Reverse Engineering IL2CPP-Based Unity Games

(Inżynieria wsteczna gier Unity opartych na IL2CPP)

Miriam Bouhajeb

Praca inżynierska

Promotor: dr. Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

15 czerwca 2025

Abstract

IL2CPP is a significant improvement in Unity game development as a scripting backend which transpiles Microsoft Intermediate Language (MSIL) into C++, enabling AOT (ahead-of-time) compilation for improved performance and cross-platform compatibility.

This process generates native binary code, which is significantly harder to reverse engineer. Despite this, specific methods can be used to systematically analyze IL2CPP-compiled applications. This technical analysis demonstrates a structured process of IL2CPP reverse engineering through a case study of a game in which multiple layers of obfuscation have been implemented by developers. The analysis reveals how using both known and original techniques allows for extraction of crucial information from the IL2CPP-generated files, providing a feasible approach for examining such applications from a security analysis perspective.

Streszczenie

IL2CPP to znaczne ulepszenie w tworzeniu gier za pomocą silnika Unity. Jako backend skryptowy, transpiluje on kod pośredni MSIL (Microsoft Intermediate Language) na C++, co pozwala na kompilację AOT (ahead-of-time). Skutkuje to wyższą wydajnością i lepszą kompatybilnością międzyplatformową.

Proces ten generuje natywny kod binarny, który jest znacznie trudniejszy do poddania inżynierii wstecznej, niż MSIL. Mimo tego, istnieją różnego rodzaju metody umożliwiające systematyczną analizę aplikacji skompilowanych przy użyciu IL2CPP, korzystające z powszechnie dostępnej wiedzy. W niniejszej analizie technicznej przedstawiono ustrukturyzowany proces inżynierii wstecznej aplikacji skompilowanych za pomocą IL2CPP. Wykorzystano do tego studium gry, w której celowo zaimplementowano wielostopniową obfuskację kodu oraz danych. Analiza ta ujawnia, w jaki sposób stosowanie zarówno znanych, jak i autorskich sposobów pozwala wydobyć kluczowe informacje z plików generowanych przez IL2CPP i modyfikowanych przez deweloperów.

Contents

Chapter 1

Introduction

Unity is an extremely popular game engine used by large game companies and indie developers alike. Its high level of abstraction and user-friendly interface make game development accessible, even for individuals without prior programming experience. The engine supports development across multiple platforms, from mobile devices to computers running major operating systems (Linux, Windows, MacOS).

The widespread popularity of Unity games, however, makes them lucrative targets for exploits and attacks. Cheating has always been part of the gaming scene, but in today's landscape, where games generate substantial revenue, their security is a critical concern. Modern cheats not only disrupt the player experience but can also cause significant financial losses for companies. Consequently, developers employ various countermeasures, yet the effectiveness of these measures is often questionable. This work addresses these security concerns, specifically by analyzing a backend technology within Unity considered more secure than its alternative. It demonstrates how obfuscation techniques can be circumvented, thereby providing insight into the inner workings of such applications and enabling the retrieval of crucial, protected information.

Chapter 2

IL2CPP Internals

2.1 Mono vs IL2CPP in Unity game development

The Unity engine is mainly associated with the C# language, which is often considered relatively easy to decompile since it typically compiles into Common Intermediate Language (CIL). CIL is not fully machine code; rather, it retains many high-level abstractions and metadata elements associated with the original code, such as type information and variable names. Initially, CIL is also less optimized because it is not the final code executed by the machine. Instead, at runtime, CIL is passed through a just-in-time (JIT) compiler, which then generates native code and performs necessary optimizations. This JIT compilation process is key to improving the portability of applications, allowing them to be executed on various platforms. However, this approach can lead to slower application performance compared to languages like C++, which are compiled directly to native code using ahead-of-time (AOT) compilation.

In Unity, developers can choose between two primary backends for their applications[1]. Mono is the traditional and more popular backend, primarily due to its ease of use, and it employs JIT compilation. While convenient for development, this also makes applications using the Mono backend relatively straightforward to reverse engineer, often requiring only an IL decompiler.

The second backend offered by Unity is IL2CPP. IL2CPP, instead, utilizes AOT compilation, which significantly improves application performance and can make analysis significantly harder, as the resulting binary consists of native machine code.

Figure 2.1 presents a diagram illustrating the IL2CPP build process. Initially, the C# code is compiled into standard .NET assemblies (DLLs). These assemblies are then processed by a tool that strips unneeded classes and methods to reduce their size. Subsequently, a Unity-shipped program called IL2CPP.exe transpiles the IL assemblies into C++ code. Finally, this C++ code, along with the IL2CPP runtime library, is compiled into the final product.

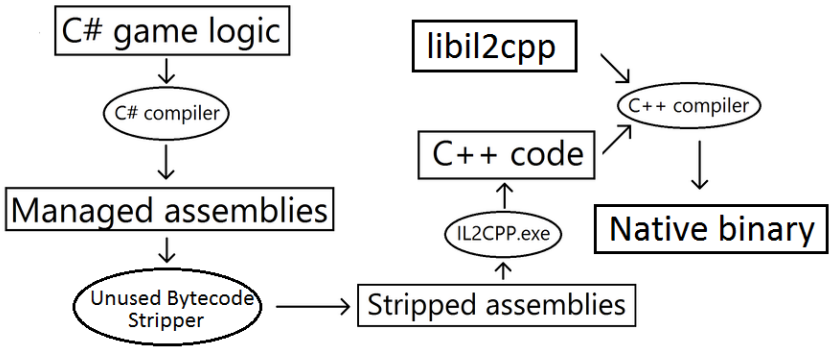


Figure 2.1: A diagram in the steps when building an IL2CPP project. Source: [2]

This process generates several key components. One is the main game binary, usually named GameAssembly.dll, which contains the original application logic compiled into a C++ binary. The main executable of the application, however, is often a small program that loads and runs another generated DLL, typically UnityPlayer.dll [3]. This UnityPlayer.dll is primarily responsible for initializing and managing the game engine, including the previously mentioned GameAssembly.dll.

Another crucial generated file is the metadata file, commonly named `global-metadata.dat`. This file contains a metadata from the original project's .NET assemblies in a serialized format, including information about types, classes, structs, methods, fields, properties, attributes, (and more) and their relationships. It also stores string data, such as symbol identifiers (e.g., class and method names) and string literals used by the application.

This information itself can reveal much about the application, allowing for a near-reconstruction of its original structure, excluding the exact source code. This is why IL2CPP, despite its AOT nature, does not inherently prevent reverse engineering, as this metadata is essential for the application's correct functioning due to .NET's reliance on reflection (obtaining information about loaded assemblies and the types defined within them) and attributes [4]. Because of that, developers try to protect their applications and often use various techniques to obfuscate this metadata file, effectively making the recovery of this information harder. This work will focus on circumventing such techniques by deobfuscating the metadata file, thereby demonstrating that these measures are not always enough.

2.2 Process toolchain

To effectively attempt reverse engineering analysis on an IL2CPP application, it's fundamental to first understand how its generation process works. While both Mono and IL2CPP backends in Unity use C# as the primary language, IL2CPP applications ultimately consist of native machine code. This is because the C# code is transpiled into C++ (as the "IL2CPP" name suggests); however, several steps occur before this C++ code is compiled into the final executable.

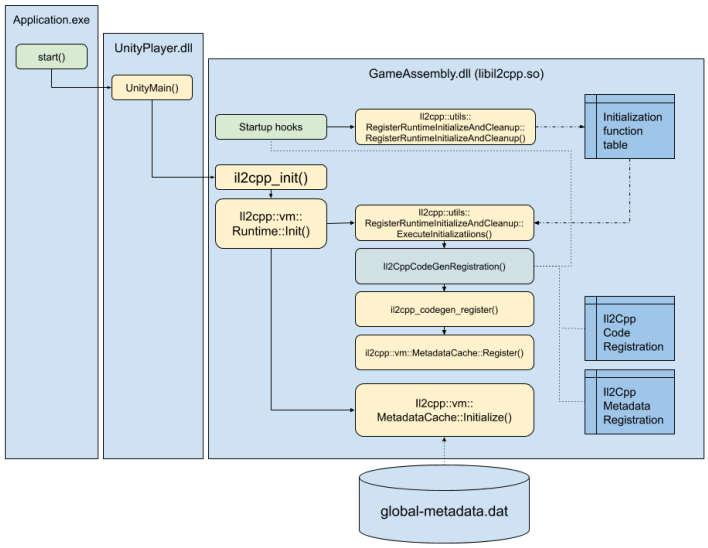


Figure 2.2: The execution process of an IL2CPP application visualized. Source: [3]

The execution chain can be seen in Figure 2.2. It shows that `UnityPlayer.dll` calls `GameAssembly.dll`. Then, through `IL2CPP::vm::Runtime::Init`, two functions are called that process metadata. Before this, however, a startup hook is executed. DLLs have the capability to run certain functions when they are loaded, even before any explicit calls are made to their exported functions. One such startup function in this context is the constructor for `IL2CPP::utils::RegisterRuntimeInitializeAndCleanup`.

This constructor receives a pointer to another function, `s_IL2CPPCodegenRegistration()`, and saves this pointer in an initialization function table. This table is, in turn, accessed later by `RegisterRuntimeInitializeAndCleanup::ExecuteInitializations()`, which is one of the functions called by the aforementioned `Init::ExecuteInitializations()`. Which then calls `s_IL2CPPCodegenRegistration()` via the previously saved pointer. Finally, `s_IL2CPPCodegenRegistration()` through few other functions, calls `IL2CPP::vm::MetadataCache::Register()`. The crucial action within this last function is storing pointers to `IL2CPPCodeRegistration` and `IL2CPPMetadataRegistration`. These are the main tables, embedded within the binary, that store critical runtime metadata.

While the `global-metadata.dat` file provides a wealth of descriptive information about the application's structure (like types and names), it does not inherently contain the actual executable code instructions or the direct pointers to type-related runtime data structures that are initialized and used within the compiled C++ binary. This latter information is instead stored and accessed via the `IL2CPPCodeRegistration` and `IL2CPPMetadataRegistration` tables.

With the knowledge of the process it can be seen that the metadata file works in tandem with the Registration tables, and probably the most crucial point will be finding the function `IL2CPP::vm::MetadataCache::Register()`.

Chapter 3

The case analysis

3.1 The target

Numerous video games use IL2CPP as their backend and apply different degrees of obfuscation. Some developers choose not to make reverse engineering significantly harder, as the benefits from doing so are often seen as small compared to the cost of introducing these measures. This is often the case for single-player games, where cheating mostly affects the fairness of only the individual playing and doesn't impact other players, neither the game environment. Additionally, larger companies are usually more concerned with protecting proprietary code and intellectual property compared to smaller indie companies. Smaller developers sometimes even share technical details to encourage community collaboration on product maintenance and improvement.

The game selected for this case study is a popular video game developed by a company known for its intricate obfuscation schemes and its ongoing efforts to counter reverse engineering. Most of this company's games employ similar techniques, as they are generally consistent in terms of technology stack, gameplay, and general concepts. Many individuals attempt to recover useful information about these games' architectures for a variety of reasons. Motivations range from less ethical ones, such as developing cheats, and in-depth gameplay analysis to determine optimal strategies or event probabilities, to efforts focused solely on research and education, such as this current work. Additionally, some reverse engineering attempts aim to enable players to host their own game instances, commonly referred to as private servers. The games from this particular company are known for their aggressive monetization schemes, with many features often locked behind a paywall. Because of that, some players want the whole game experience without having to pay much, which is why they use those private servers. Implementations of those private servers exist, which are open source and can be found on sites like Github. However the community still has issues with reverse engineering the client, as the information about its interaction with the servers is also valuable for the development.

3.2 Metadata and binary files

3.2.1 Metadata files

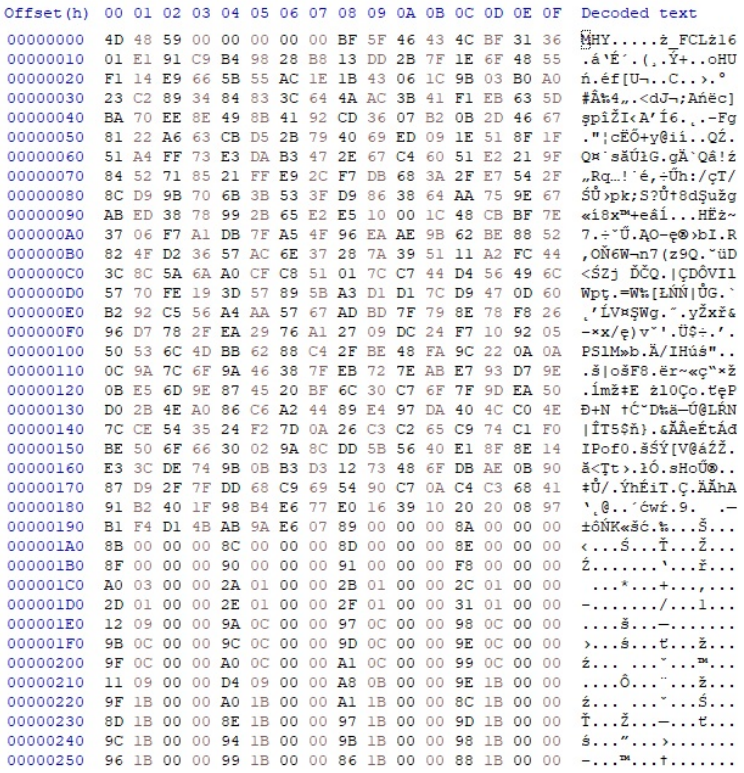
The standard procedure begins with identifying the metadata file. To achieve this, the game's main directory was search for any files having "metadata" in their name. Surprisingly instead of one, four files were found:

- `D:\...\GenshinImpact_Data\Managed\Metadata\global-metadata.dat`
- `D:\...\GenshinImpact_Data\Managed\Metadata\startup-metadata.dat`
- `D:\...\GenshinImpact_Data\Native\Metadata\global-metadata.dat`
- `D:\...\GenshinImpact_Data\Native\Metadata\startup-metadata.dat`

This is not a standard practice in IL2CPP; usually there exists one file containing metadata, `global-metadata.dat` which is present in the `Managed` directory. As listed, an additional file named `startup-metadata.dat` was included. Further investigation revealed that this file is not documented by Unity Technologies and that no references to its usage could be found in other known Unity-based games. What's more, after inspecting the files' contents it became apparent that each pair of files which were named in the same way were identical — confirmed by calculating their MD5 sums.

The next step is examining the files and comparing the contents with the example file from an empty project which was presented in the previous chapter. To do so, both were opened in HxD. As we can see in Figure 3.1a the header is way different from the standard format. There are no magic bytes or a version denominator. What instead can be seen are bytes `4D 48 59` translating to a string `MHY`. Which is an abbreviation of the game's company name.

The usual header seems to be encrypted due to high entropy of the data. Unlike typical IL2CPP data files or other sections of the analyzed file (which often contain high smattering of zeroes), this suspected encrypted header section lacks such patterns. Furthermore, this initial encrypted section spans 0x198 bytes, which is more than the standard header length. It suggests that not only the header but also the file's overall content structure or order might have been altered. During the further examination of the file other patterns of encryption were noticed that appeared periodically, with rest of the file seeming relatively normal (Fig 3.1b).



