted of the MIFARE Classic protocol specification, standard security practises for access control systems, and the observed pattern from three extracted keys. The MIFARE Classic standard, as defined in ISO/IEC 14443 and NXP's

technical documentation [160], specifies that each card should use unique sector keys to prevent cross-card vulnerabilities. Industry best practice further recommends that these keys be generated using cryptographically secure random number generators during card provisioning, ensuring that the compromise of one card does not facilitate attacks against others.

However, our observations directly contradicted this expectation. The presence of constant byte suggested either a systematic implementation flaw or a deliberate design decision that prioritised operational convenience over security. We hypothesised that the variable bytes might be derived from the card's UID, which is transmitted in plaintext during the ISO 14443-3 anti-collision phase and can be read by any NFC reader without authentication. If this hypothesis proved to be correct, it would represent a severe security vulnerability, as an attacker could compute authentication keys through passive eavesdropping alone, without requiring physical access to the card or sophisticated key extraction attacks.

Each card had a different UID, and we arranged these UIDs alongside their corresponding Sector 0 Key A values to search for correlations. Card 1 had a UID `0xFA 0x5A 0x91 0xF8` and key `0xA9 0x31 0x00 0x01 0xAA 0xA2`. Card 2 had UID `0xF8 0x2B 0x7C 0xD4` and key `0xAB 0x31 0x00 0x01 0x86 0xA2`. Card 3 had UID `0x77 0x1E 0x63 0xB7` and key `0x24 0x31 0x00 0x01 0xE5 0xA2`. Examining these values revealed that bytes at positions 1,2,3 and 4 of the key remained a contant across all cards, while bytes at position 0 and 4 varied.

Table 5.5 Calculating Key A for sector 0

Key Byte	A0 Key Formula
0	Unknown
1	0x31
2	0x00
3	0x01
4	Unknown
5	0xA2

The component to reverse in this scenario was the key generation algorithm used by the University's card provisioning system. This algorithm takes a card's UID as input and produces sector keys as output. The exact function was unknown, but we could model it abstractly as a transformation from the four byte key format as shown below.

$$
\begin{aligned}
& K_{s,t} = f(\text{UID}, s, t) \\
& \text{where} \\
& \qquad s \in \{0, 1, \ldots, 15\} \text{ (sector index)} \\
& \qquad t \in \{\text{A}, \text{B}\} \text{ (key type)}
\end{aligned}
\tag{5.1}
$$

The environment consisted of the card readers deployed across campus, which implement the MIFARE Classic authentication protocol. The boundry between component and environment occured at the AUTH command, where the reader attempts authentication with a candidate key and card responds with either success or failure.

Instantiate Model

To test our hypothesis systematically, we need to instantiate both the component model and the environment model in forms that would allow controlled experimentation. For the component model, we implemented a brute force search algorithm that would test potential key derivation functions by enumerating all reasonable transformation possibilities. Given that simple operations are often preferred in embedded systems due to computational constraints, we focused initially on bitwise XOR operations, which provide a reversible one to one mapping between input and output bytes while requiring minimal processing overhead.

The algorithm operated by taking each byte of the UID and each byte of the known key, then testing all 256 possible XOR masks to determine if any consistent relationship existed. For each combination of UID byte position, key byte position, and XOR mask value, we checked whether the transformation held across all three cards in our initial sample. This exhaustive enumeration guaranteed that if a simple XOR based derivation existed, we could discover it. While more complex derivation functions were theoretically possible, the presence of constant bytes in the key pattern suggested that the implementation favoured simplicity over cryptographic sophistication.

Test Instances

We began systematic testing by performing cross card pattern analysis on the extracted keys from our initial three card sample. Each card had a different UID, and we arranged these UIDs alongside their corresponding Sector 0 Key A values to search for correlations. Card 1 had a UID `0xFA 0x5A 0x91 0xF8` and key `0xA9 0x31 0x00 0x01 0xAA 0xA2`. Card 2 had UID `0xF8 0x2B 0x7C 0xD4` and key `0xAB 0x31 0x00 0x01 0x86 0xA2`. Card 3 had UID `0x77 0x1E 0x63 0xB7` and key `0x24 0x31 0x00 0x01 0xE5 0xA2`.

For the first variable byte position, we observed that that Card 1's key byte `0xA9` could be produced by XORing Car 1's first UID byte `0xFA` with `0x53`. Testing this hypothesis

against Card 2, we found that `0xF8 XOR 0x53 = 0xAB`, which matched Card 2's key byte at position 0. Applying the same operation to Card 3 yielded `0x77 XOR 0x53 = 0x24`, again matching hte observed key byte. This consistent relationship across all three independent samples provided strong evidence that the first key byte was indeed derived from the first UID byte through a simple XOR operation with a fixed mask of `0x53`.

Synthesise Knowledge

Our newly acquired knowledge from this first iteration revealed a concerning pattern for key byte 0. We had discovered that this byte was deterministically computable from the UID using a simple XOR operation with a fixed mask (`0x53`. Since the UID is transmitted in plaintext during the anti-collision phase and can be captured through passive eavesdropping, this represented a partial compromise of the key generation security. However, with only one of the six bytes understood, we could not yet compute complete authentication keys or assess the full security impact of the implementation.

With our synthesised knowledge, we revised the component model to reflect our current understanding of the key generation algorithm. Table 5.6 shows the updated derivation formula for Sector 0 Key A, incorporating the discovered XOR relationship for byte 0 while explicitly marking byte 4 as still unknown.

Table 5.6 Calculating Key A for sector 0 after one cycle of the iterative process

Key Byte	A0 Key Formula
0	uid[0] $\oplus$ 0x53
1	0x31
2	0x00
3	0x01
4	Unknown
5	0xA2

The revised model transformed our understanding from hypothesis to partial specification. We could now generate byte 0 correctly for any UID without requiring access to physical cards, though the complete key remained incomputable due to the unknown derivation for byte 4. The model made explicit what we knew (bytes 0, 1, 2, 3, 5) verses what required further investigation (byte 4), providing clear direction for the next iteration. However, our acceptance criteria remains unsatisfied as we cannot yet compute all the bytes in the authentication keys from the UIDs alone. The systematic methodology dictates another iteration focusing on the remaining variable byte at position 4.

Revise Model

We can now revise our model based on the knowledge we obtained in the previous iteration. We return to the instantiation phase, applying the same XOR enumeration algorithm to byte 4 across our three card sample.

Test Instances

Our brute force search tested all combinations of the UID byte positions (0-3) against all possible XOR masks (0x00-0xFF) for key byte position 4. Testing uid[0] first, Card 1 showed `0xFA XOR 0x50 = 0xAA`, matching key[4]. However, Card 3 yielded `0xF8 XOR 0x50 = 0xA8`, which did not match the observed key byte (0x86). This eliminated uid[0] as the source byte. We then repeat this process for the subsequent UId byte positions.

Testing uid[3], Card 1 demonstrated `0xF8 XOR 0x52 = 0xAA`, matching key[4]. Validating against Card 2, `0xD4 XOR 0x52 = 0x86`, also matched key[4]. Additionally, Card 3 confirmed `0xB7 XOR 0x52 = 0xE5`. This consistent relationship across all three independent samples established that key byte 4 derived from `uid[3] XOR 0x52`.

Synthesise Knowledge

With both variable bytes now understood, we synthesised a complete knowledge of the Sector 0 Key A derivation function. The key generation algorithm used a template of four constant bytes (position 1, 2, 3, 5) combined with two UID derived bytes (position 0 and 4) computed through simple XOR operations with fixed masks. This meant that anyone with the knowledge of the two XOR masks (`0x53` and `0x52`) could compute valid Sector 0 Key A values for any card without requiring access to physical cards or backend systems by passively observing its UID during normal reader interactions.

At this point, our accumulated knowledge satisfied the acceptance criteria we had defined. The reverse engineering process for this specific component was therefore complete. However, the systematic methodology provides a clear path for extending this analysis. The same iterative process XOR enumeration, cross card validation, knowledge synthesis, and model revision, could be applied to the remaining 31 keys (15 additional sectors for Key A, plus all 16 sector for Key B). Each iteration would follow the identical iteration structure, testing whether the UID based derivation pattern generalised across all sectors and both key types. This demonstrates a key advantage of the systematic approach, once the methodology successfully reverse-engineers one instance, it can be systematically replicated across the entire problem space without requiring ad-hoc adaptation or intuitive leaps.

