Anti-Cheat Ban Evasion Detection

by

Alain Siegrist

A Bachelor's Thesis
Submitted to the Department Informatik
Hochschule Luzern
In Partial Fulfillment of the Requirements
For the Degree of Bachelor in Information & Cyber Security
June 5, 2025

Supervised by Dr Tim Blazytko

Bachelor Thesis at Lucerne University of Applied Sciences and Arts
School of Computer Science and Information Technology

Title of Bachelor Thesis: Anti-Cheat Ban Evasion Detection

Name of Student: Alain Siegrist

Degree Program: BSc Information & Cyber Security

Year of Graduation: 2025

Main Advisor: Tim Blazytko

External Expert: Yves Kraft

Industry partner/provider: HSLU

Code / Thesis Classification:
☒ Public (Standard)
☐ Private

Declaration
I hereby declare that I have completed this thesis alone and without any unauthorized or external help. I further declare that all the sources, references, literature and any other associated resources have been correctly and appropriately cited and referenced. The confidentiality of the project provider (industry partner) as well as the intellectual property rights of the Lucerne University of Applied Sciences and Arts have been fully and entirely respected in completion of this thesis.

Place / Date, Signature _____

Submission of the Thesis to the Portfolio Database:
Confirmation by the student
I hereby confirm that this bachelor thesis will be correctly uploaded to the Portfolio Database in line with the code of practice of the University. I rescind all responsibility and authorization after upload so that no changes or amendments to the document may be undertaken.

Place / Date, Signature _____

Expression of thanks and gratitude

I would like to express my sincere gratitude to my advisor, Tim Blazytko, for his constant encouragement, advice, and feedback. I am deeply thankful to my family for their patience and understanding throughout this stressful period. Finally, I owe a special thanks to my friends, who stuck by me even as I missed out on our usual outings.

I Abstract

Online multiplayer games are persistently challenged by cheating, which undermines fair play and player trust. A significant exacerbating issue is ban evasion, where banned players circumvent penalties by creating new accounts, a trivial process in free-to-play games. This practice diminishes the effectiveness of bans and necessitates continuous enforcement cycles. This paper presents a technical exploration and proof-of-concept system designed to improve the identification of returning banned players, thereby strengthening the overall effectiveness of anti-cheat measures.

The proposed solution architecture comprises two primary components: a client-side agent embedded within the game application and a server-side ban management service. Upon a player connecting, the client-side agent gathers multiple identifiers, including the account ID, a comprehensive hardware fingerprint, software-based markers, network data, and a persistent client token. This data is used to track users across account changes or hardware modifications. A cryptographic challenge-response protocol is employed to prevent replay of raw hardware identifiers by malicious servers, transmitting only derived public keys.

On the server side, a ban database stores records of banned players and their associated identifiers. An evasion check service cross-references incoming identifiers against this database during the login handshake. Based on a configurable confidence scoring system, the service can block connections, flag accounts for review, or permit access.

Evaluation of the proof-of-concept was conducted using test scenarios simulating various evasion attempts, such as hardware spoofing and VPN usage. The system successfully achieved its expected outcomes, correctly identifying evasion tactics and legitimate users based on the confidence scoring mechanism. Performance analysis of the client-server handshake revealed a total duration of approximately 11 seconds in the prototype, with TPM operations identified as the primary bottleneck. While initial tests are promising, challenges remain in thoroughly validating resilience against sophisticated commercial cheat tools and mitigating performance impacts of cryptographic operations.

Future work will focus on transitioning the proof-of-concept into a production-ready system. This entails large-scale deployment and testing across heterogeneous devices to accurately measure identifier collision rates and uncover edge cases. Rigorous testing against commercially available and private spoofing tools is critical. Further analysis of the cryptographic robustness of the identifier protocol, client-side hardening against reverse engineering, and the potential incorporation of machine learning-driven behavioral analytics are key areas for future development.

II Table of contents

1. Introduction

Online multiplayer games face a persistent problem with cheating, which undermines fair play and player trust. The issue has become an ongoing struggle between cheat developers and anti-cheat systems (Beigi, 2024). Cheating is not just a fringe activity; it fuels a lucrative underground industry, with cheat developers earning tens of millions of dollars annually from selling hacks (Collins et al., 2025). Game companies respond by banning offending players to maintain competitive integrity. However, determined cheaters rarely give up after being banned; many simply create new accounts or modify their setups to bypass detection (Beigi, 2024). This act of circumventing a ban by using alternate accounts or changed identifiers is known as ban evasion (Niverthi et al., 2022). Ban evasion diminishes the effectiveness of bans and forces game administrators into a continual cycle of enforcement.

1.1. Goals

This paper is guided by the objectives outlined in the project assignment. The primary goals are:

- Research and Analysis: Explore and assess various techniques for detecting ban evasion, focusing on identifying persistent traces such as file artifacts, hardware serial numbers, Trusted Platform Module (TPM) data, and other potential indicators.
- Prototype Development: Build a proof-of-concept system that integrates userland and kernel components to record, analyze, and compare these identifiers.
- Implementation Strategies: Investigate secure and ethical methods for implementing fingerprinting techniques in games, particularly those with community servers.
- Documentation of Results: Clearly document the findings, methodologies, and outcomes of the project to ensure replicability and provide insights for future research.

To address these goals, this paper presents a technical exploration of ban evasion detection mechanisms. We review state-of-the-art techniques, detail the design and implementation of a proof-of-concept system, and discuss its evaluation. This work demonstrates a viable approach to identify ban evaders and contributes insights for future developments in game security.

2. State of the art

This chapter surveys the state-of-the-art techniques used to detect and prevent ban evasion. We organize the discussion into different areas: hardware-based identification methods, trusted hardware solutions like TPMs and Secure Boot for ensuring platform integrity, and network fingerprinting techniques. By examining these approaches, we highlight how current systems attempt to make bans persist and the ongoing arms race between cheat developers and security teams. The focus is on providing a clear foundation for understanding how ban evasion is addressed in practice.

2.1. Anti-cheat

Anti-cheat systems are software solutions created to uphold the integrity of online multiplayer video games. Their core objective is to detect and counteract actions or external programs that provide players with unauthorized advantages, such as displaying enemies through walls, thereby ensuring a fair competitive environment for all participants (Collins et al., 2025). These systems typically function by actively monitoring the game client's execution and scrutinizing its interaction with the operating system. This can involve examining running processes, memory segments utilized by the game, and network communications, often employing techniques analogous to those used by general-purpose security software to identify malicious behavior.

The effective operation of an anti-cheat system is a precursor to the enforcement actions detailed in subsequent sections, such as account and hardware bans. When an anti-cheat system flags an account for violating fair play policies, the resulting ban is intended to remove the offending player from the game ecosystem (Lehtonen, 2020). The landscape of anti-cheat technology is characterized by a persistent "arms race" between game security teams and cheat developers, with each side continually evolving their tactics and countermeasures. This dynamic necessitates constant innovation in detection techniques to maintain the efficacy of anti-cheat measures and, by extension, the penalties they trigger.

2.2. Ban enforcement

Modern online games typically employ a layered approach to ban enforcement. The simplest and most direct form is an account ban, which involves blocking the specific username or account ID of a cheater, preventing them from accessing the game with that identity (Dorner & Klausner, 2024).

However, in free-to-play (F2P) titles, where creating an account is often as simple as supplying a disposable email address, this measure can be easily circumvented (Fabri, 2025). A determined cheater can simply register a fresh

account in seconds and reenter the game, continuing to disrupt other players. As a result, many developers complement account bans with additional safeguards such as hardware ID bans or IP address fingerprinting. These extra layers help ensure that repeat offenders face more significant hurdles than just filling out a registration form, thereby strengthening overall enforcement integrity.

2.3. Hardware fingerprinting

In theory, a hardware ban works by extracting unique identifiers from the user's PC, for instance Media Access Control (MAC) addresses of network cards, CPU or motherboard serial numbers, or hard drive IDs, and combining them into a fingerprint (Collins et al., 2025). Because players rarely change all their hardware at once, parts of this fingerprint ideally remains constant for a given person. If a banned player creates a new account using the same computer, the anti-cheat client will recognize the hardware fingerprint and flag a ban evasion attempt. Many commercial anti-cheat solutions such as BattlEye, Easy Anti-Cheat, Riot Vanguard, and Faceit AC gather a broad array of system information in this manner.

While hardware-based detection raises the bar for evaders, it is not foolproof. Skilled cheaters develop spoofers to falsify hardware fingerprints, which are discussed in Section 2.5.1. A cheat program can hook the system calls that report hardware IDs and substitute fake values (Hogan, 2024). With kernel-level privileges or virtualization, cheaters can make a new PC "identity" appear, defeating naive Hardware ID (HWID) checks. To combat this, anti-cheat developers have escalated with kernel-level anti-cheat drivers, which are more resistant to tampering, and by fingerprinting more components (Rendenbach, 2022). The landscape becomes an arms race: stronger anti-cheat protections drive up the effort and cost for cheat developers; studies show cheat prices correlate strongly with the robustness of the targeted anti-cheat (Collins et al., 2025).

2.3.1. CPU

Anti-cheat systems often harvest the processor's built-in identifier, known in Windows as `ProcessorId` (meekochii, 2024). This 64-bit value is derived directly from the CPU's CPUID leaf (EAX=1), incorporating the family, model, and stepping fields along with feature flags. Because it is fused into the silicon, it cannot be altered by software or firmware updates. During system boot, the BIOS reads the CPUID data and publishes it via the System Management BIOS (SMBIOS) tables, from which it can be queried through WMI's `Win32_Processor` provider (Microsoft, 2022). In legacy CPUs there once existed a true "Processor Serial Number" but privacy concerns led Intel and AMD to disable that feature years ago (Kalayci, 2007). Therefore `ProcessorId` must be used in combination

with other system identifiers to create a unique hardware fingerprint, as it is not unique on its own.

2.3.2. Disk and filesystem

Anti-cheat systems can extend their hardware fingerprint beyond the CPU to include storage identifiers at both the device and filesystem levels. Every ATA or NVMe drive ships from the factory with a unique serial string, typically between ten and twenty ASCII characters, programmed into the controller's onboard ROM (Jeong & Lee, 2019). This "drive-firmware" serial is returned verbatim by the storage controller in response to IDENTIFY DEVICE or NVMe Identify commands and remains constant unless the controller's firmware is reflashed at a very low level.

When Windows first brings a disk online, it also embeds a "disk signature" into the Master Boot Record (MBR) or, on GUID Partition Table (GPT)-formatted media, generates a globally unique identifier in the partition header (Jeong & Lee, 2019). These values become part of the disk's partition table metadata: they endure so long as the table isn't deliberately wiped or rebuilt, making them reliable identifiers for anti-cheat checks. Finally, each formatted volume carries its own serial number, a 32-bit value written into the filesystem's boot sector (whether NTFS or FAT). This identifier is less stable, as a user can reset this number by reformatting.

Under the hood, Windows uses storage I/O Controls (IOCTLs) (such as `IOCTL_STORAGE_QUERY_PROPERTY`) to read the factory-programmed serial from the drive controller (Microsoft, 2024a), then pulls partition and volume metadata from the Logical Disk and Partition Manager. All of these identifiers are surfaced through the Windows Management Instrumentation classes `Win32_DiskDrive`, `Win32_DiskPartition`, and `Win32_LogicalDisk`.

2.3.3. Monitors

Even displays carry unique IDs. Nearly all modern monitors conform to the VESA EDID standard, which embeds a 128-byte descriptor in a tiny EEPROM on the screen's board (Video Electronics Standards Association, 2000). Within that block is a 32-bit serial number. When the GPU initializes the display over the DDC/I²C bus, it reads this serial number and hands it off to Windows (linuxdev, 2023). The OS then caches the raw EDID blob in the registry under the DISPLAY enumeration and exposes the serial via the `root\\wmi` namespace's `WmiMonitorID` provider (Microsoft, 2024b).

2.4. Network Fingerprinting

Beyond analyzing device hardware, analyzing network signatures, particularly through IP fingerprinting, serves as an additional method to identify and track

cheaters. By examining the unique characteristics of a user's IP address and associated network behaviors, anti-cheat systems can link suspicious activities to specific users, even when they attempt to conceal their identity.

2.4.1. IPv4

Historically, home network connections often received a dedicated public IPv4 address from the Internet Service Provider (ISP). Devices within the home network would then share this address using Network Address Translation (NAT) (Mahesh, 2025). This dedicated public IP serves as a unique identifier which can be used by anti-cheat systems to attempt user identification.

Tracking users using IPv4 addresses has been complicated by the widespread adoption of Carrier Grade NAT (CGNAT) (Asturias, 2023). Due to the exhaustion of available IPv4 addresses, some ISPs now assign a single public IPv4 address to multiple customers simultaneously. This practice is particularly common in mobile phone networks but is also increasingly used for fixed line home connections.

2.4.2. IPv6

The increasing usage of IPv6 opens new possibilities. An IPv6 address is extremely large, and internet service providers often allocate a unique /64 network prefix or more to each customer or device (RIPE, 2020). An anti-cheat could fingerprint a player's IPv6 prefix, which is unlikely to be shared with others, to recognize returning ban evaders even if they make new accounts (Herbert, 2018). Because the address space is essentially infinite, banning or tracking at the subnet level can effectively "tag" a particular user's connection without the collateral damage of IP bans in IPv4, which can hit many users when ISPs use CGNAT. However, some ISPs implement policies where customer prefixes change frequently to improve privacy, making long term tracking based solely on IPv6 addresses unreliable for those users (White, 2023).

As of 2025, IPv6-based banning is not common because not all players or servers use IPv6 (Rodrigues, 2023). But as IPv6 adoption grows, anti-cheats are expected to leverage it as a quasi-identifier, possibly in combination with other signals.

2.5. Evasion Techniques

Ban evasion is not a static problem: cheaters actively adapt to every new detection scheme. Their toolkit ranges from low-effort fixes, such as deleting game files, to sophisticated hardware- and firmware-level attacks that mutate the machine's identity. In the following subchapters we dissect the most common evasive tactics, explain why they work, and outline the defenses currently employed by anti-cheat vendors.

2.5.1. Hardware Spoofing

Given the hardware-based nature of modern bans, cheat developers often create HWID spoofers, which are tools that intercept or modify system calls to present fake hardware identifiers to the game or anti-cheat (Hogan, 2024). These spoofers may hook into low-level APIs, drivers, or firmware interfaces to manipulate values, such as substituting a different disk serial or MAC address in memory. In practice, a spoofer might install its own kernel driver that patches functions used by the anti-cheat to read hardware information, ensuring the anti-cheat sees a fictitious set of IDs (Skyfail, 2018). This kind of hooking is analogous to rootkit behavior, and cutting-edge anti-cheat systems deploy multiple countermeasures to detect it.

Detection of hardware spoofing often leverages the anti-cheat's kernel presence. Anti-cheat drivers monitor for unusual modifications in system structures and perform integrity checks on identification interfaces (meekochii, 2024). For instance, Easy Anti-Cheat (EAC)'s kernel module actively scans for hooks: if it detects that a system function or value has been tampered with outside the game's memory space, it flags and blocks it (Dorner & Klausner, 2024). Anti-cheat software also keeps an eye on known cheat or spoofer drivers: EAC and others maintain blacklists of driver signatures and scan for them in memory. Some anti-cheats also require the presence of hardware features that are difficult to virtualize or fake. For instance, TPM 2.0 enforcement can serve as a hard-to-bypass hardware root of trust, as discussed in Section 2.6.2.

2.5.2. Virtual Private Networks

The widespread use of Virtual Private Networks (VPNs) poses significant challenges to detection methods based on tracking IP addresses. VPNs allow users to mask their true IP addresses, making it difficult for anti-cheat systems to accurately trace network signatures (Venkateswaran, 2001). To counteract this, detection techniques have been developed to identify and block VPN usage, such as monitoring for known VPN server IP addresses (IPQualityScore, n.d.).

2.6. Trusted Computing

Modern anti-cheat solutions increasingly lean on trusted computing primitives to anchor their security guarantees in hardware rather than software alone. Technologies such as Secure Boot and TPM 2.0 provide cryptographically verifiable assurances that the platform, and by extension the anti-cheat itself, has not been tampered with. This chapter explains how these mechanisms work, why they present a high bar for cheaters, and where attackers are still finding weaknesses to exploit.

2.6.1. Secure Boot and EFI Drivers

Secure Boot, a Unified Extensible Firmware Interface (UEFI) feature, ensures the operating system boots only with code signed by trusted keys, blocking unsigned or malicious drivers at startup. Requiring Secure Boot can thus prevent cheat rootkits from loading during boot sequence, effectively forcing cheats to operate within the confines of the OS's security model. Riot's Vanguard anti-cheat exemplifies this approach: it includes a kernel driver that starts with the OS boot to establish a guarded environment before any user programs run (Laserface, 2024). Valorant, a free-to-play multiplayer game which uses Vanguard, will refuse to launch if Secure Boot is disabled, underscoring how tightly the anti-cheat integrates with firmware security to maintain integrity . This early-launch strategy means that by the time Windows is running, Vanguard is already active at Ring-0, monitoring the system from the earliest moments. The effectiveness of Secure Boot here is in guaranteeing Vanguard's driver are trusted and un-tampered, while preventing unsigned cheat drivers from preempting it.