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	4D	48	59	00	00	00	00	00	BF	5F	46	43	4C	BF	31	36	MHY....._FCL.16
00000010	01	E1	91	C9	B4	98	28	B8	13	DD	2B	7F	1E	6F	48	55	.á'É´.(¸.Ý+..oHU
00000020	F1	14	E9	66	5B	55	AC	1E	1B	43	06	1C	9B	03	B0	A0	ń.éf[U¬..C..›.°
00000030	23	C2	89	34	84	83	3C	64	4A	AC	3B	41	F1	EB	63	5D	#Â‰4„.<dJ¬;Ańëc]
00000040	BA	70	EE	8E	49	8B	41	92	CD	36	07	B2	0B	2D	46	67	şpîŽI‹A'Í6.².-Fg
00000050	81	22	A6	63	CB	D5	2B	79	40	69	ED	09	1E	51	8F	1F	.¦cËŐ+y@ií..QŹ.
00000060	51	A4	FF	73	E3	DA	B3	47	2E	67	C4	60	51	E2	21	9F	Q¤˙săÚłG.gÄ`Qâ!ź
00000070	84	52	71	85	21	FF	E9	2C	F7	DB	68	3A	2F	E7	54	2F	„Rq…!˙é,÷Űh:/çT/
00000080	8C	D9	9B	70	6B	3B	53	3F	D9	86	38	64	AA	75	9E	67	ŚŮ›pk;S?Ů†8dŞužg
00000090	AB	ED	38	78	99	2B	65	E2	E5	10	00	1C	48	CB	BF	7E	«í8x™+eâĺ...HËż~
000000A0	37	06	F7	A1	DB	7F	A5	4F	96	EA	AE	9B	62	BE	88	52	7.÷ˇŰ.ĄO–ę®›bI.R
000000B0	82	4F	D2	36	57	AC	6E	37	28	7A	39	51	11	A2	FC	44	‚OŇ6W¬n7(z9Q.˘üD
000000C0	3C	8C	5A	6A	A0	CF	C8	51	01	7C	C7	44	D4	56	49	6C	<ŚZj ĎČQ.|ÇDÔVIl
000000D0	57	70	FE	19	3D	57	89	5B	A3	D1	D1	7C	D9	47	0D	60	Wpţ.=W‰[ŁŃŃ|ŮG.`
000000E0	B2	92	C5	56	A4	AA	57	67	AD	BD	7F	79	8E	78	F8	26	˛'ÍV¤ŞWg.˝.yŽxř&
000000F0	96	D7	78	2F	EA	29	76	A1	27	09	DC	24	F7	10	92	05	–×x/ę)vˇ'.Ü$÷.'.
00000100	50	53	6C	4D	BB	62	88	C4	2F	BE	48	FA	9C	22	0A	0A	PSlM»b.Ä/IHúś"..
00000110	0C	9A	7C	6F	9A	46	38	7F	EB	72	7E	AB	E7	93	D7	9E	.š|ošF8.ër~«ç“×ž
00000120	0B	E5	6D	9E	87	45	20	BF	6C	30	C7	6F	7F	9D	EA	50	.ĺmž‡E żl0Ço.tęP
00000130	D0	2B	4E	A0	86	C6	A2	44	89	E4	97	DA	40	4C	C0	4E	Đ+N †Ć˘D‰ä–Ú@LŔN
00000140	7C	CE	54	35	24	F2	7D	0A	26	C3	C2	65	C9	74	C1	F0	|ÎT5$ň}.&ĂÂeÉtÁđ
00000150	BE	50	6F	66	30	02	9A	8C	DD	5B	56	40	E1	8F	8E	14	IPof0.šŚÝ[V@áŹŽ.
00000160	E3	3C	DE	74	9B	0B	B3	D3	12	73	48	6F	DB	AE	0B	90	ă<Ţt›.łÓ.sHoŰ®..
00000170	87	D9	2F	7F	DD	68	C9	69	54	90	C7	0A	C4	C3	68	41	‡Ů/.ÝhÉiT.Ç.ÄĂhA
00000180	91	B2	40	1F	98	B4	E6	77	E0	16	39	10	20	20	08	97	'˛@..˝ćwŕ.9. .—
00000190	B1	F4	D1	4B	AB	9A	E6	07	89	00	00	00	8A	00	00	00	±ôŃK«šć.‰...Š...
000001A0	8B	00	00	00	8C	00	00	00	8D	00	00	00	8E	00	00	00	‹...Ś...Ť...Ž...
000001B0	8F	00	00	00	90	00	00	00	91	00	00	00	F8	00	00	00	Ź.......'...ř...
000001C0	A0	03	00	00	2A	01	00	00	2B	01	00	00	2C	01	00	00	...*...+...,...
000001D0	2D	01	00	00	2E	01	00	00	2F	01	00	00	31	01	00	00	-......./...1...
000001E0	12	09	00	00	9A	0C	00	00	97	0C	00	00	98	0C	00	00	....š...—.......
000001F0	9B	0C	00	00	9C	0C	00	00	9D	0C	00	00	9E	0C	00	00	›...ś...ť...ž...
00000200	9F	0C	00	00	A0	0C	00	00	A1	0C	00	00	99	0C	00	00	ź... ...ˇ...™...
00000210	11	09	00	00	D4	09	00	00	A8	0B	00	00	9E	1B	00	00	....Ô...¨...ž...
00000220	9F	1B	00	00	A0	1B	00	00	A1	1B	00	00	8C	1B	00	00	ź... ...ˇ...Ś...
00000230	8D	1B	00	00	8E	1B	00	00	97	1B	00	00	9D	1B	00	00	Ť...Ž...—...ť...
00000240	9C	1B	00	00	94	1B	00	00	9B	1B	00	00	98	1B	00	00	ś...”...›.......
00000250	96	1B	00	00	99	1B	00	00	86	1B	00	00	88	1B	00	00	–...™...†.......

(a) Beginning of the file with custom header.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
01706180	2E	00	00	00	25	00	00	00	48	52	00	00	13	00	00	00	....%...HR.....
01706190	2F	02	00	00	29	01	00	00	49	52	00	00	4A	52	00	00	/...)...IR..JR..
017061A0	A5	0E	00	00	95	01	00	00	4B	52	00	00	44	00	00	00	Ą...•...KR..D...
017061B0	4C	52	00	00	4D	52	00	00	13	00	00	00	E3	0B	00	00	LR..MR......ă...
017061C0	25	00	00	00	4E	52	00	00	4F	52	00	00	15	01	00	00	%...NR..OR.....
017061D0	50	52	00	00	DE	1F	00	00	51	52	00	00	10	01	00	00	PR..Ţ...QR.....
017061E0	52	52	00	00	53	52	00	00	45	01	00	00	92	01	00	00	RR..SR..E...'...
017061F0	54	52	00	00	2D	00	00	00	00	00	00	00	55	52	00	00	TR..-.......UR..
01706200	56	52	00	00	2F	00	00	00	13	00	00	00	42	0F	00	00	VR../.......B...
01706210	DF	00	00	00	9F	01	00	00	57	52	00	00	42	00	00	00	ß...ź...WR..B...
01706220	25	00	00	00	8C	00	00	00	9F	01	00	00	00	00	00	00	%...Ś...ź.......
01706230	25	00	00	00	58	52	00	00	59	52	00	00	39	00	00	00	%...XR..YR..9...
01706240	5A	52	00	00	5B	52	00	00	5C	52	00	00	19	01	00	00	ZR..[R..\R......
01706250	5D	52	00	00	13	00	00	00	5E	52	00	00	9E	0E	00	00	]R......^R..ž...
01706260	13	00	00	00	92	05	00	00	2F	00	00	00	5F	52	00	00	....'.../..._R..
01706270	60	52	00	00	0D	01	00	00	E5	04	00	00	61	52	00	00	`R......Í...aR..
01706280	D9	00	00	00	62	52	00	00	42	00	00	00	42	00	00	00	Ů...bR..B...B...
01706290	12	16	00	00	23	00	00	00	63	52	00	00	E5	00	00	00	....#...cR...Í...
017062A0	64	52	00	00	65	52	00	00	66	52	00	00	58	00	00	00	dR..eR..fR..X...
017062B0	8C	00	00	00	67	52	00	00	22	01	00	00	68	52	00	00	Ś...gR.."...hR..
017062C0	69	52	00	00	00	00	00	00	8E	00	00	00	92	01	00	00	iR......Ž...'...
017062D0	6A	52	00	00	13	00	00	00	9A	01	00	00	6B	52	00	00	jR......š...kR..
017062E0	6C	52	00	00	46	CC	16	B0	D2	A2	E6	AD	AF	9A	2C	63	lR..FĚ.°Ňˇć.š,c
017062F0	DC	F0	FC	60	07	71	68	3A	2B	0A	38	38	59	3F	7E	ED	Üđü`.qh:+.88Y?~í
01706300	B5	37	4E	EB	1D	16	BA	C4	F4	6C	82	C2	81	E4	CF	77	µ7Në..şÄôl‚Â.äĎw
01706310	0E	9B	9F	75	1B	BC	0B	4F	4D	B2	D3	4C	CA	89	21	02	.›źu.Ľ.OM¸ÓLĘ‰!.
01706320	67	00	EA	FF	59	57	37	B5	01	30	07	B3	10	2F	73	8C	g.ę˙YW7µ.0.ł./sŚ
01706330	40	47	3B	8A	0A	FD	88	3F	5A	75	51	3D	F5	D3	C4	16	@G;Š.ý.?ZuQ=ŐÓÄ.
01706340	99	AA	94	14	C4	A1	DA	C9	33	D8	A2	C7	AD	79	16	A1	™Ş”.Ä˘ÚÉ3Ř˘Ç.y.˘
01706350	72	11	E6	9E	F4	46	2C	54	7C	3F	F4	51	6B	1E	68	2B	r.ćžôF,T|?ôQk.h+
01706360	CB	54	38	29	61	EC	7D	DE	D5	82	4D	DC	4B	C3	B9	B5	ËT8)aě}ŢŐ‚MÜKĂąµ
01706370	14	BC	81	B3	DE	90	CF	68	AE	E9	9F	66	62	68	0B	40	.Ľ.łŢ.Ďh®éźfbh.@
01706380	ED	1E	D3	3D	30	36	21	F3	07	4D	E9	F0	27	0E	5D	CA	í.Ó=06!ó.Méđ'.]Ę
01706390	46	64	2D	C8	57	DC	72	7D	60	92	3A	7B	A3	B3	AE	54	Fd-ČWÜr}`':{Łł®T
017063A0	9F	CB	7E	52	75	81	C4	07	39	F9	94	05	2A	4F	DA	BA	źË~Ru.Ä.9ů”.*OÚş
017063B0	53	27	A2	B8	91	26	16	92	92	5C	E6	8F	31	F4	2B	45	S'˘¸'&.’’\ćŹ1ô+E
017063C0	9C	8A	F3	42	E1	CA	67	1C	EB	A3	37	1A	A1	99	7D	CF	śŠóBáĘg.ëŁ7.˘™}Ď
017063D0	75	F1	4D	CD	EF	6F	B9	A6	B4	06	81	A4	B0	3E	CF	59	uńMÍďoą¦´..¤°>ĎY

(b) Another part of the file with encrypted contents.

Figure 3.1: Hex dump of the `global-metadata.dat` file.

3.2.2 The main binary

In the game currently being analyzed, the Managed folder is conspicuously empty, containing only a few minor files. However, within the Native folder, a file named UserAssembly.dll was discovered. This file was identified as a strong candidate for the main game binary, a suspicion that was later confirmed through investigation.

Interestingly, the Native folder also contains a surprisingly small number of files. This is likely because many library files, which would typically reside there, are instead located in the game's root directory. This arrangement suggests a highly customized game build.

3.3 Reverse engineering the game binary

The current main objective is to recover the `global-metadata.dat` file. The `startup-metadata.dat` file will be set aside for the time being, as its purpose is binary progresses, its role may become more evident.

3.3.1 Tools and preparations for analysis

Methods of analysis as well as tools used for analysis in IL2CPP are often highly specific to the platform. While reverse engineering C++ code is generally a daunting task, the analysis of IL2CPP is significantly easier thanks to its open-source codebase. The core components that make up the IL2CPP runtime environment are written in C++ and included with Unity installations. This access to the source code proved invaluable during this analysis, allowing for direct comparison between the decompiled output and the original C++ code, which helped in identifying key functionalities and points of interest.

Another valuable technique involves building an empty Unity project configured with the IL2CPP backend. A key in this method is compiling with a Program Database (PDB) file. PDB files store crucial debugging information for binaries, such as symbols, denoting information like type definitions and function signatures [5]. This method yields a binary structurally similar to the target game's executable but without its specific customizations or obfuscation. This approach provides a reference point, acting as an analytical middle-ground between the obscure output from de-compiling the obfuscated game and the original and clean IL2CPP source code.

The reason why decompiled code can look so different from the original source code is twofold, involving both the compiler and the decompiler. On one side the decompiler applies optimizations, among which are: reordering instructions or changing structures of loops and conditionals. It also is responsible for inlining functions,

which directly copies the function body into code, and effectively makes it harder to follow functions, alters control flow, manages memory (e.g. by optimizing structs' fields layouts), lowering abstractions (by transforming loops or lambdas into much simpler forms) and more.

The PDB file is then loaded into the subsequent tool in the analysis workflow: a decompiler. For this, IDA Pro was used. IDA Pro's capabilities include disassembling and decompiling binary files, identifying their sections (e.g., `.text`, `.data`), and supporting the development of custom helper plugins using Python, among other features. Another key tool used in this analysis was Process Monitor (ProcMon). According to its official documentation:

> "Process Monitor is an advanced monitoring tool for Windows that shows real-time file system, Registry and process/thread activity. [...] rich and non-destructive filtering, comprehensive event properties such as session IDs and user names, reliable process information, full thread stacks with integrated symbol support for each operation, simultaneous logging to a file, and much more" [6].

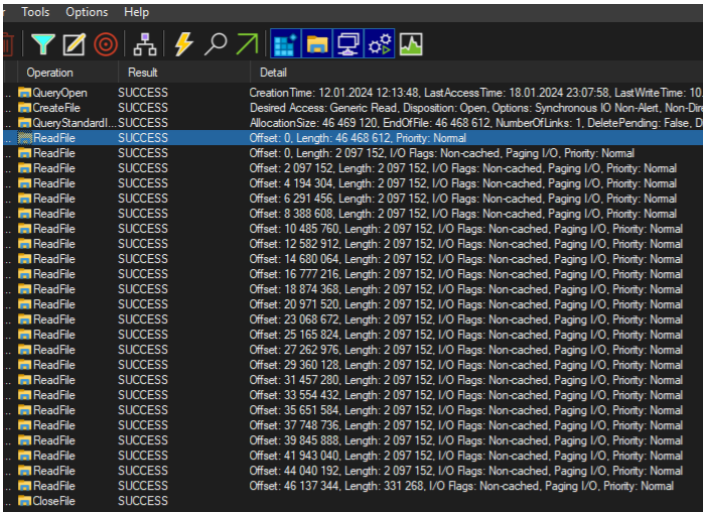
Other tools that were initially considered included debuggers and memory manipulation tools. However, research revealed that the game features an extensive anti-cheat system that blocks process attachment and uses anti-debugging techniques[7]. As a result, static analysis was chosen as the primary method for recovering information.

3.3.2 Finding places of interest

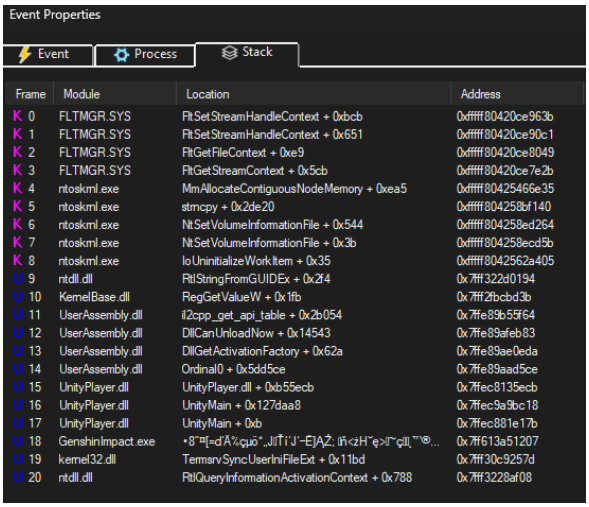
An effective starting point for the analysis is identifying the code that accesses the primary metadata file. Once the file is accessed, its contents are assumed to be used and therefore must be decrypted.