5.4.3 Validation Example: Warhammer Online Archive Format Analysis

In section 4.1, we presented the reverse engineering of the proprietary .MYP format used by Warhammer Online to store game assets. This section revisits that analysis, explicitly mapping the reverse engineering process to the systematic iterative methodology proposed in this chapter. By doing so, we demonstrate how the structured approach enables the discovery of undocumented file formats through systematic observation and analysis.

Context and Initial Observations

Warhammer Online: Age of Reckoning was a massively multiplayer online role-playing game (MMORPG) that ceased operations in 2013. Community-driven preservation efforts such as Return of Reckoning required reverse engineering of the game's proprietary data formats to enable server emulation and asset extraction. The game installation directory contains several large files with the .MYP extension, including `art.myp`, interface.myp, `world.myp`, `video_english.myp`, and `vo_english.myp`. These appeared to be archive files containing game assets such as textures, 3D models, audio files, and interface elements.

Unlike the NFC protocol analysis where specifications existed (ISO 14443, MIFARE documentation), the .MYP format was entirely proprietary and undocumented. No public specification defined the file structure, compression methods, or indexing mechanisms. This represented a pure black-box reverse engineering challenge where all knowledge had to be derived through systematic observation of the structure and program behaviour.

Initial Model

Our initial knowledge consisted of general understanding of archive file formats from prior experience with ZIP, TAR, MPQ, and CASC formats (described in Section 4.1.2.1). Industry practice suggests that game archive formats typically consist of a header containing metadata, a file index mapping file names to data locations, and the actual file data, often compressed. However, the specific implementation details, byte ordering, header structure, hashing algorithms, and compression methods, remained completely unknown for the .MYP format.

The component to reverse in this scenario was the .MYP archive file format itself, a data structure specification that governed how the game's asset files were packaged, indexed, and stored. The environment consisted of the game executables that read and write archives, specifically `warpatcher.exe` (the game updater) and `WAR.exe` (the

main game client). The boundary between component and environment occurred at the file I/O system calls where these executables performed read operations on .MYP files, extracting specific assets for rendering or patching.

Table 5.7 Initial hypothesized .MYP format structure

Field	Initial Hypothesis
Header	Unknown magic bytes, version information
File Index	Unknown structure for mapping filenames to offsets
Compression	Possibly zlib, LZ77, or proprietary
Data Layout	Sequential or random-access
Encryption	Possibly encrypted, unknown if present

Our acceptance criteria were defined as the ability to extract individual diles from .MYP archives, reconstructing them byte for byte identical to their original uncompressed form. This would enable asset analysis, modification, and repackaging for the Return of Reckoning private server project.

Instaniate Model

To test our hypotheses systematically, we needed to instantiate both component and environment models in forms that allowed controlled experimentation. For the component model (the .MYP file structure), we employed multiple analysis tools. HxD hex editor allowed direct inspection of the file bytes to identify structural patterns, magic numbers, and repeated sequences. IDA Pro disassembler enabled examination of `warpatcher.exe` and `WAR.exe` to understand how these programs parsed .MYP files. Process Monitor (ProcMon) provided dynamic analysis by capturing file access patterns, revealing what the executables read during specific operations.

For the environment model (the programs that consume .MYP files), we analysed string references within the executables. Opening `warpatcher.exe` in IDA Pro revealed multiple string constants containing "MYP, suggesting dedicated code paths for handling this file type. Cross-referencing these strings led to functions responsible for archive parsing, decompression, and file extraction. This provided entry points for tracing execution flow through the parsing logic.

The systematic approach required the generation of controlled test cases. We identified that `warpatcher.exe` attempted HTTP requests to download manifest files from remote servers (visible in `patch.cfg` configuration). Although the original servers were offline, the configuration file structure provided valuable information about the patcher's architecture. It referenced two products: "EAMythic Patcher" and "Warhammer Early Release", each with associated manifest repositories. This suggests a content delivery system where .MYP files served as the container format for downloadable game assets.

Test Instances

We began systematic testing by performing static file analysis on the .MYP archives. Opening `art.myp` in HxD hex editor revealed the file's binary structure. The first several bytes did not match known magic numbers for ZIP (PK), TAR (ustar), or other standard file formats, confirming this was a proprietary specification. However, ASCII readable strings appear at various offsets, suggesting an unencrypted file index.

Examining `warpatcher.exe` through IDA Pros string search functionality (searching for "MYP") identified multiple references. Cross-referencing these led to functions that performed file operations. One particularly relevant function contained code that opened .MYP files., performed a seek operation to specific offsets, and read blocks of data. By setting breakpoints and stepping though execution with a debugger, we observed the parsing sequence. The program fist read a fixed sized header, then read an index structure, and finally used offset values from the index to seek to and decompress individual files.

Process Monitor traces captured during attempted patcher execution showed `warpatcher.exe` opening .MYP files with read access and performing multiple seek and read operations. The access patterns revealed that the executables first read from the file's beginning (header), and accessed scattered locations throughout the file (individual file data). This access pattern was consistent with formats like ZIP that store the central directory at the end for fast enumeration without scanning the entire archive.

Static analysis of the .MYP file structure using hex editor inspection revealed repeating patterns. The file contained sequences of fixed length entries, each containing what resembled a 32 bit integer (possibly an offset), followed by another 32-bit integer (possibly size). This suggested a file allocation table index structure.

Synthesis Knowledge

Synthesising these observations producing an emerging model for the .MYP format structure. The index entries contained three primary fields, an offset value pointing to where the file data begins within the archive, a size value indicating compressed data length, and a hash identifying the file. The use of fixed length entries simplified parsing, the program could calculate an entries position via simple arithmetic rather than parsing variable length records.

The actual file data appeared to be compressed, as indicated by the size discrepancy between index reported sizes and actual byte sequences in the hex editor. Compression algorithm identification required examining the data more closely. Compressed data typically exhibits high entropy and lack of readable ASCII patterns. Additionally, cross

referencing against known compression algorithm signature (zlib header 0x78 0x9C, gzip magic number 0x1F 0x8B) could identify the compression method.

At this stage, our knowledge remained incomplete. We understood the high level structure (header, sequential file data) but lacked precise specifications for field sizes, byte ordering (little-edian vs big-edian), hash algorithms, and compression parameters. The model made explicit what we knew versus what required further investigation (exact field layouts, compression method, optional encryption).

Revise Model

With our knowledge synthesised, we revised the component model to reflect our current understanding of the .MYP format. The revised model specified a three part structure, header region, file tables, and individual file entries.

Table 5.8 Revised .MYP format model after first iteration

Component	**Specification**
Header	Small fixed-size header, exact format unknown
File Table Header	Tables storing information about files and other file tables
File Table Entry	Fixed-size records: [offset][size][filename/hash] with file data
File Data	Likely zlib compressed based on data patterns

This revised model transformed our understanding from vague hypothesis to a structured specification, though still incomplete. We could now predict that parsing a .MYP file requires: (1) reading the header to locate the index of the first file table, (2) seeking to the offset of the file table, (3) reading the file table, (4) seeking to file entry offsets, (5) reading the file entry data, (6) seeking to the compressed data, and (5) decompressing the data using the appropriate algorithm.

However, our acceptance criteria remained unsatisfied, we could not yet extract files because crucial details were missing, the exact byte layout of the file tables, the exact byte layout of the file entries, and the compression parameters needed for decompression.

Repeat

The systematic methodology dictated another iteration focusing on the unknown components. We returned to a static analysis with a more specific goal, determine the exact structures of the file table and the file entry. By manually examining multiple file tables and file entries in the hex editor and comparing offset values against actual file positions, we could infer field boundaries.

Measuring consistent byte patterns revealed that the File Header was exactly `0x28` bytes followed by padding. It also revealed that the File Table header consisted of 12 bytes which the first 4 bytes (interpreted as little-endian uint32) was the amount of File Entries after this header block, followed by 8 bytes which were an offset to the next file table.

The File Entry is made up of 34 bytes, which consisted off a pointer to the offset where the data is contained as well as other fields and flags.

Testing the compression hypothesis require extracting a block of data at a known offset and attempting decompression with various algorithms. The zlib library successfully decompressed the data when provided with the compressed size from the File Table Entry, confirming zlib as the compression method.

Locating the index of the first File Table required examining the header structure. The first 4 bytes of the .MYP file appeared to be a File Header as the data was "MYP\0". The 8 bytes of data at offset 12 from the file start correctly aligned with the start of the first File Table Header. This confirmed that this field contained the offset of the first File Table.