However, cheat developers have responded with boot-level bypasses. One method is using custom EFI drivers that load before or alongside Secure Boot. These are essentially malicious UEFI modules that a user installs to memory via the motherboard firmware interface, so that a cheat's kernel driver can be mapped into memory ahead of the anti-cheat. For instance, tools like "xigmapper" leverage a UEFI driver to manually map an unauthorized Windows driver prior to Vanguard's initialization (xtremegamer1, 2025). In a Secure Boot-enabled system, this is non-trivial: the EFI driver itself must be trusted or else Secure Boot would forbid it. Indeed, cheat communities discuss workarounds such as enrolling custom keys or exploiting the Secure Boot process to allow their unsigned EFI modules (otiosum, 2023). In one proof-of-concept, an EFI runtime driver hooks system firmware calls to fake the Secure Boot status to Windows, tricking the system into thinking Secure Boot is active while a cheat is actually loaded early (Shmurkio, 2025). These measures highlight an ongoing arms race: Secure Boot and boot-level anti-cheats greatly increase the difficulty of loading cheats, but sophisticated adversaries are exploring firmware-level attacks to regain that foothold. Coupling anti-cheat with Secure Boot pushes cheat developers into far more complex methods such as custom UEFI drivers, which increases the complexity of attacks.

2.6.2. Trusted Platform Modules

TPMs have emerged as a powerful tool in anti-cheat enforcement due to their ability to securely store cryptographic keys and device secrets. A TPM is a dedicated security chip or firmware module that can attest to hardware integrity and produce unforgeable device identifiers. Modern anti-cheats, especially on

Windows 11, leverage TPM 2.0 to bind game installations or player identities to a hardware-rooted key. Riot's Vanguard, for example, uses the TPM to obtain another unique identifier tied to the device (Klotz, 2021). Because the TPM can sign data with keys that never leave the hardware, any attempt to spoof this identity is extremely difficult without physical hardware attacks. Clearing or resetting the TPM is ineffective in defeating such checks as the underlying TPM identity, the Endorsement Key, cannot be cleared (Trusted Computing Group, 2022).

To ensure the authenticity of a TPM, a key attestation process can be employed (Trusted Computing Group, 2025). Key attestation serves the purpose of cryptographically verifying that a cryptographic key is present in a specific TPM. Each TPM is manufactured with a unique Endorsement Key (EK), and an accompanying Endorsement Key Certificate (EKCert) issued by the TPM manufacturer, which vouches for the public portion of the EK. A trusted third-party service, the Privacy Certificate Authority (Privacy CA), uses this EKCert and EK public key to establish trust that the key is indeed from an authentic TPM. The Privacy CA then issues an Attestation Identity Key (AIK) certificate after validating the EKCert, which allows the TPM to prove its identity without revealing the EK directly. This verification process means it is practically impossible for an entity other than the original manufacturer to create or spoof a TPM that can produce a valid Endorsement Key, as they would not possess the legitimate manufacturer's credentials to sign the EKCert. Consequently, obtaining a cryptographically verifiable EK necessitates acquiring a new physical TPM or a processor incorporating a firmware TPM from a recognized manufacturer, creating a monetary cost for ban evasion.

TPMs also enable secure storage of certificates or tokens that prove a device's legitimacy. For instance, an anti-cheat could issue a token sealed by the TPM such that it can be validated server-side but cannot be replayed on a different machine, since the private key is locked to the TPM hardware (Trusted Computing Group, 2022). This ensures that even if an attacker clones a disk or copies game files, they cannot carry over the trusted token without the original TPM. The robustness of TPM's secure enclaves thereby thwarts software-only spoofing of device identity.

TPM adoption has recently surged. With Windows 11 mandating TPM 2.0 for installation, a large number of active gaming PCs now have TPM 2.0 enabled (Microsoft, n.d.). This wide availability makes TPM-based anti-cheat measures viable at scale, hardware-backed security is becoming a standard expectation in gaming PCs, not a niche feature. Many game developers see this as an opportunity to enhance cheat detection. They can rely on TPMs to anchor trust in the legitimacy of the user's hardware. The trade-off is that older machines

without TPM support might be left out or face compatibility issues, but given the industry trends, this will be less of a concern moving forward.

2.7. Tracking Cookies

Some anti-cheat systems also deploy "soft ban" traces, hidden markers on the user's system to track reinstallation or circumvention attempts (Beigi, 2024). These can be thought of as the equivalent of browser cookies or evercookies, but at the system level. The idea is to leave behind a unique tag on the machine such that even if the user changes hardware or reinstalls the operating system, the tag might remain and reveal the user's past ban. For instance, anti-cheat software may create hidden files or registry entries that are not obviously related to the game, or store data in parts of the filesystem that users rarely clean.

Advanced persistent tracking might also involve dropping files on non-system drives. If a game detects the presence of a second hard drive, it might store a tracking file there. A user who only wipes their C: drive and reinstalls Windows would leave that file intact on D:, allowing the ban to re-trigger. Countermeasures by cheat users include performing extremely thorough system wipes. Communities dedicated to ban evasion share guides for "cleaning traces" which involve not just reinstalling the OS but also reformatting all drives (UnknownCheats, 2020).

The effectiveness of tracking files can be debated. On one hand, they absolutely catch out careless evaders. Banned players who naively reinstall a game and its launcher find themselves immediately re-banned, indicating a leftover trace exposed them. This raises the cost of evasion by requiring more technical steps to truly sanitize a computer. On the other hand, determined cheaters can eventually discover and remove these markers, such as the UnknownCheats community, which actively documents known registry keys and files for different games (UnknownCheats, 2021). Thus, trace files act as a simple way to increase the effort needed to evade a ban, without complex technical considerations.

2.8. Combining identifiers

An important principle evident in anti-evasion strategies is that multiple weak signals, when combined, can form a strong identifier (Collins et al., 2025). No single identifier is infallible: hardware IDs can be spoofed, IPs can change, and tracking cookies can be removed. However, aggregating these independent signals can greatly improve reliability. This is essentially a sensor fusion or ensemble learning problem, where each signal contributes some evidence, and together they produce a confident decision. For instance, consider a player who is using a VPN to hide their IP and also spoofing their HWID. Each individual deception might be able to evade a single-faceted check. But if an anti-cheat system

checks many attributes in parallel, such as hardware, network, and cookies, the probability of a ban evader successfully faking all of them simultaneously is lower compared to spoofing a single one. It only takes one or two unchanged identifiers for linkage to occur. As a simple example, a user may successfully spoof five hardware components but overlook a sixth; that remaining match could identify the evader.

Evaders must address all points of identification, whereas the defender can catch them if any one point remains unchanged. This asymmetry strongly favors the defender. Indeed, some cheaters lament that there are many hardware GUIDs and hidden markers one must clean or spoof to fully evade detection (UnknownCheats, 2024).

2.9. Replay attacks

In architectures where clients connect to potentially untrusted third-party servers, the direct transmission of unique identifiers presents a considerable risk. Malicious server operators can intercept and store these identifiers. Subsequently, these harvested identifiers can be re-transmitted in conjunction with a new, seemingly unrelated, account or session, which is called a replay attack (Syverson, 1994). This deceptive tactic can mislead system administrators or automated moderation systems into incorrectly associating new malicious activity with the original, legitimate owner of the identifiers. The consequence is often the unwarranted penalization of the innocent party, including an account or hardware ban.

A relevant occurrence of this vulnerability was observed in the context of Space Station 14, an open-source, multiplayer online game with a distributed server model (Space Station 14, n.d.). Initially, raw hardware identifiers were sent to any game server a client connected to (Space Wizards Federation, 2024). This allowed malevolent server operators to collect this data. By creating new game accounts and replaying the captured hardware identifiers, these operators could frame legitimate players, making it appear as though the original player was evading bans or engaging in disruptive behavior across multiple accounts (Chief-Engineer, 2024). This often led to the erroneous banning of the innocent player.

To address this problem, centralized hardware ID functionality was introduced to the authentication service of Space Station 14 (PJB3005, 2024). This server receives the raw hardware ID, generates a unique, anonymized token for it, and stores the mapping. Game servers then only receive this anonymized token. While this approach mitigates direct exposure of hardware IDs to game servers, it introduces other considerations. It necessitates the storage and maintenance of this mapping and creates a dependency on the central authority. Furthermore, in distributed ecosystems such as Space Station 14, where independent community

groups operate their own authentication services (Space Station Multiverse, n.d.), clients connecting to these alternative systems transmit their raw hardware identifiers directly to them, thereby reintroducing the original vulnerability. These shortcomings serve as a key motivation for the novel technique detailed in Section 5.4.

3. Concept

This chapter outlines the conceptual framework for a system designed to detect ban evasion attempts. The core intention is to move beyond singular, easily spoofed identifiers by employing a multi-faceted strategy that combines client-side data collection with server-side analysis and verification. This layered approach aims to create a more resilient profile of a player's environment, making it significantly harder for determined evaders to return undetected.

The high-level architecture, depicted in Figure 1, consists of two main components: a client-side agent embedded in the game, and a server-side ban management service. On the client side, the anti-cheat module will gather a set of identifiers each time a player connects. These identifiers include: various hardware fingerprints, a tracking cookie, assessments of system integrity, and information on network devices. An evader would have to wipe or spoof all of these identifiers to avoid detection, significantly raising the effort required to successfully return undetected.

On the server side, we maintain a Ban Database and an Evasion Check Service. The Ban DB stores records of all banned players, including every identifier collected at the time of their ban such as usernames, IPs, hardware hashes and client tokens. It can also store linkages, for example, if a new account was caught and linked to an old ban, that relation is recorded. Each time a player attempts to connect, the Evasion Check Service cross-references the incoming identifiers against the Ban DB. This is done during the login handshake. A dynamic confidence scoring system then evaluates these matches to determine the appropriate action. If the confidence score is high, indicating a strong likelihood of ban evasion, the player is immediately banned. Conversely, a moderate score may flag the player as suspicious for later moderator review, while a low score simply allows the connection. This system enhances accuracy by weighing multiple factors instead of a simple pass/fail, significantly reducing false positives.

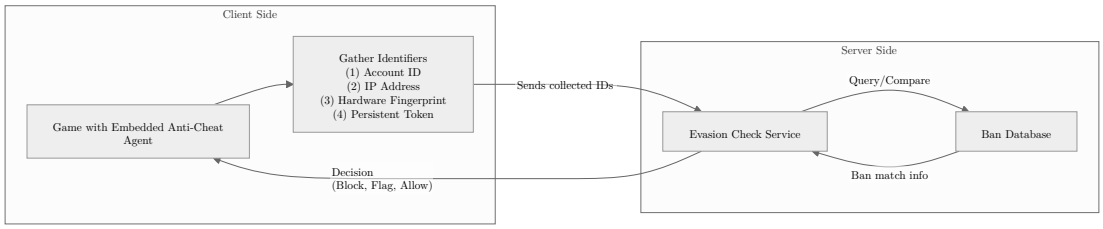

Figure 1: Server-client flow.

Figure 2 depicts the technologies used for each component. The anti-cheat module is implemented as a C++ Dynamic Link Library (DLL), a common

approach for anti cheat software because it allows direct access to system APIs. This design allows it to integrate with the client while remaining separate from the main application logic. Keeping it as an independent module makes updates and maintenance easier, as the anti-cheat system can be modified or replaced without requiring a full client rebuild. Additionally, it allows the module to be separately obfuscated or otherwise protected, without impacting the game code.

The module communicates with a kernel driver written in C using the Windows Driver Kit (WDK), which enables lower-level monitoring of system behavior. Kernel drivers are necessary for detecting advanced cheats that operate below user-mode protections, such as malicious kernel drivers. By placing certain security checks in the kernel space, the system makes it significantly harder for external programs to spoof system information used for identification.

On the server side, the ban database is built using PostgreSQL, chosen for its reliability, strong indexing and permissive license. The server itself is implemented in C#, leveraging its high level features like networking and benefiting from its ease of use.

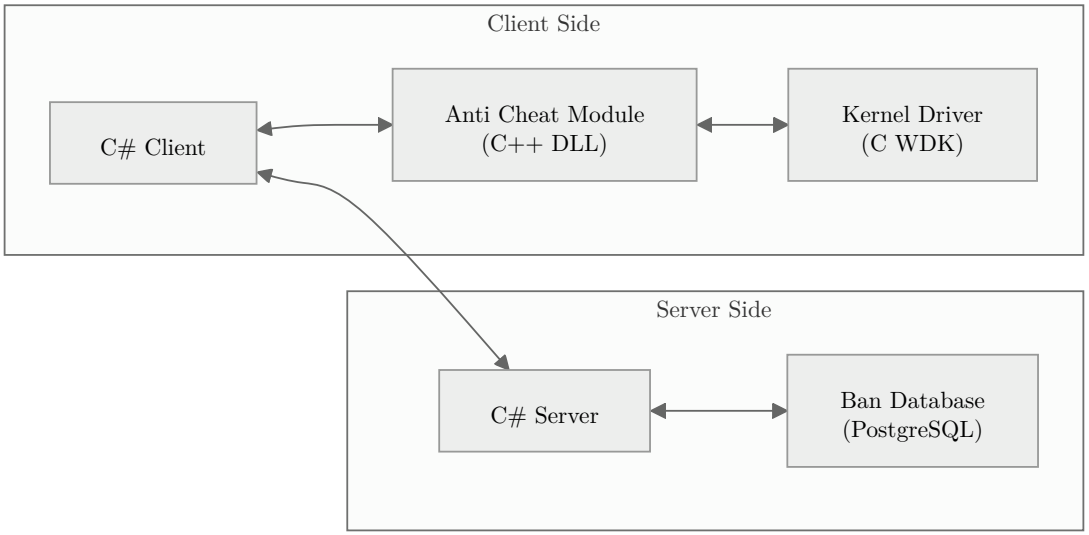

Figure 2: Technologies used for each component.

The system is currently supported only on Windows, as it represents the largest gaming platform. Additionally, many of the detection techniques rely on Windows-specific mechanisms that are difficult to replicate on Unix or macOS.

4. Methodology

This chapter details the methodological framework chosen for the development and evaluation of the ban evasion detection system. The approach integrates two principal methods:

- Literature Research
- Iterative Design, Prototyping, and Evaluation

4.1. Literature Review

The primary objective of the literature research was to build a solid theoretical foundation and gain a comprehensive understanding of current techniques, challenges, and community insights related to ban evasion in online environments. This phase adopted a broad approach to gather diverse perspectives.

The research process involved a systematic survey of various sources:

- Academic Papers: Focusing on user identification, anomaly detection, and fingerprinting techniques in the context of security and online platforms.
- Industry Whitepapers: Examining practical implementations, anti-cheat technologies, and fraud detection strategies employed by game developers and anti-cheat providers.
- Community Experiences: Analyzing discussions, technical write-ups, and shared knowledge from forums, blogs, and developer communities regarding specific evasion tactics and countermeasures.

This foundational research was crucial for identifying common and sophisticated evasion methods, understanding the limitations of existing detection mechanisms, and informing the initial design requirements and potential technical strategies for the proposed system.

4.2. Iterative Design, Prototyping, and Evaluation

Following the literature review, the core development of the system was undertaken using an iterative methodology. This practical phase focused on translating the research insights into a functional prototype through cycles of design, implementation, and testing.

The iterative process involved defining specific detection modules in each cycle, designing the data flow and logic for these modules, implementing them within the system architecture, and conducting preliminary tests to ensure basic functionality. This cyclical approach allowed for incremental feature development, continuous refinement based on intermediate results, and the flexibility to adapt the design as new challenges or insights emerged.

4.2.1. Test Environment

To ensure consistent and reproducible evaluation results, a test environment was prepared. This environment was established using two real systems, a Windows 10 laptop and a Windows 11 laptop. Real devices were used instead of VMs to ensure the hardware and network fingerprinting is performed under real-world conditions. Network conditions were simulated using Proton VPN and switching to a mobile hotspot. A user mode and kernel mode tool for hardware ID spoofing were installed on the client devices representing malicious actors. The user mode spoofer used is SecHex-Spoofy (SecHex, 2025), and the kernel mode spoofer used is Metamorph (Liverus, 2025), which is based on the earlier spoofer Mutante (Tulach, 2024). The goal was to create a setting that could mimic legitimate user actions as well as various ban evasion attempts realistically.

All server-side components were deployed on a CX22 instance hosted by Hetzner (2 vCPU, 4 GiB RAM, Ubuntu 24.04 LTS). PostgreSQL 16 was installed to manage the ban database. This configuration approximates a realistic production environment in which the detection logic is executed on remote hardware rather than on a device in the local network. It also served to validate the feasibility of running the proof-of-concept on a Linux host with limited resources.

The client devices chosen for the test environment are detailed in Table 1. To ensure correct behavior on the most used operating systems for video games both Windows 10 and Windows 11 were used while testing.

Device	**Operating system**	**Date of purchase**
Laptop A	Windows 10 22H2	2021
Laptop B	Windows 11 23H2	2024

Table 1: Client devices used for testing.

To allow the unsigned proof-of-concept kernel driver to be loaded, both machines were ran with the driver signature enforcement disabled. On Windows 11, memory virtualization was also disabled as it ignores the driver signature enforcement settings. As this mode would typically affect the trust score assigned by the proof-of-concept, it was ignored in the evaluation, and the kernel spoofer was deployed using a vulnerable driver as if driver signature enforcement was enabled.

To guarantee isolation between testing scenarios, all users and fingerprints were deleted from the ban database between each scenario. Data such as the `fingerprint_kind` table were kept, as they contain no user- or device-specific data.

4.2.2. Evaluation Scenarios

To comprehensively assess the system's performance, a suite of test scenarios was designed. These scenarios, summarized in Table 2, cover a spectrum from legitimate use to sophisticated evasion techniques, allowing for evaluation of both detection accuracy (true positives, true negatives) and potential weaknesses (false positives, false negatives).