To spot such places, Process Monitor (ProcMon) was used. The program's functionality to monitor Windows API calls allows one to check when the process opens a file. What is more, it can look for a particular file. With these filters configured, the main binary was executed, revealing crucial information. From Figure 3.2a, it can be seen that there are multiple functions that operate on the file. `CreateFile` has a somewhat misleading name as it is also responsible for opening a file. By examining the first `ReadFile` call, which reads the entire file starting at offset 0 and spanning its full length, the corresponding stack trace leading to the call can be inspected. The most recent function call made within the application resides in `UserAssembly.dll` as seen in Figure 3.2b. The function names listed, such as `Ordinal0`, are not valid due to the absence of debugging symbols, and the program's attempts to deduce them are inaccurate. Instead, what indicates where the call was made is the absolute address displayed on the right-hand side — `0x7FFE89B55F64`.

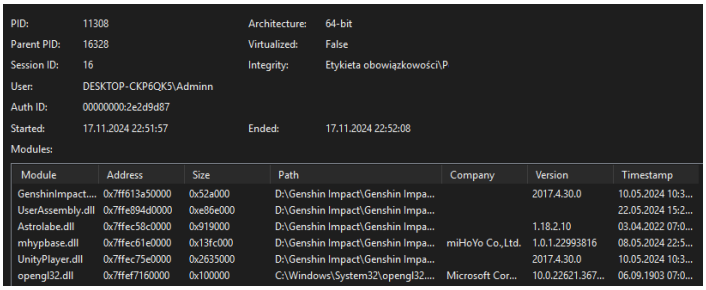
This address reveals the exact location of the function call, relative to the base memory address at which the DLL was loaded. This address, in turn, can be inferred by further inspecting the process itself — that is, where and which DLLs were loaded. In Figure 3.2c, it can be seen that the base address of `UserAssembly.dll` is `0x7FFE894D0000`. With that information, the binary can be loaded inside IDA Pro for further inspection.



(a) Calls accessing the metadata file.



(b) Stack trace of the first `ReadFile` call.



(c) DLL's loaded by the process.

Figure 3.2: Procmon process inspection.

3.3.3 Static analysis

Following the analysis phase, a static inspection of the binary was performed using IDA Pro. To make the process easier, structure definitions were imported to IDA, as the headers containing them are available due to IL2CPP being open-source. The source for the header is a Github repository [8] that extracts the headers from source code to make a IDA friendly format for import. Furthermore, since it had been established that the metadata file was obfuscated, it was assumed that the corresponding struct fields were likely obfuscated as well. Therefore, all existing field definitions were deleted and set to "unknown" + offset for later assignment.

```
typedef struct IL2CPPGlobalMetadataHeader {
    int32_t unknown00;
    int32_t unknown04;
    int32_t unknown08;
    int32_t unknown0C;
    // ...
    int32_t unknown18C;
    int32_t unknown190;
    int32_t unknown194;
} IL2CPPGlobalMetadataHeader;
```

Listing 3.1: IL2CPPGlobalMetadataHeader struct

Once the file is loaded in IDA Pro, inspecting the previously found address is futile. This is because IDA Pro, like many disassemblers, defaults to loading the DLL at a placeholder base address. To work with the actual addresses the DLL used during runtime, the file was rebased within IDA. This procedure shifted all internal addresses, setting the new base address to `0x7FFE894D0000` which as mentioned earlier, was the base address of `UserAssembly.dll`, when loaded into memory. After rebasing, navigating to the point of interest becomes simpler. Using IDA's "jump to address" feature, the disassembly view where `ReadFile` is called can be accessed (as shown in Figure 3.3). However, since analyzing assembly language exclusively can be very time-consuming, the code was primarily inspected from this point onward using the built-in decompiler. This tool generates C++-like code which, though often verbose, is significantly easier to comprehend than raw assembly.

```
.text:00007FFE89B55F5B                mov     r8d, ebx
.text:00007FFE89B55F5E                call    cs:ReadFile
.text:00007FFE89B55F64                test    eax, eax
.text:00007FFE89B55F66                jz      short loc_7FFE89B55F86
.text:00007FFE89B55F68                xor     eax, eax
.text:00007FFE89B55F6A                cmp     [rsp+58h+var_2C], r14d
.text:00007FFE89B55F6F                jz      short loc_7FFE89B55F98
.text:00007FFE89B55F71                jmp     short loc_7FFE89B55F9C
.text:00007FFE89B55F73 ; ---------------------------------------------------------------------------
.text:00007FFE89B55F73
.text:00007FFE89B55F73 loc_7FFE89B55F73:                      ; CODE XREF: sub_7FFE89B55F10+19↑j
.text:00007FFE89B55F73                call    cs:GetLastError
.text:00007FFE89B55F79                test    eax, eax
.text:00007FFE89B55F7B                jz      short loc_7FFE89B55FC0
.text:00007FFE89B55F7D                lea     rcx, aMemorymappedfi ; "########MemoryMappedFile::ReadToMemoryD"...
.text:00007FFE89B55F84                jmp     short loc_7FFE89B55FAC
.text:00007FFE89B55F86 ; ---------------------------------------------------------------------------
```

Figure 3.3: Readfile function under the found address — `0x7FFE89B55F64`.

The current function appears to read a file and then map its contents into memory. To explore more relevant sections of the code, one can navigate upwards through the call tree to identify the calling function. This process involves finding cross-references to the current function, jumping to its callers, and then repeating these steps. In this particular case, this navigation is straightforward because each function in the call stack is referenced only once. After bypassing an intermediate function named `sub_7FFFE8896E630`, which also seemed to handle file operations, an interesting function was found (Fig 3.4a). As seen, this function is the one called with arguments, which are the names of the files that were inspected earlier. During the analysis of this function, the empty reference project loaded into IDA Pro was used as a reference point. A search for the `global-metadata.dat` string within its code revealed a very similar code snippet, as shown in Figure 3.4b.

These observations also suggest that a simpler method for locating the relevant code snippet would be searching for the string `global-metadata.dat` and then navigating to its cross-references. However, this approach may not be effective in all scenarios, as the underlying process could be more heavily obfuscated, or the string itself might be absent or further obscured, and the one showed above, should generally work in most cases.

```
}
sub_7FFE89AB2080();
v26 = (char *)sub_7FFE89AFEA30("global-metadata.dat");
v27 = sub_7FFE89AFEA30("startup-metadata.dat");
dword_7FFE8BE65538 = *(_DWORD *)(qword_7FFE8B96E618 + 4);
qword_7FFE8B96E628 = (__int64)(v26 + 408);
qword_7FFE8B96E630 = (__int64)v27;
v28 = *(signed int *)(qword_7FFE8B96E608 + 160) ^ 0xC9C11F6i64;
v29 = sub_7FFE89B561C0(qword_7FFE8B96E1D0, v28, 8i64);
v30 = v29;
if ( (_DWORD)v28 )
  sub_7FFE89BBADA0(v29, 0i64, 8 * v28);
qword_7FFE8B96E638 = v30;
v31 = *(_DWORD *)(qword_7FFE8B96E620 + 320) ^ 0x272A7182;
v32 = (unsigned __int64)(0xDD67C8A60DD67C8Bui64
                       * (unsigned __int128)(*(signed int *)(qword_7FFE8B96E620 + 320) ^ 0x272A7182ui64) >> 64) >> 6;
v33 = sub_7FFE89B561C0(qword_7FFE8B96E1D0, v32, 8i64);
v34 = v33;
if ( v31 >= 0x4A )
  sub_7FFE89BBADA0(v33, 0i64, 8 * v32);
qword_7FFE8B96E640 = v34;
v35 = *(_DWORD *)(qword_7FFE8B96E620 + 60) - 355368699;
v36 = sub_7FFE89B561C0(
        qword_7FFE8B96E1D0,
        (unsigned __int64)(0x4924924924924925i64 * (unsigned __int128)((unsigned __int64)v35 >> 2) >> 64) >> 1,
        8i64);
v37 = v36;
```

(a) Snippet in the main binary.

```
v0 = il2cpp::vm::MetadataLoader::LoadMetadataFile("global-metadata.dat");
s_GlobalMetadata = v0;
s_GlobalMetadataHeader = v0;
s_TypeInfoTable = il2cpp::utils::Memory::Calloc(s_Il2CppMetadataRegistration->typesCount, 8ui64);
s_TypeInfoDefinitionTable = il2cpp::utils::Memory::Calloc(
                              (s_GlobalMetadataHeader->typeDefinitionsCount * 0x4EC4EC4EC4EC4EC5ui64 >> 64) >> 5,
                              8ui64);
s_MethodInfoDefinitionTable = il2cpp::utils::Memory::Calloc(
                                ((s_GlobalMetadataHeader->methodsCount * 0x2492492492492493ui64 >> 64)
                                 + ((s_GlobalMetadataHeader->methodsCount
                                   - (s_GlobalMetadataHeader->methodsCount * 0x2492492492492493ui64 >> 64)) >> 1)) >> 5,
                                8ui64);
s_GenericMethodTable = il2cpp::utils::Memory::Calloc(s_Il2CppMetadataRegistration->methodSpecsCount, 8ui64);
s_ImagesCount = s_GlobalMetadataHeader->imagesCount >> 5;
s_ImagesTable = il2cpp::utils::Memory::Calloc(s_ImagesCount, 0x40ui64);
s_AssembliesCount = (s_GlobalMetadataHeader->assembliesCount * 0xF0F0F0F0F0F0F0F1ui64 >> 64) >> 6;
s_AssembliesTable = il2cpp::utils::Memory::Calloc(s_AssembliesCount, 0x60ui64);
v1 = s_GlobalMetadataHeader;
v2 = s_GlobalMetadata;
v3 = (s_GlobalMetadata + s_GlobalMetadataHeader->imagesOffset);
```

(b) Snippet in the sample project binary.

Figure 3.4: Equivalent code snippets in both binaries.

A key observation is that the memory location where `global-metadata.dat` was initially loaded is not subsequently accessed, even though data from such a file is utilized by `IL2CPP::utils::Memory::Calloc` in the empty (reference) project. The only place where this loaded data appears to be used in the relevant disassembly segment is immediately after the file is opened:

```
1 v26 = sub_7FFE89AFEA30("global-metadata.dat");
2 ...
3 qword_7FFE8B96E628 = v26 + 408;
```

Here, surprisingly, the file pointer `v26` is advanced by 408 bytes, which is 0x198 in hexadecimal. This number, 0x198, was previously identified as the size of the metadata header. Thus, the variable holding the file pointer is moved to a position immediately after the header, effectively skipping it. This is unexpected, as the header is crucial for initializing IL2CPP structures, strongly suggesting that the actual header data is initialized or accessed from a different source.

The code snippets also suggest that a function, `sub_7FFE89B561C0`, is the `IL2CPP::utils::Memory::Calloc` equivalent in the analyzed binary. This conclusion is based on it being called the same number of times and with similar arguments: its last argument corresponds to the size passed to `Calloc`, and its second argument is a specific variable. Those variables match the following definition of the `Calloc` function in IL2CPP source code:

```
1 #define IL2CPP_CALLOC(count, size)
      IL2CPP::utils::Memory::Calloc(count,size)
```

Interestingly, this second argument, `v32` corresponds in purpose to the first argument used with `Calloc` in the empty project. For instance, in the second `Calloc` call within the empty project, this first argument is sourced from `s_GlobalMetadataHeader->typeDefinitionsCount`. This suggests that `v32` in the analyzed code also accesses this specific header field.

Further examination reveals that `v32` is assigned from `qword_7FFE8B96E620`. Interestingly, this same `qword_7FFE8B96E620` is also used in all other instances where the metadata header is accessed in the reference code. This naturally raises the suspicion that `qword_7FFE8B96E620` (or the data it points to) is the actual metadata header, and the initially loaded `global-metadata.dat` data (whose memory region was not accessed) is merely a decoy. To confirm this suspicion, it's necessary to determine when the global variable `qword_7FFE8B96E620` receives its value. Listing cross-references (xrefs) to this variable reveals thousands of instances. However, IDA Pro allows these references to be sorted by the type of operation (read or write) and doing so shows only a single write operation to this variable.

```
void *sub_7FFE89ADCC00()
{
  void *result; // rax

  qword_7FFE8B96E600 = (__int64)&unk_7FFE8A7407B8;
  qword_7FFE8B96E608 = (__int64)&unk_7FFE8A743148;
  qword_7FFE8B96E610 = (__int64)&unk_7FFE8A7437A0;
  qword_7FFE8B96E618 = (__int64)&unk_7FFE8A740848;
  qword_7FFE8B96E620 = (__int64)&unk_7FFE8A740620;
  dword_7FFE8B96E1A0 = 3458;
  qword_7FFE8B96E1A8 = (__int64)&qword_7FFE8B967200;
  result = &unk_7FFE8A6E2100;
  qword_7FFE8B96E1B0 = (__int64)&unk_7FFE8A6E2100;
  return result;
}
```

Figure 3.5: Assignment of the real header.