With these discoveries, we synthesised complete knowledge of the .MYP format sufficient to meet our acceptance criteria. We could now programmatically parse any .MYP file by reading the header to get to the File Table offset, then traversing all the file table entries until we find one with the correct file key, reading the data stored at the offset, and decompressing the data.

Final Model and Validation

We revised the model to its complete specification:

Offset	Size	Name	Comment
0x00	4 Bytes	File Tag	Always "MYP\0"
0x04	4 Bytes	Unknown	
0x08	4 Bytes	Unknown	
0x0C	8 Bytes	Table Offset	Offset of start of the first file table
0x18	4 Bytes	Number of Files	Number of files in archive
0x1C	4 Bytes	Unknown 1	Unused
0x20	8 Bytes	Unknown 2	Unused

Offset	Size	Name	Comment
0x00	4 Bytes	Amount of File Table Entries	
0x04	8 Bytes	Offset to the next File Table	

Offset	Size	Name	Comment
0x00	8 Bytes	Data Offset	Pointer to the file data
0x08	4 Bytes	Header Size	Size of the data header
0x0C	4 Bytes	Compressed Size	Size of the data on disk, after compression if compressed
0x10	4 Bytes	Uncompressed Size	Size of the data before compression
0x14	8 Bytes	Key	A unique key to identify the file
0x1C	4 Bytes	Checksum	Checksum of the file data
0x1D	1 Bytes	Compressed Flag	Boolean to represent if compressed with deflate
0x1F	1 Bytes	Unknown	Possibly padding for alignment

We implemented an extraction tool based on this specification and validated it against all .MYP files in the games installation. The tool successfully extracted thousands of assets including textures, 3D models, and audio files. Files extracted and subsequently recompressed produced byte identical archives, confirming our model's accuracy. This satisfied our acceptance criteria completely, we had reverse engineered the .MYP format in order to enable asset extraction and repacking.

Contribution to Validation of Proposed Methodology

This validation example demonstrates the iterative model generalises effectively to file format reverse engineering, a domain fundamentally different from both network protocol analysis, and cryptographic access control. The methodology proved equally applicable to static data structures as to dynamic communication protocols, confirming its domain independence.

The example highlights how the methodology handles undocumented proprietary formats. Unlike standards based system where specifications provide ground for truth and validation, propriety formats require building the specification through observation. The iterative method, initial hypothesis, observation, synthesis, model revision, repeat, provided a systematic path from ignorance to understanding without requiring privileged access to source code internal documentation.

The file format domains demonstrates how tool selection (Phase 1 instantiation) adapts to problem characteristics. Network protocol analysis required packet capture tools and emulation devices. File format analysis required hex editors, disassemblers, and file I/O monitoring. The methodology's abstraction of "instantiate tools appropriate to observe the boundary" allows domain specific tool selection while maintaining process consistency.

Chapter 6

Conclusion

This chapter concludes the report, discussing its main contribution, and identifying future work.

This report has presented a detailed exploration of reverse engineering in two distinct yet comparable domains, Near Field Communication (NFC) systems, and video games. By investigating both areas, common challenges, methodologies, and opportunities have been revealed, leading to the proposal of a systematic model for iterative deduction in reverse engineering.

In the NFC domain, we reverse engineer both access control systems and consumer products, identifying critical vulnerabilities in widely used technologies such as MIFARE Classic and MIFARE DESFire. A particular focus is placed on modelling the behaviour of NFC systems and developing a flexible tooling framework to assist in the analysis and emulation of NFC devices. Case studies, such as the University of Southampton ID card system, demonstrate the practical applications and real-world implications of the techniques employed.

In the domain of video games, we explore both the reverse engineering of archive file formats and anti-cheat systems. Through a combination of system-level and static file analysis, we successfully reconstruct the behaviour of update mechanisms and game resource containers. The study of anti-cheat software further highlights the adversarial nature of this environment and the sophisticated methods used to detect and prevent cheating.

To address the commonalities across both domains, a new systematic model for reverse engineering is defined. This model emphasises an evidence-based iterative approach, guided by acceptance criteria, boundary setting, trace analysis, and knowledge acquisition. The application of this model to both NFC systems and video game software demonstrates its adaptability and effectiveness.

Despite the successes, there remain inherent limitations. Some NFC protocols and anti-cheat systems employ advanced obfuscation and cryptographic protections that resist analysis without privileged access or intensive computational resources. Similarly, legal and ethical considerations restrict the depth of reverse engineering permissible, particularly in commercial software domains.

6.1 A Systematic Model for Iterative Deduction of Reverse Engineering

This thesis presents an iterative methodology for reverse engineering components by progressively building and refining models through controlled experimentation and knowledge synthesis. Beginning with complete initial models based on limited observations, we demonstrate how the cycle of instantiation, testing, learning, and revision allows for the gradual construction of a comprehensive and accurate understanding of complex systems.

The approach emphasises not only the importance of structured experimentation, but also the critical role of continual model revision based on new traces and knowledge. By enabling two-way verification between a modelled component and its real environment and vice versa, the process ensures that assumptions are continually challenged and refined.

Through a worked example focusing on network packet reverse engineering, we have shown that the method is not only theoretically sound but also practically effective. By adhering to well-defined acceptance criteria, we are able to validate the accuracy and utility of the reverse-engineered models.

Ultimately, this iterative framework supports a systematic, reliable, and scalable approach to reverse engineering. It transforms what is an ad hoc, intuition-driven task into a repeatable and rigorous methodology, offering a path from uncertainty to insight, one iteration at a time.

6.2 Future Work

The methodology presented in this thesis opens up several promising directions for future research and application.

An immediate extension would be the creation of additional worked examples in different domains. In particular, applying the methodology to the reverse engineering of automotive systems and Near Field Communication (NFC) technologies would provide further validation and refinement of the approach. Extending NFC tooling to support

more secure and modern standards such as DESFire EV2 and EV3 would represent a significant step forward, enabling the methodology to tackle increasingly complex and security-hardened systems.

Another important area for future exploration is the application of this methodology to the field of video game preservation. The techniques developed could greatly assist efforts to document, understand, and preserve legacy video games, where access to the original source code or documentation is often limited or unavailable. In particular, the methodology could be applied to the reverse engineering of various internal components of video games, such as:

- **3D Model Formats**: Reverse engineering the structure of terrain data stored within game archives, deducing the underlying systems used by developers, and reconstructing editable formats. Techniques such as deserialisation, data splicing, and graphical-based visualisation would facilitate not only the recovery of assets but also their modification and reintegration into the original engine, and this could be used to provide enhancements such as improved AI through generated pathfinding data.
- **3D Skinned Geometry**: Investigating character and armour models, developing tools to import and export proprietary mesh and skinning data into industry standard format such as Wavefront OBJ and Autodesk 3D Studio Max. Further work would include handling animation and skeletal data to enable full fidelity of editing in-game models.
- **Client-Side Database Formats**: Analysing and deserialising client-side asset databases to reconstruct the relationships between game objects. Human-readable formats such as CSV and JSON would aid in understanding and modifying these relationships, supporting broader preservations and modding efforts.
- **Lightmaps**: Reverse engineering lighting systems by combining GPU debugging with file analysis, enabling the extraction and recreation of lightmaps outside the original game engine. This would would contribute to understanding the original content pipeline and offer tools to generate new lighting assets for preservation and restoration.
- **Anti-Cheat Technologies**: Extending the methodology to explore the evolution of anti-cheat mechanisms, including machine learning-based detection methods. This would involve reverse engineering modern anti-cheat systems and researching how to design a more robust countermeasures and fairer gameplay environments.

In addition to extending the application of this methodology, future work could focus on further refinement and formalisation of the iterative process itself. This includes

exploring alternative approaches to trace collection, knowledge synthesis, and model revision, particularly in domains where data are noisy, incomplete, or adversarial.

One promising avenue would be the development of dedicated tool support to manage the lifecycle of model construction and refinement. This could include integration with version control systems such as Git to check changes in models across iterations. Such tools could also automate aspects of the process, including trace collection, difference detection between models, and guided synthesis of new knowledge. The use of constraint-solving tools could be explore to help validate model consistency across iterations. Furthermore, a visual dashboard for monitoring progress across iterations, highlighting tested hypotheses, unresolved gaps, and evidence chains, would improve transparency and reproducibility in complex reverse engineering scenarios. This type of toolchain would not only enhance methodological rigor but also provide practical utility to practitioners working across security, software preservation, and system analysis.

Each of these directions not only serves as a valuable test bed for the methodology but also addresses real-world needs in areas ranging from digital preservation to cybersecurity. Continued exploration along these lines would help validate, generalise, and strengthen the iterative reverse engineering framework proposed in this thesis.