Three primary metrics were recorded:

- Detection outcome
- Confidence score
- Trust score

False-positive and false-negative counts were derived by comparing the observed outcome with the ground truth specified in the scenario definition. Through the execution of these scenarios within the controlled test environment, the system's detection logic, robustness against evasion tactics, and potential for false positives were thoroughly evaluated.

ID	Player Status	Hardware	Token State	Network	Expected result	Ideal result
T01	Unbanned	Genuine new	N/A	N/A	Allow	Allow
T02	Unbanned	Genuine new	N/A	Public Wi-Fi hotspot	Flag	Allow
T03	Unbanned	Second-hand hardware (used)	N/A	N/A	Flag	Allow
T04	Unbanned	Genuine new	N/A	Dynamic-IP ISP	Allow	Allow
T05	Unbanned	Genuine new	N/A	VPN	Block	Block
T06	Banned	Unchanged	Preserved	Same network	Block	Block
T07	Banned	Unchanged	Removed	Same network	Block	Block
T08	Banned	Unchanged	Removed	Different network	Block	Block
T09	Banned	Genuine new	N/A	Same network	Flag	Block
T10	Banned	Genuine new	N/A	VPN	Block	Block
T11	Banned	Genuine new	N/A	Different network	Allow	Block
T12	Banned	User-mode spoof	Preserved	Same network	Block	Block
T13	Banned	User-mode spoof	Removed	Same network	Block	Block
T14	Banned	User-mode spoof	Removed	VPN	Block	Block
T15	Banned	User-mode spoof	Preserved	Different network	Block	Block
T16	Banned	User-mode spoof	Removed	Different network	Block	Block
T17	Banned	User-mode spoof / no TPM	Removed	Different network	Block	Block
T18	Banned	Kernel-mode spoof	Preserved	Same network	Block	Block
T19	Banned	Kernel-mode spoof	Removed	Same network	Block	Block
T20	Banned	Kernel-mode spoof	Removed	VPN	Block	Block

ID	Player Status	Hardware	Token State	Network	Expected result	Ideal result
T21	Banned	Kernel-mode spoof	Preserved	Different network	Block	Block
T22	Banned	Kernel-mode spoof	Removed	Different network	Block	Block
T23	Banned	Kernel-mode spoof / no TPM	Removed	Different network	Block	Block

Table 2: Testing scenarios used in the evaluation.

Some expected results in the different scenarios are more conservative than the ideal results, due to limits of the detection methods used. The gap between the expected results and the ideal outcomes are explained in Table 3.

ID	Expected result	Ideal result	Reason
T02	Flag	Allow	Cannot be differentiated from T09
T03	Flag	Allow	Cannot be differentiated from evader reusing the same monitor
T09	Flag	Block	Cannot be differentiated from T02
T11	Allow	Block	Cannot use hardware or network to identify user

Table 3: Mismatched expected vs ideal test results.

4.3. Project management

The project timeline was divided into three parts: research, implementation and evaluation as shown in Figure 3. This division helped determine whether the project was still within the expected timeframe.

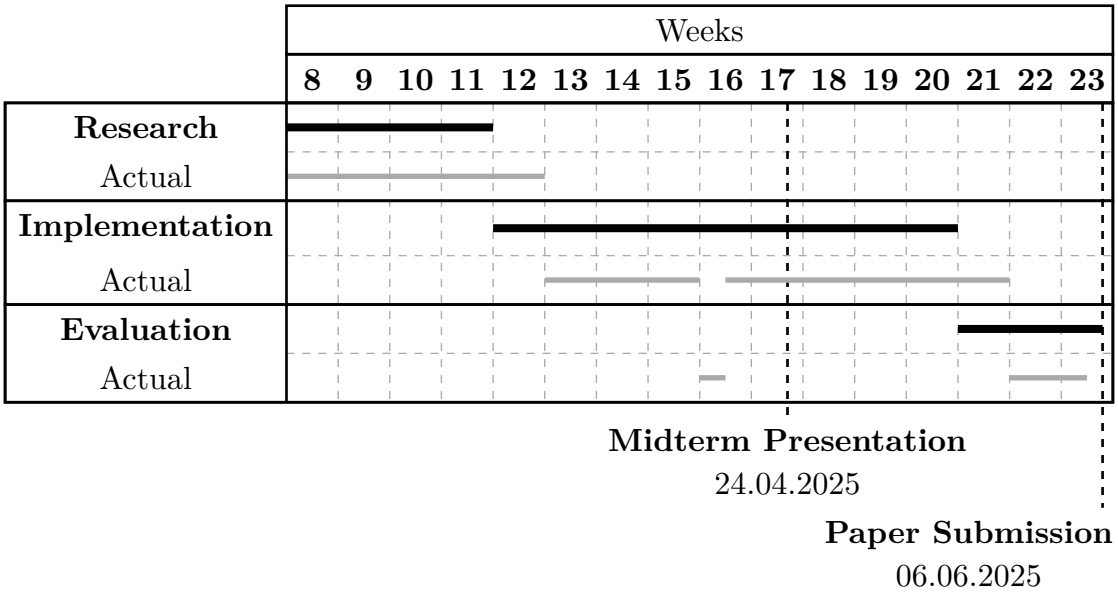

Figure 3: Project timeline.

The first phase, Research (Weeks 1–4), is dedicated to grounding the work in existing knowledge. During this period we perform an exhaustive literature review

of ban-evasion techniques. This foundational work establishes key terminology and helps us refine our technical approach before committing to development.

The second phase, Implementation (Weeks 5–13), focuses on building the proof-of-concept system. Starting from the architecture defined in the research phase, we develop the different parts of the proof-of-concept, such as network communication, device fingerprinting, and the scoring system. This extended window allows for iterative prototyping: early versions are tested and insights are fed back into design revisions. Buffer time is included to accommodate unforeseen technical challenges.

The final phase, Evaluation (Weeks 14–16), is reserved for rigorous testing and analysis. We run the anti-cheat in a controlled environment, measure detection accuracy, false-positive rates, and computational overhead. These results show how effective the implemented detections are.

4.4. Use of generative artificial intelligence

In the creation of this paper, generative artificial intelligence tools were used to support various aspects of the research and writing process. The following machine learning models were utilized: OpenAI's o3 and o4-mini, as well as Google's Gemini 2.5 Flash and Pro. These tools were used for:

- Identifying potential sources for the state-of-the-art chapter
- Conducting initial reviews of academic papers and large codebases
- Refining text to improve flow and clarity
- Proofreading and checking for spelling and grammatical errors

All AI-generated material was thoroughly reviewed and verified by the author to ensure accuracy and avoid the inclusion of incorrect or misleading information. The intellectual contributions presented in the paper are solely the result of the human author's efforts.

5. Implementation

This chapter details the implementation of an anti-cheat system designed to link users to a stable identity, thereby hindering ban evasion attempts. The discussion begins with the project structure and the server-side infrastructure supporting the system. Following this, various types of identifiers utilized for fingerprinting are examined and their implementation described. Finally, the kernel component of the system is explained.

5.1. Project structure

The implementation consists of several projects in order to emulate a realistic anti-cheat system. A "game" project serves as the primary test environment. While not a fully functional game, it replicates the context where an anti-cheat system would typically be integrated. This project is developed using C# on the .NET 9 platform. Data persistence is handled using Dapper for object-relational mapping. Communication between the game client and the anti-cheat module relies on Platform Invocation Services (P/Invoke) for calling native code, with Protocol Buffers (Protobuf) employed for data serialization.

The core user-mode anti-cheat logic resides in the "anticheat user" project. This component is written in C++ and utilizes the Meson build system. It compiles into a DLL that is subsequently loaded by the "game" project during runtime. The "anticheat kernel" project is implemented in C using the Windows Driver Kit and is loaded by the user-mode anti-cheat on initialization.

5.2. Ban database

PostgreSQL was selected as the database management system due to its open-source nature and widespread adoption in industry environments. The database schema is designed to accommodate identifiers that are associated with multiple user accounts, which indicates that multiple accounts belong to the same user. User authentication credentials are not stored within this database. Authentication is considered to be a separate concern, and it is assumed that any connecting player can be reliably associated with a user or account ID through external mechanisms.

The core database structure is illustrated in Listing 1. It consists of tables for users, fingerprints, and the many-to-many relationship between them.

```sql
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    banned BOOLEAN NOT NULL DEFAULT false,
    trust INT NOT NULL DEFAULT 100,
    suspicious BOOLEAN NOT NULL DEFAULT false
);

CREATE TABLE fingerprint_kind (
    kind TEXT PRIMARY KEY,
    -- How confident are we in the users identity with this kind of fingerprint?
    confidence INT NOT NULL CHECK (confidence BETWEEN 0 AND 100),
    -- How is the trustworthiness of the client increased if this identifier is present?
    present_trust INT NOT NULL DEFAULT 0,
    -- How is the trustworthiness of the client reduced if this identifier is absent?
    missing_trust INT NOT NULL DEFAULT 0
);

CREATE TABLE fingerprints (
    fingerprint_id SERIAL PRIMARY KEY,
    kind TEXT NOT NULL,
    data BYTEA NOT NULL,
    banned BOOLEAN NOT NULL DEFAULT false,
    UNIQUE (kind, data),
    FOREIGN KEY (kind) REFERENCES fingerprint_kind(kind)
);

CREATE TABLE user_fingerprints (
    user_id INT NOT NULL,
    fingerprint_id INT NOT NULL,
    PRIMARY KEY (user_id, fingerprint_id),
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (fingerprint_id) REFERENCES fingerprints(fingerprint_id)
);
```

Listing 1: Ban database schema.

5.2.1. Confidence system

The program implements a nuanced player assessment through two primary scoring metrics: a confidence score and a trust score, which together determine the appropriate response rather than relying on a simple pass or fail. The confidence score quantifies the likelihood of a ban evasion occurring by checking a player's identifiers against a database of known banned entries. Each category of identifier, such as a disk serial or an IP address, is associated with a predefined

`confidence` value stored in the `fingerprint_kind` table, reflecting its perceived reliability and the severity if such an identifier is found on a ban list. As detailed in Listing 2, when any of a player's submitted identifiers matches an entry marked as banned in the database, the `confidence` value associated with the type of that banned identifier is contributed to the player's cumulative confidence score. This aggregate confidence score directly influences subsequent actions: a total score reaching or exceeding 100 results in an automatic ban of the user, while a score of 30 or higher flags the user's account as suspicious, potentially for further review.

```C#
// componentResults is a list of all fingerprint values
var bannedComponents = componentResults.Where(x => x.Banned);
int totalScore = bannedComponents.Sum(x => x.Confidence);

bool thresholdBan = totalScore >= 100;
bool suspicious = totalScore >= 30;

if (thresholdBan)
{
    Console.WriteLine("Automatic ban - total score threshold exceeded.");
    _dbConnection.Execute("SELECT ban_user(@UserId)", user);
    // The server then prepares a ban message and disconnects the peer.
}
else if (suspicious)
{
    Console.WriteLine("Client is connected but marked as suspicious");
    _dbConnection.Execute("UPDATE users SET suspicious = TRUE WHERE user_id =
    @UserId", user);
}
```

Listing 2: Calculation of the confidence score and the resulting enforcement actions.

In parallel, a distinct trust score is calculated to evaluate the overall integrity and authenticity of the player's reported hardware and software environment. This trust score begins at a default baseline of 100 points. It is then dynamically adjusted based on several factors, primarily the presence or absence of various identifier types, where each type has `present_trust` and `missing_trust` values defined in the `fingerprint_kind` database table. For instance, the successful validation of a TPM would contribute its `present_trust` value positively to the score modifier. Conversely, missing expected identifiers would subtract their respective `missing_trust` values. Furthermore, the trust score is significantly impacted by critical integrity verifications; detections such as active kernel hooks or discrepancies between BIOS serial numbers reported from user-mode and

those retrieved from the kernel lead to substantial penalties. The method for calculating this trust score and its direct consequences, including player disconnection for scores below a critical threshold, are shown in Listing 3. A final trust score below 20 results in the player being disconnected, overriding the confidence score evaluation, as the system cannot sufficiently verify their device's integrity.

```C#
// Fetch trust modifiers (present_trust, missing_trust) for all identifier kinds
var fingerprintKinds = _dbConnection
    .Query<(string Kind, int PresentTrust, int MissingTrust)>(
        @"SELECT kind, present_trust, missing_trust
          FROM fingerprint_kind");

// Calculate an initial trust modifier based on the presence or absence of expected identifier kinds
var trustModifier = fingerprintKinds
    .Select(k => fingerprintComponents.Any(f => f.Label == k.Kind) ? k.PresentTrust : -k.MissingTrust)
    .Sum();

// Apply penalties from kernel driver feedback
if (!fingerprint.BiosSerial.IsEmpty && !fingerprint.KernelBiosSerial.IsEmpty &&
    fingerprint.BiosSerial != fingerprint.KernelBiosSerial)
{
    trustModifier -= 200;
}
if (fingerprint.TestSigning)
{
    trustModifier -= 200;
}
if (fingerprint.KernelHooks)
{
    trustModifier -= 200;
}

// Calculate the final trust score
var trust = 100 + trustModifier;
_dbConnection.Execute("UPDATE users SET trust = @Trust WHERE user_id = @UserId", new { user.UserId, Trust = trust });

if (trust < 20)
{
    Console.WriteLine($"Client has low trust ({trust}), disconnecting");
    ServerConfirm disconnectMessage = new(
        "Cannot verify integrity of your device, try again later or contact support"
    );
    // The server then disconnects the peer.
}
```

Listing 3: Calculation of the trust score.

5.3. Network protocol

Communication between the client and server components utilizes the LiteNetLib library for networking functionalities. Data serialization and deserialization are handled using Protobuf, ensuring efficient and type-safe data exchange.

5.4. Identifier challenge response protocol

A significant challenge in hardware identification is the potential for malicious server operators to replay hardware identifiers if they are transmitted directly, as discussed in Section 2.9. To mitigate this risk, a challenge-response protocol based on asymmetric cryptography is employed.

The protocol, visualized in Figure 4, proceeds as follows:

1. The client gathers relevant hardware information, such as the disk serial number.
2. This information is then processed using a computationally expensive hashing algorithm, producing a hash result of 256 bits.
3. This hash serves as the seed for generating a public-private key pair.
4. The client transmits only the public key to the server.
5. The server generates a challenge consisting of random bytes and sends it to the client.
6. The client adds its user id and signs the challenge using its private key.
7. The server receives the signed challenge and verifies the signature using the client's public key.
8. The server records the public key in its database, representing the effective hardware ID.

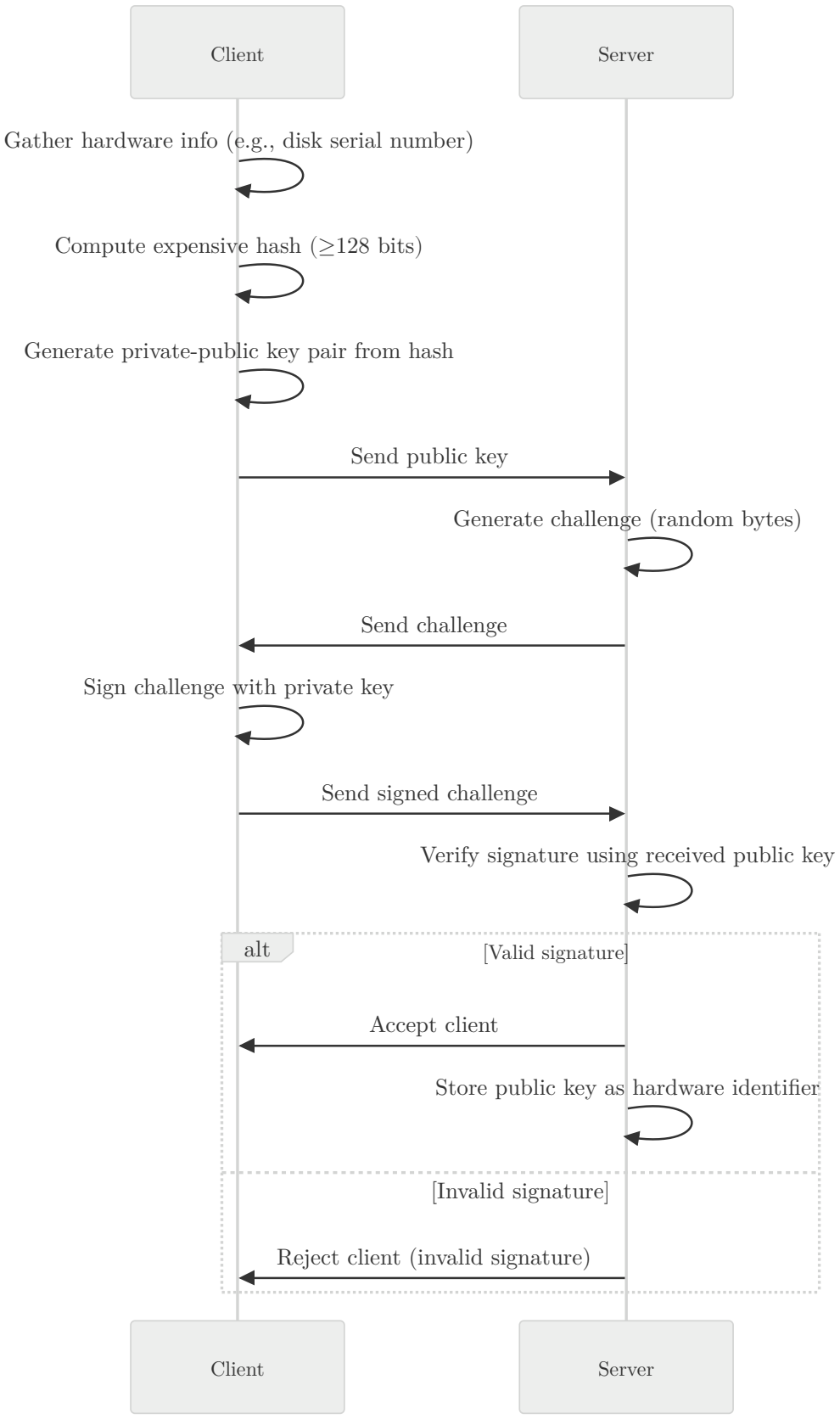

Figure 4: Hardware id server-client protocol.