In that piece of code, multiple global variables get their values assigned to data stored in the .rodata section (which contains static constants, read-only data). Specifically, in this instance, the variable `qword_7FFE8B96E620` is assigned the address `0x7FFE8B96E620`, which points to data residing in this section. Navigating to this address and inspecting its contents in a hex viewer confirms the theory that this is indeed the actual header. A clear indicator is that the first three bytes at this location form the developer's tag (`MHY`), the same tag previously observed in the decoy metadata file. It also has the same length as the one in the `global-metadata.dat` file.

```
00007FFE8A740620  4D 48 59 00 00 00 00 00  41 53 AC 34 11 DE 09 35  MHY.....AS¬4.Ţ.5
00007FFE8A740630  8B F2 A3 63 90 A2 7C 3F  40 35 32 05 57 8D 20 20  ‹ňŁc.˜|?@52.WŤ··
00007FFE8A740640  D3 1F 6E 07 9E 93 49 05  D3 BA 98 6F B5 52 94 55  Ó.n.ž“I.Óş.oµR”U
00007FFE8A740650  21 0A 70 16 43 9C F8 63  25 69 64 76 BF 85 ED 15  !.p.Cśřc%idvż…í.
00007FFE8A740660  10 61 CC 43 5F 0A BF 40  BB 46 6B 39 51 25 AF 03  .aĚC_.ż@»Fk9Q%Ż.
00007FFE8A740670  C0 7E 8F 69 79 A5 EE 2A  6E A0 3A 23 25 7E EA 00  Ŕ~ŹiyĄî*n·:#%~ę.
00007FFE8A740680  FE 6B 9C 19 8F 50 B2 1D  A2 6D 10 74 24 98 45 6A  ţkś.ŹP².˘m.t$.Ej
00007FFE8A740690  42 4E 99 25 2F 44 75 19  F3 C3 AD 3F 4B 5F 83 2D  BN™%/Du.óĂ?K_.-
00007FFE8A7406A0  05 4D 7E 45 47 12 ED 75  99 62 F8 17 64 67 BD 61  .M~EG.íu™bř.dg˝a
00007FFE8A7406B0  A8 9C FC 62 A8 EC 36 7C  9A E6 1D 60 84 BB CF 5C  ¨śüb¨ě6|šć.`„»Ď\
00007FFE8A7406C0  2D DC 07 44 79 14 CB 6C  69 18 4A 21 DA 95 8E 79  -Ü.Dy.Ëli.J!Ú•Žy
00007FFE8A7406D0  8D 80 46 09 EF A1 AC 5F  A6 B3 B6 41 33 F9 AF 72  Ť€F.ď˘¬_¦ł¶A3ůŻr
00007FFE8A7406E0  7B A3 49 17 73 FE 3E 62  DE B2 05 49 48 3E 50 04  {ŁI.sţ>bŢ².IH>P.
00007FFE8A7406F0  0A C0 82 27 36 0A 27 38  4D 47 ED 71 B7 5F B3 0A  .Ŕ‚'6.'8MGíq·_ł.
00007FFE8A740700  D9 91 3D 2F 58 1A 54 33  A1 45 CC 68 91 07 7F 0B  Ů‘=/X.T3˘ĚĚh‘...
00007FFE8A740710  68 AE 51 23 3B 44 55 4B  89 59 EB 03 77 7C 80 58  h®Q#;DUK‰Yë.w|€X
00007FFE8A740720  84 51 1F 1E C2 08 C0 46  EA A5 59 7F 8D FC 89 7E  „Q..Â.ŔFęĄY.Ťü‰~
00007FFE8A740730  A7 7A 2D 10 4E 56 5C 01  28 04 D3 2F E8 3C F3 1F  §z-.NV\.(.Ó/č<ó.
00007FFE8A740740  F2 B3 3E 28 92 A4 EF 44  C0 96 6D 1C C0 93 3B 68  ňł>(’¤ďDŔ–m.Ŕ“;h
00007FFE8A740750  33 ED 59 72 A4 88 41 35  86 6F 97 7F 50 46 4C 3F  3íYr¤.A5†o—.PFL?
00007FFE8A740760  47 6C 10 27 72 83 F3 42  8E 4B FD 03 68 5C 25 7D  Gl.'r.óBŽKý.h\%}
00007FFE8A740770  7E 89 B9 03 91 4D F7 6C  86 C6 F2 26 3D 4C 86 3F  ~‰ą.‘M÷l†ĆŇ&=L†?
00007FFE8A740780  80 A9 CA 68 1F A8 7D 5D  A1 04 04 44 CA A6 85 6E  €©Ęh.¨}]˘..DĘ¦…n
00007FFE8A740790  E9 84 2C 0D 71 C1 14 11  91 58 36 22 A2 9F 00 33  é„,.qÁ..‘X6"˘ź.3
00007FFE8A7407A0  50 BD C1 71 9B C1 8C 29  10 1C CF 3B FC 1D F3 79  P˝Áq›ÁŚ)..Ď;ü.óy
00007FFE8A7407B0  58 50 EE 47 01 13 22 16  40 BF 50 8A FE 7F 00 00  XPîG.."."@żPŠţ...
00007FFE8A7407C0  99 5A B2 6C 02 92 BF 31  80 F7 67 8A FE 7F 00 00  ™Z˛l.’ż1€÷gŠţ...
00007FFE8A7407D0  C0 90 8E 8B FE 7F 00 00  30 51 6D 8A FE 7F 00 00  Ŕ.Ž‹ţ...0QmŠţ...
00007FFE8A7407E0  74 E2 D8 59 2C ED 2C 20  90 2B 32 8A FE 7F 00 00  tâŘY,í,·.+2Šţ...
00007FFE8A7407F0  DC E9 2C 0B 05 25 EE 6D  00 21 6E 8A FE 7F 00 00  Üé,..%îm.!nŠţ...
00007FFE8A740800  00 72 96 8B FE 7F 00 00  F4 86 B9 36 00 00 00 00  .r–‹ţ...ô†ą6....
00007FFE8A740810  00 00 00 00 00 00 00 00  89 8B 68 09 DE 4C 14 70  ........‰‹h.ŢL.p
00007FFE8A740820  B0 87 94 8B FE 7F 00 00  F0 75 61 8A FE 7F 00 00  °‡”‹ţ...đuaŠţ...
00007FFE8A740830  30 33 74 8A FE 7F 00 00  83 A6 F0 50 00 00 00 00  03tŠţ....¦đP....
00007FFE8A740840  20 46 66 8A FE 7F 00 00  01 01 00 00 07 00 00 00  ·FfŠţ...........
00007FFE8A740850  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Figure 3.6: Hex dump of the embedded metadata header.

3.4 Header recovery

Now that the header has been identified, it is quite straightforward to examine its cross-references and identify locations where it is accessed. The harder challenge, however, is determining which field within the header is being accessed at each location.

To achieve this using static analysis, it is necessary to identify the code segments surrounding these access calls, as that context reveals where in the code the fragment under inspection is. This would normally be extremely tedious; however, with access to the IL2CPP source code and the decompiled output of an empty reference project, the process can be significantly sped up. The initial step is to identify the function under inspection, as this is an anchor point for the analysis. Since once the function is deduced, examining its definition in the IL2CPP source code or its disassembly (from the reference project as it's populated with symbols) can reveal which header fields it accesses. This information then allows these fields to be correctly identified and renamed in the decompiled view of the target binary.

While the previously described approach to function identification might sound trivial in theory, its practical application within an obfuscated binary can be a significant challenge. Identifying the specific function of interest often requires navigating the call tree to locate any previously identified functions that could serve as landmarks. This task is complicated by the fact that, initially, most functions in the binary are unnamed, apart from imported library functions. One effective technique in the initial stage is searching for unique hardcoded strings, as these sometimes appear in only a single function within the binary. Although identifying such a function via a unique string might not seem immediately valuable, it can serve as a crucial hint if encountered later during further analysis, helping in the identification of other related code segments. There is a chance, that such a seemingly insignificant function may be called by a more critical one. This relationship, identifiable via its cross-references (especially if the function has few callers), can then lead to the discovery and analysis of the more significant calling function.

Even after a function is correctly identified by its name and corresponding source code, expected references to header fields within that function might still be absent in the decompiled output, despite being present in the original source. This discrepancy can occur if the original functions are encapsulated by wrapper functions, a technique that can obscure direct call flows and data access patterns. In other cases, discrepancies can present in opposite way. A decompiled function might appear substantially larger or more complex than its original source code counterpart, potentially including header accesses not found in the original. Furthermore, repeated code segments may be observed within a single function, often as a result of compiler optimizations such as function inlining.

3.4.1 Initial observations and modifications

As en example of a process of analyzing the unknown header fields we will look at the found snippet with `IL2CPP::utils::Memory::Calloc` calls. Locating the counterpart to the previously discussed code snippet within the reference project revealed the function's name to be `MetadataCache::Initialize`. For an additional point of reference, the IL2CPP source code for this function was also examined.

```
void MetadataCache::Initialize()
{
    s_GlobalMetadata = vm::MetadataLoader::LoadMetadataFile("global-metadata.dat");
    s_GlobalMetadataHeader = (const Il2CppGlobalMetadataHeader*)s_GlobalMetadata;
    IL2CPP_ASSERT(s_GlobalMetadataHeader->sanity == 0xFAB11BAF);
    IL2CPP_ASSERT(s_GlobalMetadataHeader->version == 24);

    const Il2CppAssembly* assemblies = (const Il2CppAssembly*)((const char*)s_GlobalMetadata + s_GlobalMetadataHeader->assembliesOffset);
    for (uint32_t i = 0; i < s_GlobalMetadataHeader->assembliesCount / sizeof(Il2CppAssembly); i++)
        il2cpp::vm::Assembly::Register(assemblies + i);

    // Pre-allocate these arrays so we don't need to lock when reading later.
    // These arrays hold the runtime metadata representation for metadata explicitly
    // referenced during conversion. There is a corresponding table of same size
    // in the converted metadata, giving a description of runtime metadata to construct.
    s_TypeInfoTable = (Il2CppClass**)IL2CPP_CALLOC(s_Il2CppMetadataRegistration->typesCount, sizeof(Il2CppClass*));
    s_TypeInfoDefinitionTable = (Il2CppClass**)IL2CPP_CALLOC(s_GlobalMetadataHeader->typeDefinitionsCount / sizeof(Il2CppTypeDefinition), sizeof(Il2CppClass*));
    s_MethodInfoDefinitionTable = (const MethodInfo**)IL2CPP_CALLOC(s_GlobalMetadataHeader->methodsCount / sizeof(Il2CppMethodDefinition), sizeof(MethodInfo*));
    s_GenericMethodTable = (const Il2CppGenericMethod**)IL2CPP_CALLOC(s_Il2CppMetadataRegistration->methodSpecsCount, sizeof(Il2CppGenericMethod*));
    s_ImagesCount = s_GlobalMetadataHeader->imagesCount / sizeof(Il2CppImageDefinition);
    s_ImagesTable = (Il2CppImage*)IL2CPP_CALLOC(s_ImagesCount, sizeof(Il2CppImage));
```

Figure 3.7: Source code reference for the inspected snippet.

Initial inspection suggests that in the decompiled code (Figure 3.4a), the control flow has not been altered; therefore, it can be assumed that the order of `IL2CPP::utils::Memory::Calloc` calls has also not been modified. For the ease analysis, variable types were temporarily hidden, and previously identified functions and variables were named.

Additionally, a pointer to t`IL2CPPGlobalMetadataHeader` was assigned to the identified header variable. This change was made because this variable was initially an int64 (64 bit signed integer), and its counterpart in the source code (Figure 3.7) is also a pointer. Following these modifications, the decompiled code appears as shown in Figure 3.8. With these changes, the decompiled code is easier to handle, as it now looks more like the empty project.

3.4.2 Identifying the header fields

The analysis starts with looking at the first `IL2CPP::utils::Memory::Calloc` call (referred to simply as `Calloc` from this point forward) - in Figure 3.8. The second argument in this call is not the header pointer, but is instead another global variable: `qword_7FFE8B96E608`. This matches the pattern in the source code, where the same kind of argument is `s_IL2CPPMetadataRegistration`.

As this global variable is likely to appear again in later parts of the analysis, it's worth to rename it in the decompiler view from `qword_7FFE8B96E608` to its source code name, `s_IL2CPPMetadataRegistration`. Furthermore, its type was adjusted to match the original — a pointer to this structure. After these adjustments, the

code segment now looks like this:

```
1 v28 = s_IL2CPPMetadataRegistration[1].genericMethodTableCount ^
      0xC9C11F6i64;
2 v29 = IL2CPP::utils::Memory::Calloc(qword_7FFE8B96E1D0, v28, 8i64);
```

There are a few strange things about this snippet. First, in both the source code (Figure 3.7) and the empty project (Figure 3.4b), the accessed field is `typesCount`. Second, the field in the decompiled code is XORed with a hardcoded value. This implies that this structure has also been reordered and obfuscated.

```
v26 = sub_7FFE89AFEA30("global-metadata.dat");
v27 = sub_7FFE89AFEA30("startup-metadata.dat");
dword_7FFE8BE65538 = *(qword_7FFE8B96E618 + 4);
metadataContentsNoHeader = v26 + 408;
startupMetadataFile = v27;
v28 = *(qword_7FFE8B96E608 + 160) ^ 0xC9C11F6i64;
v29 = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, v28, 8i64);
v30 = v29;
if ( v28 )
  memset(v29, 0, 8 * v28);
unk_7FFE8B96E638 = v30;
v31 = globalMetadataRealHeader->unknown140 ^ 0x272A7182;
v32 = (0xDD67C8A60DD67C8Bui64 * (globalMetadataRealHeader->unknown140 ^ 0x272A7182ui64) >> 64) >> 6;
v33 = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, v32, 8i64);
v34 = v33;
if ( v31 >= 0x4A )
  memset(v33, 0, 8 * v32);
unk_7FFE8B96E640 = v34;
v35 = globalMetadataRealHeader->unknown3C - 355368699;
v36 = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, (0x4924924924924925i64 * (v35 >> 2) >> 64) >> 1, 8i64);
v37 = v36;
if ( v35 >= 0x1C )
  memset(v36, 0, 8 * ((0x4924924924924925i64 * (v35 >> 2) >> 64) >> 1));
unk_7FFE8B96E648 = v37;
v38 = (globalMetadataRealHeader->unknownCC >> 3) ^ 0x8EF0C8;
v39 = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, v38, 8i64);
v40 = v39;
if ( v38 )
  memset(v39, 0, 8i64 * v38);
unk_7FFE8B96E650 = v40;
v41 = (0xCCCCCCCCCCCCCCCDui64 * (globalMetadataRealHeader->unknown118 - 802355392) >> 64) >> 5;
dword_7FFE8B96E658[0] = v41;
v42 = v41;
v43 = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, v41, 96i64);
```

Figure 3.8: Source code reference for the inspected snippet.

In the second `Calloc` call within the analyzed binary, the header field accessed is `unknown140`. This field is also XORed and subsequently modified by arithmetic operations. However, these operations (e.g., bitwise shifts) don't resemble typical encryption methods. An examination of the decompiled empty reference project confirms that these are likely some transformation operations whose complex appearance may be a result of the compilation process, as they also appear there (Fig 3.4b). This `unknown140` field corresponds to `typeDefinitionCount` in the reference materials. From this, it can also be deduced that the variable `v33` is equivalent to `s_TypeInfoDefinitionTable`, as it stores the result of this `Calloc` call, mirroring the pattern in both the source code and the empty reference project. A similar pattern was noted for the previous `Calloc` call involving `s_IL2CPPMetadataRegistration`.

Now the same process can be applied to all other `Calloc` calls. However after analyzing all of them, a discrepancy appears. In both reference sources, `s_IL2CPPMetadataRegistration` is used twice for `Calloc`, but in the decompilation, it's just once. While the exact reason for this difference is still unclear for

now, what has been found so far indicates that some fields originally belonging to the `s_IL2CPPMetadataRegistration` structure might have been moved to the main header. This aligns with earlier observations that the header itself was modified and is larger than the usual one.

The results of this analysis of the code snippet are presented in Figure 3.9. In this figure, global variables and header fields have been renamed to their identified counterparts, and other functions have also been labeled accordingly.