Appendix A

Yale Smart Lock Cards

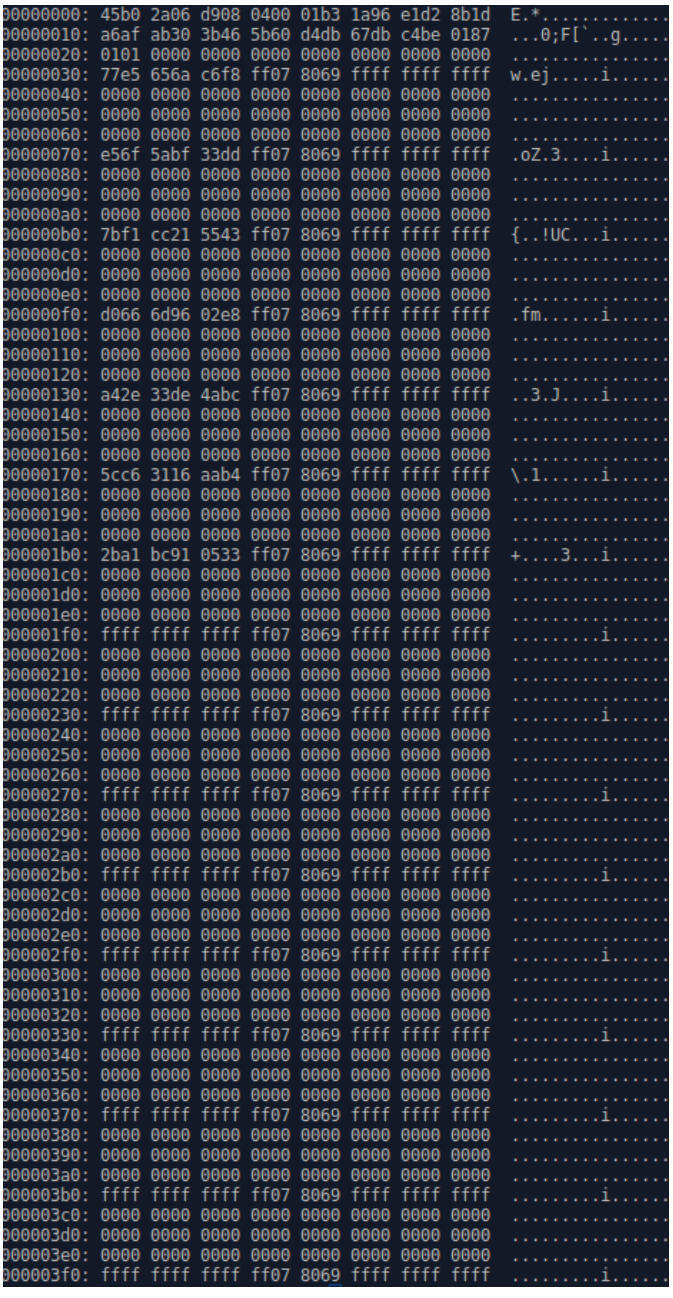


FIGURE A.1: Dump of Yale Smart Lock Card

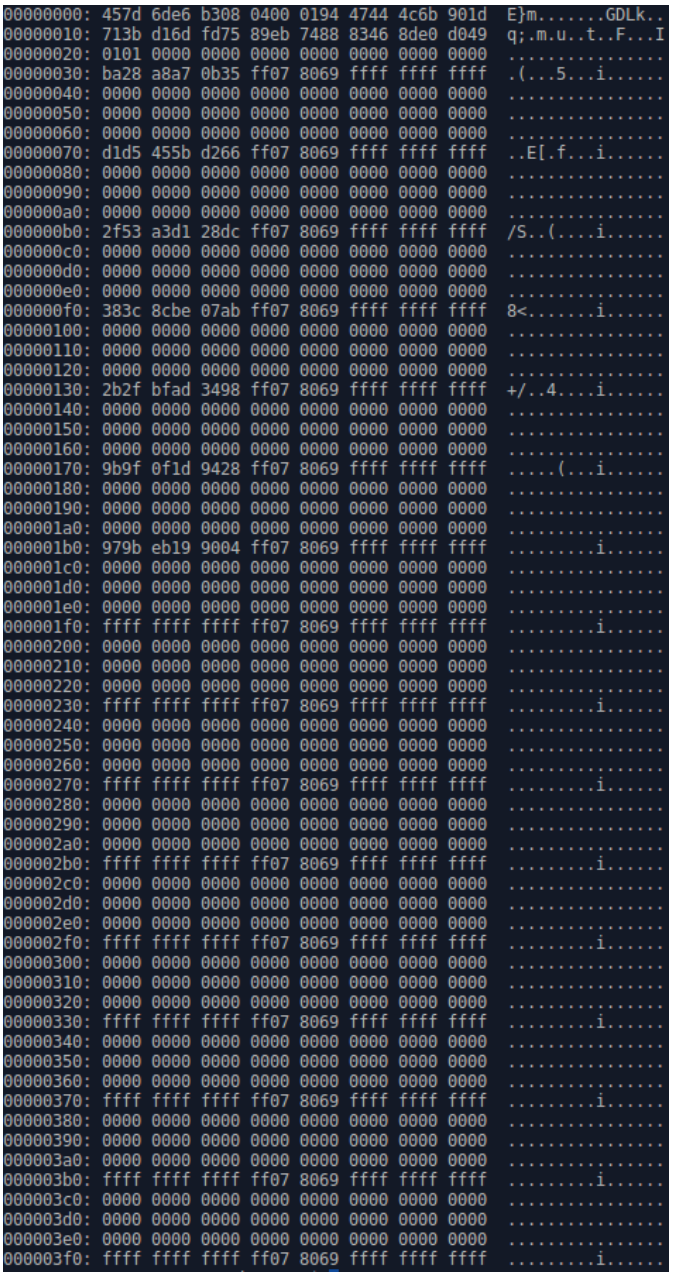


FIGURE A.2: Dump of Yale Smart Lock Fob

Appendix B

Calculating University of Southampton Keys from UUIDs

Table B.1 Calculating Key A for sector 0

Key Byte	A0 Key Formula
0	uid[0] $\oplus$ 0x53
1	0x31
2	0x00
3	0x01
4	uid[3] $\oplus$ 0x52
5	0xA2

Table B.2 Calculating Key A for sector 1

Key Byte	A1 Key Formula
0	0x4E
1	0x06
2	uid[1] $\oplus$ 0x43
3	uid[3] $\oplus$ 0x50
4	0x43
5	uid[2] $\oplus$ 0x4C

Table B.3 Calculating Key A for sector 2

Key Byte	A2 Key Formula
0	0x0D
1	uid[1] $\oplus$ 0x45
2	0x02
3	0x41
4	uid[2] $\oplus$ 0x53
5	uid[0] $\oplus$ 0x4C

Table B.4 Calculating Key A for sector 3

Key Byte	A3 Key Formula
0	uid[1] $\oplus$ 0x4E
1	0x04
2	uid[3] $\oplus$ 0x43
3	uid[2] $\oplus$ 0x50
4	uid[0] $\oplus$ 0x53
5	0x4C

Table B.5 Calculating Key A for sector 4

Key Byte	A4 Key Formula
0	0x14
1	uid[3] $\oplus$ 0x45
2	0x73
3	uid[0] $\oplus$ 0x4A
4	0x41
5	0x1B

Table B.6 Calculating Key A for sector 5

Key Byte	A5 Key Formula
0	uid[3] $\oplus$ 0x55
1	0x64
2	uid[2] $\oplus$ 0x53
3	0x4A
4	0x0B
5	uid[1] $\oplus$ 0x51

Table B.7 Calculating Key A for sector 6

Key Byte	A6 Key Formula
0	0x51
1	uid[2] $\oplus$ 0x50
2	uid[0] $\oplus$ 0x50
3	0x10
4	uid[1] $\oplus$ 0x4C
5	0x0A

Table B.8 Calculating Key A for sector 7

Key Byte	A7 Key Formula
0	uid[2] $\oplus$ 0x53
1	uid[0] $\oplus$ 0x4F
2	0x14
3	uid[1] $\oplus$ 0x32
4	0x00
5	uid[3] $\oplus$ 0x30

Bibliography

[1] BitBetter, "PowerPC FAQ," 2013, Accessed: 28 November 2024. [Online]. Available: https://web.archive.org/web/20130510103008/http://www.bitbetter.com/powerfaq.htm

[2] M. Almeida, "Hacking MiFare Classic Cards," *Black Hat*, 2014.