The process begins before the client sends its fingerprint to the server. Listing 4 shows the C++ routine that derives an Ed25519 key pair from an arbitrary byte string. It uses the Argon2id hashing algorithm to hash the raw hardware identifier. It deliberately fixes the Argon2 salt to zero so that identical hardware produces identical keys. Brute-force pre-computation is made more difficult by Argon2id's high memory cost, which can be tweaked to adapt to the anticipated likelihood of such an attack. Depending on this likelihood, higher memory and iteration parameters can be chosen, increasing both the memory and compute cost for an attacker. However, this also increases the time taken for the legitimate client during authentication.

```cpp
#include "crypto.h"

unsigned char salt[crypto_pwhash_SALTBYTES] = {};

std::pair<PublicKey, PrivateKey>
derive_keypair_from_bytes(const std::string& raw)
{
    Seed seed;

    // interactive: 64 MiB RAM, 3 passes
    crypto_pwhash(seed.data(), seed.size(),
                  raw.data(), raw.size(),
                  salt,
                  crypto_pwhash_OPSLIMIT_INTERACTIVE,
                  crypto_pwhash_MEMLIMIT_INTERACTIVE,
                  crypto_pwhash_ALG_ARGON2ID13);

    PublicKey  pk;
    PrivateKey sk;
    crypto_sign_seed_keypair(pk.data(), sk.data(), seed.data());
    return {pk, sk};
}
```

Listing 4: Deterministic derivation of an Ed25519 key pair from a hardware sample using Argon2id.

The helper templates in Listing 5 iterate over every collected identifier, replace the original byte string in-place by the public key, and cache the private key for later signing.

```cpp
template<typename Strings, typename Privs>
void derive_keys(Strings& serials, Privs& privOut)
{
    for (auto& s : serials) {
        auto keypair = derive_keypair_from_bytes(s);
        auto pub = keypair.first;
        auto fingerprintPrivate = keypair.second;
        // public key replaces raw serial
        s.assign(pub.begin(), pub.end());
        // store private key for later use
        privOut.emplace_back(std::move(fingerprintPrivate));
    }
}
```

Listing 5: Key derivation for a container of serial strings.

A full fingerprint is converted and packed into a protocol buffer by `make_fingerprint()` as shown in Listing 6. A progress bar informs the user because key-generation may take several seconds, depending on the number of identifiers.

```cpp
Fingerprint make_fingerprint()
{
    Fingerprint fp = /* various identifiers are collected */;

    FingerprintPrivate keys;
    derive_keys(*fp.mutable_monitorids(),  keys.monitorIds);
    derive_keys(*fp.mutable_macaddresses(), keys.macAddresses);
    derive_keys(*fp.mutable_diskserials(),  keys.diskSerials);
    derive_keys(*fp.mutable_volumeserials(), keys.volumeSerials);

    derive_key(fp.mutable_registrykey(), keys.registryKey);
    derive_key(fp.mutable_cpuserial(),   keys.cpuSerial);
    derive_key(fp.mutable_biosserial(),  keys.biosSerial);

    fingerprintPrivate = std::move(keys);
    return fp;
}
```

Listing 6: Fingerprint conversion on the client.

The server allocates an independent, unpredictable 256-bit nonce for every public key in the client fingerprint. Listing 7 shows the C# code that fills a `FingerprintChallenge` message.

```csharp
static ByteString RandomChallenge()
{
    Span<byte> buf = stackalloc byte[32];
    RandomNumberGenerator.Fill(buf);
    return ByteString.CopyFrom(buf);
}

challenge.RegistryKey = RandomChallenge();
challenge.CpuSerial   = RandomChallenge();
challenge.BiosSerial  = RandomChallenge();

for (int i = 0; i < fingerprint.MonitorIds.Count; i++)
{
    challenge.MonitorIds.Add(RandomChallenge());
}
// Identical loops for MAC, Disk and Volume identifiers
```

Listing 7: Generation of one-time challenges for all public keys.

Upon reception the client signs every challenge with the matching private key. The signatures are returned in a `FingerprintProof` message as shown in Listing 8.

```cpp
FingerprintProof make_proof(
  const FingerprintChallenge& ch, uint32_t userId
)
{
    FingerprintProof proof;

    auto sign = [&](std::string_view msg, const PrivateKey& sk) {
        unsigned char sig[crypto_sign_BYTES];
        unsigned long long siglen;

        // Concatenate with userId to prevent relay attack
        std::string buffer;
        buffer.reserve(4 + msg.size());
        buffer.append(reinterpret_cast<char*>(&userId), 4);
        buffer.append(msg);

        if (crypto_sign_detached(sig, &siglen, reinterpret_cast<const unsigned
        char*>(buffer.data()), buffer.size(), key.data()) != 0) {
            throw std::runtime_error("Signature generation failed");
        }
        return std::string(reinterpret_cast<char*>(sig), siglen);
    };

    proof.set_registrykey(sign(ch.registrykey(), fingerprintPrivate.registryKey));
    proof.set_cpuserial (sign(ch.cpuserial(),  fingerprintPrivate.cpuSerial));
    proof.set_biosserial(sign(ch.biosserial(), fingerprintPrivate.biosSerial));

    for (int i = 0; i < ch.monitorids_size(); ++i)
    {
        proof.add_monitorids(sign(ch.monitorids(i),
        fingerprintPrivate.monitorIds[i]));
    }

    // Identical loops for MAC, Disk and Volume identifiers

    return proof;
}
```

Listing 8: Signing every challenge with the corresponding private key.

The verifier in Listing 9 reconstructs every Ed25519 public key. It then checks all signatures, and if any signature fails, it rejects the client.

```C#
public static class FingerprintVerifier
{
    public static bool Verify(
        int userId,
        Fingerprint clientFingerprint,
        FingerprintChallenge issuedChallenge,
        FingerprintProof proof)
    {
        byte[] user = BitConverter.GetBytes(userId);

        // Scalar fields
        if (!VerifySig(user, issuedChallenge.RegistryKey, proof.RegistryKey,
        clientFingerprint.RegistryKey)) return false;
        if (!VerifySig(user, issuedChallenge.CpuSerial, proof.CpuSerial,
        clientFingerprint.CpuSerial)) return false;
        if (!VerifySig(user, issuedChallenge.BiosSerial, proof.BiosSerial,
        clientFingerprint.BiosSerial)) return false;

        // Repeated fields
        if (!VerifyRepeated(user, issuedChallenge.MonitorIds, proof.MonitorIds,
        clientFingerprint.MonitorIds)) return false;
        if (!VerifyRepeated(user, issuedChallenge.MacAddresses,
        proof.MacAddresses, clientFingerprint.MacAddresses)) return false;
        if (!VerifyRepeated(user, issuedChallenge.DiskSerials, proof.DiskSerials,
        clientFingerprint.DiskSerials)) return false;
        if (!VerifyRepeated(user, issuedChallenge.VolumeSerials,
        proof.VolumeSerials, clientFingerprint.VolumeSerials)) return false;

        return true;
    }

    private static bool VerifySig(byte[] userId, ByteString message, ByteString
    signature, ByteString publicKey)
    {
        // Add userId to the challenge to prevent relay attack
        byte[] combined = new byte[userId.Length + message.Length];
        Buffer.BlockCopy(userId, 0, combined, 0, userId.Length);
        message.CopyTo(combined, userId.Length);

        var pk = PublicKey.Import(SignatureAlgorithm.Ed25519, publicKey.Span,
        KeyBlobFormat.RawPublicKey);
        return SignatureAlgorithm.Ed25519.Verify(pk, combined, signature.Span);
    }
}
```

Listing 9: Signature verification for all identifiers on the server.

The server does not need to perform the computationally expensive Argon2 key derivation that the client undertakes. Instead, the server's role is limited to verifying the digital signatures using the public keys provided by the client. This design ensures that the server-side verification process remains highly efficient and scalable, as cryptographic signature validation is significantly less resource-intensive than the client's key generation. The server treats the received public key as the client's unique hardware identifier, and successful signature verification confirms the client's legitimate ownership of that identifier without ever needing to know the raw hardware information.

5.5. Identifiers

Various data points can be used to attempt to identify a player uniquely. These identifiers generally fall into three main categories: hardware components, software configuration, and network configuration.

5.5.1. TPM

TPMs are hardware components that securely store cryptographic keys. Each TPM contains a unique EKCert, created and signed by the manufacturer. The system utilizes the Privacy CA key attestation protocol. Through this protocol, a client possessing a TPM can cryptographically prove ownership of the physical TPM associated with a specific EKCert. By establishing a chain of trust anchored to the TPM hardware manufacturers' certificates, the system aims to prevent the forging of fake TPM identities, forcing potential ban evaders to acquire entirely new TPM-equipped hardware.

5.5.1.1. TSS.CPP

The implementation leverages TSS.CPP, a C++ library developed by Microsoft Research for interacting with TPMs (versions for other languages are also available). A significant challenge arose during implementation related to credential activation using the SHA-256 hash algorithm. Initial attempts failed, while activation using SHA-1 succeeded. This was problematic because manufacturer provided EK typically use SHA-256, and the hash algorithm used for activation must match the EK's algorithm specification. The possibility of hardware incompatibility was considered, as older TPM 1.2 specifications only mandated SHA-1 support. However, the TPM version being used was 2.0, and there were no user reports indicating hardware limitations. Testing on multiple newer devices yielded the same results.

Further investigation into the TSS.CPP library source code revealed the root cause. The `RSA_Encrypt` function within the library incorrectly ignored the provided `hashAlgorithm` parameter, consistently using SHA-1 regardless of the specified algorithm. This bug was corrected in a local copy of the library,

resolving the issue and enabling successful SHA-256 credential activation. The existence of this bug suggests that this specific TPM functionality might not be frequently utilized within the scope of the Microsoft TSS.CPP implementation, as it likely would have been reported otherwise.

5.5.1.2. Key Attestation

The key attestation process simulates a remote attestation procedure involving a Privacy CA. The EK is generally restricted from direct use in encryption or decryption operations for privacy reasons. Therefore, a temporary signing key, known as an AIK, is generated. This AIK is certified by the EK, proving it originates from the specific TPM. In a standard Privacy CA scenario, a trusted third party verifies the TPM's identity. In this implementation, the game server assumes the role of the verifier, receiving the necessary information to identify and attest the client's TPM directly. The overall flow is depicted in Figure 5.

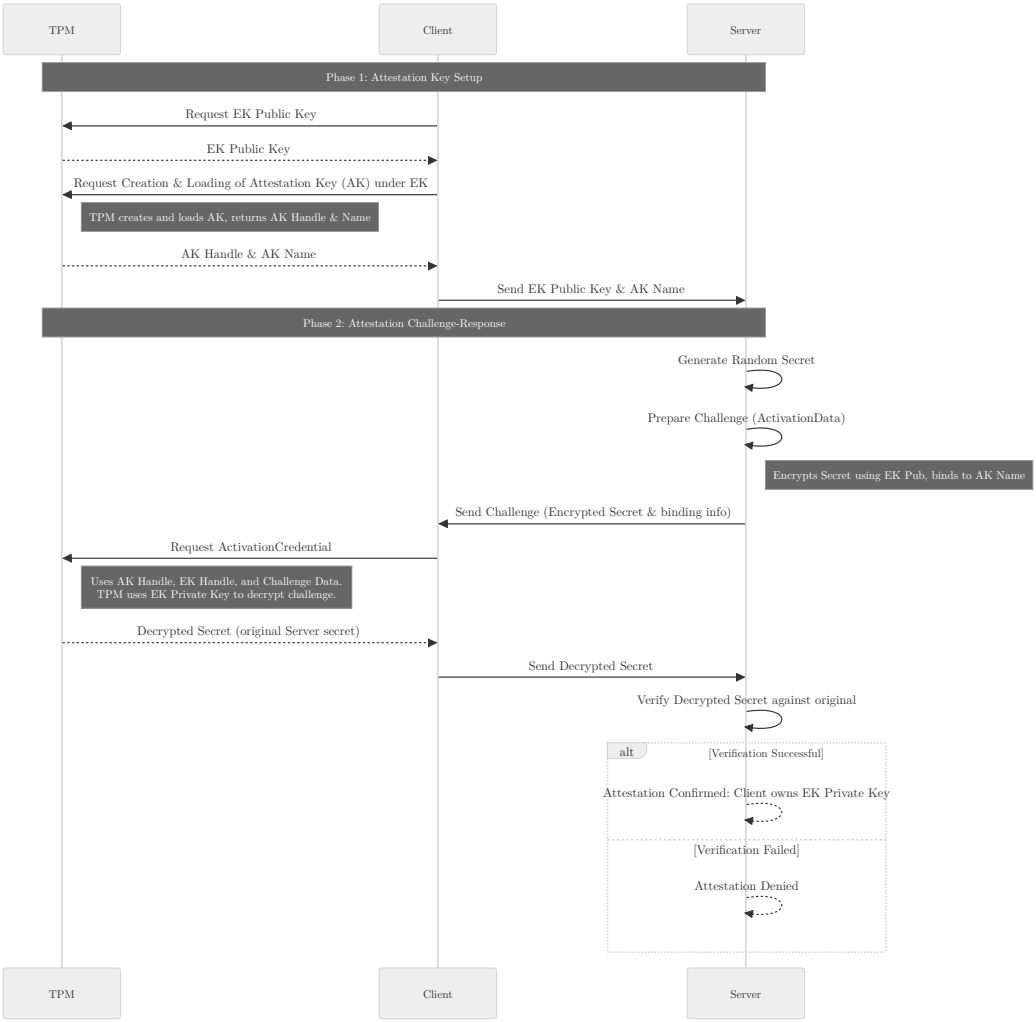

Figure 5: Attestation protocol to link EK to physical TPM.

The process begins by reading the public part of the EK from the TPM, as shown in Listing 10. First, the EK handle is initialized with the value listed in the TPM specification. Then, the `ReadPublic` method is called to retrieve the EK's public key.

```cpp
// TCG TPM v2.0 Provisioning Guidance 7.8 NV Memory EK Reserved Handle
TPM_HANDLE ekHandle(0x81010001);
// Get the public key of the EK
auto ekPubX = tpm.ReadPublic(ekHandle);
ekHandle.SetName(ekPubX.outPublic.GetName());
ekHandle.SetAuth(ByteVec{});
TPMT_PUBLIC &ekPub = ekPubX.outPublic;
auto ekString = ekPub.ToString(true);

printf("EK PUB:\n%s\n\n", ekString.c_str());
```

Listing 10: Reading the Endorsement Key.

Next, an AIK is generated. This key is specifically created for signing operations and is linked to the EK through a policy that requires authorization using the Endorsement Hierarchy (which typically has empty authorization). The generation process is outlined in Listing 11.

```cpp
TPMT_PUBLIC akTemplate(
    ekPub.nameAlg,
    TPMA_OBJECT::sign | TPMA_OBJECT::restricted |
    TPMA_OBJECT::fixedTPM | TPMA_OBJECT::fixedParent |
    TPMA_OBJECT::sensitiveDataOrigin | TPMA_OBJECT::userWithAuth,
    ByteVec{},
    TPMS_RSA_PARMS(
        TPMT_SYM_DEF_OBJECT{},
        TPMS_SCHEME_RSASSA(ekPub.nameAlg),
        2048,
        65537),
    TPM2B_PUBLIC_KEY_RSA()
);

auto akParts = tpm._Sessions(policy).Create(ekHandle,
  TPMS_SENSITIVE_CREATE{},
  akTemplate,
  ByteVec{},
  std::vector<TPMS_PCR_SELECTION>{}
);
tpm.FlushContext(policy);

// Load the generated key
auto policy2 = tpm.StartAuthSession(TPM_SE::POLICY, ekPub.nameAlg);
tpm.PolicySecret(TPM_RH::ENDORSEMENT, policy2, {}, {}, {}, 0);
keyToActivate = tpm._Sessions(policy2).Load(ekHandle,
  akParts.outPrivate,
  akParts.outPublic
);

keyToActivate.SetName(akParts.outPublic.GetName());
keyToActivate.SetAuth(ByteVec{});
auto activateString = akParts.outPublic.ToString(true);
printf("ACTIVATE KEY:\n%s\n\n", activateString.c_str());
```

Listing 11: Generating a new key to use in the attestation protocol.

