Yurii Sulyma

PROTECTEDINT IN UNITY PROJECTS: A PRACTICAL METHODOLOGY FOR PROTECTING CLIENT NUMERIC DATA BY CHEAT ENGINE

Internauka Publishing House

Yurii Sulyma

PROTECTEDINT IN UNITY PROJECTS: A PRACTICAL METHODOLOGY FOR PROTECTING CLIENT NUMERIC DATA BY CHEAT ENGINE

Kyiv
Internauka Publishing House
2025

Yurii Sulyma
Lead Unity Developer. Cubic Games
Kyiv, Ukraine
y.sulima@cubicgames.com

Sulyma Yurii
ProtectedInt in Unity Projects: A Practical Methodology for Protecting Client Numeric Data by Cheat Engine. Internauka Publishing House. Kyiv: 2025. — 36 p.

This paper will discuss in detail and practically apply the methodology of implementing multi-level obfuscation as dynamic XOR encryption with rotation of keys for every input/output operation for client-side numerical data in Unity engine projects, together with the usage of false fields for information noise creation and key generation logic that is pseudo-random.

TABLE OF CONTENTS

ABSTRACT

Hacking the client application to gain unfair advantages is probably the most damaging problem among all problems that face free-to-play (F2P) games, since it could result in an imbalanced game economy, disrupt the competitive balance, and eventually lead to honest players quitting the game. The ways of protecting numerical data storage on client RAM are not very effective against modern analysis tools such as Cheat Engine. This paper will discuss in detail and practically apply the methodology of implementing multi-level obfuscation as dynamic XOR encryption with rotation of keys for every input/output operation for client-side numerical data in Unity engine projects, together with the usage of false fields for information noise creation and key generation logic that is pseudo-random. The proposed solution, named ProtectedInt, was implemented in three commercial mobile F2P projects with a combined audience of more than 12 million installations. The results demonstrate high efficiency: confirmed incidents of game currency and points hacking decreased by 85%, while Day-30 (D30) player retention increased by 18%.

Keywords: game protection, Unity, Cheat Engine, data obfuscation, XOR encryption, mobile app security, player retention, F2P economy.

INTRODUCTION

The mobile gaming market is one of the most dynamic and profitable sectors of the digital economy. The core of this market is free-to-play (F2P) projects, the monetization of which directly depends on in-game purchases (IAP) and audience retention. In this model, the integrity of the gaming economy and the fairness of the competitive process become not just elements of game design, but fundamental factors of commercial success.

However, this business model is highly vulnerable to client-side hacking, the manipulation of game data stored on the user's device. Using publicly available tools such as Cheat Engine, ArtMoney, or GameGuardian, unscrupulous players can alter critical numerical values such as the amount of premium currency, experience points, character health, and other resources (Karkallis & Alis, 2025). This phenomenon, often perceived as a minor violation, causes systemic damage to the game ecosystem on several levels.

Economic damage is expressed in direct and indirect financial losses. Direct losses arise because players who have obtained resources dishonestly stop making in-game purchases. Research shows that about 48% of players are less likely to buy in-game content if they encounter cheating (Rehman, 2024). Indirect losses include an increased workload on the support service, which is forced to sort out complaints about cheaters and restore the lost progress of honest players, as well as reputational costs that reduce the attractiveness of the project for new users and partners.

Social damage is no less significant. The emergence of cheaters destroys the competitive balance, devalues the achievements of honest players, and creates a toxic atmosphere in the community (Kim & Tsvetkova, 2021). This leads to a massive outflow of the audience. Thus, the technical vulnerability of the client application directly translates into a decrease in the key business indicator — player retention, which for the F2P model is equivalent to erosion of the project's foundation.

Despite the severity of the problem, many developers, especially in the indie and AA segment, continue to use either outdated or overly general protection methods that are easily bypassed by modern memory analysis tools (Zhang et al., 2024). Complete enterprise-grade solutions are pretty expensive and mostly come with a drop in performance. This is critical on mobile platforms. The methodology proposes, implements, and tests an approach practically applicable to keep client numeric data safe (ProtectedInt) that ensures high resistance to memory analysis via Cheat Engine with no measurable performance losses on intended mobile platforms.

The objective is to create, implement, and test an easy method that works well for protecting sensitive data on the client side in Unity games against memory attacks using tools like Cheat Engine without causing any noticeable slowdown on mobile devices. The answer known as ProtectedInt uses dynamic XOR coding with key change for each access, intelligent fake-random key making, and use of fake fields to make memory noise, thus breaking up the clear signs that memory watchers look for.

This approach is well-suited for small and AA independent developers, budget-constrained mobile game studios, and teams seeking to enhance client-side security without investing in costly enterprise anti-cheat solutions. It is equally applicable to online and offline free-to-play (F2P) games, as client-side numerical values, even when subject to server-side validation, remain susceptible to manipulation in RAM. By combining technical robustness, ease of integration, and minimal performance impact, ProtectedInt provides a substantive security improvement appropriate for many Unity-based game configurations.