```
il2cpp::os::Thread::Init(v25, -5501853106176i64, a4, a3);
s_GlobalMetadata = LoadMetadataFileReal("global-metadata.dat");
startup = LoadMetadataFileReal("startup-metadata.dat");
dword_7FFE8BE65538 = *(qword_7FFE8B96E618 + 4);
globalMetadataNoHeader = (s_GlobalMetadata + 408);
startupMetadataFile = startup;
v30 = *(s_Il2CppMetadataRegistration + 160) ^ 0xC9C11F6i64;
s_TypeInfoTable = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, v30, 8i64);
v32 = s_TypeInfoTable;
if ( v30 )
  memset(s_TypeInfoTable, 0, 8 * v30);
qword_7FFE8B96E638 = v32;
v33 = globalMetadataHeaderReal->typeDefinitionsCount ^ 0x272A7182;
v34 = (0xDD67C8A60DD67C8Bui64 * (globalMetadataHeaderReal->typeDefinitionsCount ^ 0x272A7182ui64) >> 64) >> 6;
s_TypeInfoDefinitionTable = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, v34, 8i64);
v36 = s_TypeInfoDefinitionTable;
if ( v33 >= 0x4A )
  memset(s_TypeInfoDefinitionTable, 0, 8 * v34);
::s_TypeInfoDefinitionTable = v36;
v37 = globalMetadataHeaderReal->methodsCount - 355368699;
s_MethodInfoDefinitionTable = il2cpp::utils::Memory::Calloc(
                                qword_7FFE8B96E1D0,
                                (0x4924924924924925i64 * (v37 >> 2) >> 64) >> 1,
                                8i64);
v39 = s_MethodInfoDefinitionTable;
if ( v37 >= 0x1C )
  memset(s_MethodInfoDefinitionTable, 0, 8 * ((0x4924924924924925i64 * (v37 >> 2) >> 64) >> 1));
qword_7FFE8B96E648 = v39;
v42 = (globalMetadataHeaderReal->methodSpecsCountMaybe >> 3) ^ 0x8EF0C8;
v41 = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, v42, 8i64);
rdi56 = v41;
if ( v42 )
  memset(v41, 0, 8i64 * v42);
qword_7FFE8B96E650 = rdi56;
s_ImagesCount = (0xCCCCCCCCCCCCCCCDui64 * (globalMetadataHeaderReal->imagesCount - 802355392) >> 64) >> 5;
dword_7FFE8B96E658 = s_ImagesCount;
v44 = s_ImagesCount;
s_ImagesTable = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, s_ImagesCount, 96i64);
```

Figure 3.9: Snippet after full analysis.

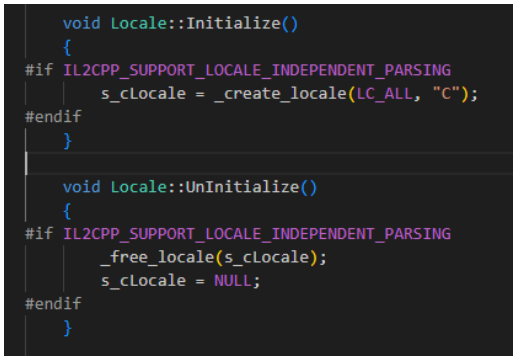
3.4.3 Investigating the snippet's context

It is worth noting that the matching code snippet in the analyzed game binary is found inside a much larger function than the `MetadataCache::Initialize` function from the reference project. This larger function in the game binary contains many more operations, suggesting it is a combined block of code into which `MetadataCache::Initialize`, and perhaps other functions, have been inlined. This inlining also helps explain the significant size difference: the function in the decompiled game binary spans over 5000 lines, whereas in the reference project it is only approximately 300 lines.

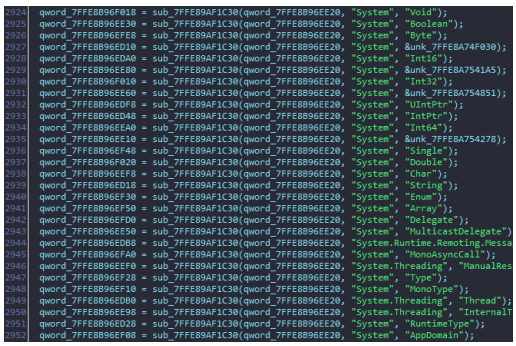
This suspected inlining can be investigated by examining the surrounding code within this large function and by navigating up its call tree. For example, searching for hardcoded strings within this function can reveal other potential points of interest. In the beginning of the function under analysis a hardcoded string `C` can be found as well as a named variable (Fig 3.10a). Generally a named variable like this shouldn't exist, however when the binary was loaded into IDA a notification about existence of debugging symbols appeared, which may be responsible for that. Even though the PDB file wasn't loaded, it is probable that just the generation of it left some named variables. Looking for the string "C" in the source code, showed a function that also uses it, as well as variables also using they keyword "locale". What's more, the `LC_ALL` value is defined in the source code as equal to 0, in a file `locale.h`.

Now the evidence points that the function `Locale::Initialize` is located in the same place as `MetadataCache::Initialize`, and their bodies are directly copied there. Another intriguing, very specific piece of code is one shown in 3.10c, where multiple calls of the same function are made with specific strings that seem to denote variable types. It was found far into the decompiled code, after the snippet with `Callocs`. Searching for those in source code was simple, as it is done only in one function, `Runtime::Init`, as seen in 3.10d. Interestingly enough, in the beggining of that function (Fig 3.11) there is a call to `os::Locale::Initialize()`, and after that a call to `MetadataCache::Initialize()`, which matches the order of those functions and the snippet in the decompiled code.

(a) Named variable Locale and string "C".



(b) Similar code in Locale::Initialize.



(c) Characteristic calls with strings.

(d) Identical calls in Runtime::Init.

Figure 3.10: Comparable code snippets. Left side — decompiled main binary, right side — source code.

This suspected inlining can be further investigated by examining the surrounding code within this large function and by navigating up its call tree. For example, searching for hardcoded strings within this function can reveal other potential points of interest. Near the beginning of the function under analysis, a hardcoded string "C" and a named variable are present (Figure 3.10a). The presence of such a descriptively named variable is uncommo for a release binary. However, upon loading the binary into IDA Pro, a notification indicated the existence of debugging symbols (and a path under which they were saved!), which likely explains this naming. Even if a PDB file was not explicitly loaded for this analysis, it might be possible that the build process itself embedded some symbol information, leading to descriptive variable names.

A search for the string "C" in the IL2CPP source code revealed a function that utilizes this string, alongside variables associated with the keyword "locale". Furthermore, the symbolic constant `LC_ALL` is defined with a value of 0 within the `locale.h` source file. After those findings, it can be safely assumed that the function that unpacks metadata is actually `Runtime::Init`, with many of its functions inlined. For confirmation other functions like `os::Thread::Init()` and `Thread::Initializ e()` were investigated to see if that's the same case for them and indeed it was. The analysis done so far helped identify several different functions within the large, decompiled code block. This, in turn, enables the identification of header fields accessed

```
void Runtime::Init(const char* filename, const char *runtime_version)
{
    SanityChecks();

    os::Initialize();
    os::Locale::Initialize();
    MetadataAllocInitialize();

    s_FrameworkVersion = framework_version_for(runtime_version);

    os::Image::Initialize();
    os::Thread::Init();

    il2cpp::utils::RegisterRuntimeInitializeAndCleanup::ExecuteInitializations();

    MetadataCache::Initialize();
    Assembly::Initialize();
    gc::GarbageCollector::Initialize();

    // Thread needs GC initialized
    Thread::Initialize();

    // Reflection needs GC initialized
    Reflection::Initialize();

    memset(&il2cpp_defaults, 0, sizeof(Il2CppDefaults));

    const Il2CppAssembly* assembly = Assembly::Load("mscorlib.dll");
```

Figure 3.11: Beginning of the `Runtime::Init` function.

in other segments of this block, as knowing which original function is responsible for an access helps narrow down the candidate fields from the header structure.

3.4.4 Deobfuscating deliberate misdirection

The process described previously is the general framework for identifying header fields. Initially, this involves identifying functions by searching for distinctive code patterns or data references, such as unique strings. These candidate functions are then cross-referenced with the IL2CPP source code or the decompiled output of the empty reference project. Once a function (and potentially surrounding functions within the inlined block) is identified, the specific header fields it accesses can be recognized. This investigative process can then be applied to "neighbouring" or related functions. Sometimes, however, even after a function is identified, further detailed analysis may be necessary. This is because the decompiled code can still differ significantly from both the original source code and the reference project's output, due to the mentioned changes introduced by the compiler or deliberate modifications by the developers.

One such example is a code snippet within the `MetadataCache::Initialize` function. This snippet extracts `ImageDefinition` data from the metadata file to populate an `ImagesTable` structure. This segment is located immediately after the previously analyzed `Calloc` calls and is present in both the decompiled target binary and the reference project.

```
s_ImagesCount = (0xCCCCCCCCCCCCCCCDui64 * (globalMetadataHeaderReal->imagesCount - 802355392)
real_s_ImagesCount = s_ImagesCount;
v44 = s_ImagesCount;
s_ImagesTable = il2cpp::utils::Memory::Calloc(qword_7FFE8B96E1D0, s_ImagesCount, 96i64);
if ( s_ImagesCount << 32 )
{
  v46 = s_ImagesTable;
  do
  {
    v46[1] = 0i64;
    v46[2] = 0i64;
    v46[3] = 0i64;
    v46[4] = 0i64;
    v46[5] = 0i64;
    *v46 = xmmword_7FFE8A742BE0;
    *(v46 + 2) = 2008088359908919802i64;
    *(v46 + 6) = 1123462149;
    *(v46 + 4) = 873729663504399224 3i64;
    *(v46 + 5) = 664265444953316231 5i64;
    *(v46 + 6) = 570542401629937238 9i64;
    *(v46 + 14) = 1612286113;
    *(v46 + 8) = 1377039419313529108i64;
    *(v46 + 18) = 749164775;
    *(v46 + 10) = 6282057078567468209i64;
    *(v46 + 22) = 1806965411;
    v46 += 6;
    --v44;
  }
  while ( v44 );
}
real_s_ImagesTable = s_ImagesTable;
```

(a) ImagesTable population in the main binary.

```
s_ImagesCount = s_GlobalMetadataHeader->imagesCount / sizeof(Il2CppImageDefinition);
s_ImagesTable = (Il2CppImage*)IL2CPP_CALLOC(s_ImagesCount, sizeof(Il2CppImage));

// setup all the Il2CppImages. There are not many and it avoid locks later on
const Il2CppImageDefinition* imagesDefinitions = (const Il2CppImageDefinition*)((const char*)
for (int32_t imageIndex = 0; imageIndex < s_ImagesCount; imageIndex++)
{
    const Il2CppImageDefinition* imageDefinition = imagesDefinitions + imageIndex;
    Il2CppImage* image = s_ImagesTable + imageIndex;
    image->name = GetStringFromIndex(imageDefinition->nameIndex);

    std::string nameNoExt = utils::PathUtils::PathNoExtension(image->name);
    image->nameNoExt = (char*)IL2CPP_CALLOC(nameNoExt.size() + 1, sizeof(char));
    strcpy(const_cast<char*>(image->nameNoExt), nameNoExt.c_str());

    image->assemblyIndex = imageDefinition->assemblyIndex;
    image->typeStart = imageDefinition->typeStart;
    image->typeCount = imageDefinition->typeCount;
    image->exportedTypeStart = imageDefinition->exportedTypeStart;
    image->exportedTypeCount = imageDefinition->exportedTypeCount;
    image->entryPointIndex = imageDefinition->entryPointIndex;
    image->token = imageDefinition->token;
}
```

(b) ImagesTable population in source code.

Figure 3.12: ImagesTable populations in source code and a misleading one in the main binary.

Looking at the decompiled code from the reference project (Listing 3.2) next to the original IL2CPP source code (Figure 3.12b), it's clear they are functionally the same. Although the loop in the reference project's output appears highly convoluted, its core logic mirrors that of the source code. For instance, both versions assign a name to each image definition: the reference project's code does so by directly accessing metadata (using `imagesOffset` field from the header), while the source code employs a higher-level abstraction, `GetStringFromIndex`. Similarly, the variable `v3` in the reference project, which corresponds to the `imagesDefinitions` pointer from the source code, is also incremented within its loop. The direct addition of 8 to `v3` in the reference project mirrors the `imageIndex++` iterator increment in the source code's loop.

However the situation is vastly different in how this snippet looks in the main binary (Fig 3.12a). Notably, the `imagesDefinitions` variable isn't created, as there's no access to an `imagesOffset` from the metadata header as seen in the references. Instead, the `s_ImagesTable` global variable receives hardcoded values for each of its members. This behavior seems atypical for genuine data population and likely is a confusion tactic against reverse engineering efforts. This shows that behavior can deviate significantly from standard patterns due to purposeful obfuscation.

To find the actual image data assignment, it was necessary to trace other occurrences of `s_ImagesTable`. Although direct cross-references were scarce (only three, including its initial assignment), further investigation revealed that another global variable was assigned the value ofm `s_ImagesTable`. Tracing this second global variable, in turn, led to the relevant code sections where the true data assignment occurs. This path allowed for the identification and assignment of the correct header field, `imagesOffset`, because a structurally similar code snippet was found — one that accesses a header field and then uses it in calculations involving the actual images table. As shown in Listing 3.3, the `stringOffset` header field is also accessed in this context. The variable `v116` (representing the true images table, `real_s_ImagesTable`) has its individual images accessed by adding multiples of 96 (the size of the `IL2CPPImage` struct, stored in `v118`) to `v116`, with fields subsequently accessed via offsets.

Furthermore, the values assigned to these image table fields are also XORed, implying that the image definitions themselves are encrypted within the metadata file. Another key takeaway from this deeper analysis is the first clear use of the `startup-metadata.dat` file; it appears to hold some of the data structures typically found in the main metadata file. Since `startup-metadata.dat` lacks its own header, and the global metadata header is used for extracting its contents, this implies that the header in `global-metadata.dat` file stores offset tables for both itself and the `startup-metadata.dat` file.

To validate these conclusions, the value in the `global-metadata.dat` header at offset 0x194 was extraced and XORed with the constant `0x162168BDi64`, resulting in the number `0x37BBC`. To confirm if data at this resulting offset within `startup-metadata.dat` was encrypted, the file was opened and its contents at this address were inspected. Indeed, as shown in Figure 3.12 (presumably a hex view or similar), the data at this location appeared to be encrypted, and the encrypted parts started exactly at this offset, which confirmed the hypothesis.

```
1  s_ImagesCount = s_GlobalMetadataHeader->imagesCount >> 5;
2  s_ImagesTable = IL2CPP::utils::Memory::Calloc(s_ImagesCount,
       0x40ui64);
3  ...
4  v1 = s_GlobalMetadataHeader;
5  v2 = s_GlobalMetadata;
6  v3 = (s_GlobalMetadata + s_GlobalMetadataHeader->imagesOffset);
7  v4 = 0;
8  v5 = 0;
9  v6 = s_ImagesCount;
10 if ( s_ImagesCount > 0 )
11 {
12   v7 = 0i64;
13   do
14   {
15     v8 = v7 + s_ImagesTable;
16     v9 = v2 + v1->stringOffset + *v3;
17     *(v7 + s_ImagesTable) = v9;
18       [...]
19     *(v8 + 16) = v16;
20     *(v8 + 24) = v3[2];
21     *(v8 + 28) = v3[3];
22     *(v8 + 32) = v3[4];
23     *(v8 + 36) = v3[5];
24     *(v8 + 40) = v3[6];
25     *(v8 + 56) = v3[7];
26     *(v8 + 60) = 0;
27     if ( v50 >= 0x10 )
28     [...]
29     ++v5;
30     v3 += 8;
31     v7 += 64i64;
32     [...]
33   }
34   while ( v5 < s_ImagesCount );
35 }
```

Listing 3.2: ImagesTable population snippet from the decompiled empty project.