[3] W. H. Tan, "Practical attacks on the Mifare Classic," September 2009, Accessed: 19 May 2019. [Online]. Available: http://www.doc.ic.ac.uk/{~}mgv98/MIFARE{_}files/report.pdf

[4] D. Carluccio, "Electromagnetic Side Channel Analysis for Embedded Crypto Devices Chair for Communication Security," 2005.

[5] D. Oswald and C. Paar, "Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World," 2011.

[6] Microsoft, "winscard Module," 2006. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms924513.aspx

[7] B. S., "46 DC EA D3 17 FE 45 D8 09 23 EB 97 E4 95 64 10 D4 CD B2 C2," March 2011, Accessed: 3 December 2024. [Online]. Available: https://yalelawtech.org/2011/03/01/46-dc-ea-d3-17-fe-45-d8-09-23-eb-97-e4-95-64-10-d4-cd-b2-c2/

[8] T. Kasper, I. Von Maurich, D. Oswald, and C. Paar, "Cloning Cryptographic RFID Cards for 25$," *5th Benelux Workshop on Information and System Security*, 2010.

[9] F. D. Garcia, P. Van Rossum, R. Verdult, and R. W. Schreur, "Wirelessly pickpocketing a Mifare Classic card," *IEEE Symposium on Security and Privacy*, pp. 3–15, 2009.

[10] C. Kriegs, "Warhammer Online Patcher," August 2008, Accessed: 12 November 2024. [Online]. Available: https://kriegs.net/2008/08/warhammer-online-patcher/

[11] "Deceiving Blizzard Warden," *HackMag*, 2016, Accessed: 24 December 2024. [Online]. Available: https://hackmag.com/uncategorized/deceiving-blizzard-warden/

[12] D. Prizmant and M. Sandt, "A Directory Traversal Attack on Punkbuster Server can be Leveraged to Gain Remote Code Execution (CVE-2020–24102, CVE-2020–26037)," *Medium*, 2020, Accessed: 1 January 2025. [Online]. Available: https://medium.com/@prizmant/hacking-punkbuster-e22e6cf2f36e

[13] "History of the Enigma," 2012, Accessed: 16 November 2024. [Online]. Available: https://www.cryptomuseum.com/crypto/enigma/hist.htm

[14] B. Johnson, *The secret war*. British Broadcasting Corporation, 1978.

[15] S. Borowski, "Code-Breaking Instrumental in Ending World War II," December 2012, Accessed: 17 November 2024. [Online]. Available: https://www.aaas.org/taxonomy/term/10/code-breaking-instrumental-ending-world-war-ii

[16] A. Lam, "ThoughtSTEM Blog," 2019, Accessed: 17 November 2024. [Online]. Available: https://www.thoughtstem.com/blog/2019/11/how-advancements-in-computer-science-helped-us-win-wwii.html

[17] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, pp. 13–17, January 1990, Accessed: 28 November 2024. [Online]. Available: https://web.archive.org/web/20180417124021/http://win.ua.ac.be/~lore/Research/Chikofsky1990-Taxonomy.pdf

[18] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic Reverse Engineering of Malware Emulators," in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 94–109.

[19] S. Al-Sharif, F. Iqbal, T. Baker, and A. Khattack, "White-Hat Hacking Framework for Promoting Security Awareness," in *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016, pp. 1–6.

[20] D. Pinchbeck, D. Anderson, J. Delve, G. Alemu, A. Ciuffreda, and A. Lange, "Emulation as a strategy for the preservation of games: the KEEP project," *DiGRA*, 2009.

[21] Redherring32, "OpenTendo: An Open-Source Hardware (OSHW) Recreation of the Original 1985 Front-Loading NES Motherboard," 2020, Accessed: 17 November 2024. [Online]. Available: https://github.com/Redherring32/OpenTendo

[22] M. V. Wilkes and W. Renwick, "The EDSAC (Electronic Delay Storage Automatic Calculator)," *Mathematics of Computation*, vol. 4, pp. 61–65, 1950.

[23] "EDSAC - The National Museum of Computing," 2019, Accessed: 26 November 2024. [Online]. Available: https://www.tnmoc.org/edsac

[24] B. News, "Vintage computer brought back to life at Bletchley Park," February 2015, Accessed: 28 November 2024. [Online]. Available: https://www.bbc.co.uk/news/technology-31100014

[25] J. Markoff, "Mouse and new WP program join Microsoft product lineup," *InfoWorld*, p. 10, May 1983, Accessed: 28 November 2024. [Online]. Available: https://books.google.co.uk/books?id=4S8EAAAAMBAJ&pg=PA10&redir_esc=y#v=onepage&q&f=false

[26] J. Spolsky, "Why Are the Microsoft Office File Formats So Complicated? (And Some Workarounds)," February 2008, Accessed: 28 November 2024. [Online]. Available: https://www.joelonsoftware.com/2008/02/19/why-are-the-microsoft-office-file-formats-so-complicated-and-some-workarounds/

[27] E. Wells and S. Harshbarger, *Microsoft Excel 97 Developers Handbook: Example Filled Solutions Oriented Guide That Helps You*, 1st ed. USA: Microsoft Press, 1997.

[28] R. Weir, "OpenOffice.org Conference," IBM, September 2006, Accessed: 28 November 2024. [Online]. Available: http://www.robweir.com/blog

[29] W. Wilen, "Royalty-free specifications for Microsoft Office binary file formats," September 2007, Accessed: 28 November 2024. [Online]. Available: https://www.wictorwilen.se/blog/royaltyfree-specifications-for-microsoft-office-binary-file-formats/

[30] P. Moloney, "ECMA 376 Edition 2 Microsoft Patent Declaration," December 2008, Accessed: 28 November 2024. [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-376-Edition-2-Microsoft-Patent-Declaration.pdf

[31] "A Brief History of OpenOffice.org," 2007, Accessed: 28 November 2024. [Online]. Available: https://wiki.openoffice.org/w/index.php?title=A_Brief_History_Of_OpenOffice.org&oldid=186681

[32] S. Shankland, "Sun shelled out $73.5 million for Star Division," *CNET News*, November 1999, Accessed: 28 November 2024. [Online]. Available: https://web.archive.org/web/20150626090829/http://news.cnet.com/Sun-shelled-out-73.5-million-for-Star-Division/2100-1001_3-232561.html

[33] R. Hillesley, "OpenOffice at the crossroads: Every bug is a feature," *The H Open*, p. 2, June 2010, Accessed: 28 November 2024. [Online]. Available: https://web.archive.org/web/20131208042359/http://www.h-online.com/open/features/OpenOffice-at-the-crossroads-1023702.html?page=2

[34] D. Rentz, "OpenOffice.org Component Document File Format," August 2007, Accessed: 28 November 2024. [Online]. Available: https://www.openoffice.org/sc/compdocfileformat.pdf

[35] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with RevNIC," in *5th European Conference on Computer Systems*. Association for Computing Machinery, 2010, pp. 167–180. [Online]. Available: https://doi.org/10.1145/1755913.1755932

[36] "Quick guide: Cracked DRM systems," BBC News, 2007, Accessed: 12 November 2024. [Online]. Available: http://news.bbc.co.uk/1/hi/technology/6944830.stm

[37] J. Tidy, "Police bust 'world's biggest' video-game-cheat operation," BBC News, 2021, Accessed: 12 November 2024. [Online]. Available: https://www.bbc.co.uk/news/technology-56579449

[38] T. Walshe and A. Simpson, "An Empirical Study of Bug Bounty Programs," in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, 2020, pp. 35–44.

[39] K. Ok, M. Aydin, V. Coskun, and B. Ozdenizci, "Exploring underlying values of NFC applications," *International Conference on Management Technology and Applications*, vol. 12, pp. 290–294, 2010.

[40] K. Valk and E. Poll, "Comprehensive security analyses of a toys-to-life game and possible countermeasures," July 2016.

[41] K. Ok, V. Coskun, M. N. Aydin, and B. Ozdenizci, "Current benefits and future directions of NFC services," *ICEMT 2010 - 2010 International Conference on Education and Management Technology*, pp. 334–338, 2010.

[42] E. Biffi, P. Taddeo, M. L. Lorusso, and G. Reni, "NFC-based application with educational purposes," pp. 9–11, 2014.

[43] M. Csapodi and A. Nagy, "New applications for NFC devices," *2007 16th IST Mobile and Wireless Communications Summit*, 2007.