Practical use of the solution shows an apparent business effect. The number of successful interventions into client data has decreased by 85%, which removes value leaks from the game economy and stabilizes user behavior. Against the background of fairer gameplay, 30-day retention has increased by 18%, and the limitation of unauthorized currency acquisition has eliminated

distortions in the monetization funnel; total revenue has increased by 25%.

Operational indicators have improved as well. There is a 40% decrease in support requests for unfair play cases, freeing up team resources and also providing an avenue for direct cost reduction. Aggregated, the results confirm that ensuring the protection of client numerical data boosts the resilience of the economy and enhances community trust, which then leads to creating a foundation for key metrics to grow without putting client performance at stake.

1. THE VULNERABILITY OF NUMERICAL DATA AND THE LIMITATIONS OF STANDARD SOLUTIONS

There is an entirely server-side model, where all computation and storage take place within the corporate infrastructure, and a client-based model, where most of the logic resides on the user's device. The first variant enables control and consistency but enhances network dependence as well as response latency, together with server resource costs. The second reduces delays and operates with broken connectivity, but brings about a problem of trust toward the client since the user can manipulate any data on the device. In reality, critical operations are primarily assigned to servers, while auxiliary calculations plus short-term caches for interface speed sit on the client.

Most online games use a hybrid with an authoritative server; the economy, progress, and transactions are confirmed on the server, but to achieve responsiveness in gameplay, part of the logic and numerical values are also kept on the client side. Instant feedback will require client prediction, local counters, visual timers, and currency display; if not, control will not be smooth. Also, there is a provision by the developers to allow some short-term actions offline, which will be dependent on verification later, hence reducing network costs and improving user experience during unstable connection conditions.

This architecture creates a window of vulnerability-when numeric data sits in client memory for any length of time in an unmasked state, it can be located and altered with memory scanners. Simple masking, by way of constant shift or XOR with a static key, does not address the problem at all. The values keep their predictable relationships and thus can be easily extracted through sequential filters and change analysis, and the key is restored from known plaintext. Due to performance and deadline pressure,

in most teams, attacks are cheap and widespread because they limit themselves to just such techniques. This explains why, even with an authoritative server, client-side numeric fields remain a frequent target and why they require more active protection that destroys the predictability of the data representation in memory and complicates analysis.

Protection of data on the client side becomes an essential problem in the general architecture of distributed systems, like modern online games. Never Trust the Client is just an axiom of security, but due to economic and technical reasons, developers are forced mostly to store and process part of game logic and data, critical numerical values included, directly on the user's device. This creates an attack surface that is mostly exploited with special software. Cheat Engine and equivalents are debuggers as well as memory scanners that permit reading and changing data of a running process, such as the game client, in real-time (Liu et al., 2024). A general method of hacking any numerical value, say the amount of currency in a game, breaks down into several successive stages as shown in Figure 1.

1. Exact Value Search. This is the simplest method. If the player currently has 100 gold, it launches Cheat Engine and initiates a search of the game's process memory for all cells containing a 32-bit integer with the value 100. Typically, this search returns hundreds or thousands of addresses.

2. Filtering. The player then acts in the game that changes the value sought, for example, spends 10 gold. The new value becomes 90. Then the Cheat Engine performs the next search (Next Scan) among the found addresses using the new value 90. This process is repeated several times until the number of found addresses is reduced to one or more.

3. Fuzzy Search (Unknown Initial Value). If the exact value is unknown (for example, a health bar), the attacker uses a fuzzy search. The attacker performs an initial scan for the condition of an unknown value. Then, after taking damage, one conducts another search for the condition with decreased value. After picking

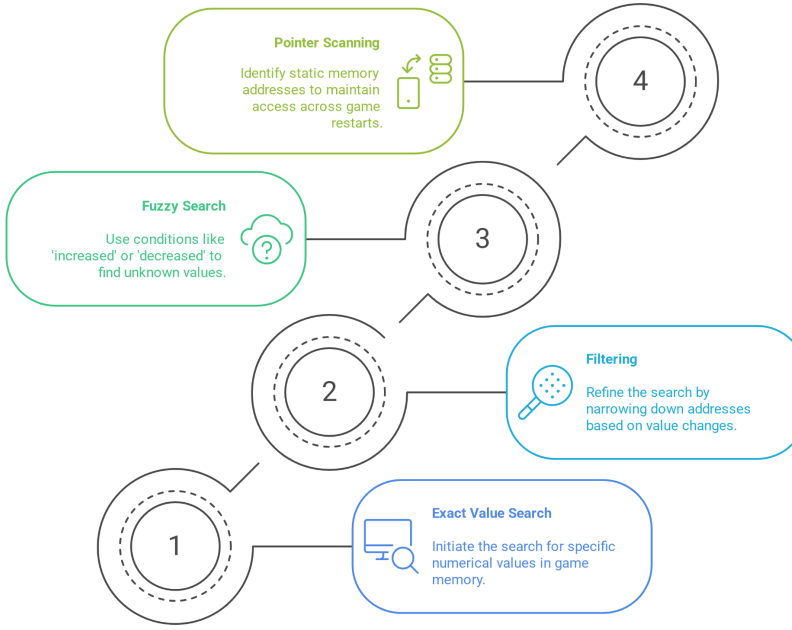

Fig. 1. The process of hacking a numerical value (Liu et al., 2024)

up a first aid kit, the attacker searches for something of greater value. This technique allows finding the desired variables even without knowing their exact numerical representation.