The server then creates a challenge. This involves generating a secret value and encrypting it using the public part of the client's EK. This encrypted data forms the challenge sent to the client, as seen in Listing 12.

```C#
// Generate a new secret value the client will send us as proof
byte[] secret = new byte[30];
RandomNumberGenerator.Fill(secret);
// Encrypt the secret using the TPM EK
ActivationData activation = TpmCredentialBlob.Create(secret,
fingerprint.DerivedKeyName.Span, fingerprint.AdditionalCertificates.First().Span);
var challenge = new FingerprintChallenge()
{
    TpmChallenge = ByteString.CopyFrom(activation.EncIdentity),
    TpmSecret = ByteString.CopyFrom(activation.EncryptedSecret),
    TpmIntegrity = ByteString.CopyFrom(activation.Integrity)
};
```

Listing 12: Creating the challenge to attest TPM integrity.

Finally, the client uses the `ActivateCredential` command. This command utilizes both the AIK and the EK. The TPM uses the private portion of the EK to decrypt the secret embedded in the challenge. The decrypted secret is then returned to the server as proof that the client possesses the private key corresponding to the claimed EK, confirming the TPM's authenticity. The client-side decryption step is shown in Listing 13.

```cpp
AUTH_SESSION policy = tpm.StartAuthSession(
                        TPM_SE::POLICY,
                        TPM_ALG_ID::SHA256);
// We don't send the policy command code ActivateCredential because most implementations
// don't actually require it and won't work if its used.
tpm.PolicySecret(TPM_RH::ENDORSEMENT, policy,
                 ByteVec{}, ByteVec{}, ByteVec{}, 0);
//  Prepare the second session: a simple PW session for the AIK handle
AUTH_SESSION pw = AUTH_SESSION::PWAP();
// Get the encrypted data from the server
ActivationData cred{};
cred.Secret = ByteVec(challenge.tpmsecret().begin(), challenge.tpmsecret().end());
cred.CredentialBlob = TPMS_ID_OBJECT(
    ByteVec(challenge.tpmintegrity().begin(), challenge.tpmintegrity().end()),
    ByteVec(challenge.tpmchallenge().begin(), challenge.tpmchallenge().end())
);
// We use our EK to decrypt the server challenge, proving that we own the EK private key
// Run activate credential with both sessions
ByteVec decryptedChallenge =
    tpm._Sessions(pw, policy)
    .ActivateCredential(keyToActivate,   // activateHandle (AIK)
                        ekHandle,        // keyHandle      (EK)
                        cred.CredentialBlob,
                        cred.Secret);
// The decryptedChallenge now matches the randomly generated secret on the server
```

Listing 13: Decrypting the server challenge using the TPM EK.

The decrypted secret is then sent back to the server, where it will be checked against the generated secret, as seen in Listing 14. If the data matches, the server knows the client possesses the certified TPM.

```C#
var client = _clients[new User { UserId = clientConfirm.UserId }];
if (client is not ClientState.Connecting(var clientHello, var tpmSecret))
{
    Console.WriteLine($"Client sent proof while not in connecting state");
    break;
}
var proof = FingerprintProof.Parser.ParseFrom(clientConfirm.FingerprintProof);
var tpmProof = proof.TpmChallenge.ToByteArray();
Console.WriteLine($"TPM client proof: {BitConverter.ToString(tpmProof)}");
Console.WriteLine($"TPM server proof: {BitConverter.ToString(tpmSecret)}");

ServerConfirm serverConfirm;
if (tpmProof.SequenceEqual(tpmSecret))
{
    // The proof matches the secret, accept the client
    Console.WriteLine("Valid");
    serverConfirm = new(null);
}
else
{
    // The proof is incorrect, reject the client
    Console.WriteLine("Invalid");
    serverConfirm = new("Invalid TPM signature");
}
```

Listing 14: Verifying the TPM proof on the server.

5.5.1.3. Certificate verification

Before a remote TPM's attestation can be relied upon, it is necessary to establish two conditions. Firstly, the EKCert must be authentic, meaning it chains to a trusted manufacturer-controlled root Certificate Authority (CA). Secondly, the certificate needs to have been issued specifically for the TPM use-case and not for an unrelated purpose.

TPM vendors publish their EK root certificates. Microsoft periodically packages the complete set in the publicly available `TrustedTpm.cab` archive. Instead of distributing numerous `.cer` files alongside an executable, these were bundled into a single ZIP archive. A loader can then extract all certificate from this archive.

The code in Listing 15 first opens the ZIP archive, filters for files that appear to be certificates and deserializes them using `X509Certificate2`. The helper returns an `X509Certificate2Collection`, which can later be used as a trust store.

```C#
static class TpmCertificateAuthorities
{
    public static X509Certificate2Collection Load(string zipFilePath)
    {
        var trustedRoots = new X509Certificate2Collection();

        using FileStream zipStream = File.OpenRead(zipFilePath);
        using var archive = new ZipArchive(zipStream, ZipArchiveMode.Read);

        foreach (var entry in archive.Entries)
        {
            bool looksLikeCert = entry.Length > 0 &&
              (entry.Name.EndsWith(".cer", StringComparison.OrdinalIgnoreCase) ||
              entry.Name.EndsWith(".crt", StringComparison.OrdinalIgnoreCase));

            if (!looksLikeCert) continue;

            try
            {
                using var es = entry.Open();
                using var ms = new MemoryStream();
                es.CopyTo(ms);
                var cert = new X509Certificate2(ms.ToArray());
                trustedRoots.Add(cert);
            }
            catch (CryptographicException ex)
            {
                Console.WriteLine($"Bad cert {entry.FullName}: {ex.Message}");
            }
        }

        return trustedRoots;
    }
}
```

Listing 15: Extracting trusted TPM CAs from a ZIP archive.

Once a trust list is available, an `X509Chain` can be configured to bypass the system-wide root store and rely exclusively on the established TPM roots. The `X509ChainPolicy` class offers several relevant settings. The `CustomRootTrust` and `CustomTrustStore` properties limit the chain engine to the previously loaded collection of certificates. The `RevocationMode` and `RevocationFlag` properties determine whether and how online Certificate Revocation List (CRL) or Online Certificate Status Protocol (OCSP) checks are performed. Furthermore, `VerificationFlags` allow for the relaxation of specific errors that might otherwise cause validation

to fail. In Listing 16, the helper method constructs the chain, prints any errors, and terminates the process upon failure.

```csharp
static class TpmCertificate
{
    public static bool Verify(X509Certificate2 ekCert, byte[][] embeddedChain,
    X509Certificate2Collection trustedRoots)
    {
        // Set up the chain with custom trusted roots
        using var chain = new X509Chain();
        // Use only the custom root store (ignore system root store for this
        verification)
        chain.ChainPolicy.TrustMode = X509ChainTrustMode.CustomRootTrust;
        chain.ChainPolicy.CustomTrustStore.Clear();
        chain.ChainPolicy.CustomTrustStore.AddRange(trustedRoots);
        // Add any certificates that are embedded in the TPM (newer Intel CPUs)
        foreach (var item in embeddedChain)
        {
            X509Certificate2 chainCert =
            X509CertificateLoader.LoadCertificate(item);
            // Make sure we are not adding any root certificates
            if (ekCert.Issuer == ekCert.Subject) { return false; }
            chain.ChainPolicy.ExtraStore.Add(chainCert);
        }

        // Enable revocation checking (online if possible)
        chain.ChainPolicy.RevocationMode = X509RevocationMode.Online;
        chain.ChainPolicy.RevocationFlag = X509RevocationFlag.EntireChain;

        // Ignore errors during revocation checking, not all manufacturers provide
        standardized revocation endpoints
        chain.ChainPolicy.VerificationFlags =
X509VerificationFlags.IgnoreCertificateAuthorityRevocationUnknown |
X509VerificationFlags.IgnoreCtlSignerRevocationUnknown |
X509VerificationFlags.IgnoreEndRevocationUnknown;

        bool isChainValid = chain.Build(ekCert);
        if (!isChainValid)
        {
            // Chain build failed, certificate is not trusted
            foreach (X509ChainStatus status in chain.ChainStatus)
            {
                Console.WriteLine($"Chain error: {status.StatusInformation.Trim()}
                (Status: {status.Status})");
            }
            return false;
        }

        return CheckTpmSpecificPurpose(ekCert);
    }
```

Listing 16: Certificate-chain validation against the custom TPM root store.

A valid chain is a necessary condition but not sufficient on its own. An attacker could potentially present a valid certificate issued for a different purpose that chains to a recognized public CA, such as Intel's. The Trusted Computing Group assigns Object Identifier (OID) `2.23.133.8.1` to "TPM Endorsement Key Certificate". Genuine EK certificates include this OID within the "Enhanced Key Usage" extension.

Consequently, `ekCert.Extensions` are parsed, and certificates lacking the OID are rejected, as demonstrated in Listing 17.

```C#
private const string TcgEkOid = "2.23.133.8.1";

private static bool CheckTpmSpecificPurpose(X509Certificate2 cert)
{
    foreach (X509Extension ext in cert.Extensions)
    {
        if (ext is X509EnhancedKeyUsageExtension eku)
            foreach (Oid oid in eku.EnhancedKeyUsages)
                if (oid.Value == TcgEkOid)
                    return true;
    }

    Console.WriteLine(
        $"EK certificate missing mandatory OID {TcgEkOid}");
    return false;
}
```

Listing 17: Rejecting certificates that are not flagged as TPM EK certs.

The `TpmCertificateAuthorities.Load` method combined with `TpmCertifiate.Verify` forms a complete validation pipeline. The code snippet in Listing 18 shows how the server validates the certificate after receiving the TPM attestation.

```C#
ServerConfirm serverConfirm;
// Check client proof against generated secret to validate EK private key ownership
if (tpmProof.SequenceEqual(tpmSecret))
{
    Console.WriteLine("Valid");
    serverConfirm = new(null);
}
else
{
    Console.WriteLine("Invalid");
    serverConfirm = new("Invalid TPM signature");
}

if (serverConfirm.DisconnectReason is null)
{
    // TPM signature is valid, check certificate chain
    var authorities = TpmCertificateAuthorities.Load("tpm-certificates.zip");
    if (!TpmCertificate.Verify(tpmCertificate, authorities))
    {
        Console.WriteLine("Untrusted TPM certificate chain");
        serverConfirm = new("Not a trusted TPM");
    }
}
```

Listing 18: EK certificate validation on the server.

After verifying both the attestation and the certificate's authenticity, the server can trust that the client owns a unique hardware security module. The certificate serves as proof that the hardware component was produced and signed by a trusted manufacturer.

5.5.2. Querying WMI

Windows Management Instrumentation (WMI) is a Component Object Model (COM)-based infrastructure that lets user-mode code read kernel objects that Microsoft exposes as "management classes". Every class can be queried with the WMI Query Language (WQL) and returns one or more `IWbemClassObjects`. To make repetitive WMI access easier we wrap the raw COM interfaces in a utility class `WmiSession` which handles `IWbemLocator`, `IWbemServices`, security initialisation, and provides a single `executeQuery(L"...", Fingerprint&, Processor)` helper

The skeleton in Listing 19 shows how a caller connects to a namespace and runs an arbitrary query; the same pattern is reused in all later sections involving WMI.

```cpp
WmiSession session;
if (session.connect(L"root\\cimv2")) {
    session.executeQuery(
        // WQL
        L"SELECT SerialNumber FROM Win32_DiskDrive",
        // Fingerprint object
        fp,
        // Result callback
        WmiProcessors::ProcessDiskDriveInfo
    );
}
```

Listing 19: Minimal use of `WmiSession` to run a WQL query.

5.5.3. Disk serial

Every mass-storage device ships with a firmware serial number that the controller returns over ATA or NVMe. Windows projects this value through the `Win32_DiskDrive.SerialNumber` property.

The actual WMI request that harvests the numbers is issued in Listing 20. Each returned object is passed to `ProcessDiskDriveInfo`, reproduced afterwards in Listing 21.

```cpp
sessionRootCimV2.executeQuery(
    L"SELECT SerialNumber FROM Win32_DiskDrive",
    fp,
    WmiProcessors::ProcessDiskDriveInfo
);
```

Listing 20: Collecting `SerialNumber` from `Win32_DiskDrive`.

Listing 21 strips vendor padding and appends every non-empty identifier to the fingerprint container.

```cpp
void ProcessDiskDriveInfo(ComPtr<IWbemClassObject>& clsObj, Fingerprint&
fp)
{
    std::wstring serial;
    if (!WmiHelpers::GetStringProperty(clsObj.get(), L"SerialNumber", serial))
        return;

    size_t first = serial.find_first_not_of(L" \t\n\r\f\v");
    if (first == std::wstring::npos)
        return;

    size_t last = serial.find_last_not_of(L" \t\n\r\f\v");
    std::wstring trimmed = serial.substr(first, last - first + 1);

    if (!trimmed.empty()) {
        std::string ascii(trimmed.begin(), trimmed.end());
        fp.add_diskserials(ascii);
    }
}
```

Listing 21: Sanitising the raw string from the disk controller.

5.5.4. Disk volume IDs

Whereas the disk serial originates in the controller firmware, every file system instance written to that disk gets its own volume serial number. NTFS and FAT derive the 32-bit value during the format operation and store it in the partition's boot sector. Windows surfaces the number through the `Win32_LogicalDisk.VolumeSerialNumber` property for each mounted drive.

The anti-cheat iterates over all logical disks by running the WQL statement in Listing 22; each result row is forwarded to the handler shown in Listing 23.

```cpp
sessionRootCimV2.executeQuery(
    L"SELECT VolumeSerialNumber FROM Win32_LogicalDisk",
    fp,
    WmiProcessors::ProcessLogicalDiskInfo
);
```

Listing 22: Collecting `VolumeSerialNumber` from `Win32_LogicalDisk`.

```cpp
void ProcessLogicalDiskInfo(ComPtr<IWbemClassObject>& clsObj, Fingerprint&
fp)
{
    std::wstring vsn;
    if (WmiHelpers::GetStringProperty(clsObj.get(), L"VolumeSerialNumber", vsn)
        && !vsn.empty())
    {
        std::string ascii(vsn.begin(), vsn.end());
        fp.add_volumeserials(ascii);
    }
}
```

Listing 23: Appending the volume serial number to the fingerprint.

5.5.5. CPU serial

Modern x86 processors fuse a unique identifier inside the silicon itself. Intel exposes it as `ProcessorId`, AMD uses an equivalent scheme. WMI maps the value to the `Win32_Processor.ProcessorId` field.

The query in Listing 24 returns exactly one object on single-socket systems. Its contents are forwarded to the callback in Listing 25 for conversion to ASCII and storage.

```cpp
sessionRootCimV2.executeQuery(
    L"SELECT ProcessorId FROM Win32_Processor",
    fp,
    WmiProcessors::ProcessProcessorInfo
);
```

Listing 24: Fetching `ProcessorId` from `Win32_Processor`.

```cpp
void ProcessProcessorInfo(ComPtr<IWbemClassObject>& clsObj, Fingerprint&
fp)
{
    std::wstring pid;
    if (WmiHelpers::GetStringProperty(clsObj.get(), L"ProcessorId", pid)
        && !pid.empty())
    {
        std::string ascii(pid.begin(), pid.end());
        fp.set_cpuserial(ascii);
    }
}
```

Listing 25: Storing the processor identifier.

5.5.6. Monitor serial numbers

Flat-panel displays embed both a `ProductCodeID` and a `SerialNumberID` in the EDID block stored inside the monitor's EEPROM. Through `root\wmi!WmiMonitorID` Windows exposes each array as 16 `LONG`s.

First, the anti-cheat runs the query in Listing 26; next, the bytes are concatenated as demonstrated in Listing 27 to build a unique 32-byte monitor key.

```cpp
sessionRootWmi.executeQuery(
    L"SELECT * FROM WmiMonitorID",
    fp,
    WmiProcessors::ProcessMonitorInfo
);
```

Listing 26: Reading EDID data via `WmiMonitorID`.

```cpp
void ProcessMonitorInfo(ComPtr<IWbemClassObject>& clsObj, Fingerprint& fp)
{
    std::string id(32, '\0');
    bool haveProduct = false, haveSerial = false;

    std::vector<LONG> product;
    if (WmiHelpers::GetLongArrayProperty(clsObj.get(), L"ProductCodeID", product)
        && product.size() == 16)
    {
        for (size_t i = 0; i < 16; ++i)
            id[i] = static_cast<char>(product[i] & 0xFF);
        haveProduct = true;
    }

    std::vector<LONG> serial;
    if (WmiHelpers::GetLongArrayProperty(clsObj.get(), L"SerialNumberID", serial)
        && serial.size() == 16)
    {
        for (size_t i = 0; i < 16; ++i)
            id[i + 16] = static_cast<char>(serial[i] & 0xFF);
        haveSerial = true;
    }

    if (haveProduct || haveSerial)
        fp.add_monitorids(id);
}
```

Listing 27: Building the 32-byte monitor identifier.

5.5.7. Tracking cookies

Aside from using existing system identifiers, a custom marker, also referred to as a cookie, can be stored on the system by the anti cheat. Instead of dropping a browser cookie, we embed a value in the Windows registry. The sub-key `HKEY_LOCAL_MACHINE\SOFTWARE\PocAntiCheat` is created on first start and holds the string value `MachineUUID`. The helper routine shown in Listing 28 encapsulates the complete life-cycle: open or create the key, attempt to load an existing identifier, or fall back to generating and persisting a new one.