[44] D. Sethia, D. Gupta, T. Mittal, U. Arora, and H. Saran, "NFC based secure mobile healthcare system," *2014 6th International Conference on Communication Systems and Networks, COMSNETS 2014*, 2014.

[45] B. Ozdenizci, K. Ok, V. Coskun, and M. N. Aydin, "Development of an indoor navigation system using NFC technology," *4th International Conference on Information and Computing, ICIC 2011*, pp. 11–14, 2011.

[46] Sony, "FeliCa Contactless IC Card Techonology," Accessed: 23 January 2020. [Online]. Available: http://www.sony.net/Products/felica/

[47] NXP, "MIFARE DESFire EV1 contactless multi-application IC," *Fortune*, vol. 1, pp. 1–12, January 2015.

[48] G. D. K. Gans and J. Hoepman, "A practical attack on the MIFARE Classic," *Smart Card Research and Advanced Applications*, vol. 5189, pp. 267–282, September 2008. [Online]. Available: http://www.springerlink.com/index/u6v0027436h11471.pdf

[49] F. D. Garcia, G. De Koning Gans, R. Muijrers, P. Van Rossum, R. Verdult, R. W. Schreur, and B. Jacobs, "Dismantling MIFARE classic," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5283 LNCS, pp. 97–114, 2008.

[50] C. Meijer and R. Verdult, "Ciphertext-only Cryptanalysis on Hardened Mifare Classic Cards," *22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pp. 18–30, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2810103.2813641

[51] F. Dang, P. Zhou, Z. Li, E. Zhai, A. Mohaisen, Q. Wen, and M. Li, "Large-scale invisible attack on AFC systems with NFC-equipped smartphones," *IEEE INFOCOM*, 2017.

[52] Dentsu, "GAMING: NO LONGER JUST A SIDE QUEST FOR BRANDS 2024: STATE OF GAMING REPORT," 2024, Accessed: 17 November 2024. [Online]. Available: https://s3.amazonaws.com/media.mediapost.com/uploads/StateOfGamingReport.pdf

[53] E. Kain, ""Diablo III" Fans Should Stay Angry About Always-Online DRM," July 2012, Accessed: 17 November 2024. [Online]. Available: https://www.forbes.com/sites/erikkain/2012/05/17/diablo-iii-fans-should-stay-angry-about-always-online-drm/

[54] B. Sinclair, "Warhammer Online Shutting Down," 2013, Accessed: 17 November 2024. [Online]. Available: https://www.gamesindustry.biz/warhammer-online-shutting-down

[55] C. Kerr, "Carbine Studios Will Shut Down WildStar on November 28," 2018, Accessed: 17 November 2024. [Online]. Available: https://www.gamedeveloper.com/business/carbine-studios-will-shut-down-i-wildstar-i-on-november-28

[56] M. Wales, "Ubisoft reportedly revoking The Crew from owners' libraries following server shutdown," April 2024, Accessed: 28 November 2024. [Online]. Available: https://www.eurogamer.net/ubisoft-reportedly-revoking-the-crew-from-owners-libraries-following-server-shutdown

[57] "Archived petition: Require videogame publishers to keep games they have sold in a working state," 2024, Accessed: 26 November 2024. [Online]. Available: https://petition.parliament.uk/archived/petitions/659071

[58] M. Steil, "Lecture: The Ultimate Game Boy Talk," December 2016, Accessed: 26 November 2024. [Online]. Available: https://fahrplan.events.ccc.de/congress/2016/Fahrplan/events/8029.html

[59] Pret, "Pret/pokered: Disassembly of Pokémon Red/Blue," 2010, Accessed: 26 November 2024. [Online]. Available: https://github.com/pret/pokered

[60] "Directive 2002/95/EC of the European Parliament and of the Council of 27 January 2003 on the restriction of the use of certain hazardous substances in electrical and electronic equipment (RoHS)," 2003, Accessed: 26 November 2024. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32002L0095

[61] CNET, "Xbox 360 defects: Red-ringed and red-faced," 2009, Accessed: 26 November 2024. [Online]. Available: https://web.archive.org/web/20091120112304/http://crave.cnet.co.uk/gamesgear/0,39029441,49304288,00.htm

[62] SquareTrade, "Game Console Reliability Report: Xbox 360, PS3, and Wii," 2009, Accessed: 26 November 2024. [Online]. Available: https://www.squaretrade.com/htm/pdf/SquareTrade_Xbox360_PS3_Wii_Reliability_0809.pdf

[63] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation—A performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[64] M. Stevens, "Nostradamus: The first collision of SHA-1 hash functions," 2007, Accessed: 26 November 2024. [Online]. Available: https://marc-stevens.nl/research/hashclash/Nostradamus/

[65] "Air Force completes first successful test of new missile defense system," 2010, Accessed: 26 November 2024. [Online]. Available: https://web.archive.org/web/20101209141211/http://www.wpafb.af.mil/news/story.asp?id=123231285

[66] Cleveland.com, "Defense Department discusses new strategies for future missile defense systems," 2010, Accessed: 26 November 2024. [Online]. Available: https://www.cleveland.com/metro/2010/11/defense_department_discusses_n.html

[67] M. V. Gamer, "Why Is PS3 Emulation So Hard?" 2020, Accessed: 26 November 2024. [Online]. Available: https://www.youtube.com/watch?v=lLebZyha74o

[68] J. M. Libres, J. A. C. Etom, and N. M. Dimal, "An Analysis of the CELL Broadband Engine Architecture and its Implications on the Difficulty of Emulating the PlayStation 3 Console," May 2023,

Accessed: 26 November 2024. [Online]. Available: https://www.researchgate.net/profile/Janriz-Mathew-Libres/publication/375600995_An_Analysis_of_the_CELL_Broadband_Engine_Architecture_and_its_Implications_on_the_Difficulty_of_Emulating_the_PlayStation_3_Console/links/65521ba8ce88b87031e51caa/An-Analysis-of-the-CELL-Broadband-Engine-Architecture-and-its-Implications-on-the-Difficulty.pdf

[69] R. Contributors, "PPU scheduler 2349," 2017, Accessed: 26 November 2024. [Online]. Available: https://github.com/RPCS3/rpcs3/pull/2349

[70] R. Team, "RPCS3 Compatibility List," 2017, Accessed: 26 November 2024. [Online]. Available: https://rpcs3.net/compatibility

[71] M. Cerny, "Patent US20240211380A1: System and Method for Virtualizing Hardware Resources for Video Game Emulation," 2024, Accessed: 26 November 2024. [Online]. Available: https://patents.justia.com/patent/20240211380

[72] G. B. Debbie, "Key2 the kingdom: Sony offers copy protection for CD-R: The Magazine for Electronic Media Producers Users," *Emedia Magazine*, vol. 14, no. 6, pp. 10–11, June 2001. [Online]. Available: https://www.proquest.com/trade-journals/key2-kingdom-sony-offers-copy-protection-cd-r/docview/224088957/se-2

[73] K. Pigna, "EA Hit with Class Action Lawsuit over Spore DRM," *1UP.com*, September 2008, Accessed: 24 December 2024. [Online]. Available: https://web.archive.org/web/20091001171241/http://www.1up.com/do/newsStory?cId=3170131

[74] G. Howson, "Spore becomes the most pirated game ever - but why?" *The Guardian - Games Blog*, December 2008, Accessed: 24 December 2024. [Online]. Available: https://www.theguardian.com/technology/gamesblog/2008/dec/09/games

[75] R. Petrut, ""Why you mad?" - User and media perception on game design anti-piracy measures," June 2020, Accessed: 24 December 2024. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1440310/FULLTEXT01.pdf

[76] T. Staff, "Game Devs Trolling Pirates Goes All The Way Back To At Least The Playstation Days With Spyro 2," May 2019, Accessed: 24 December 2024. [Online]. Available: https://www.techdirt.com/2019/05/03/game-devs-trolling-pirates-goes-all-way-back-to-least-playstation-days-with-spyro-2/

[77] D. Sims, "Rockstar sold pirated copies of its games on Steam, triggering anti-piracy measures," *TechSpot*, September 2023, Accessed: 24 December 2024. [Online]. Available: https://www.techspot.com/news/100073-rockstar-sold-pirated-copies-games-steam-triggering-anti.html

[78] W. Butler, "The Return of Reckoning: Warhammer Online and Successful Fan-Directed Preservation of Videogames," September 2024, Accessed: 16 November 2024. [Online]. Available: https://warhammer-conference.com/warhammer-online-and-preservation-of-videogames/

[79] Return of Reckoning Development Team, "Return of Reckoning - About," 2014, Accessed: 23 May 2025. [Online]. Available: https://www.returnofreckoning.com/about

[80] M. D. Hayman *et al.*, "Warhammer Online Developer Discussion #1," September 2021, Accessed: 23 May 2025. [Online]. Available: https://www.youtube.com/watch?v=aBpv-rbnQ08

[81] M. D. Hayman, "Highly scalable and flexible distributed server framework," Bachelor's Thesis, May 2016.