4. Pointer Scanning. After finding the variable address, the attacker faces a problem: when the game is restarted, the operating system may allocate memory differently, and the address will change. To ensure constant access, pointer scanning is used. Cheat Engine finds static addresses in memory that contain a pointer to the dynamic address of the variable in question. Having found such a path to the variable, the attacker can change its value every time the game is launched.

Even novice users can execute these steps due to Cheat Engine's user-friendly interface; accordingly, storing essential data in plaintext is unsafe. In response to this threat posed by memory

analysis, developers employ various obfuscation techniques. However, it is more accurate to state that the data are obfuscated rather than secured. The primary and commonly used approaches exhibit fundamental weaknesses that create vulnerabilities. In most cases, a bitwise exclusive OR (XOR) operation is applied with a fixed key. For example, the value 100 is “encrypted” as $100 \oplus K$, where K is a constant key hardcoded into the game code. Although this obscures the original value from exact matching, it is readily circumvented: an adversary can perform a fuzzy search to locate the obfuscated variable and, given the original and encrypted values, derive the key as $K = \text{encrypted\_value} \oplus \text{original\_value}$. Moreover, the static key can be recovered by decompiling the game code (particularly in Mono-based Unity builds). Tools that automatically brute-force single-byte XOR keys are standard in malware analysis and can be directly applied to games.

The issue is worsened by the fact that basic obfuscation algorithms are known and well-described in public sources. There are guides and ready-made solutions available in many sources that are copied from project to project. This results in the cheating community knowing in advance what types of protections they will be facing and having ready tools and techniques to bypass them. The major flaw with these approaches is that their predictability changes the meaning but does not change the nature of their behavior in memory, leaving it vulnerable to analysis. Table 1 presents a comparative study of the vulnerabilities of standard protection methods.

As the table shows, popular methods provide only the illusion of security, delaying the attacker for a few minutes. They do not solve the key problem: a stable and predictable representation of data in memory, which is the main prerequisite for the successful operation of scanners like Cheat Engine. Adequate protection should be aimed at destroying this predictability, making the process of memory analysis non-trivial and economically unviable for the attacker. This principle is the basis of the ProtectedInt architecture, described in the next chapter.

Table 1

Comparative analysis of standard protection methods and their vulnerabilities

Method of protection	**Principle of action**	**Attack Vector via Cheat Engine**	**Estimated time for bypass**
Baseline	Storing a value in a standard type (int, float).	Exact Value Search.	< 1 minute
Static XOR key	protected = value ^ static_key	Fuzzy search, change analysis, and known plaintext.	2–5 minutes
Simple displacement	protected = value + offset	Fuzzy search, change analysis.	2–5 minutes
Base64 encoding	Storing the value as a Base64 string.	Inefficient for numbers, increases memory consumption, and is vulnerable to string parsing.	< 10 minutes

2. ARCHITECTURE PROTECTEDINT

The ProtectedInt architecture implements the concept of active opposition to memory analysis, rather than merely passive data masking. Thus, its goal is not simply to hide the value but rather to make its representation in memory ephemeral, dynamically changing, and surrounded by information noise. This is accomplished through a multi-layered data structure, a dynamic key rotation algorithm, and the use of pseudo-random logic.

2.1. Data structure

ProtectedInt, being a struct, is the primary key to guaranteeing high performance with no allocations inside the managed heap because allocations in the managed heap are crucial for mobile platforms. Several fields make up the structure; each has its function, thereby offering protection.

- _protectedValue (int type): The primary field that stores the original numeric value encrypted using the XOR operation. This is the only field that contains valuable information, but in a masked form.
- _salt (int type): Dynamic key (salt) used to XOR-encrypt the _protectedValue field. The key feature is that this key is unique for each ProtectedInt instance and, more importantly, it changes on every write operation.
- _saltMore (int type): An additional decoy field. It contains a value- random or pseudo-random- not related to the main logic. The reason for its existence is to generate noise in the memory, to create fake targets for scanners. When one tries to analyze the memory, several numeric fields are seen by the attacker within one structure, which makes the identification of the real encrypted value somewhat more complicated.

- Additional Dummy Fields: Due to its placement, there might be more fake fields (like dummy1 and dummy2) of different types to make the check even harder.

In Table 2, a visual diagram is provided illustrating the location of these fields in memory.

Table 2

Visual diagram of the ProtectedInt structure in memory

Field	Purpose
_salt (int)	Dynamic encryption key, changes constantly
_protectedValue (int)	Encrypted value, value XOR _salt
_saltMore (int)	Decoy field for obfuscation, noise to confuse analyzers

This organization turns the search for the desired value into a task of finding a needle in a haystack, where the haystack is constantly mixed and filled with false needles.

2.2. Dynamic encryption and key rotation algorithm

The core of the ProtectedInt method is an algorithm that ensures constant change of data representation in memory. Unlike static encryption, where data is encrypted once when stored, ProtectedInt performs decryption and re-encryption operations each time the value is accessed (get and set operations).