If the registry key is present, it is fetched into a wide string, converted to narrow, parsed as UUIDv7 and copied into the binary field `registrykey` of the outgoing fingerprint object. When the registry key is absent, a new UUIDv7 is generated for the device, based on the timestamp and a random number generator. The byte sequence is then converted back into the canonical textual form, widened and written to the registry with `RegSetValueExW`. On every subsequent launch the identifier is then read, allowing easy identification of the device.

```cpp
constexpr LPCWSTR MACHINE_UUID_NAME = L"MachineUUID";
bool registry_trace(Fingerprint &fp) noexcept
{
    RegKey key;
    if (RegCreateKeyExW(HKEY_LOCAL_MACHINE, L"SOFTWARE\\PocAntiCheat", /* ... */) != ERROR_SUCCESS)
    {
        return false;
    }
    DWORD type{}, size{};
    // Query existing value
    if (RegQueryValueExW(key.h, MACHINE_UUID_NAME, nullptr, &type, nullptr, &size) == ERROR_SUCCESS && type == REG_SZ && size > 0)
    {
        std::wstring value(size / sizeof(wchar_t), L'\0');
        if (RegQueryValueExW(key.h, MACHINE_UUID_NAME, nullptr, nullptr, wstring_as_byte(value), &size) != ERROR_SUCCESS) { return false; }
        // Convert wide-string to narrow for parsing
        std::string ascii{value.begin(), value.end()};
        fp.set_registrykey(std::string(16, '\0'));
        if (uuidv7_from_string(ascii.c_str(), string_as_byte(*fp.mutable_registrykey())) != 0)
            return false;
        return true;
    }
    // Generate new UUIDv7
    std::array<uint8_t, 10> randBytes;
    if (BCryptGenRandom(nullptr, randBytes.data() /* ... */) != 0)
        return false;
    fp.set_registrykey(std::string(16, '\0'));
    uuidv7_generate(string_as_byte(*fp.mutable_registrykey()), current_timestamp(), randBytes.data(), nullptr);
    // Convert to string
    std::array<char, 37> uuidStr{};
    uuidv7_to_string(string_as_byte(*fp.mutable_registrykey()), uuidStr.data());
    std::wstring wide(uuidStr.begin(), uuidStr.end());
    return RegSetValueExW(key.h, MACHINE_UUID_NAME, 0, REG_SZ, reinterpret_cast<const BYTE *>(wide.c_str()) /* ... */) == ERROR_SUCCESS;
}
```

Listing 28: Creating or loading the persistent machine UUID.

5.5.8. Network

Network fingerprinting uses various identifiers and characteristics associated with the networking infrastructure a client uses to connect to the game server.

This can be integrated into the fingerprint and help identify and track clients more reliably.

5.5.8.1. IP address

Because the client must use a valid IP address to communicate with the game server, the public IP address can be taken directly from the game connection. This is implemented by taking the peer address after an initial connection is established, as shown in Listing 29.

```
1  public void OnNetworkReceive(NetPeer peer, NetPacketReader reader, byte
   channelNumber, DeliveryMethod deliveryMethod)                                C#
2  {
…                                          …
3    switch (message)
4    {
5        case ClientHello hello:
…                                          …
6          // ... other fingerprint components
7          .Append((Label: "IP", Data: peer.Address.GetAddressBytes()))
```

Listing 29: Adding a component for the public IP address of a client.

By saving the raw IP address bytes both IPv4 and IPv6 addresses are supported. The prevalence of IP address sharing between customers significantly reduces the certainty associated with an IPv4 address as a unique identifier. Consequently, it is assigned a low confidence score within the system. This low score ensures that individuals who happen to share an IPv4 address but are otherwise distinct users, such as neighbors using the same ISP with CGNAT, are not automatically banned. However, shared IP addresses might still trigger alerts for moderators to investigate further if other suspicious activity is detected.

5.5.8.2. VPN detection

To prevent users from obscuring their true IP address, the system maintains a list of known VPN and datacenter IP ranges. This list is periodically fetched from a remote source and cached locally to reduce network overhead. First, a `VpnList` object is created and initialized, downloading the latest Classless Inter-Domain Routing (CIDR) entries if the cache is stale and parsing them into internal arrays. As shown in Listing 30, the `VpnList` is constructed with the source URL and cache filename, then `InitializeAsync()` is called to ensure the cache is up to date.

```csharp
// Prepare and initialize VPN list, downloading if cache is older than one day
VpnList vpnList = new("https://raw.githubusercontent.com/X4BNet/lists_vpn/refs/heads/main/output/datacenter/ipv4.txt", "vpns.txt");
await vpnList.InitializeAsync();
new Server(port, conn, vpnList).Run();
```

Listing 30: Instantiating and initializing the VPN list before server startup.

Internally, the `VpnList` class downloads the remote text file of CIDR blocks, saves it to a local cache, and then parses each non-comment line into a `(start, end)` tuple representing the IPv4 range. These tuples are sorted by their starting IP to enable efficient binary search. The core parsing logic and array setup are presented in Listing 31.

```C#
public class VpnList
{
    private uint[] _starts = [];
    private uint[] _ends = [];

    private void LoadAndParseCache()
    {
        var entries = File.ReadAllLines(_cacheFile)
                          .Where(l => !string.IsNullOrWhiteSpace(l) && !
                          l.StartsWith('#'))
                          .Select(ParseCidr)
                          .Where(r => r != null)!
                          .Select(r => r!.Value)
                          .ToList();

        // Sort entries by starting address for later binary search
        entries.Sort((a, b) => a.start.CompareTo(b.start));

        _starts = entries.Select(r => r.start).ToArray();
        _ends   = entries.Select(r => r.end).ToArray();
    }

    private static (uint start, uint end)? ParseCidr(string line)
    {
        var parts = line.Trim().Split('/');
        if (parts.Length != 2) return null;
        if (!IPAddress.TryParse(parts[0], out var ip)) return null;
        if (!int.TryParse(parts[1], out var prefixLen)) return null;
        if (ip.AddressFamily != System.Net.Sockets.AddressFamily.InterNetwork ||
            prefixLen < 0 || prefixLen > 32) return null;

        uint ipUint = IpToUint(ip);
        uint mask = 0xFFFFFFFF << (32 - prefixLen);

        uint start = ipUint & mask;
        uint end = start + (uint)((1UL << (32 - prefixLen)) - 1);
        return (start, end);
    }
}
```

Listing 31: Parsing VPN address ranges.

Once the list is loaded into memory, each incoming connection request is checked against the stored ranges. Using a binary search on the sorted `_starts` array, the code determines if the client's IP falls within any of the known VPN or

datacenter subnets, as shown in Listing 32. If a match is found, the connection is rejected immediately, as illustrated in Listing 33.

```C#
public class VpnList
{
    /// <summary>
    /// Checks if the given IPv4 address is within any of the banned subnets.
    /// </summary>
    public bool Contains(IPAddress ip)
    {
        if (_starts.Length == 0)
            throw new InvalidOperationException("Call InitializeAsync() first.");
        if (ip.AddressFamily != System.Net.Sockets.AddressFamily.InterNetwork)
            return false;

        uint ipUint = IpToUint(ip);
        int idx = Array.BinarySearch(_starts, ipUint);
        if (idx >= 0)
        {
            // Exact match at the start of a subnet
            return true;
        }
        else
        {
            int ins = ~idx - 1;
            if (ins >= 0 && _ends[ins] >= ipUint)
            {
                // Falls within the previous subnet
                return true;
            }
        }
        return false;
    }
}
```

Listing 32: Checking if an IP address is in a banned range.

```C#
public void OnConnectionRequest(ConnectionRequest request)
{
    var clientAddress = request.RemoteEndPoint.Address;
    if (_vpnList.Contains(clientAddress))
    {
        Console.WriteLine($"Connection from VPN/datacenter detected: {clientAddress}, blocking");
        var serverConfirm = new ServerConfirm("Connections from VPNs are not allowed");
        byte[] payload = Encoding.UTF8.GetBytes(
          JsonSerializer.Serialize((Message)serverConfirm)
        );
        request.Reject(payload);
        return;
    }

    // Normal connection handling...
}
```

Listing 33: Rejecting connections from VPN or datacenter IP addresses.

By integrating this VPN detection mechanism, the anti-cheat can enforce IP-based policies more reliably. Disallowing VPNs can mitigate attempts to hide evasion or bypass regional restrictions.

5.5.8.3. Local devices

Since the anti cheat software runs locally on the user's device, it has access to the local network environment. This allows it to gather information about other devices connected to the same network segment.

Sophisticated users might employ Virtual LANs (VLANs) to segment their network and prevent the anti cheat software from discovering other devices. However, nearly all local networks require a gateway device (typically a router) to facilitate communication with the internet. The MAC address of this gateway router can be fingerprinted. Spoofing the gateway's MAC address is generally more difficult than spoofing the MAC address of the local computer itself. Modifying the router's MAC address might require flashing custom firmware. Simply overwriting or spoofing the gateway MAC locally on the client machine would likely disrupt network routing, preventing connectivity. Actively spoofing it only for the game or anti cheat process requires more sophisticated techniques.

The system can be configured to collect a specific number `n` of MAC addresses observed on the local network. To minimize generating additional network traffic, the implementation selects the lowest `n` MAC addresses currently present in the system's Address Resolution Protocol (ARP) table. While the specific set of

visible MAC addresses might change over time as devices connect and disconnect, collecting multiple MAC addresses increases the probability of finding a match across different connection sessions for the same user. By sorting the list of addresses first, the chance of repeatedly finding the same devices increases. Tuning the value of `n` represents a trade off: increasing `n` improves the chances of finding a matching identifier but also increases the amount of data that needs to be stored and processed.

The code determines which network interface is used for internet traffic and then finds the gateway's IP address. This step filters out virtual adapters and focuses on the real LAN segment used by the system, as outlined in Listing 34.

```cpp
// Pick a well-known public IP (8.8.8.8) to identify the main interface
SOCKADDR_INET dst{};
dst.si_family = AF_INET;
dst.Ipv4.sin_addr.s_addr = htonl(0x08080808);

DWORD bestIfIdx = 0;
if (GetBestInterface(dst.Ipv4.sin_addr.s_addr, &bestIfIdx) != NO_ERROR) {
    return false;
}

// Retrieve the route to that IP, to extract the gateway address
uint32_t gatewayIp = 0;
MIB_IPFORWARD_ROW2 bestRoute{};
SOCKADDR_INET bestSourceAddress;
if (GetBestRoute2(nullptr, bestIfIdx, nullptr, &dst, 0, &bestRoute,
&bestSourceAddress) == NO_ERROR) {
    gatewayIp = ntohl(bestRoute.NextHop.Ipv4.sin_addr.s_addr);
}
```

Listing 34: Identifying the interface index and gateway IP address.

Next, the ARP table is retrieved and iterated. Valid entries are converted to MAC arrays, with the gateway's MAC captured separately. Duplicates are skipped to minimize noise, as shown in Listing 35.

```cpp
PMIB_IPNET_TABLE2 tbl = nullptr;
if (GetIpNetTable2(AF_UNSPEC, &tbl) != NO_ERROR || !tbl) {
    return false;
}

std::vector<std::array<uint8_t, 6>> macs;
macs.reserve(tbl->NumEntries);

std::array<uint8_t, 6> gatewayMac{};
bool haveGatewayMac = false;

for (DWORD i = 0; i < tbl->NumEntries; ++i) {
    const auto& row = tbl->Table[i];
    if (!arp_row_is_valid(row))
        continue;

    auto mac = arp_row_to_mac(row);

    // Check if this row is the gateway, and remember its MAC
    if (!haveGatewayMac && row.Address.si_family == AF_INET) {
        uint32_t ip = ntohl(row.Address.Ipv4.sin_addr.s_addr);
        if (ip == gatewayIp) {
            gatewayMac     = mac;
            haveGatewayMac = true;
        }
        continue;
    }

    // Skip duplicates
    if (std::find_if(macs.begin(), macs.end(),
        [&mac](const auto& m) { return mac_equal(m, mac); }) == macs.end())
    {
        macs.push_back(mac);
    }
}
FreeMibTable(tbl);
```

Listing 35: Fetching the ARP table and extracting unique MAC addresses.

Finally, to ensure consistent selection the MAC list is sorted, as this increases the chance the same MAC addresses are gathered from the network. If there are more entries than needed, it is truncated to the lowest `n` addresses. The router's MAC address is added to the front of the list, to ensure it is always present. Each chosen MAC is then appended to the fingerprint object, as detailed in Listing 36.

```cpp
// Sort to make selection repeatable across runs
std::sort(macs.begin(), macs.end());
// Ensure gateway is always included at position zero
if (haveGatewayMac) {
    macs.insert(macs.begin(), gatewayMac);
}
if (macs.empty()) {
    return false;
}
// Trim to desired count
if (macs.size() > n) {
    macs.resize(n);
}
// Add to the fingerprint
for (const auto& m : macs) {
    fp.add_macaddresses(std::string(m.begin(), m.end()));
}
return true;
```

Listing 36: Processing MAC addresses.

5.6. Kernel module

To improve client-side security, a kernel-mode component is deployed by the anti-cheat system. This module executes integrity checks directly within the operating system kernel, providing a more privileged position for detecting common evasion and tampering techniques. Upon loading, it performs three assessments: retrieval of the baseboard serial number from SMBIOS, scanning for unauthorized modifications to kernel dispatch routines, and verifying the system's test signing mode. The findings from these checks are subsequently communicated to the user-mode component of the anti-cheat system and then forwarded to the server in the fingerprint.

The initialization of the driver and execution of these checks occur within the `DriverEntry` routine, which is the standard entry point for Windows drivers. As demonstrated in Listing 37, after the necessary device object and symbolic link are created to enable user-mode communication, the functions performing the assessments are invoked.

```cpp
// DriverEntry: Initializes the driver.
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
    NTSTATUS status;
    PDEVICE_OBJECT DeviceObject = NULL;
    UNICODE_STRING deviceName;
    UNICODE_STRING symLinkName;

    RtlInitUnicodeString(&deviceName, L"\\Device\\AnticheatPoc");
    RtlInitUnicodeString(&symLinkName, L"\\DosDevices\\AnticheatPoc");

    status = IoCreateDevice(
        DriverObject,
        0,
        &deviceName,
        FILE_DEVICE_UNKNOWN,
        0,
        FALSE,
        &DeviceObject);
    status = IoCreateSymbolicLink(&symLinkName, &deviceName);

    DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchDeviceControl;
    for (int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++) {
        if (DriverObject->MajorFunction[i] == NULL) {
            DriverObject->MajorFunction[i] = DispatchDefault;
        }
    }
    DriverObject->DriverUnload = DriverUnload;

    // Run checks
    SmbiosIdentifiers();
    ScanIrpHandlers();
    g_TestSign = IsTestSignModeEnabled();

    return STATUS_SUCCESS;
}
```

Listing 37: Kernel module initialization and execution of primary integrity checks in `DriverEntry`.

A unique hardware identifier, the baseboard serial number, is retrieved directly from the SMBIOS data. This is performed to detect attempts by user-mode

applications to spoof this identifier. The `SmbiosIdentifiers` function, shown in Listing 38, initiates this process by calling `ZwQuerySystemInformation` with the `SystemFirmwareTableInformation` class and `FIRMWARE_TABLE_PROVIDER_SMBIOS` provider. The returned SMBIOS table is then parsed. The code iterates through SMBIOS structures until it locates the Type 2 structure, which corresponds to Baseboard Information. The serial is cached and later compared by the anti-cheat server against the serial number obtained through standard user-mode interfaces. A mismatch between these values results in a significant reduction of the client's trust score, leading to rejection of the client.

```cpp
static CHAR g_BiosSerial[BIOS_SERIAL_LEN] = {0};

VOID SmbiosIdentifiers(VOID) {
    NTSTATUS status;
    PSYSTEM_FIRMWARE_TABLE_INFORMATION pFirmwareTableInfoAlloc = NULL;
    PVOID pSmbiosData = NULL;
    ULONG smbiosDataLengthActual = 0;
    // ... determining table size
    pFirmwareTableInfoAlloc->ProviderSignature = FIRMWARE_TABLE_PROVIDER_SMBIOS;
    pFirmwareTableInfoAlloc->Action = FIRMWARE_TABLE_ACTION_GET_TABLE;
    pFirmwareTableInfoAlloc->TableID = 0;
    pFirmwareTableInfoAlloc->TableBufferLength = smbiosDataSizeNeeded;
    ULONG bytesWritten = 0;
    status = ZwQuerySystemInformation(
        SystemFirmwareTableInformation,
        pFirmwareTableInfoAlloc,
        fullAllocSize,
        &bytesWritten
    )
    // ... error handling
    PRAW_SMBIOS_DATA raw = (PRAW_SMBIOS_DATA)pFirmwareTableInfoAlloc->TableBuffer;
    pSmbiosData = raw->SMBIOSTableData;
    smbiosDataLengthActual = raw->Length;
    PCHAR smbiosDataEnd = (PCHAR)pSmbiosData + smbiosDataLengthActual;
    PUCHAR currentPtr = (PUCHAR)pSmbiosData;

    while (currentPtr < (PUCHAR)smbiosDataEnd) {
        // ... looping through entries in the table
        if (pHeader->Type == 2) { // Baseboard Information
            PSMBIOS_TYPE2_INFO pType2 = (PSMBIOS_TYPE2_INFO)pHeader;
            if (pHeader->Length >= FIELD_OFFSET(SMBIOS_TYPE2_INFO, SerialNumber) +
            sizeof(pType2->SerialNumber)) {
                PCHAR serial = GetSmbiosStringInternal(pHeader, pType2-
                >SerialNumber, smbiosDataEnd);
                RtlStringCchCopyA(g_BiosSerial, BIOS_SERIAL_LEN, serial);
            }
        }
        // ...
    }
    // ...
}
```

Listing 38: Extracting the baseboard serial number from SMBIOS.