[82] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14 516–14 539, 2017.

[83] T. Zhang, C. Taylor, B. Coppens, W. Mebane, C. Collberg, and B. De Sutter, "Tools and models for software reverse engineering research," in *Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*. Salt Lake City, UT, USA: ACM, 2024, pp. 44–57.

[84] S. Collins, A. Poulopoulos, M. Muench, and T. Chothia, "Anti-cheat: Attacks and the effectiveness of client-side defences," in *Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, ser. CheckMATE '24. Salt Lake City, UT, USA: ACM, 2024, pp. 30–43.

[85] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 363–379.

[86] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 58–75.

[87] V. Coskun, "A Survey on Near Field Communication (NFC) Technology Vedat Coskun," August 2013.

[88] Federal Office for Information Security, "Advanced Security Mechanisms for Machine Readable Travel Documents," pp. 1–24, March 2012.

[89] O. Jensen, T. O'Meara, and M. Gouda, "Securing NFC credit card payments against malicious retailers," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 214–228, 2016.

[90] ISO/IEC, "Identification cards — Contactless integrated circuit cards - Proximity cards — Part 1: Physical characteristics," 1998.

[91] International Organization for Standardization (ISO), "ISO7816-4 Organization, security and commands for interchange," 2005.

[92] BQT Solutions, "Mifare Classic ® to Mifare DESFire ® Migration Path SOLUTION FACT," 2013. [Online]. Available: https://www.bqtsolutions.com/system/files/mifare{_}classic{_}to{_}mifare{_}desfire{_}migration{_}reader.pdf

[93] L. Francis, G. Hancke, K. Mayes and K. Markantonakis, "Practical nfc peer-to-peer relay attack using mobile phones," *6th International Conference on Radio Frequency Identification: Security and Privacy Issues (RFIDSec)*, pp. 35–49, 2010. [Online]. Available: http://www.mendeley.com/research/lecture-notes-computer-science-2/

[94] K. Nohl and H. PLOTZ, "Mifare: Little Security, Despite Obscurity." 24th Chaos Communication Congress, December 2007. [Online]. Available: https://www.youtube.com/watch?v=QJyxUvMGLr0

[95] K. Nohl and D. Evans, "Reverse-Engineering a Cryptographic RFID Tag," pp. 185–193, July 2008. [Online]. Available: http://www.usenix.org/event/sec08/tech/full{_}papers/nohl/nohl{_}html/

[96] F. Specification, "Standard Card IC MF1 IC S50 Functional Specification," May 2001.

[97] R. Verdult, "Classic Mistakes." *Hacking at Random*, vol. 38, no. 16, p. 36, 2009. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true{&}db=bth{&}AN=12979445{&}site=ehost-live

[98] N. T. Courtois, "The Dark Side of Security by Obscurity and Cloning MiFare Classic Rail and Building Passes Anywhere, Anytime," Cryptology ePrint Archive, Report 2009/137, 2009, https://eprint.iacr.org/2009/137.

[99] B. Boni, "The dark side of e-commerce - Cracking the code or passing the buck?" *Network Security*, vol. 2002, no. 4, pp. 18–19, 2002.

[100] Advanced Card Systems, "EMV Specifications: Book 3 - Application Specification," 2008. [Online]. Available: http://www.emvco.com/specifications.aspx?id=155

[101] J. V. D. Breekel, D. A. Ortiz-yepes, E. Poll, and J. D. Ruiter, "EMV in a nutshell," pp. 1–37, 2016.

[102] Global System for Mobile Communications, "TS 151 011 - V4.15.0 - Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface (3GPP TS 51.011 version 4.15.0 Release 4)," *Identity*, 1999.

[103] International Organization for Standardization, "ISO/IEC 7810:2003 - Identification cards – Physical characteristics," Accessed: 18 May 2019. [Online]. Available: https://www.iso.org/standard/31432.html

[104] ——, "ISO/IEC 7816-4:2013 - Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange," Accessed: 18 May 2019. [Online]. Available: https://www.iso.org/standard/31432.html

[105] Eduard De Jong, Pieter Hartel, Patrice Peyret and P. Cattaneo, "Java Card: An analysis of the most successful smart card operating system to date," vol. 1, pp. 0–40, 2002.

[106] S. Vernois and V. Alimi, "WinSCard Tools : a software for the development and security analysis of transactions with smartcards WinSCard Tools : a software for the development and security analysis of transactions with smartcards," March 2017.

[107] S. J. Murdoch, S. Drimer, R. Anderson, and M. Bond, "Chip and PIN is broken," *IEEE Symposium on Security and Privacy*, pp. 433–446, 2010.

[108] M. Bond, O. Choudary, S. J. Murdoch, S. Skorobogatov, and R. Anderson, "Chip and skim: Cloning EMV cards with the pre-play attack," *IEEE Symposium on Security and Privacy*, pp. 49–64, May 2014.

[109] PCSC Workgroup, "Homepage," Accessed: 16 May 2019. [Online]. Available: https://web.archive.org/web/20170710095308/http://www.pcscworkgroup.com/

[110] D. Sabatier and P. Lartigue, "The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1708, pp. 348–368, 1999.

[111] H. Shan and J. Yuan, "Man in the NFC," *DEFCON 25*, 2017. [Online]. Available: https://media.defcon.org/DEFCON25/DEFCON25presentations/DEFCON25-Haoqi-Shan-and-Jian-Yuan-Man-in-the-NFC.pdf

[112] C. E. Ross, "United States Patent 5,751,073 - Vehicle passive keyless entry and passive engine starting system," *United States Patent and Trademark Office*, 1998.

[113] R. P. Richner, "Research Collection," *BRISK Binary Robust Invariant Scalable Keypoints*, pp. 12–19, 2011. [Online]. Available: https://doi.org/10.3929/ethz-a-010025751

[114] Transport for London, "TfL confirms introduction of cash free bus travel from Sunday 6 July," April 2014. [Online]. Available: https://tfl.gov.uk/info-for/media/press-releases/2014/april/tfl-confirms-introduction-of-cash-free-bus-travel-from-sunday-6-july

[115] U. Stopka, G. Schafer, and A. K. Technische, "NFC-Enabled eTicketing in Public Transport – Aims, Approaches and First Results of the OPTIMOS Project Ulrike," pp. 84–98, 2017.

[116] A. Testhouse, "Certified Mifare Products." [Online]. Available: http://arsenal-testhouse.com/certified-mifare-products/

[117] S. Klee, M. Strüber, M. Maass, M. Hollick, and K. Bräunlich, "Nfcgate: Opening the door for nfc security research with a smartphone-based toolkit," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/klee

[118] M. Roland, J. Langer, and J. Scharinger, "Applying relay attacks to google wallet," in *2013 International Conference on Availability, Reliability and Security*, 2013, pp. 599–604.

[119] M. Authors, "Near-field communication (nfc) cyber threats and mitigation solutions in payment transactions: A review," *Sensors*, vol. 24, no. 23, p. 7423, 2024.

[120] NIST, "CVE-2023-35671: Android framework vulnerability in screen pinning feature," National Vulnerability Database, August 2023, published: 2023-08-14. CVSS v3.1 Base Score: 7.8 (High). Allows unauthorized NFC tag reading while device is locked in screen pinning mode. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-35671

[121] A.-I. Radu, T. Chothia, C. J. Newton, I. Boureanu, and L. Chen, "Practical emv relay protection," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1737–1756.

[122] T. Lis, "How to change ordinary EXE file into a DLL v1.3," 2008-2010, Accessed: 18 December 2024. [Online]. Available: http://web.archive.org/web/20211016044233/https://lubiki.keeperklan.com/other_docs/change_exe_to_dll.htm

[123] C. Dorner and L. D. Klausner, "If It Looks Like a Rootkit and Deceives Like a Rootkit: A Critical Examination of Kernel-Level Anti-Cheat Systems," in *19th International Conference on Availability, Reliability and*

Security, ser. ARES 2024. ACM, July 2024, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1145/3664476.3670433

[124] A. Jonnalagadda, I. Frosio, S. Schneider, M. McGuire, and J. Kim, "Robust vision-based cheat detection in competitive gaming," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 4, no. 1, pp. 1–18, 2021.