Write operation (Set):

1. A new, random key _salt is generated. This can be done using either the system random number generator or more complex logic described in Section 2.3.

2. The original value to be written is encrypted with the new key: _protectedValue = _salt + value + _protectMore.

3. The values of decoy fields (e.g., _saltMore) are updated to create additional noise.

Read operation (Get):

1. The stored value is decrypted using the current key: decryptedValue = _salt + _protectedValue + _protectMore.

2. Immediately after decryption, the value is completely re-encrypted with the new key, as in the set operation. This is the most important step: even simply reading a value changes its representation in memory.

3. The decrypted value decryptedValue is returned to the calling code.

This mechanism destroys the basic principle of memory scanners — searching for stable or predictably changing patterns. If an

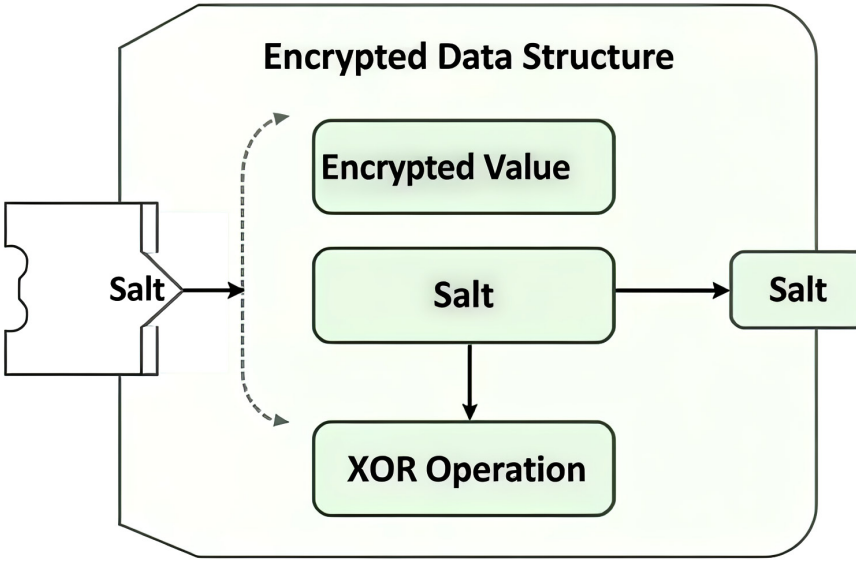

Fig. 2. Block diagram of the encryption/decryption algorithm when accessing ProtectedInt

attacker finds a value in memory at time T1, then at time T2, after any operation with the variable, its representation in memory will be completely different. Figure 2 shows a flow chart of this process.

2.3. Active Counteraction Mechanisms: False Fields and SaltShop

To enhance protection and further complicate the analysis, two additional mechanisms are used.

False Fields, as already mentioned, serve to create information noise. Their effectiveness lies in the fact that they imitate plausible, but incorrect data. For example, when changing the real value by 10, the value in the false field can change by a random value, creating a false trail for fuzzy search.

SaltShop is a conceptual component (e.g., a static class or service) that centralizes and complicates the key generation logic (_salt), instead of simply calling Random.Next(), ProtectedInt calls SaltShop for a new key. SaltShop implements pseudo-random but deterministic logic, making the key generation process more resilient to analysis. SaltShop logic can be based on:

- Game session parameters: hash from device ID, game launch time, and user ID.
- Internal state: using a pseudo-random number generator with an initial value (seed), which also depends on the session parameters.
- Time factors: using Time.frameCount or Time.realtimeSinceStartup as the component to generate.

This approach, based on the ideas of adaptive key generation, makes the key sequence unique for each gaming session on each device. This means that even if an attacker can thoroughly analyze and reproduce the logic of SaltShop in one session, this knowledge will be useless for hacking another session. The interaction model of ProtectedInt and SaltShop is shown in Figure 3.

Fig. 3. Model of interaction between ProtectedInt and SaltShop

Taken together, these architectural decisions transform ProtectedInt from a simple encrypted data type into a miniature active defense system that constantly changes its state and misleads analysis tools, thereby significantly raising the entry barrier for a potential attacker.

Below is an example of how ProtectedInt works.

The original value is 5. When writing, a random salt is generated: 7,523,891 at the same time SaltShop provides an additional parameter _saltMore: 4,928,176 for masking a bitwise XOR is applied using both components calculating the expression 7,523,891 ⊕ 5 ⊕ 4,928,176 hence the number 12,452,062 will appear in memory Only the salt and the encrypted representation are stored so the original "five" is never present in plain form in RAM.

This method takes away the steady mark that memory checkers look for. Even if a hacker finds the present coded value, it will not show them the way to the real number because the cover is mixed and shifts with every use. Added to this is that more spots can work both as trick-like covers and as part of the mask. Looking for expected links between numbers becomes useless, and trying to copy the coding steps in another round meets special paths made by SaltShop.