The integrity of I/O Request Packet (IRP) dispatch routines for all loaded kernel drivers is also checked. This allows detection of kernel-level hooks that redirect

system operations for spoofing hardware results. The `ScanIrpHandlers` function, shown in Listing 40, orchestrates this by first obtaining a list of all loaded kernel modules using `ZwQuerySystemInformation` within the `RefreshSystemModuleInformation` function. It then enumerates driver objects within the system's `\Driver` directory using `ZwOpenDirectoryObject` and `ZwQueryDirectoryObject`. For each driver object, the `ScanDriverIrpHandlers` function, shown in Listing 39, iterates through its `MajorFunction` array, which contains pointers to its IRP handlers. The `IsAddressInValidKernelModule` function, also shown in Listing 39, validates each handler's address, ensuring it falls within the memory range of a known, legitimately loaded kernel module. If a handler points outside of any recognized module, such as a malicious module manually placed into the kernel, it is flagged as a hook. Such a finding severely reduces the client's trust score.

Some spoofing tools modify IRP tables to point to existing routines in legitimate kernel modules which return no data and report an error. This cannot be detected using this IRP address range check. The system instead relies on the trust score being affected because no data is available, potentially preventing the client from connecting.

```cpp
static BOOLEAN g_DriverHook = FALSE;

BOOLEAN IsAddressInValidKernelModule(PVOID Address) {
    for (ULONG i = 0; i < g_ModuleInformation->NumberOfModules; ++i) {
        PRTL_PROCESS_MODULE_INFORMATION module = &g_ModuleInformation->Modules[i];
        if (Address >= module->ImageBase &&
            Address < (PVOID)((PUCHAR)module->ImageBase + module->ImageSize)) {
            return TRUE;
        }
    }
    return FALSE;
}

VOID ScanDriverIrpHandlers(PDRIVER_OBJECT DriverObject) {
    // ... sanity checks
    for (ULONG i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; ++i) {
        PDRIVER_DISPATCH irpHandler = DriverObject->MajorFunction[i];
        if (irpHandler != NULL) {
            if (!IsAddressInValidKernelModule(irpHandler)) {
                g_DriverHook = TRUE;
            }
        }
    }
}
```

Listing 39: Validating kernel IRP routines by checking valid module ranges.

```cpp
NTSTATUS ScanIrpHandlers() {
    // ...
    status = RefreshSystemModuleInformation();
    // ... error handling

    // Enumerate Driver Objects from \Driver directory
    RtlInitUnicodeString(&dirName, L"\\Driver");
    InitializeObjectAttributes(&objAttr, &dirName, OBJ_KERNEL_HANDLE |
    OBJ_CASE_INSENSITIVE, NULL, NULL);
    status = ZwOpenDirectoryObject(&dirHandle, DIRECTORY_QUERY |
    DIRECTORY_TRAVERSE, &objAttr);
    // ... error handling

    ULONG context = 0;
    BOOLEAN firstQuery = TRUE;
    UCHAR buffer[1024];
    POBJECT_DIRECTORY_INFORMATION dirInfo = (POBJECT_DIRECTORY_INFORMATION)buffer;
    while (TRUE) {
        status = ZwQueryDirectoryObject(dirHandle, dirInfo, sizeof(buffer), TRUE,
        firstQuery, &context, &requiredSize);
        firstQuery = FALSE;
        // ... error handling

        if (/* sanity check dirInfo */) {
            // We are interested in objects of type "Driver"
            if (RtlCompareUnicodeString(&dirInfo->TypeName, &g_DriverType, TRUE)
            == 0) {
                UNICODE_STRING objectName;
                // ... construct full path: \Driver\<name>
                PDRIVER_OBJECT pDriverObject = NULL;
                NTSTATUS refStatus = ObReferenceObjectByName(&objectName,
                OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, NULL, 0,
                *IoDriverObjectType, KernelMode, NULL, (PVOID*)&pDriverObject);

                if (NT_SUCCESS(refStatus) && pDriverObject) {
                    // Scan the driver object
                    ScanDriverIrpHandlers(pDriverObject);
                    ObDereferenceObject(pDriverObject);
                }
            }
        }
    }
    // ... cleanup
    return status;
}
```

Listing 40: Scanning all loaded driver objects.

The system's test signing mode is also verified, as its activation significantly lowers the security of the system by allowing unsigned or self-signed drivers to load. The `IsTestSignModeEnabled` function, detailed in Listing 41, queries the system's code integrity status using `ZwQuerySystemInformation` with the `SystemCodeIntegrityInformation` class. The `CodeIntegrityOptions` field within the returned `SYSTEM_CODEINTEGRITY_INFORMATION` structure is examined. If the `CODEINTEGRITY_OPTION_TESTSIGN` flag (0x2) is set, or if the `CODEINTEGRITY_OPTION_ENABLED` flag (0x1) is not set, it indicates that test signing is active or standard integrity checks are disabled. This condition results in a severe reduction of the client's trust score.

```cpp
static BOOLEAN g_TestSign = FALSE;

BOOLEAN IsTestSignModeEnabled()
{
    SYSTEM_CODEINTEGRITY_INFORMATION sci;
    NTSTATUS status;

    sci.Length = sizeof(sci);
    status = ZwQuerySystemInformation(
        SystemCodeIntegrityInformation, // Value 103
        &sci,
        sizeof(sci),
        NULL);

    // CODEINTEGRITY_OPTION_ENABLED             0x01
    // CODEINTEGRITY_OPTION_TESTSIGN            0x02
    if (NT_SUCCESS(status) &&
        ((sci.CodeIntegrityOptions & 0x2) || !(sci.CodeIntegrityOptions & 0x1)))
    {
        return TRUE;
    }
    return FALSE;
}
```

Listing 41: Verifying system code integrity status.

The results gathered from these kernel-level checks are consolidated into a `FINGERPRINT_KERNEL` structure. This structure (containing `biosSerial`, `kernelHooks`, and `testSigning` members) is then made accessible to the user-mode anti-cheat component via an IOCTL request. As shown in Listing 42, the `DispatchDeviceControl` routine handles the `IOCTL_GET_FINGERPRINT` request by populating the user-supplied output buffer with the collected integrity data. This