```
1  if ( real_s_ImagesCount > 0 )
2    {
3      v113 = (char *)startupMetadataFile +
           (globalMetadataHeaderReal->unknown194 ^ 0x162168BDi64);
4      v114 = 0i64;
5      v759 = (__int64)startupMetadataFile +
           (globalMetadataHeaderReal->unknown194 ^ 0x162168BDi64);
6      do
7      {
8        v115 = ((629846663 * (43086 * v114 ^ 0x343BDEE8) + 716843428) ^
             0x33FC08D4) + 1562760747;
9        v116 = real_s_ImagesTable;
10       v117 = (signed __int64)GetStringFromIndex(v115 ^ *(_DWORD
             *)&v113[40 * v114 + 16] ^ 0x5C34F96u);
11       v118 = 96 * v114;
12       [...]
13       *(_QWORD *)(v116 + v118 + 80) = v134;
14       *(_DWORD *)(v116 + v118 + 24) = (v115 ^ *((_DWORD *)v132 + 8) ^
             0xE9E10CD) + 1123462149;
15       *(_DWORD *)(v116 + v118) = (v115 ^ *((_DWORD *)v132 + 7) ^
             0x4FE26013) + 2025947892;
16       *(_DWORD *)(v116 + v118 + 12) = (v115 ^ (*((_DWORD *)v132 + 2)
             - 1408302495)) + 1295256737;
17       *(_DWORD *)(v116 + v118 + 72) = v115 ^ (*(_DWORD *)v132 -
             338802593) ^ 0x2CA758E7;
18       [...]
```

Listing 3.3: Actual ImagesTable fields assigning code.

A similar pattern of misdirection and data relocation occurred with other structures (e.g., `s_AssembliesTable`) and their corresponding header fields. An analogous technique, involving tracing variable assignments and cross-references, was used to identify their true locations and values.

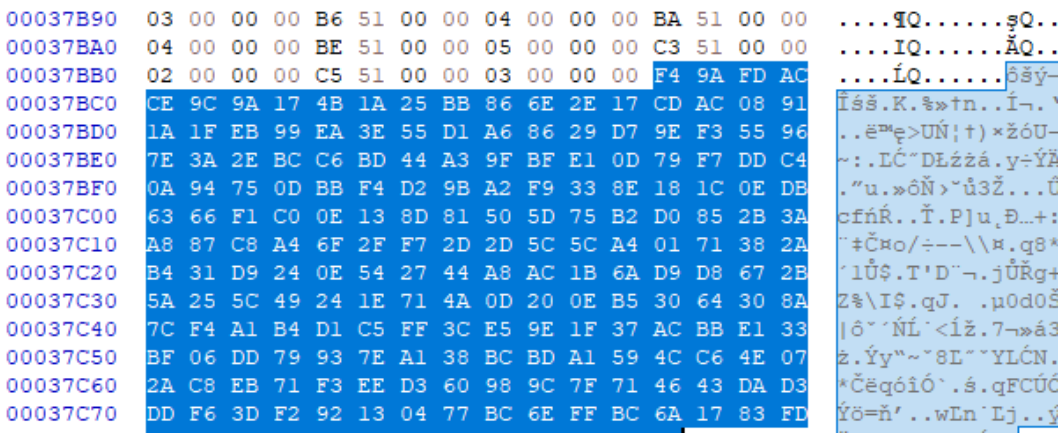


Figure 3.13: Encrypted contents of the `startup-metadata.dat` file, present as expected.

3.4.5 Code recovery

Ultimately, these techniques create a compounding effect: the more code that is analyzed, the easier subsequent analysis becomes. As more functions are identified, it is progressively simpler to correlate and understand others. Consistent renaming and retyping of identified functions and variables are therefore crucial practices that should be maintained throughout the analysis. By applying these methods, a substantial part of the header was successfully recovered: approximately 29 offset fields, most of which are significant (such as those related to methods), and 9 count fields.

Notably, the number of recovered count fields (9) is considerably lower than the number of offset fields (29), despite offset fields often having a corresponding count field in typical data structures. This discrepancy is likely because, in IL2CPP, such count fields are mainly used for runtime assertions rather than, for example, managing loop iterations. In the decompiled binary, these assertions were often absent — a finding that strongly suggests deliberate developer tampering, as it is unlikely a compiler would optimize away assertions based on values read from a file (and thus unknown at compile-time). The absence of these explicit count fields, however, does not necessarily prevent file reconstruction. Once the offsets of various data sections are known, the corresponding counts can often be inferred, as one data section begins where the previous one ends.

A substantial amount of functions and global variables have also been uncovered, with many functions performing critical tasks known, which will prove helpful in the following chapters.

Chapter 4

String decryption

Recovering the header and decrypting most of its fields can often be the end of analysis, as the metadata file is fully recovered. However in the case under the analysis, that's not the end of all as developers took great care into making the feat harder. In chapter 2 it was noted that the metadata file does not only store an encrypted header, but also parts of it are encrypted. As the structure of the file is now more clear, it all points to the string tables being encrypted. As in plain metadata files, there are usually chunks that store all kinds of strings

To achieve full recovery of the metadata file, decrypting these strings is crucial. Fortunately, with the code structure partially recovered and the functions operating on strings — along with the header field responsible for string table access already identified, locating the relevant decryption snippets is an easier task. It's important to note that there are two kinds of strings — normal strings and string literals. The first ones are identifiers and labels, for example class or method names. The literals are used in application code, for example error messages. Within the metadata, string literals themselves are managed using two distinct tables: `stringLiterals` and `stringLiteralsData`.

The `stringLiterals` table contains entries, each consisting of two 32-bit integers: the first is an offset into the `stringLiteralsData` table, and the second specifies the length of the string. Using these offset and length pairs, it's possible to read the actual string content from the `stringLiteralsData` table. The strings in this data table are stored one after another, without any alignment or null terminators, hence the need for the length information for correct retrieval.

The situation with normal strings is simpler, as they are typically accessed directly by an index. The table storing them is often structured such that each string begins at the location indicated by its index and is then null-terminated, making retrieval straightforward once the index is known. This chapter will mainly focus on string literals, with only a brief mention of normal strings, as the process of retrieving normal strings can be considered a simpler version of the methods required

for string literals.

4.1 Code overview

4.1.1 Finding relevant code

A function that accesses string literal data, as identified from the IL2CPP source code, is `GetStringLiteralFromIndex`. However, this function was not initially found or named in the decompiled target binary. To locate its equivalent, the source code was examined to determine which functions call `GetStringLiteralFromIndex`. This revealed that it is exclusively called by `MetadataCache::InitializeMethodMetadata()`. Since `MetadataCache::InitializeMethodMetadata()` had already been identified during the previous analysis, it's simple to navigate to its location.

As shown in Figure 4.1 (from the source code), `GetStringLiteralFromIndex` is invoked within a switch statement when the result of `GetEncodedIndexType(encodedSourceIndex)` corresponds to `kIL2CPPMetadataUsageStringLiteral`, which has a value of 5. The decompiled version of `MetadataCache::InitializeMethodMetadata()` in the target binary also contains a switch statement whose cases appear to directly correspond to those in the original source, albeit with some alterations to the control flow.

Examining the case for the value 5 in the decompiled code (Figure 4.2), only one function call is present: `sub_7FFE89B01120`. This function is suspected to either be `GetStringLiteralFromIndex` or `IL2CPP::vm::String::NewLen`, as the only the former is called within this case, and the latter is called within it, and there's a possibility of inlining.

```
void MetadataCache::InitializeMethodMetadata(uint32_t index)

    for (uint32_t i = 0; i < count; i++)
    {
        uint32_t offset = start + i;
        IL2CPP_ASSERT(s_GlobalMetadataHeader->metadataUsagePairsCount >= 0 && offset <= static_cast<uint32_t>(s_GlobalMetadataHeader->metadataUsagePairsCount));
        const Il2CppMetadataUsagePair* metadataUsagePairs = MetadataOffset<const Il2CppMetadataUsagePair*>(s_GlobalMetadata, s_GlobalMetadataHeader->metadataUsage
        uint32_t destinationIndex = metadataUsagePairs->destinationIndex;
        uint32_t encodedSourceIndex = metadataUsagePairs->encodedSourceIndex;

        Il2CppMetadataUsage usage = GetEncodedIndexType(encodedSourceIndex);
        uint32_t decodedIndex = GetDecodedMethodIndex(encodedSourceIndex);
        switch (usage)
        {
            case kIl2CppMetadataUsageTypeInfo:
                *s_Il2CppMetadataRegistration->metadataUsages[destinationIndex] = GetTypeInfoFromTypeIndex(decodedIndex);
                break;
            case kIl2CppMetadataUsageIl2CppType:
                *s_Il2CppMetadataRegistration->metadataUsages[destinationIndex] = const_cast<Il2CppType*>(GetIl2CppTypeFromIndex(decodedIndex));
                break;
            case kIl2CppMetadataUsageMethodDef:
            case kIl2CppMetadataUsageMethodRef:
                *s_Il2CppMetadataRegistration->metadataUsages[destinationIndex] = const_cast<MethodInfo*>(GetMethodInfoFromIndex(encodedSourceIndex));
                break;
            case kIl2CppMetadataUsageFieldInfo:
                *s_Il2CppMetadataRegistration->metadataUsages[destinationIndex] = GetFieldInfoFromIndex(decodedIndex);
                b enum Il2CppMetadataUsage::kIl2CppMetadataUsageStringLiteral = 5
            case kIl2CppMetadataUsageStringLiteral:
                *s_Il2CppMetadataRegistration->metadataUsages[destinationIndex] = GetStringLiteralFromIndex(decodedIndex);
                break;
            default:
                NOT_IMPLEMENTED(MetadataCache::InitializeMethodMetadata);
                break;
        }
    }
```

Figure 4.1: `MetadataCache::InitializeMethodMetadata` function in source code.

```
        case 5u:
          v16 = v14;
          if ( *(*s_StringLiteralTableMaybe + 8i64 * v14) )
            goto LABEL_5;
          v27 = v4 + v3->stringLiteralOffset - 366108145;
          v24 = sub_7FFE89B01120(
                  v15,
                  (-404766271 * (1609814976 * (59818i64 * v15 ^ 0x68A8FCAEui64) >> 23)
                 + (*(v27 + 4i64 * v15) ^ 0x57564700)
                 - 1984691239)
                + v4
                + (v3->stringLiteralDataOffset ^ 0x714D8F09i64),
                  -404766271 * (1609814976 * ((59818i64 * v15 + 59818) ^ 0x68A8FCAEui64) >> 23)
                - (-404766271 * (1609814976 * (59818i64 * v15 ^ 0x68A8FCAEui64) >> 23)
                 + (*(v27 + 4i64 * v15) ^ 0x57564700))
                + (*(v27 + 4i64 * v15 + 4) ^ 0x57564700));
          v25 = s_StringLiteralTableMaybe;
          goto LABEL_4;
        case 7u:
          v16 = v14;
          if ( !*(*(s_StringLiteralTableMaybe + 8) + 8i64 * v14) )
          {
            LOBYTE(v12) = 1;
            v28 = il2cpp::vm::MetadataCache::GetTypeInfoFromTypeIndex(v12, v15, v10, v8, v4, a2, a3);
            v24 = sub_7FFE89B01490(v14, v28);
            v25 = (s_StringLiteralTableMaybe + 8);
LABEL_4:
            *(*v25 + 8 * v16) = v24;
            v4 = globalMetadataNoHeader;
            v3 = globalMetadataHeaderReal;
          }
          goto LABEL_5;
        default:
```

Figure 4.2: `MetadataCache::InitializeMethodMetadata` function in the decompiled code.

4.1.2 Identifying variables and their purposes

To understand the decryption process, it is crucial to analyze the code's behavior, with variables being crucial. This analysis begins with the previously identified function of interest, `sub_7FFE89B01120`. While initially thought to be either `GetStringLiteralFromIndex` or `IL2CPP::vm::String::NewLen`, the arguments passed to `sub_7FFE89B01120` in the target binary differ significantly from those documented for `GetStringLiteralFromIndex` in its source code. In the source, the only argument passed to `GetStringLiteralFromIndex` is `decodedIndex`, which is the result of the `GetDecodedMethodIndex` function called with `metadataUsagePairs ->encodedSourceIndex`. In contrast, the decompiled function `sub_7FFE89B01120` is called with three arguments. All of these arguments appear to have undergone transformations (such as multiplication, XORing, and bitwise shifting), and the variables supplying these arguments have also been heavily modified in previous code sections. This is particularly evident for the variable `v15`, which is used in numerous places throughout the function, including as an argument in every function call within each case of the switch statement. For example, it is passed to a function in the sixth case as follows:

```
case 6u:
    v16 = v14;
    if ( *(_QWORD *)(*(_QWORD *)(s_StringLiteralTableMaybe + 72) +
        8i64 * v14) )
        goto LABEL_5;
    if ( v15 ) {
        [ ... ]
        v24 = (char *)
        MetadataCache::GetMethodInfoFromMethodDefinitionIndex
        (v15, v4, a2, a3);
    }
    else {
        v24 = 0i64;
    }
```

This pattern strongly suggests that `v15` likely holds values corresponding to both `encodedSourceIndex` and `decodedIndex` from the source code, depending on how it's utilized in different operations. The second variable used in the investigated `Calloc` call is `v27`. Upon initial inspection, `v27` appears to be a pointer to a particular string literal. This can be inferred by examining its member variables and their assignments, as illustrated below:

```
3 = globalMetadataHeaderReal;
v4 = globalMetadataNoHeader;
[...]
v27 = v4 + v3->stringLiteralOffset - 366108145;
```

The typical pattern of accessing a metadata field is evident here: first, the offset of the target table (`stringLiteralOffset`) is extracted from the header, and then this offset is added to the base address of the metadata file (`v4`) to determine the beginning of the table. The subtraction of a constant instead of the expected XOR operation for decryption is not too strange; this could be a compiler optimization or an alternative interpretation by the decompiler.

While the contents of `v27` are clear, the origin of `v15` requires further investigation. A closer look at the beginning of the large function (Figure 4.3) shows that variables used in its initial calculations are either global, derived from previously identified header fields, or are arguments to this encompassing function. The only significant input variable whose value is not immediately available from static globals or header fields is the index parameter with which the original `MetadataCache::InitializeMethodMetadata` (and by extension, the inlined `GetStringLiteralFromIndex`) is effectively called. As its name implies, this index is used in subsequent calculations for method metadata.

Fortunately, an examination of the calls to this function reveals that this index is always supplied as a hardcoded value, not a variable. To retrieve all such indices, a simple IDAPython plugin was developed, as the large number of cross-references

made manual extraction of these arguments unfeasible. The plugin's output is visible in IDA Pro's output window, and can be written to a file using Python. The two other variables passed as an input to this function, but not in its original source code are not critical to this specific analysis path, as they are used only in switch cases not currently under investigation.

4.2 Writing decryption code

4.2.1 Preparing inputs for the decryption function

The preceding findings suggest that there is a viable path to retrieve the string literals: by reimplementing the relevant function logic in an external script or program. This script would replicate the operations performed by the decompiled code up to the point of the end of case 5, then apply the logic of `sub_7FFE89B01120` (previously identified as `GetStringLiteralFromIndex`) to decrypt and retrieve the string. Several other methods were considered before opting for this solution, as rewriting potentially messy decompiled code can be error-prone and time-consuming. One alternative was to call the function directly from the DLL using its known offset. However, this approach would omit initial assignments of global variables related to metadata, rendering their pointers invalid. Another solution considered was attaching a debugger to a running instance of the game to call the function after all variables were initialized, but the previously mentioned anti-cheat and anti-debugging measures would probably prevent this. Thus, the approach of reimplementing the decryption logic, while harder, was chosen.

In the beginning of the decrypting code, files are mapped in the memory, and needed header fields are assigned to variables:

```
1    MemoryMappedFile mmf1(headerPath.c_str(), 0x0);
2    MemoryMappedFile mmf2(metadataPath.c_str(), 0x198); // Skip the
         fake header.
3    uint32_t valueMetadataUsageLists = read32(mmf1.getData(), 0x28) ^
         0x6F7A555B;
4    uint32_t valuestringLiteralData = read32(mmf1.getData(), 0xD8) ^
         0x714D8F09;
5    uint32_t valuestringLiteral = read32(mmf1.getData(), 0x30) -
         366108145;
6    uint32_t valuemetadataUsagePairs = read32(mmf1.getData(), 0x138)
         - 2106723582;
7    uint32_t index = 0x17A1u;
```

Reading the identified header fields is now straightforward. Since the fields and their offsets within the actual header structure (found embedded in the binary) are known, their values can be read directly. Furthermore, previous analysis of the decompiled code has revealed the XOR keys necessary for decrypting these field values. For the current test, a hardcoded value has been assigned to the `index`

variable.

Let's now analyze the operations occurring at the beginning of the `IL2CPP::vm::MetadataCache::InitializeMethodMetadata` function, before its main switch statement, as depicted in Figure 4.3. Initially, the variable `v6` is assigned a value:

```
1 v5 = globalMetadataNoHeader +
      (globalMetadataHeaderReal->metadataUsageListsOffset ^
      0x6F7A555Bi64);
2 v6 = *(_DWORD *)(v5 + 4i64 * a1);
```

```
v3 = globalMetadataHeaderReal;
v4 = globalMetadataNoHeader;
v5 = globalMetadataNoHeader + (globalMetadataHeaderReal->metadataUsageListsOffset ^ 0x6F7A555Bi64);
v6 = *(_DWORD *)(v5 + 4i64 * a1);
v7 = (1014436965 * ((16032818195816i64 * (unsigned __int64)a1 + 13507582996138228 3i64) >> 8) >> 19) ^ 0x990884EE;
v8 = *(_DWORD *)(v5 + 4i64 * (a1 + 1))
   + ((unsigned int)(1014436965 * ((16032818195816i64 * (unsigned __int64)(a1 + 1) + 13507582996138228 3i64) >> 8) >> 19) ^ 0x990884EE)
   - (v6
   + v7);
if ( *(_DWORD *)(v5 + 4i64 * (a1 + 1))
   + ((unsigned int)(1014436965 * ((16032818195816i64 * (unsigned __int64)(a1 + 1) + 13507582996138228 3i64) >> 8) >> 19) ^ 0x990884EE) != v6 + v7 )
{
  v10 = (unsigned int)(v6 + v7 - 851580944);
  while ( 2 )
  {
    v11 = v4 + v3->metadataUsagePairsOffset - 2106723582;
    v12 = 742369971 * ((956247808 * (57417i64 * (unsigned int)v10 ^ 0x72878364ui64) >> 13) ^ 0x6BDB5ACC) >> 15;
    v13 = v12 ^ *(_DWORD *)(v11 + 8i64 * (unsigned int)v10) ^ 0x207BBCFC;
    v14 = v12 ^ (*(_DWORD *)(v11 + 8i64 * (unsigned int)v10 + 4) - 1043264072);
    v15 = v13 & 0x1FFFFFFF;
    switch ( v13 >> 29 )
```

Figure 4.3: Operations in the beginning of the `IL2CPP::vm::MetadataCache::InitializeMethodMetadata` function.

From prior breakdown, it's obvious that `v5` points to the beginning of the metadataUsageLists table within the metadata file; in the preceding line, `v5` was assigned the base address of the metadata (`globalMetadataNoHeader`) plus the table's offset retrieved from the header. The assignment to `v6` therefore dereferences this table pointer (`v5`) after incrementing it by 4 * a1 (where `a1` is the `index`). This effectively retrieves the i-th (i corresponding to `index`) DWORD value from the `metadataUsageLists` table. This operation is notably similar to the source code line assigning the `metadataUsagePairs` variable (previously shown in Figure 4.1), which uses the MetadataOffset() function. The `MetadataOffset()` function takes the metadata pointer, `s_GlobalMetadataHeader->metadataUsageListsOffset`, and the `index` as arguments, performing an equivalent calculation to that observed in the `v6` assignment.

A very similar pattern occurs with the variable `v8`. A key difference is that the `index` (passed as `a1`) is incremented by one before being multiplied by 4 and used in the offset calculation. While additional intricate operations involve `v8`, the current focus is on this indexed access. The incrementation of the `index` by 1 for `v8` suggests access to a subsequent element or a related field within a structure.

To gain more insight into these operations, it's helpful to examine the `IL2CPPMetadataUsageList` structure:

```
struct IL2CPPMetadataUsageList
{
    uint32_t start;
    uint32_t count;
};
```

It is very simple — it has two fields of size 4 bytes. In the source code two variables are assigned both `start` and `count`, and named accordingly.