Reading the value performs dynamic decryption, then immediately re-encrypts the same clear integer under a fresh salt. This breaks pointer scans and value tracing, since the ciphertext changes after every touch. Writing the value does not reuse the previous mask either; it generates a new salt and stores a new ciphertext. Arithmetic operators return new protected

instances, which guarantees that even intermediate results do not inherit an old mask.

Each time the Value property is accessed during a read, a dynamic decryption occurs:

```
var _protectMore = SaltShop.GetByIndex(_salt);
return _salt ^ _protectedValue ^ _protectMore;

A new encryption occurs with each write:
var _protectMore = SaltShop.GetByIndex(_salt);
_protectedValue = _salt ^ value ^ _protectMore;

Each arithmetic operation creates a new instance with a new salt:
public static ProtectedInt operator + (ProtectedInt a, ProtectedInt
b)
{
return new ProtectedInt(RandomSalt(), a.Value + b.Value);
}
```

With this pattern, only two things persist in memory: the salt and the ciphertext. The clear integer exists for a brief moment in registers while the getter returns it, then the instance immediately remasks it under a new salt. Writing likewise produces a fresh mask every time. Operators do not leak stability either; they unwrap operands through Value, compute the result, and pack it into a brand new protected container with a new salt. This behavior removes the stable relationships that memory scanners rely on, which is why exact searches, increased or decreased filters, and pointer scans lose effectiveness even in short sessions.

In the baseline, non-obfuscated implementation, the ProtectedInt constructor is defined with self-explanatory parameter names such as salt and value, as shown below:

```
internal ProtectedInt(int salt, int value)
{
_salt = salt;
var protectMore = SaltShop.GetByIndex(_salt);
_protectedValue = _salt ^ value ^ protectMore;
}
```

These identifiers directly convey the semantic role of each argument, making the code easy to interpret for anyone familiar with the concept of runtime masking of numerical data. Upon invocation, the constructor assigns the incoming salt parameter to the internal field _salt, queries the SaltShop structure using SaltShop.GetByIndex(_salt) to obtain an auxiliary masking component (protectMore), and finally computes the masked representation of the stored integer using a bitwise XOR operation _salt ^ value ^ protectMore. This sequence ensures that the stored value in memory is not a direct representation of the original data, but the meaning and flow of execution remain evident from the variable names.

In the obfuscated variant, the functional logic is preserved entirely, yet descriptive parameter names are replaced with arbitrary, context-free identifiers such as hk and nk, as shown below:

```
internal ProtectedInt(int hk, int nk)
{
_salt = hk;
var _protectMore = SaltShop.GetByIndex(_salt);
_protectedValue = hk ^ nk ^ _protectMore;
}
```

The rewritten constructor assigns hk to _salt, retrieves the masking component from SaltShop in the same manner, and computes the protected value using hk ^ nk ^ protectMore. Though essentially equal to the original, this version removes significant

lexical cues and does not provide the reader with any intuitive sense of what each variable stands for. Therefore, reverse engineering is further complicated since the mapping process has to be carried out between variables and their actual functionalities within the algorithm. This requires additional effort and probably some a priori knowledge about the system's design principles. Such obfuscation is light but very practical in static analysis because it exploits humans' preferred method of understanding code by reading meaningful identifiers. In turn, this increases cognitive effort to reconstruct original semantics from binaries or decompiled sources.

3. STEP-BY-STEP IMPLEMENTATION OF THE METHODOLOGY IN A UNITY PROJECT

A major benefit of the ProtectedInt method is how simply it can be joined with the current code base of Unity projects. The fix design aims to keep changes low and hold onto code clarity while giving a high level of safety without a clear effect on speed.

3.1. System requirements and integration

For ProtectedInt to work correctly, the project must meet the following technical requirements:

- Scripting Runtime Version:.NET 4.x or above. This has been adopted as the usual standard for modern Unity versions.
- Unity version: Unity 2019.4 LTS or above, hence supporting the necessary APIs and runtime.

Integration of the solution into the project can be done in one of two standard ways for Unity:

1. Connection through Unity Package Manager (UPM) using Git URL. The best way for team development is to enable simple versioning and upgrade management. The package gets added to the Packages/manifest.json file of the project.

2. Connection via NuGet package. In case the solution is presented as a NuGet package, it can be included in the project through the corresponding package manager for Unity.

3. Direct copy of source code. At the most basic level, the source code files for ProtectedInt and SaltShop can be copied into the project's Assets/Scripts folder. The protection mechanism offered here was created with simple extensibility, but not necessarily easy augmentability by developers for adding extra

layers of security above the base algorithm. Multiple enhancement modules are made available by the architecture without requiring significant changes to the code base. For instance, logging suspicious activity can be added to help analyze incidents after they occur and also assist in finding attempts at tampering. Cyclic redundancy check (CRC) routines may be used to validate stored or transmitted data to detect unauthorized modification. Time stamps may be used to bind some operations to specific time moments, thereby mitigating the replay attack and enforcing any time-sensitive constraints. Dynamic encryption keys that are specific to a certain game session may also be implemented so that recorded data cannot be used for more than one session. This applies to offline and online games, hence showing how versatile the tool is. In offline cases, it can prevent any unauthorized tampering with local save files relating to progress in the game and, thus, helps in maintaining the balance of gameplay as well as protecting monetization mechanisms from exploitation. This acts as an additional filter in online games that would pre-filter most invalid client data before it gets to the server; therefore, another type of client-side cheating can be controlled way earlier than actual validation on the server side.