enables the user-mode client to transmit these crucial kernel-level indicators to the server for a comprehensive trust assessment.

```cpp
#define IOCTL_GET_FINGERPRINT CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define BIOS_SERIAL_LEN 64
typedef struct {
    CHAR  biosSerial[BIOS_SERIAL_LEN];
    BOOLEAN kernelHooks;
    BOOLEAN testSigning;
} FINGERPRINT_KERNEL, *PFINGERPRINT_KERNEL;

NTSTATUS
DispatchDeviceControl(_In_ PDEVICE_OBJECT DeviceObject, _Inout_ PIRP Irp)
{
    // ...
    NTSTATUS status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG bytes  = 0;

    // Request for fingerprint data
    if (code == IOCTL_GET_FINGERPRINT) {
        ULONG outLen = irpSp->Parameters.DeviceIoControl.OutputBufferLength;
        if (outLen < sizeof(FINGERPRINT_KERNEL)) {
            status = STATUS_BUFFER_TOO_SMALL;
            bytes  = sizeof(FINGERPRINT_KERNEL);
        } else {
            PFINGERPRINT_KERNEL out = (PFINGERPRINT_KERNEL)Irp->AssociatedIrp.SystemBuffer;
            RtlZeroMemory(out, sizeof(FINGERPRINT_KERNEL));

            out->kernelHooks = g_DriverHook;
            out->testSigning = g_TestSign;
            RtlStringCchCopyA(out->biosSerial, BIOS_SERIAL_LEN, g_BiosSerial);
            status = STATUS_SUCCESS;
            bytes  = sizeof(FINGERPRINT_KERNEL);
        }
    }
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = bytes;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

Listing 42: Handling requests from the user-mode anti-cheat component.

By incorporating these verification mechanisms directly within a kernel module, the anti-cheat system establishes a more privileged position for assessing the integrity of the client environment. This approach is necessitated by the prevalence of cheats leveraging kernel modules for hardware spoofing.

6. Evaluation

The proof-of-concept anticheat underwent testing to assess its effectiveness in realistic conditions. This section presents the results of all test scenarios and evaluates the system's impact on connection performance.

6.1. Test scenarios

The results of all test cases defined in Section 4.2.2 are recorded in Table 4. All scenarios executed as expected: no test case produced an unexpected server decision. The score column in these results indicates the system's confidence that a user is attempting evasion, with scores above 30 leading to flagging for later moderator review, and scores above 100 resulting in an immediate ban. The Trust score column measures the system's confidence in the integrity of the player's device. This score starts at 100, is beneficially influenced if a TPM is present, as seen in the many trust scores of 200, and is reduced if spoofing attempts are detected, as seen in the later examples utilizing spoofers. A Trust score below 20 prevents the client from joining, as their system cannot be confidently fingerprinted. The observed outcome column shows the decision taken for each test, with the pass/fail column showing if the system performed as expected. An example of the server's analysis can be seen in Figure 6.

Scenario	Observed Outcome	Pass/Fail	Score	Trust Score
T01	Allow	Pass	0	200
T02	Flag	Pass	40	200
T03	Flag	Pass	60	200
T04	Allow	Pass	20	200
T05	Block	Pass	N/A	N/A
T06	Block	Pass	600	200
T07	Block	Pass	500	200
T08	Block	Pass	430	200
T09	Flag	Pass	40	200
T10	Block	Pass	N/A	N/A
T11	Allow	Pass	0	200
T12	Block	Pass	780	200
T13	Block	Pass	680	200
T14	Block	Pass	N/A	N/A
T15	Block	Pass	710	200
T16	Block	Pass	610	200
T17	Block	Pass	510	100
T18	Block	Pass	730	−50
T19	Block	Pass	630	−50
T20	Block	Pass	N/A	N/A
T21	Block	Pass	710	−50
T22	Block	Pass	610	−50
T23	Block	Pass	420	−150

Table 4: Test results of the proof-of-concept system in detecting ban evasion.

Figure 6: Screenshot of the server output for scenario T08.

Despite matching all of the expected test results, the proof-of-concept does not match the ideal outcomes of allowing all unbanned players and blocking all evading players. In two cases, a normal player was flagged as suspicious, due to certain similarities with an actual evasion attempt. In two other cases, an evading player was either flagged or allowed instead of being blocked. This results in a false-positive ban rate of 0%, a false-positive flag rate of 50% as well as a false-negative rate of 10.5%. The false-positive flag rate is high due to the test scenarios being deliberately chosen to challenge the system, and cases such as sharing a monitor or public wifi are overrepresented and they are expected to be uncommon in practice.

No false-positive ban was observed. However, the experiment comprised only two physical devices. The absence of duplicate identifiers across this limited sample does not guarantee a low false-positive rate at scale. A trial involving hundreds of clients is recommended before enforcing automated bans. The production system should additionally ignore any identifier that is shared by a large number of accounts, thereby treating common values as non-discriminative.

Data center VPNs (ProtonVPN) were flagged and blocked as intended. Residential VPNs are able to masquerade as home connections and therefore bypass the IP-based block list. Because residential VPNs are more expensive, their use increases the monetary cost of evasion.

The publicly available user-mode spoofer had no effect on detection, despite most fingerprint gathering occurring in user mode. The kernel-mode spoofer was able to prevent the gathering of MAC addresses, as well as spoofing the serial number of one of the two disks present in the system. Due to the large number of other identifiers that were unchanged, the evasion was still detected. Due to the spoofer using detectable kernel hooks and blocking all MAC addresses, its

trust score was low and it would not have been allowed to connect to the server, even if no other identifier was matched. No open-source spoofer tested during the evaluation succeeded in bypassing the proof-of-concept.

The proof-of-concept matched its specification across twenty-three representative scenarios. Open-source HWID spoofers, datacenter VPNs and naive account resets were effectively detected and blocked. No false-positive bans were recorded, though broader trials are necessary to establish reliability at scale. Resource overhead on commodity server hardware is modest, indicating suitability for real-world deployment after further validation.

6.2. Performance

Figures 7 and 8 are flame graphs created using the Tracy profiler, where each image contains two panels: the top panel shows server-side activity and the bottom panel shows client-side activity. Figure 8 is a zoomed-in view of the final segment of Figure 7. Profiling the fingerprinting and authentication workflow reveals a total handshake duration of approximately 11 seconds. Within this interval, TPM-based operations account for about 8 seconds, while the remaining 3 seconds contain software-based fingerprint gathering, key derivation, network transmission, and server-side verification.

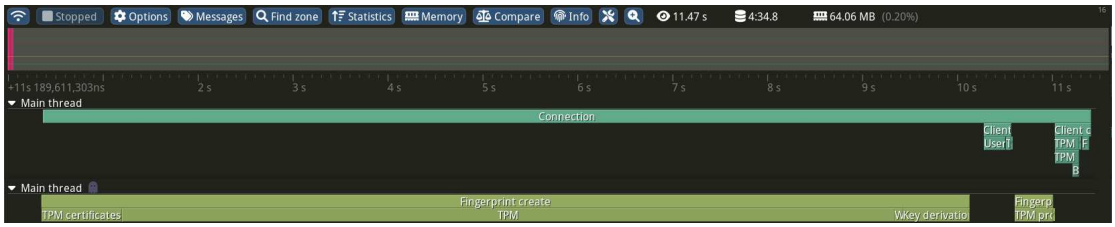

Figure 7: Tracy profile of server-client handshake.

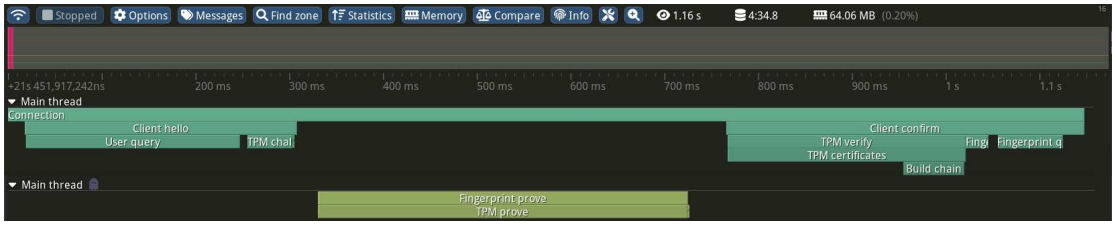

Figure 8: Zoomed-in final part of Tracy profile.

The dominant client-side bottleneck is TPM attestation key generation. The TPM must create a new key pair which takes a considerable amount of time on the tested TPMs. To reduce this component's duration, selecting faster

cryptographic parameters could be an option. Using elliptic-curve algorithms in place of a 2048-bit RSA key could shorten TPM key-generation time, provided that the chosen parameters are compatible with the TPM's firmware and the attestation process.

Software-based fingerprinting tasks, such as collecting system properties, hashing configuration files, and querying registry values, complete in less than 100 milliseconds. The key-derivation routine using Argon2 takes approximately 700 milliseconds. Scheduling the key-derivation in parallel with the TPM operations would allow higher difficulty settings for Argon2, increasing resistance to precomputation attacks without extending the time taken to connect to a server.

Running all fingerprinting tasks in a background thread could allow the main game client to begin loading assets simultaneously. If asset loading finishes before fingerprinting completes, the server could admit the player into a lobby or limited state while awaiting the remaining fingerprint data. To prevent a client from delaying fingerprint submission indefinitely, a server-enforced timeout should be applied. Any client that fails to complete fingerprint transmission within this window would then be disconnected.

On the server side, generating challenges and verifying non-TPM keys is negligible, as seen in Figure 8. The aspect that requires more time is TPM certificate verification, which involves checking the client endorsement certificate against a trusted certificate chain. One improvement would be to load the entire list of TPM root certificates into memory once at startup, thereby avoiding repeated disk reads or parsing for every incoming connection. Ensuring any certificate revocation lists are cached could also reduce the time needed when verifying the certificate chain.

Implementing faster TPM cryptographic parameters, running key derivation alongside hardware operations, performing fingerprint collection in a background thread, allowing players to join before finishing the fingerprint, and caching certificate data in memory together will streamline the handshake process. With these optimizations in place, fingerprinting can occur transparently during the game's loading sequence without perceptible delays.

7. Outlook

The prototype presented here establishes that hardware-anchored, multi-signal fingerprints can reliably expose casual ban evaders, but moving from an academic proof-of-concept to a production-grade system demands several additional efforts. First, the evaluation must be repeated at much greater scale. Rather than testing on a handful of lab machines, the system should be deployed across hundreds or even thousands of heterogeneous devices to measure identifier collision rates in the wild and to surface edge cases in detection.

It is critical to subject the identifier protocol to adversaries wielding mature, commercially available spoofing tools. While initial experiments used open-source kernel and user-mode spoofers to validate basic resistance, underground cheat vendors frequently offer paid, feature-rich binaries tailored to popular titles. Acquiring and testing these tools against the prototype will shed light on which hardware and network fields they already spoof flawlessly and quantify the effort that cheaters must expend to successfully bypass each identifier.

The cryptographic robustness of the identifier protocol also warrants deeper analysis. Although the current challenge–response protocol leverages a deterministic Argon2id key derivation to avoid transmitting raw fingerprints to the server, the possibility of a precomputation attack remains. Modern consumer-grade GPUs with tens of gigabytes of VRAM are increasingly affordable, making large-scale brute-force enumeration of likely hardware states feasible. If such precomputed tables are feasible to construct, the protocol must evolve to incorporate longer derived keys or increase the difficulty used for the algorithm.

Hardening the client implementation against reverse engineering and tampering is another vital step. The current user-mode DLL and unsigned kernel driver can be relatively easily dismantled through static analysis. A production-grade client should perform self-hashing, cross-verify its integrity against a signed boot-time driver and apply code obfuscation to frustrate disassembly. Acquiring a legitimate driver signing certificate will ensure that the kernel component loads seamlessly under Windows security policies such as Device Guard. Alternatively, integrating the fingerprint logic directly into an established anti-cheat framework would inherit their hardened security, albeit at the cost of relying on proprietary software.

To lower the barrier for adoption by existing developers, the anti-cheat functionality must be decoupled into consumable packages. At a minimum, this entails splitting out the core finger-printing code into a software development kit, with bindings for popular engines such as Unity, Unreal Engine, and Godot, and building the kernel-level component as a standalone, signed driver that can be straightforwardly integrated into custom launchers. By providing these artifacts

under a permissive licence for open-source projects and a commercial licence for closed-source games, the project could foster broad uptake while ensuring sustainable funding for continued maintenance.

Beyond hardware and network signals, a fundamentally different approach to identification is machine learning-driven behavioural analytics. Prior research in fraud detection demonstrates that fine-grained user behaviors can be linked even when users switch hardware or IP address (Niverthi et al., 2022). A preliminary behavioural pipeline would collect high-resolution input telemetry, extract statistical features such as average turn rates, jump-to-crouch ratios, and recoil compensation curves, and employ unsupervised clustering methods to group behavioural traces. When a connection's hardware fingerprint appears novel but its behavioural profile clusters closely with a known banned user, the system can flag this as potential ban evasion. Because this approach targets the human patterns rather than the device, it could catch evasion in scenarios where hardware-based fingerprinting is not effective.

Bringing these elements together enables a layered and resilient anti-evasion strategy. Hardware-based roots of trust, such as TPM-sealed keys, tie identity directly to the physical device. Cryptographic mechanisms prevent replay attacks. Tamper resistance through code signing and obfuscation hinders reverse engineering. Broad, real-world deployment surfaces rare edge cases and informs empirical tuning. Behavioural analytics complement hardware-based techniques by identifying repeat offenders through their unique in-game actions, even when they change devices or networks. While the current prototype serves as a foundation, this roadmap outlines how it can evolve into a production-ready solution to ban evasion.

8. Appendix

The paper is submitted alongside a ZIP archive containing the files listed in Table 5.

Folder / File	Description
Test Report.xlsx	Excel file used to track test scenarios and the evaluation results.
Web-Abstract.pdf	Web Abstract for use in the Portfolio Database.
proof-of-concept/	Folder containing the code for the proof-of-concept system.

Table 5: Appendix contents.

Aufgabenstellung

Modul:	Dept I BAA FS25

Titel:	Anti-Cheat Ban Evasion Detection
Ausgangslage und Problemstellung:	Ban evasion by malicious players remains a persistent challenge in free-to-play online games. Since creating new accounts typically incurs no financial cost, banned users can easily rejoin games and continue disruptive behavior. This undermines the integrity of the gaming environment and diminishes the experience of legitimate players. Effective detection and prevention mechanisms for ban evasion are therefore critical for maintaining fair play and trust within gaming communities.
Ziel der Arbeit und erwartete Resultate:	* Research and Analysis: Explore and assess various techniques for detecting ban evasion, focusing on identifying persistent traces such as file artifacts, hardware serial numbers, Trusted Platform Module (TPM) data, and other potential indicators. * Prototype Development: Build a proof-of-concept system that integrates userland and kernel components to record, analyze, and compare these identifiers. * Implementation Strategies: Investigate secure and ethical methods for implementing fingerprinting techniques in games, particularly those with community servers. * Documentation of Results: Clearly document the findings, methodologies, and outcomes of the project to ensure replicability and provide insights for future research.
Gewünschte Methoden, Vorgehen:	* Conduct a literature review of current approaches and best practices in ban evasion detection and anti-cheat technologies. * Design and implement a prototype that a range of various identifiers and uses them for detecting ban evasion. * Evaluate the prototype within a controlled test environment to assess its accuracy, reliability, and potential impact.
Kreativität, Methoden, Innovation:	There is a lot of room to explore different fingerprinting/detection approaches. A real-world example game could also be used to show some of the techniques that were evaluated in the paper.
Sonstige Bemerkungen:	

Projektteam

Student:in 1:	Alain Siegrist

Betreuer:in:	Blazytko

Auftraggeber

Firma:	HSLU
Ansprechperson:	Tim Blazytko
Funktion:	
Strasse:	
PLZ/Ort:	
Telefon:	+41 41 757 68 07
E-Mail:	tim@blazytko.to
Website:	

Version 13.06.2023 / bcl

9. List of abbreviations, listings, figures, tables

Glossary

EAC – Easy Anti-Cheat: A commercial anti-cheat product. 6

AIK – Attestation Identity Key: A temporary signing key, generated by a TPM and certified by the Endorsement Key to prove its origin from a specific TPM without revealing the EK directly. 8, 34, 36, 38

Argon2id : A computationally expensive hashing algorithm used for key derivation. 28, 74

CA – Certificate Authority: A trusted entity that issues digital certificates. 40, 44

CRL – Certificate Revocation List: A list of digital certificates that have been revoked by the issuing Certificate Authority before their scheduled expiration date. 41

EK – Endorsement Key: A unique cryptographic key embedded in each Trusted Platform Module (TPM) during manufacturing, used for secure storage and identity attestation. 8, 33, 34, ...

EKCert – Endorsement Key Certificate: A certificate issued by the TPM manufacturer that vouches for the public portion of the Endorsement Key. 8, 33, 40

OCSP – Online Certificate Status Protocol: An Internet protocol used for obtaining the revocation status of an X.509 digital certificate, typically in real-time. . 41

OID – Object Identifier: A numerical identifier used in cryptography to uniquely identify objects, such as certificate policies or extensions. . 44

Privacy CA – Privacy Certificate Authority: In the context of TPM attestation, a trusted third-party service that verifies a TPM's identity by validating its Endorsement Key Certificate and issues an Attestation Identity Key certificate. 8, 33, 34

SHA-1 : A cryptographic hash function that produces a 160-bit (20-byte) hash value. . 33

SHA-256 : A cryptographic hash function that produces a 256-bit (32-byte) hash value. 33, 34

F2P – free-to-play: A business model for video games where players can access the core game without payment, but may have optional purchases. 2

GPT – GUID Partition Table: A standard for the layout of partition tables on a physical hard disk drive. .4

HWID – Hardware ID: A unique identifier derived by combining various hardware components of a user's computer to create a persistent fingerprint. 3, 6, 9, 72

MBR – Master Boot Record: A special type of boot sector at the beginning of a partitioned computer mass storage device. 4

SMBIOS – System Management BIOS: A standard that defines data structures for BIOS and motherboard components. 3, 59, 60, 61

TPM – Trusted Platform Module: A dedicated security chip or firmware module that securely stores cryptographic keys, attests to hardware integrity, and produces unforgeable device identifiers, used for hardware-rooted identity. 1, 6, 7, ...

ARP – Address Resolution Protocol: A network protocol used to resolve IP addresses to Media Access Control (MAC) addresses on a local network segment. 56, 57

CGNAT – Carrier Grade NAT: A network address translation technique where a single public IPv4 address is shared among multiple customers simultaneously, often used by ISPs due to IPv4 address exhaustion. 5, 52

CIDR – Classless Inter-Domain Routing: A method for allocating IP addresses and routing IP packets, used to define IP address ranges. 52, 53

ISP – Internet Service Provider: A company that provides internet access to individuals and organizations, responsible for assigning IP addresses to customers. . 5

MAC – Media Access Control: A unique identifier assigned to a network interface controller (NIC) for communications within a network segment. 3, 6, 56, 57

NAT – Network Address Translation: A method of remapping one IP address space into another, used by home networks to share a single public IP address among multiple devices. 5

VLAN – Virtual LAN: A logical network segment that can span multiple physical network devices, allowing for the segmentation of network traffic and isolation of devices. 56

VPN – Virtual Private Network: A technology that creates a secure, encrypted connection over a less secure network, by passing network traffic over a different device. 6, 9, 52, ...

IOCTL – I/O Control: A system call used by Windows to directly communicate with device drivers for performing input/output operations. 4, 66

IRP – I/O Request Packet: A data structure used by the Windows kernel to represent an I/O request, processed by driver dispatch routines. 62, 63

UEFI – Unified Extensible Firmware Interface: A software interface between an operating system and platform firmware, replacing the legacy BIOS. 7

WMI – Windows Management Instrumentation: A COM-based infrastructure that exposes kernel objects as "management classes", allowing user-mode code to query system information. 4, 45, 46, 48

WQL – WMI Query Language: A query language used to retrieve information from Windows Management Instrumentation (WMI) classes. 45

COM – Component Object Model: A Microsoft technology that defines an object-oriented, programming-language-independent system for creating reusable software components. 45

DLL – Dynamic Link Library: A module containing functions and data that can be used by other applications. 12, 21, 74

Meson : An open-source build system designed for efficiency and ease of use. 21

P/Invoke – Platform Invocation Services: A .NET feature that allows managed code (such as C#) to call unmanaged functions implemented in native libraries. 21

PostgreSQL : An open-source relational database management system. 13, 15, 21

Protobuf – Protocol Buffers: A language-neutral, platform-neutral, extensible mechanism for serializing structured data. 21, 26

TSS.CPP : A C++ library developed by Microsoft Research for interacting with Trusted Platform Modules (TPMs). 33, 34

Tracy : A real-time, high-precision, and cross-platform performance profiling tool. 72

UUIDv7 : A version of the Universally Unique Identifier (UUID) standard that includes a timestamp component. .50

WDK – Windows Driver Kit: A software development kit that enables developers to create kernel-mode drivers for Windows. 13

Listings

Figures

Tables

10. Bibliography

Asturias, D. (2023, August). *CGNAT: The Workaround to IPv4 Depletion*. https://www.rapidseedbox.com/blog/cgnat

Beigi, M. (2024, October). *How we Outsmarted CSGO Cheaters with IdentityLogger*. https://mobeigi.com/blog/gaming/how-we-outsmarted-csgo-cheaters-with-identitylogger/

Chief-Engineer. (2024, November). *Departure*. https://chief-engineer.github.io//2024/11/10/departure.html

Collins, S., Poulopoulos, A., Muench, M., & Chothia, T. (2025, February). Anti-Cheat: Attacks and the Effectiveness of Client-Side Defences. *CheckMATE '24*. https://doi.org/10.1145/3689934.3690816

Dorner, C., & Klausner, L. D. (2024). If It Looks Like a Rootkit and Deceives Like a Rootkit: A Critical Examination of Kernel-Level Anti-Cheat Systems. *Proceedings of the 19th International Conference on Availability, Reliability and Security*, 1–11. https://doi.org/10.1145/3664476.3670433

Fabri, G. (2025, April). *What Happens if You Evade a Ban on Roblox?*. https://www.exitlag.com/blog/how-serious-is-ban-evasion-in-roblox/

Herbert, T. (2018). *Privacy in IPv6 Network Prefix Assignment* (Issue draft-herbert-ipv6-prefix-address-privacy-00). https://datatracker.ietf.org/doc/draft-herbert-ipv6-prefix-address-privacy-00

Hogan, A. (2024, May). *Don't get spoofed - how cheats are avoiding hardware bans*. https://www.intorqa.gg/post/don-t-be-spoofed-how-cheats-are-avoiding-hardware-bans

IPQualityScore. *Proxy Detection API*. Retrieved March 6, 2025, from https://www.ipqualityscore.com/proxy-vpn-tor-detection-service

Jeong, D., & Lee, S. (2019). Forensic signature for tracking storage devices: Analysis of UEFI firmware image, disk signature and windows artifacts. *Digital Investigation*, *29*, 21–27. https://doi.org/10.1016/j.diin.2019.02.004

Kalayci, B. (2007, October). *P3 Serial Un-Support Page*. https://www.buraks.com/p3uns/

Klotz, A. (2021, September). *Valorant Enforcing TPM on Windows 11 PCs to Keep Cheaters at Bay*. https://www.tomshardware.com/news/valorant-requires-tpm-on-windows-11

Laserface. (2024, May). *Riot Vanguard FAQ*. https://support-leagueoflegends.riotgames.com/hc/en-us/articles/24169857932435-Riot-Vanguard-FAQ-League-of-Legends

Lehtonen, S. (2020). *Comparative Study of Anti-cheat Methods in Video Games*.

linuxdev. (2023, July). *Writing over the display EDID*. https://forums.developer.nvidia.com/t/writing-over-the-display-edid/259202

Liverus. (2025, May). *Metamorph*. https://github.com/Liverus/Metamorph

Mahesh. (2025, January). *NAT in Networking | Importance and How It Works*. https://nitizsharma.com/nat-explained-in-networking/

meekochii. (2024, July). *Understanding Kernel-Level Anticheats in Online Games*. https://research.meekolab.com/understanding-kernel-level-anticheats-in-online-games

Microsoft. *Windows 11 Specs and System Requirements*. Retrieved March 10, 2025, from https://www.microsoft.com/en-us/windows/windows-11-specifications

Microsoft. (2022, April). *Win32_Processor class*. https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-processor

Microsoft. (2024a, February). *IOCTL_STORAGE_QUERY_PROPERTY*. https://learn.microsoft.com/en-us/windows/win32/api/winioctl/ni-winioctl-ioctl_storage_query_property

Microsoft. (2024b, September). *Using an INF File to Override EDIDs*. https://learn.microsoft.com/en-us/windows-hardware/drivers/display/overriding-monitor-edids

Niverthi, M., Verma, G., & Kumar, S. (2022). Characterizing, Detecting, and Predicting Online Ban Evasion. *Proceedings of the ACM Web Conference 2022*, 2614–2623. https://doi.org/10.1145/3485447.3512133

otiosum. (2023, May). *Patching AMI Aptio V firmware to circumvent Secure Boot checks*. https://www.unknowncheats.me/forum/anti-cheat-bypass/585526-patching-ami-aptio-firmware-circumvent-secure-boot-checks.html

PJB3005. (2024, September). *Auth-managed HWIDs*. https://github.com/space-wizards/SS14.Web/pull/26

Rendenbach, C. A. (2022). *Anti-Cheating Measures in Video Games*.

RIPE. (2020). *IPv6 Address Allocation and Assignment Policy*. https://www.ripe.net/publications/docs/ripe-738/

Rodrigues, C. (2023, December). *Using DNS to estimate the worldwide state of IPv6 adoption*. https://blog.cloudflare.com/ipv6-from-dns-pov/

SecHex. (2025, June). *SecHex-Spoofy*. https://github.com/SecHex/SecHex-Spoofy

Shmurkio. (2025, January). *FakeSecureBoot*. https://github.com/Shmurkio/FakeSecureBoot

Skyfail. (2018, June). *Kernelmode SMBIOS Hardware ID Spoofing*. https://www.unknowncheats.me/forum/anti-cheat-bypass/287926-kernelmode-smbios-hardware-id-spoofing.html

Space Station 14. *About Space Station 14*. Retrieved June 4, 2025, from https://spacestation14.com/about/about/

Space Station Multiverse. *Hub Info*. Retrieved June 4, 2025, from https://spacestationmultiverse.com/hub-info/

Space Wizards Federation. (2024, August). *NetManager.ClientConnect.cs*. https://github.com/space-wizards/RobustToolbox/blob/31292fe4b8a7a2cdcb04ae58882833b9deffbc9b/Robust.Shared/Network/NetManager.ClientConnect.cs#L141

Syverson, P. (1994). A taxonomy of replay attacks [cryptographic protocols]. *Proceedings The Computer Security Foundations Workshop VII*, 187–191. https://doi.org/10.1109/CSFW.1994.315935

Trusted Computing Group. (2022, January). *TCG EK Credential Profile for TPM Family 2.0*. https://trustedcomputinggroup.org/resource/http-trustedcomputinggroup-org-wp-content-uploads-tcg-ek-credential-profile-v-2-5-r2_published-pdf/

Trusted Computing Group. (2025). *TPM 2.0 Keys for Device Identity and Attestation*. https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-Keys-for-Device-Identity-and-Attestation-v1.10r9_pub.pdf

Tulach, S. (2024, January). *Mutante*. https://github.com/SamuelTulach/mutante

UnknownCheats. (2020, November). *Should i change my MB and other parts if im HWID banned ?*. https://www.unknowncheats.me/forum/rainbow-six-siege/427672-change-mb-im-hwid-banned.html

UnknownCheats. (2021, April). *Apex/EAC/Origin Registry Trace Files Cleaner*. https://www.unknowncheats.me/forum/apex-legends/448248-apex-eac-origin-registry-trace-files-cleaner.html

UnknownCheats. (2024, July). *HWID Ban - Which parts to replace?*. https://www.unknowncheats.me/forum/escape-from-tarkov/648065-hwid-ban-replace.html

Venkateswaran, R. (2001). Virtual private networks. *IEEE Potentials*, *20*(1), 11–15. https://doi.org/10.1109/45.913204

Video Electronics Standards Association. (2000, February). *VESA Enhanced Extended Display Identification Data Standard*. https://glenwing.github.io/docs/VESA-EEDID-A1.pdf

White, R. (2023, April). *Privacy And Networking Part 8: IPv6 Addresses And Privacy*. https://packetpushers.net/blog/privacy-and-networking-part-8-ipv6-addresses-and-privacy/

xtremegamer1. (2025, March). *xigmapper*. https://github.com/xtremegamer1/xigmapper