[125] M. M. Z. Loo, G. Lužkov, and P. Burelli, "Anticheatpt: A transformer-based approach to cheat detection in competitive computer games," in *2025 IEEE Conference on Games (CoG)*, 2025, pp. 1–4.

[126] "Lexmark International, Inc. v. Static Control Components, Inc." 2014, Accessed: 3 December 2024. [Online]. Available: https://supreme.justia.com/cases/federal/us/572/12-873/case.pdf

[127] H. Wen, "Developing the Battle.net Emulator BNETD," *ONLamp.com*, May 2002, Accessed: 3 December 2024. [Online]. Available: https://web.archive.org/web/20160229080530/http://www.onlamp.com/pub/a/onlamp/2002/05/09/bnetd.html

[128] P. Beruk, "SPA Cease and Desist Notice," Email, April 1998, Accessed: 3 December 2024. [Online]. Available: https://web.archive.org/web/19990220043305/http://starhack.ml.org/cease.txt

[129] M. Baysinger, "SPA Cease and Desist Notice," 1999, Accessed: 3 December 2024. [Online]. Available: https://web.archive.org/web/19990225142437/http://starhack.ml.org/spa.shtml

[130] E. Miller, "Analysis of BNETD and Blizzard," February 2002, Accessed: 3 December 2024. [Online]. Available: https://web.archive.org/web/20130918135959/http://lawmeme.research.yale.edu/modules.php?name=News&file=article&sid=149

[131] P. Masons, "Vivendi wins games copyright battle," October 2004, Accessed: 3 December 2024. [Online]. Available: https://www.pinsentmasons.com/out-law/news/vivendi-wins-games-copyright-battle

[132] T. N. Yorker, "Machine Politics: The Man Who Started the Hacker Wars," *The New Yorker*, May 2012, Accessed: 3 December 2024. [Online]. Available: https://www.newyorker.com/magazine/2012/05/07/machine-politics?currentPage=2

[133] M. J. Kronfeld and T. Liddy, "IHACKED YOU! CODE-CRACKING N.J. KID 'FREES' APPLE CELL," *New York Post*, August 2007, Accessed: 3 December 2024. [Online]. Available: https://nypost.com/2007/08/25/ihacked-you/

[134] J. Fildes, "iPhone hacker publishes secret Sony PlayStation 3 key," *BBC News*, January 2011, Accessed: 3 December 2024. [Online]. Available: https://www.bbc.co.uk/news/technology-12116051

[135] fail0verflow, bushing, marcan, segher, sven, "Console Hacking 2010: PS3 Epic Fail," Presented at the 27th Chaos Communication Congress, 2010, Accessed: 3 December 2024. [Online]. Available: https://fahrplan.events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf

[136] T. Thorsen, "Sony/Hotz settlement details surface," *GameSpot*, April 2011, Accessed: 3 December 2024. [Online]. Available: https://www.gamespot.com/articles/sony-hotz-settlement-details-surface/1100-6308347/

[137] Lydian, "Court of Justice of the European Union allows Reverse Engineering to Correct Errors," 2021, Accessed: 21 April 2025. [Online]. Available: https://www.lexology.com/library/detail.aspx?g=f5b1193c-f423-4f96-bca5-03f5145ecf15

[138] Identiv, "SCL3711 Contactless USB Smart Card Reader Data Sheet," 2017. [Online]. Available: https://www.identiv.com/products/smart-card-readers/contactless-smart-card-readers/scl3711-contactless-usb-smart-card-reader

[139] J. Krieger, "Dark Age of Camelot – Patcher," July 2009, Accessed: 12 January 2025. [Online]. Available: https://www.kriegs.net/2009/07/dark-age-of-camelot-patcher

[140] J. Espenschied, "No security reprieve from Blizzard's Warden," *Computerworld*, May 2007, Accessed: 24 December 2024. [Online]. Available: https://web.archive.org/web/20220926200531/https://www.computerworld.com/article/2545180/no-security-reprieve-from-blizzard-s-warden.html

[141] R. Bowes, "Warden Modules," SkullSecurity Wiki, Accessed: 1 January 2025. [Online]. Available: https://www.skullsecurity.org/wiki/Warden_Modules

[142] xakepru, "x14.08-coverstory-blizzard: Materials for the article "Stretching Blizzard Warden"," 2014, Accessed: 24 December 2024. [Online]. Available: https://github.com/xakepru/x14.08-coverstory-blizzard

[143] Even Balance, Inc., "Even Balance Official Website," Accessed: 1 January 2025. [Online]. Available: https://www.evenbalance.com/index.php

[144] I. Basque-Rice, "Cheaters Could Prosper: An Analysis of the Security of Video Game Anti-Cheat," May 2023, Accessed: 1 January 2025. [Online]. Available: https://ibrice101.github.io/Uni/media/1901124-Dissertation.pdf

[145] A. Orallo, "Why should you worry about kernel-level anti-cheat?" *Micky*, May 2020, Accessed: 1 January 2025. [Online]. Available: https://web.archive.org/web/20220101082041/https://micky.com.au/why-should-you-worry-about-kernel-level-anti-cheat/

[146] R. K. Rigney, "The Gamers Do Not Understand Anti-Cheat," *Push to Talk*, February 2024, Accessed: 1 January 2025. [Online]. Available: https://www.pushtotalk.gg/p/the-gamers-do-not-understand-anti-cheat

[147] mirageofpenguins, "/dev: Vanguard x LoL," *League of Legends*, April 2024, Accessed: 1 January 2025. [Online]. Available: https://www.leagueoflegends.com/en-us/news/dev/dev-vanguard-x-lol/

[148] G. Newell, "Valve, VAC, and trust," Reddit post in r/gaming, February 2014, Accessed: 1 January 2025. [Online]. Available: https://www.reddit.com/r/gaming/comments/1y70ej/valve_vac_and_trust/

[149] B. Philip, "Kernel-level anti-cheats: a necessary devil," July 2024, Accessed: 1 January 2025. [Online]. Available: https://bphilip.uk/blog/2024-07-29-evaluating-kernel-level-anti-cheats-as-a-consumer/

[150] ch4ncellor, "EAC-Reversal - Quick and crude reversal," GitHub repository, Accessed: 1 January 2025. [Online]. Available: https://github.com/ch4ncellor/EAC-Reversal

[151] bright, IDontCode, irql0, "EasyAntiCheat Exploit to inject unsigned code into protected processes," *Back Engineering Blog*, August 2021, Accessed: 1 January 2025. [Online]. Available: https://blog.back.engineering/10/08/2021/

[152] jockyitch, "netCoders vs. PunkBuster," *Bash and Slash*, March 2008, Accessed: 1 January 2025. [Online]. Available: https://web.archive.org/web/20160618090959/http://bashandslash.com/index.php?Itemid=78&id=297&option=com_content&task=view

[153] Even Balance, Inc., "Even Balance Official Website," March 2008, Accessed: 1 January 2025. [Online]. Available: https://web.archive.org/web/20080504060716/https://evenbalance.com/

[154] UnknownCheats Community, "HackShield," UnknownCheats Wiki. [Online]. Available: https://www.unknowncheats.me/wiki/HackShield

[155] SteamDB, "HackShield AntiCheat Games on Steam," SteamDB Info. [Online]. Available: https://steamdb.info/tech/AntiCheat/HackShield/

[156] AhnLab, Inc., "HackShield Product Details." [Online]. Available: https://web.archive.org/web/20101030144803/http://global.ahnlab.com/en/site/product/productSubDetail.do?prod_type=P0&prod_class=P&prod_seq=9003

[157] digitalsleuth, "AutoIT Extractor: View and Extract Resources in an AutoIT Compiled Executable," 2022, Accessed: 27 April 2025. [Online]. Available: https://github.com/digitalsleuth/autoit-extractor

[158] Free Pascal Wiki Contributors, "Character and String Types," 2025, Accessed: 23 March 2025. [Online]. Available: https://wiki.freepascal.org/Character_and_string_types

[159] Adobe Systems Incorporated, "Photoshop File Formats Specification," 2019, Accessed: 23 March 2025. [Online]. Available: https://www.adobe.com/devnet-apps/photoshop/fileformatashtml/

[160] NXP Semiconductors, "AN12653: End to End System Security Risk Considerations for Implementing Contactless Cards and Tags," NXP Semiconductors, Tech. Rep. AN12653, May 2021, rev. 1.1. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12653.pdf