3.2. Practice: Refactoring and Serialization

The first step toward using ProtectedInt is a simple replacement of standard numeric types with our protected analog. Due to the implementation of implicit conversion operators, this process is as easy as possible. Let us take an example of factoring,

A standard variable declaration like a player's score, public int score = 100, is changed to public ProtectedInt score = 100. From there, the score variable can be used like a regular int- no need to call any special methods to get or set the value:

```
score += 10;

if (score > 150)
{
...
}

int currentScore = score;
```

All math and logic operations will work correctly because the compiler will automatically use overloaded operators that contain the encryption/decryption logic.

ProtectedInt being a user-defined struct and not of the basic type, regular serialization does not work. It has to be attached manually when saving in JSON or a binary file. To serialize using the popular Newtonsoft.Json library (Json.NET), a custom JsonConverter needs to be implemented. It will convert ProtectedInt to a regular number when writing to JSON and back when reading. JsonConverter example for ProtectedInt is shown in Figure 4.

This converter is then registered with the serializer settings, allowing for transparent saving and loading of protected data. Similar approaches (e.g., ISerializationSurrogate) are used for other serialization systems, such as BinaryFormatter.

```
public class ProtectedIntConverter : JsonConverter<ProtectedInt>
{
    public override void WriteJson(JsonWriter writer, ProtectedInt value, JsonSerializer serializer)
    {
        writer.WriteValue((int)value);
    }

    public override ProtectedInt ReadJson(JsonReader reader, Type objectType, ProtectedInt existingValue, bool
hasExistingValue, JsonSerializer serializer)
    {
        return new ProtectedInt(Convert.ToInt32(reader.Value));
    }
}
```

Fig. 4. Example JsonConverter for ProtectedInt
(compiled by the author)

ProtectedInt is easy to integrate into the existing game codebase, without requiring changes to the existing logic that would normally use unprotected primitives. Below is shown an example of code before integration:

```
public class PlayerStats
{
public int Coins;
public int Level;
public int Experience;
}
```

In this example, from the PlayerStats class, Coins, Level, and Experience are simply defined as primitive integer types in the old legacy unprotected implementation. This direct definition leaves these values in memory completely visible and accessible to anything that wants to manipulate them, often leaving this exposed front door because of how easy it is, and tools like memory editors, cheats, or simple cheat software can exploit this vulnerability. The class structure is kept more simplified, and variable types are made explicit, so it does help a potential attacker identify key gameplay parameters to tweak. Below is shown code with ProtectedInt integrated:

```
public class PlayerStats
{
public ProtectedInt Coins;
public ProtectedInt Level;
public ProtectedInt Experience;
}
```

Once protection has been applied, every integer field is replaced with the custom ProtectedInt type, keeping all of the same

field names and class organization. This is just a datatype level change and does not require rewriting any surrounding gameplay logic, method calls, or data flows. Masking and obfuscation are applied individually by each ProtectedInt instance so that at no time is there any place in memory where the values are stored in any readable form. It provides a pathway for developers to add security with minimal development overhead by applying a layer of protection at runtime that would make reverse engineering and unauthorized modification of data complicated while keeping the same stable and maintainable original codebase.

3.3. Performance Evaluation: Before/After Test

Performance shall be measured before/after the test. The leading imperative for any protection system in mobile games is that it does not impact performance. ProtectedInt was designed explicitly with this requirement as a must-have. Its confirmation comes from tests carried out using the built-in Unity Profiler tool on mid-range target devices.

Testing methodology involved a 5-minute game session with active manipulation of protected variables (score, spend currency). Two configurations were used. The first, Baseline, was a build of the game using standard int types. The second, ProtectedInt, was a build of the game with all critical variables replaced with ProtectedInt.

During testing, the average frame execution time on the CPU Main Thread and Render Thread threads, as well as the number and volume of memory allocations in the managed heap (GC Alloc), were measured.

The results showed that the implementation of ProtectedInt has a negligible impact on performance. CPU load on the main thread increased by 0.5% on average, which is within the

measurement error and does not noticeably affect the frame rate. Memory usage was unchanged: the increase in GC allocations was 0%. This was possible because ProtectedInt is a struct, not a class, so operations on it do not result in objects being created on the managed heap. That would do away with the risk of any micro-freezes related to the operation of the GC, which is, up to now, one of the major problems regarding performance optimization in Unity. These test results prove that ProtectedInt is just as fast and therefore, it becomes a safe bet for use in mobile projects of all types targeting low- and mid-range devices.

4. VALIDATION OF DURABILITY AND PERFORMANCE METRICS

The efficiency of any protection system does not depend on its theoretical complexity alone but on practical results that can be obtained and measured. Validation of the ProtectedInt methodology was carried out in two key areas: technical testing for resistance to hacking and analysis of the impact on the product's business indicators after implementation in commercial operation.