```
    uint32_t start = metadataUsageLists->start;
    uint32_t count = metadataUsageLists->count;
```

So the first is a value at the beginning of the metadataUsageLists, and the second one is on the offset of 4 bytes. And this is exactly what is happening in the decompilation with variables `v6` and `v8`. The first one — `v6` is assigned metadataUsageLists, as it just takes the index of the pair without any offset, and `v8` is metadataUsageLists incremented by one field, as there is +1 added to the index and then multipled by 4, that gives an offset of 4. Which makes the adding of 1 actually produce a displacement of 4 bytes. Therefore they are corresponding to `start` and `count`. One difference between source code and decompilation is the intricate calculations. Between v6 and v8 there is a variable v7, which just seems like an arbitrary value calculated using the index. It has no reference to any pointers, just pure arithmetic. Now the following code, mirroring the behavior can be written:

```
uint32_t index = 0x17A1u;
int32_t v6 = read32(mmf2.getData(), valueMetadataUsageLists +
    (int64_t)4 * index); // start
uint64_t v7 = (1014436965 * (((int64_t)16032818195816 * index +
    (int64_t)135075829961382283) >> 8) >> 19) ^ 0x990884EE;
int64_t v8 = read32(mmf2.getData(), valueMetadataUsageLists +
    (int64_t)4 * (index + 1)) // count
              + ((uint32_t)(1014436965 * (((int64_t)16032818195816 *
                  (uint64_t)(index + 1) + (int64_t)135075829961382283)
                  >> 8) >> 19) ^ 0x990884EE) - (v6 + v7);
```

The variable v7 and operations on v8 have been simply rewritten, as understanding their purpose is unnecessary for the time being and there is a possibility that after the whole analysis it will be possible to tell their purpose. The next segment of code encountered is an if statement, which envelops the main loop and the switch statement. Within it, an assignment is made to the variable `v10`:

```

if ( *(_DWORD *)(v5 + 4i64 * (a1 + 1))
     + ((unsigned int)(1014436965 * ((16032818195816i64 * (unsigned
         __int64)(a1 + 1) + 13507582996138228 3i64) >> 8) >> 19) ^
         0x990884EE) != v6 + v7 )
  {
    v10 = (unsigned int)(v6 + v7 - 851580944);
```

Upon closer examination, the leftmost side of the if statement's condition mirrors the expression used to calculate `v8`, but without the subtraction of `(v6 + v7)`. When this term is subtracted from both sides of the if condition, the condition simplifies to `v8 != 0`. Thus, the if statement essentially checks if `v8` is non-zero.

The value of the `v10` variable can be calculated with the information currently available. However, its specific purpose is not immediately apparent from this initial assignment, so the analysis will now proceed to the main loop containing the switch statement:

```
1  while ( 2 )
2      {
3        v11 = v4 + v3->metadataUsagePairsOffset - 2106723582;
4        v12 = 742369971 * ((956247808 * (57417i64 * (unsigned int)v10 ^
             0x72878364ui64) >> 13) ^ 0x6BDB5ACC) >> 15;
5        v13 = v12 ^ *(_DWORD *)(v11 + 8i64 * (unsigned int)v10) ^
             0x207BBCFC;
6        v14 = v12 ^ (*(_DWORD *)(v11 + 8i64 * (unsigned int)v10 + 4) -
             1043264072);
7        v15 = v13 & 0x1FFFFFFF;
8        switch ( v13 >> 29 )
9        {
10
11       [switch statement body]
12
13       }
14     }
```

Since expressions similar to the one assigning `v11` were seen before, it can now be determined that `v11` holds a pointer to the `metadataUsagePairs` table in the metadata file. This pointer's value was calculated in earlier preparation steps. Next, `v12` is set to an arithmetic value calculated using `v10`.

It can be seen that `v12` is then used as an XOR key for other variables, suggesting it's part of a more complex obfuscation scheme. The first variable XORed with `v12` is `v13`. Besides this XOR operation, the calculation for `v13` is very similar to that of `v6` (discussed previously). Specifically, `v11` (the pointer to the start of the metadataUsagePairs table) is incremented by an offset equal to 8 multiplied by an index, `v10`.

Looking at the IL2CPP source code reveals a nearly identical pattern to what was seen with `metadataUsageLists`. This similarity stems from the way `metadataUsageLists` and `metadataUsagePairs` are assigned is structurally the same. Both the `IL2CPPMetadataUsageList` and `IL2CPPMetadataUsagePair` structures are similar:

```
1  struct IL2CPPMetadataUsagePair
2  {
3      uint32_t destinationIndex;
4      uint32_t encodedSourceIndex;
5  };
```

From this, it's straightforward to deduce that `v13` and `v14` in the decompiled code correspond to `destinationIndex` and `encodedSourceIndex` respectively, from the source code loop.

The next variable, `v15`, is simply the `encodedSourceIndex` (`v14`) ANDed with a constant. This operation directly corresponds to the calculation of `decodedIndex` in the source code. Similarly, the expression used in the switch statement in the decompiled code is equivalent to the `usage` variable from the source code. These conclusions are based on comparing the decompiled logic with the following source code segment:

```
1 IL2CPPMetadataUsage usage = GetEncodedIndexType(encodedSourceIndex);
2 uint32_t decodedIndex = GetDecodedMethodIndex(encodedSourceIndex);
3
4 static inline IL2CPPMetadataUsage
      GetEncodedIndexType(EncodedMethodIndex index){
5     return (IL2CPPMetadataUsage)((index & 0xE0000000) >> 29);
6 }
7 static inline uint32_t GetDecodedMethodIndex(EncodedMethodIndex
      index){
8     return index & 0x1FFFFFFFU;
9 }
```

One caveat here is that the AND operation, which precedes the shift in the source code, has been omitted in the decompiled output. This is likely an optimization performed by the compiler, which determined the AND operation to be redundant in this context. Furthermore, the functions in question are marked with the inline keyword in the source code, explaining why they do not appear as separate function calls in the decompiled binary but are instead integrated directly into the calling code.

New step is be going back to the interesting 5th case in the switch statement and analyzing what happens to each of the variables inside it, as all of the ones that make up variables inside that case are known.

```
case 5u:
          v16 = v14;
          if ( *(_QWORD *)(*(_QWORD *)s_StringLiteralTableMaybe +
              8i64 * v14) )
            goto LABEL_5;
          v27 = v4 + v3->stringLiteralOffset - 366108145;
          v24 = (char *)sub_7FFE89B01120(
                          v15,
                          (signed int)(-404766271 * (1609814976 *
                              (59818i64 * v15 ^ 0x68A8FCAEui64) >> 23)
                                  + (*(_DWORD *)(v27 + 4i64 * v15)
                                      ^ 0x57564700)
                                  - 1984691239)
                        + v4
                        + (v3->stringLiteralDataOffset ^
                            0x714D8F09i64),
                          -404766271 * (1609814976 * ((59818i64 * v15
                              + 59818) ^ 0x68A8FCAEui64) >> 23)
                        - (-404766271 * (1609814976 * (59818i64 * v15
                            ^ 0x68A8FCAEui64) >> 23)
                         + (*(_DWORD *)(v27 + 4i64 * v15) ^
                             0x57564700))
                        + (*(_DWORD *)(v27 + 4i64 * v15 + 4) ^
                            0x57564700));
          v25 = (_QWORD *)s_StringLiteralTableMaybe;
          goto LABEL_4;
          [...]
    LABEL_4:
            *(_QWORD *)(*v25 + 8 * v16) = v24;
            v4 = globalMetadataNoHeader;
            v3 = globalMetadataHeaderReal;
          }
        goto LABEL_5;
        default:
LABEL_5:
          v10 = (unsigned int)(v10 + 1);
          if ( --v8 )
            continue;
          return (*(&IL2CPP::utils::Memory::Free + 1))(0i64);
      } // End of the switch
    } // End of the while loop
```

Let's first examine the variable `v16`. It is assigned the value of `v14` and is subsequently used only in an assignment at LABEL_4 (a label within the decompiled code). Assuming `v25` represents the `StringLiteralTable`, the operation at LABEL_4 indicates that `v16` serves as an index into this table, and the value of `v24` is stored at this indexed location. Since `v24` holds the result of the function of interest (presumably the decrypted string literal), this operation effectively saves the retrieved string literal into the `StringLiteralTable`.

Following this, an if statement checks if a value has already been written to that specific location in the `StringLiteralTable`. If data is already present, `v10` (acting as a primary loop counter or index) is incremented, and the loop proceeds to the next iteration, provided that `count` (which is decremented) is not yet zero.

Next, the variable `v27` is calculated. This involves another access to the metadata file, this time targeting the `stringLiteralData` table (or a similar table for string content), an operation similar to what was performed during the initial data preparation phase. The next key step involves examining the mysterious function call. This call takes three arguments: `v15` and two other, more complex expressions. To determine the nature of these latter two arguments, a quick examination of the original `GetStringLiteralFromIndex` function in the source code is required:

```
1  IL2CPPString*
       MetadataCache::GetStringLiteralFromIndex(StringLiteralIndex index)
2  {
3      if (index == kStringLiteralIndexInvalid)
4          return NULL;
5
6      IL2CPP_ASSERT(index >= 0 && static_cast<uint32_t>(index) <
           s_GlobalMetadataHeader->stringLiteralCount /
           sizeof(IL2CPPStringLiteral) && "Invalid string literal index
           ");
7
8      if (s_StringLiteralTable[index])
9          return s_StringLiteralTable[index];
10
11     const IL2CPPStringLiteral* stringLiteral = (const
           IL2CPPStringLiteral*)((const char*)s_GlobalMetadata +
           s_GlobalMetadataHeader->stringLiteralOffset) + index;
12
13     s_StringLiteralTable[index] = String::NewLen((const
           char*)s_GlobalMetadata +
           s_GlobalMetadataHeader->stringLiteralDataOffset +
           stringLiteral->dataIndex, stringLiteral->length);
14
15     return s_StringLiteralTable[index];
16 }
```

This structure closely resembles the operations within the fifth case of the switch statement in the decompiled code. This similarity strongly suggests that `GetStringLiteralFromIndex` was indeed inlined, and consequently, the function `sub_7FFE89B01120` (which was initially thought to be `GetStringLiteralFromIndex`) is more likely the `IL2CPP::vm::String::NewLen` function. This revised identification is supported by the observation that `sub_7FFE89B01120` takes multiple arguments which appear analogous to those expected by `String::NewLen` (such as a character pointer and a length), rather than the single index argument of `GetStringLiteral FromIndex`.

Based on this re-evaluation, it can be deduced that the second argument to `sub_7FFE89B01120` is the string literal itself, and the third argument is its length. This particular identification proved valuable during the development of the decryption script. The recovered length parameter served as something akin to a sanity check to help verify the correctness of previous calculations, especially during initial testing when wrongly calculated lengths sometimes resulted in extremely large, unreasonable values.

```
1     if (v8 == 0) {
2         cout << "v8 = 0, aborting";
3         return 1;
4     }
5     uint64_t v10 = (unsigned int)(v6 + v7 - 851580944); //
          metadataUsagePairs
6     while(1){
7
8         int64_t v12 = 742369971 * ((956247808 * ((int64_t)57417 *
              (unsigned int)v10 ^ (uint64_t)0x72878364) >> 13) ^
              0x6BDB5ACC) >> 15;
9         uint32_t v13 = v12 ^ read32(mmf2.getData(),
              valuemetadataUsagePairs + 8 * (unsigned int)v10) ^
              0x207BBCFC;       // destinationIndex
10        uint32_t v14 = v12 ^ read32(mmf2.getData(),
              valuemetadataUsagePairs + 8 * (unsigned int)v10 + 4) -
              1043264072; // encodedSourceIndex
11        uint32_t v15 = v13 & 0x1FFFFFFF;
12
13        if((v13 >> 29) == 5){
14            uint32_t stringLiteralLength = read32(mmf2.getData(),
                  valuestringLiteral + 4 * v15) ^ 0x57564700;
15            uint32_t stringLiteraldataIndex = read32(mmf2.getData(),
                  valuestringLiteral + 4 * v15 + 4) ^ 0x57564700;
16            char * str = (signed int)(-404766271 * (1609814976 *
                  ((int64_t)59818 * v15 ^ (uint64_t)0x68A8FCAE) >> 23)
17                + stringLiteralLength - 1984691239)
18                + (char *)mmf2.getData() + valuestringLiteralData;
19
20            uint32_t length = -404766271 * (1609814976 * ((59818 *
                  v15 + 59818) ^ 0x68A8FCAEui64) >> 23)
21                - (-404766271 * (1609814976 * ((int64_t)59818 * v15 ^
                      (uint64_t)0x68A8FCAE) >> 23)
22                    + stringLiteralLength)
23                + stringLiteraldataIndex;
24
25    }
```

One remaining key function is String::NewLen. Upon inspecting its decompiled code, a very complex and intricate structure presents itself:

```
[...]
      v12 = v11 & 0xFFFFFFFFFFFFFFFEui64;
      v13 = 0i64;
      v14 = _mm_load_si128((const __m128i *)&xmmword_7FFE8A742C30);
      v15 = _mm_load_si128((const __m128i *)&xmmword_7FFE8A742C40);
      v16 = _mm_load_si128((const __m128i *)&xmmword_7FFE8A742C50);
      v17 = _mm_load_si128((const __m128i *)&xmmword_7FFE8A742C60);
      do
      {
        v18 = _mm_xor_si128(_mm_loadu_si128((const __m128i *)(a2 +
            v13 + 16)), _mm_add_epi64(v10, v14));
        _mm_store_si128(
          (__m128i *)((char *)&v25 + v13),
          _mm_xor_si128(_mm_loadu_si128((const __m128i *)(a2 + v13)),
              v10));
        _mm_store_si128((__m128i *)((char *)&v25 + v13 + 16), v18);
        v19 = _mm_xor_si128(_mm_loadu_si128((const __m128i *)(a2 +
            v13 + 48)), _mm_add_epi64(v10, v16));
        _mm_store_si128(
          (__m128i *)((char *)&v25 + v13 + 32),
          _mm_xor_si128(_mm_loadu_si128((const __m128i *)(a2 + v13 +
              32)), _mm_add_epi64(v10, v15)));
        _mm_store_si128((__m128i *)((char *)&v25 + v13 + 48), v19);
        v10 = _mm_add_epi64(v10, v17);
        v13 += 64i64;
        v12 -= 2i64;
      }
      while ( v12 );
    [...]
```

The preceding code snippet represents only a portion of the decryption loop, and its complexity already suggests a challenging reverse engineering task. The functions with si128 suffixes (e.g., `_mm_load_si128`, `_mm_xor_si128`) operate on 128-bit data types and are compiler intrinsics -—- special functions provided by the compiler that often map directly to assembly instructions[9][10]. The prevalence of these intrinsics contributes to the difficulty of manual analysis, potentially making it even more time-consuming than the previous analysis..

Given this complexity, it's worth to look at previously mentioned methods for obtaining the function's results in a "black-box" manner, avoiding a full reversal of its internal logic. Previously, direct calling of functions from the DLL was dismissed due to uninitialized global variables. However, if the target function, doesn't rely on global variables, direct invocation becomes a more viable strategy, which is the case here. Therefore, instead of reimplementing its logic, it can be called directly.

4.2.2 DLL Function Invocation

In DLL files without any symbols it's near impossible to call a desired function without prior knowledge of its memory and function layout. In this case, thanks to the static analysis the exact offset of the function `sub_7FFE89B01120` is known — which as the name points is at `0x7FFE89B01120`. However when the DLL is loaded into the memory, its address is different, therefore it's needed to calculate an offset of the function into the DLL, which is relative to the address the binary was rebased at. One last piece needed to call the function is to match the type it returns, which is a pointer to `IL2CPPString`. In the source code definition the structure and dependent ones look like following:

```
1  struct IL2CPPString
2  {
3      IL2CPPObject object;
4      int32_t length;      ///< Length of string
5                           excluding the trailing null
6                           (which is included in 'chars').
7      IL2CPPChar chars[IL2CPP_ZERO_LEN_ARRAY];
8  };
9
10 struct IL2CPPObject
11 {
12     IL2CPPClass *klass;
13     MonitorData *monitor;
14 };
```

It might be troublesome to import other definitions, such as the members of `IL2CPPObject` as they contain other nested definitions. An easy way to circumvent that is using void pointers instead of those structures, as for this purpose it will be the same, since their size is also 8 bytes, and accessing the object field will not be needed.

The relevant portion of the source code, which handles the task of initiating the call to the function within the DLL can be seen in the listing 4.1.