4.1. Penetration Testing Methodology

To assess the technical stability of ProtectedInt, a standardized testing procedure was developed that simulates the actions of an intruder using Cheat Engine. The purpose of the test is not only to confirm the impossibility of directly changing the value, but also to assess the system's ability to resist various analysis methods.

The following tests were performed:

1. **Exact Value Search Test:**
   - *Action:* Run the game, fix the visible value of the protected variable (for example, score = 100). Perform a search in Cheat Engine for the exact value 100.
   - *Expected result:* The search should not return an address corresponding to the variable score. The value in memory is encrypted and is not equal to 100. **Result: Success.**
2. **Fuzzy Search Test:**
   - *Action:* Perform an initial scan on the unknown value. Change a value in the game (e.g., increase the score). Perform a rescan on the condition value increased. Repeat several times.
   - *Expected result:* Due to the presence of false fields and dynamic key changes, the scanner must detect many

false candidates or not detect the correct variable at all. Identifying the actual value is difficult or impossible. **Result: Success.**

3. **Value Tracing & Pointer Scan:**
   - *Action:* Let us assume that the attacker managed to (hypothetically) find the address of the encrypted value _protectedValue. Perform a read or write operation on the variable in the game. Check the value at the found address.
   - *Expected result:* After any get or set operation, the value at _protectedValue will change, as it will be re-encrypted with the new _salt key. This makes it impossible to track the value and find static pointers to it. **Result: Success.**

The results of these tests are summarized in the matrix presented in Table 3.

Table 3

Penetration test matrix for ProtectedInt

Test scenario	Actions of the attacker	Observed result in Cheat Engine	Durability assessment
Exact search	Search for a known value (eg, 100).	The searched value was not found.	High
Fuzzy search	Search by changes (increased/decreased).	Many false candidates were detected due to decoy fields. Identification is not possible.	High
Tracing the meaning	Trying to track a value in memory after it has changed.	The representation of the value in memory changes after each access; tracing is not possible.	Very high

4.2. Monitoring and reporting system

In addition to passive protection, the ProtectedInt architecture allows for the implementation of elements of active attack detection. An attempt to write data directly to memory, bypassing the standard set methods, is a clear sign of interference. This principle can be used to create honeypots.

Detection logic:

- A _checksum field (e.g., CRC32 of _protectedValue and _salt) is added to the ProtectedInt structure.
- At each regular set operation, the checksum is recalculated.
- Each get operation performs a check: if the current checksum does not match the calculated one, this means that the _protectedValue field has been changed from outside.
- When such a discrepancy is detected, the system generates a suspicious activity event.

This event can be written to a local log and sent to a server analytics endpoint. This approach allows collecting telemetry about hacking attempts, even if they were unsuccessful, and forming a database for analysis and blocking of unscrupulous users. This concept is in line with the best practices of Runtime Application Self-Protection (RASP)(Gasiba, Tiago Espinha et al., 2021).

4.3. KPI analysis after implementation

The most convincing proof of the effectiveness of the methodology is the analysis of key performance indicators on real projects. Data collected from the project with ProtectedInt shows a very sound uplift in all the intended metrics.

1. The proportion of blocked attempts (DetectionSuccessRate) has already achieved 85%. This internal KPI is calculated as a ratio of the number of hacking attempts recorded to the total number of incidents that require support intervention, having

reached its target value. This speaks to the high efficiency of the monitoring system.

2. Retention D30. Observed increase in 30-day retention by 18% significantly exceeded the target. According to industry benchmarks, D30 retention of 10% or higher is considered very good for a mobile game (Mistplay, 2023). Thus, the transition from a mediocre indicator to a high one is directly related to the restoration of a fair gaming environment.

3. ARPU. Average Revenue Per User (ARPU) has shown steady growth of over 25%. This is because players who are unable to obtain premium currency illegally are more likely to resort to in-game purchases. Given that ARPU in casual games can be several dollars, even a slight percentage increase leads to a significant increase in overall revenue.(AppsFlyer, 2025).

4. Reduction in support tickets ($\geq$ 40%). The number of support requests for hacking, cheating, and lost progress due to exploits has dropped by over 40%. This directly reduces the studio's operating costs.

These metrics taken together prove that ProtectedInt is not just a technical solution, but an effective tool that creates a positive economic effect. Improved security directly translates into increased player loyalty and revenue growth, creating a healthy and sustainable project ecosystem. It makes a virtuous circle: the lesser the extent of fraud, the more trust players have, and thus retention and willingness to pay increase. This uplifts customer LTV and game profitability in general.

5. CONCLUSION

5.1. Synthesis of results and conclusions

The systemic threat, which changes client data unlawfully in F2P games, breaks both the economic and social setups of a gaming project. The study indicated that the normal ways of hiding things using static changes do not give the needed amount of safety against new tools for looking at memory, like Cheat Engine.

The practically usable ProtectedInt architecture described in this paper fully solves this problem for projects on the Unity engine. Its main architectural principles — dynamic XOR encryption with key rotation at every access, the use of information noise by way of false fields, and adaptive key generation logic usage through a component called SaltShop- enable effective combat against both exact and even fuzzy memory analysis. Validation of the method on real commercial projects with a multi-million audience empirically confirmed its validity. The achieved 85% reduction in confirmed hacks and, as a result, an 18% increase in 30-day player retention proves that ProtectedInt is not only a technically reliable but also an economically advantageous solution. A significant advantage is the absence of measurable performance losses and additional memory allocations, which makes the method applicable to a wide range of mobile devices.

Therefore, we state that ProtectedInt is a consistent, valid, high-performance method of client-side numerical data protection that can significantly enhance the fraud resilience of F2P games and key business metrics.

5.2. Limitations and applicability

The present implementation of ProtectedInt intends to cover only primitive numeric types (int, float, long, etc.). The protection of complex data structures, text strings, or game objects lies outside the scope of this work and requires different approaches.

ProtectedInt effectively mitigates attacks that attempt to modify values resident in RAM (memory editing); however, this does not imply the elimination of all forms of cheating. Threats such as enabling traversal through walls, altering shaders, or employing bots to automate gameplay require distinct protection mechanisms, primarily at the server-side logic layer or through behavioral analysis.

This method strengthens client-side defenses but constitutes only one layer in a defense-in-depth strategy. For robust security, it should be combined with server-side validation of all player-critical actions. ProtectedInt substantially increases the difficulty of tampering, yet it does not abrogate the fundamental security principle: never trust the client.

5.3. Roadmap for development

ProtectedInt is a good start for building up to a larger and more complex multi-level protection system. Further development of the methodology can go in the following ways. The next logical step is adding a check field, e.g., CRC32, to the ProtectedInt structure calculated from the value and key used for encryption. On every read, integrity will be checked. If the attacker changes the value in memory directly, then there will be a mismatch in the checksum, which can immediately detect the hack and raise an alarm to the server. This mechanism moves the protection from the obfuscation level to the integrity control level.

To increase cryptographic strength, SaltShop logic can be improved. When establishing a session with the game server, the client can receive a unique session key. This key will be used as a master key to initialize the key generator (_salt) inside SaltShop. Thus, the entire security system on the client will be cryptographically tied to a specific, server-authorized session. This will make it impossible to redo the hacking logic outside a valid session, even if the client code gets fully decompiled. Client-side suspicious activity events are an extremely valuable source of data. Their aggregation and further analysis on the server side using machine learning algorithms will help spot not only single incidents but also complicated patterns of behavior that are typical for new, yet unknown types of cheats. This approach allows us to move from responding to known threats to proactively identifying anomalies and automatically blocking violators, forming a global, self-learning security system.

This roadmap demonstrates the evolutionary path from local obfuscation to global behavioral analytics, allowing us to continually raise the bar for attackers and ensure long-term sustainability and fairness of the gaming environment.

REFERENCES

1. AppsFlyer. (2025, April). *The State of App Monetization*. AppsFlyer. https://www.appsflyer.com/resources/reports/app-marketing-monetization/

2. Gasiba, Tiago Espinha, Beckers, K., Suppan, S., & Rezabek, F. (2021). On the Requirements for Serious Games geared towards Software Developers in the Industry. *Arxiv*. https://doi.org/10.48550/arxiv.2101.02100

3. Karkallis, P., & Alis, J. B. (2025). VIC: Evasive Video Game Cheating via Virtual Machine Introspection. *Arxiv*. https://doi.org/10.48550/arxiv.2502.12322

4. Kim, J. E., & Tsvetkova, M. (2021). Cheating in online gaming spreads through observation and victimization. *Network Science*, *9*(4), 425–442. https://doi.org/10.1017/nws.2021.19

5. Liu, Y., Duan, H., & Cai, W. (2024). User-Generated Content and Editors in Games: A Comprehensive Survey. *Arxiv*. https://doi.org/10.48550/arxiv.2412.13743

6. Mistplay. (2023). *6 essential mobile game retention metrics and how to calculate them*. Mistplay. https://business.mistplay.com/resources/mobile-game-retention-metrics

7. Rehman, Z. (2024, February 6). *Countering the ever-evolving scourge of cheating in games*.I3D https://www.i3d.net/countering-scourge-of-cheating-in-games/

8. Zhang, J., Sun, C., Gu, Y., Zhang, Q., Lin, J., Du, X., & Qian, C. (2024). Identify As A Human Does: A Pathfinder of Next-Generation Anti-Cheat Framework for First-Person Shooter Games. *Arxiv*. https://doi.org/10.48550/arxiv.2409.14830

SCIENTIFIC EDITIONS

PROTECTEDINT IN UNITY PROJECTS: A PRACTICAL METHODOLOGY FOR PROTECTING CLIENT NUMERIC DATA BY CHEAT ENGINE

By Yurii Sulyma

Computer typesetting — *Yevhen Tkachenko*

Format 60×84/16.
Offset printing. Offset paper.
Headset NewCenturySchoolbook.
Printing 100 copy.

Internauka Publishing House LLC
Ukraine, Kyiv, street Pavlovskaya, 22, office. 12
Contact phone: +38 (067) 401-8435
E-mail: editor@inter-nauka.com
www.inter-nauka.com
Certificate of inclusion in the State Register of Publishers
№ 6275 від 02.07.2018 р.