```
1
2  HMODULE hModule =
       LoadLibraryW(L"C:\\Windows\\System32\\UserAssembly.dll");
3  HMODULE baseAddress = GetModuleHandleW(L"UserAssembly.dll");
4  const uintptr_t offset = 0x631120;
5  void *funcAddr = reinterpret_cast<void *>((uintptr_t)hModule +
       offset);
6  uintptr_t functionAddress = reinterpret_cast<uintptr_t>(baseAddress)
       + offset;
7  StringDecryptFunc decryptFunc =
       reinterpret_cast<StringDecryptFunc>(functionAddress);
8
9  cout << "length: " << length << "\n";
10 char * result = (char *)decryptFunc(v15, str, length);
11 if (result != nullptr)
12 {
13
14     IL2CPPString* IL2CPPStr = reinterpret_cast<IL2CPPString*>(result);
15     wchar_t* wideStr = reinterpret_cast<wchar_t*>(IL2CPPStr->chars);
16     wstring_convert<std::codecvt_utf8_utf16<wchar_t>> converter;
17     wstring wstr(wideStr, IL2CPPStr->length);
18     string utf8str = converter.to_bytes(wstr);
19
20     std::cout << "UTF-8 Output: " << utf8str << std::endl;
21
22 }
23 else
24 {
25     cout << "Decryption failed." << endl;
26 }
```

Listing 4.1: Finished code handling calling the DLL function.

With the whole code finished, the program can now be executed with the previously chosen example index:

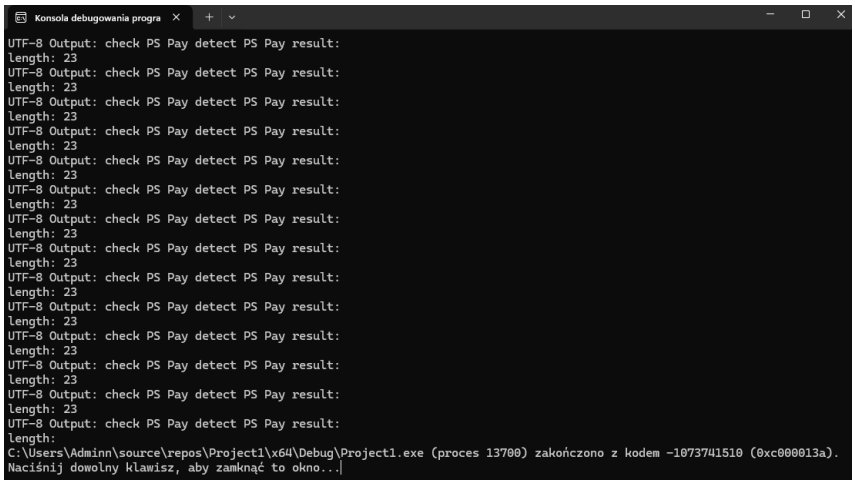


Figure 4.4: Console output for the decryption code.

As seen a fully deobfuscated string presents itself, one that seems like a debug message. Now that it's known what process occurs with string literals it's worth to look at strings.

4.2.3 Strings recovery

While the process for string literals was quite convoluted, retrieving the strings is generally simpler. This process will be described briefly, as the main decryption method is quite similar to that of string literals, though the access method differs.

A function used to access these strings is `GetStringFromIndex`. Inside this function, a decryption process occurs that is similar to the one seen for string literals, as it also uses intrinsics. However, in this case, `GetStringFromIndex` typically accepts only one argument: the string's index. This direct indexing potentially allows the function to be called without the multi-step (like offset and length lookups) that was required for string literals.

However with that, a new issue arises: the indices for these normal strings may not be hardcoded, unlike some indices encountered for string literals or in other examples. To understand how to work around that, it is worth to examine an example of how `GetStringFromIndex` is invoked in the source code:

```
1 void AssemblyName::FillNativeAssemblyName(const IL2CPPAssemblyName&
    aname, IL2CPPMonoAssemblyName* nativeName)
2     {
3         nativeName->name =
              StringUtils::StringDuplicate(IL2CPP::vm::MetadataCache::
4         GetStringFromIndex(aname.nameIndex));
5
6         [...]
7     }
```

Similar references to indexed strings appear when examining the application's code and data structures. For instance, structs stored in the metadata file often contain an index to a string, rather than embedding the string itself. Then, when these structs are populated (during runtime), a function like `GetStringFromIndex` would be called to retrieve the string corresponding to that index.

There are a few potential solutions for retrieving these strings. First, since the starting location of the string table is often known, it is possible to begin decrypting or reading the strings sequentially from that point. As the strings are null-terminated, the end of one string marks the beginning of the next. This process can be repeated until the entire table is processed. Another method involves using existing software designed for IL2CPP projects; such tools will be described in the next chapter.

Chapter 5

Recovering main binary's information

With the methods described in this research it's now possible to recover information using the metadata file, such as names of classes, methods, fields, parameters, enums, and other code elements. It can be automatically done with tools like IL2CPPDumper or IL2CPPInspector. The possible solutions for both will be described.

5.1 IL2CPPDumper

IL2CPPDumper[11] is an open-source tool developed on GitHub. It recovers the structure of an application's main DLL, though not the original C# code itself — a feature no publicly available tool had accomplished at the time of this writing (although one is currently in development [12]). The program also generates scripts to import symbols into popular reverse engineering tools like Ghidra, IDA, and Binary Ninja. Additionally, it can bypass simple metadata obfuscations, generate a file containing string literals, and perform other useful tasks. It also has the ability to recover structures and their fields which are stored in a header file, create fake (or "dummy") DLL files that represent the application's original managed code assemblies and many other features that make it easier to comprehend the application's structure.

However, the obfuscation encountered in this study was by no means simple. Extensive metadata reordering, the splitting of metadata into two files, and the presence of decoy code snippets can make automation difficult, even with smart heuristics. Consequently, to effectively use IL2CPPDumper with the target binary, it is necessary to construct an entirely new, deobfuscated metadata file. This new file needs to adhere to original formatting conventions, including an ordered header and decrypted string literals. While this reconstruction process can be tedious and

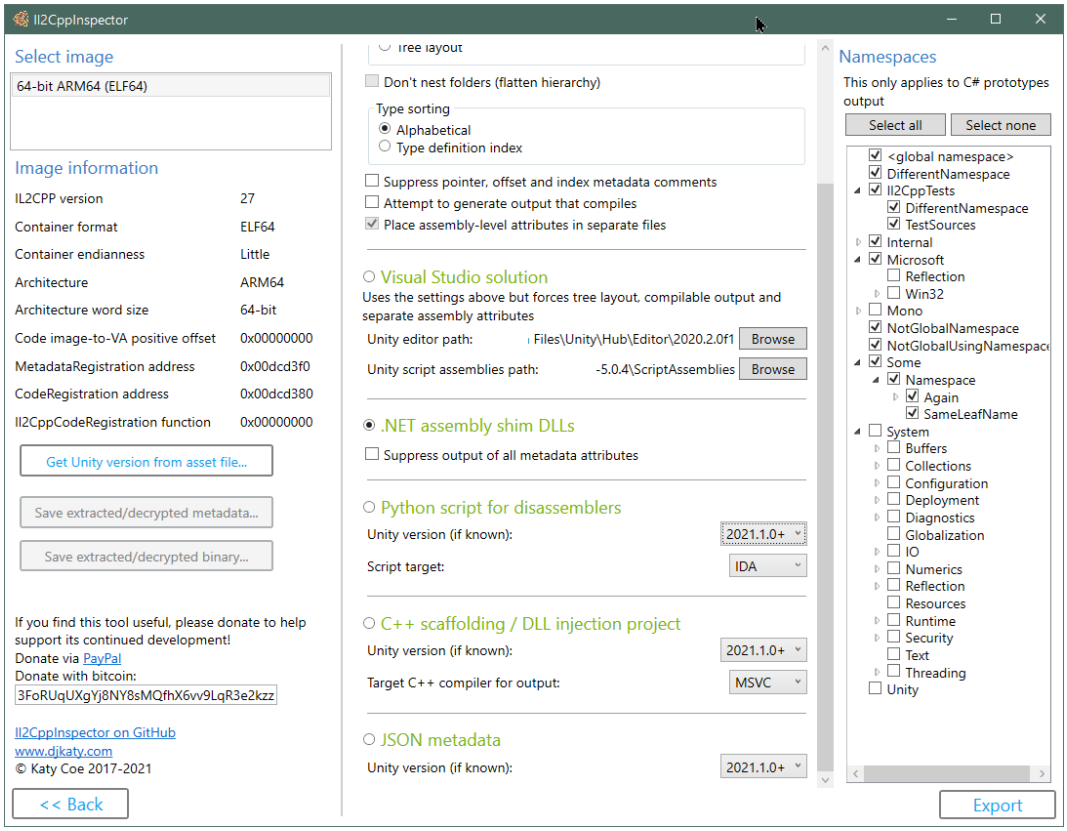


Figure 5.1: IL2CPPInspector user interface.

requires significant manual effort prone to errors, it is a feasible solution.

5.2 IL2CPPInspector

Another tool that can prove helpful in reverse engineering the main binary is IL2CPP-Inspector[13]. This tool, although perhaps less popular, offers functionality similar to IL2CPPDumper and even includes a graphical user interface (GUI), as shown in Figure 5.1.

A key differentiator from IL2CPPDumper is IL2CPPInspector's support for plugin creation. The developer designed this tool with community-shared plugins in mind, facilitating their easy development. These plugins can intervene in the binary and metadata loading process — for instance, to handle obfuscated or reordered files. The plugins are highly customizable: within them, one can correct file reordering by pointing to the correct data structures and even call functions from the main binary to decrypt files, a feature that would be extremely useful in this case. String retrieval is also supported. A plugin can scan metadata definitions that reference a string index and then call `GetStringFromIndex` to retrieve the decrypted string.

However, this approach would still require modifications to the main metadata file due to the presence of the `startup-metadata.dat` file. This file could either be manually merged into the primary `global-metadata.dat` file, or the IL2CPP-Inspector loading process itself could be further modified to accommodate it, potentially simplifying its use for other users that would want to analyze the game. What's more, a plugin written by IL2CPPInspector's author for another game from the same publisher is available [14]. However, it differs significantly from the current case, as it was developed four years ago when the obfuscation methods employed were simpler and no additional metadata files were present.

Chapter 6

Summary and conclusion

This work looked closely at how to analyze IL2CPP applications that are heavily obfuscated, making them difficult to understand. Specific techniques were introduced and tools developed to get past these complex defenses and carefully extract key information from their protected metadata and program code. By concentrating on these core tasks — getting crucial data and uncovering important code sections.

The techniques detailed were successful in retrieving this essential information, effectively bypassing deliberate efforts by developers to hinder analysis. Additionaly, combining these successful techniques with proposed tools allows restoring the original .NET environment's structure. This, in turn, makes it much easier to proceed with further analysis and ultimately helps gaining a deeper understanding of how specific IL2CPP applications function.

Bibliography

[1] Unity Technologies. *Scripting Backends: IL2CPP Overviews*. Unity Documentation, 2025. Accessed June 15, 2025. Available at: `https://docs.unity3d.com/6000.1/Documentation/Manual/scripting-backends.html`

[2] Unity Technologies. *IL2CPP - How it works (Version 5.6)*. Docs Unity3D. Accessed May 30, 2025. Available at: `https://docs.unity3d.com/560/Documentation/Manual/IL2CPP-HowItWorks.html`

[3] Katy's Code. *IL2CPP Reverse Engineering Part 2: Structural Overview & Finding the Metadata*. Accessed May 31, 2025. Katy's Code Blog, 2020. Available at: `https://katyscode.wordpress.com/2020/12/27/IL2CPP-part-2/`

[4] Microsoft. *Reflection and Attributes (C#)*. Microsoft Learn, 2023. Accessed May 31, 2025. Available at: `https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/`

[5] Wikipedia. *Program database*. Microsoft Github PDB repository, 2023. Accessed April 27, 2025. Available at: `https://github.com/Microsoft/microsoft-pdb`

[6] Microsoft. *Process Monitor*. Microsoft Docs, 2025. Accessed April 27, 2025. Available at: `https://learn.microsoft.com/en-us/sysinternals/downloads/procmon`

[7] xTaiwanPingLord. $DebuggerBypassDLL$. GenshinDebuggerBypass Github repository, 2023. Accessed April 26, 2025. Available at: `https://github.com/xTaiwanPingLord/GenshinDebuggerBypass`

[8] nneonneo. *IDA-friendly IL2CPP Headers from IL2CPPVersions Repository*. GitHub repository, 2025. Accessed April 26, 2025. Available at: `https://github.com/nneonneo/IL2CPPVersions/tree/master/headers`

[9] Intel Corporation. *Intel Intrinsics Guide*. Intel Developer Zone. Accessed May 29, 2025. Available at: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`

[10] Microsoft. *Compiler Intrinsics*. Microsoft Learn, 2025. Accessed May 29, 2025. Available at: `https://learn.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics`

[11] Perfare. *IL2CPPDumper*. GitHub. Accessed May 30, 2025. Available at: `https://github.com/Perfare/IL2CPPDumper`

[12] SamboyCoding. *Cpp2IL*. GitHub. Accessed May 30, 2025. Available at: `https://github.com/SamboyCoding/Cpp2IL`

[13] djkaty. *IL2CPPInspector*. GitHub. Accessed May 30, 2025. Available at: `https://github.com/djkaty/IL2CPPInspector`

[14] djkaty. *IL2CPPInspectorPlugins — miHoYo Loader*. GitHub. Accessed May 30, 2025. Available at: `https://github.com/djkaty/IL2CPPInspectorPlugins/blob/master/Loaders/miHoYo/Plugin.cs`

[15] Katy's Code. *IL2CPP Reverse Engineering Part 1: Hello World and the IL2CPP Toolchain*. Katy's Code Blog, 2020. Accessed May 31, 2025. Available at: `https://katyscode.wordpress.com/2020/06/24/IL2CPP-part-1/`

[16] Hex-Rays. *The IDAPython Book (v8.4)*. Hex-Rays Documentation, 2025. Accessed May 29, 2025. Available at: `https://python.docs.hex-rays.com/8.4/index.html`

[17] Hex-Rays. *Plugin focus: HrDevHelper*. Hex-Rays Blog, 2025. Accessed May 29, 2025. Available at: `https://hex-rays.com/blog/plugin-focus-hrdevhelper`

[18] Hex-Rays. *IDA SDK Developer Guide: IDAPython*. Hex-Rays Documentation, 2025. Accessed May 29, 2025. Available at: `https://docs.hex-rays.com/developer-guide/idapython`

[19] IL2CPPDumper.com. *Starting with a Clean Slate*. IL2CPPDumper.com. Accessed May 30, 2025. Available at: `https://IL2CPPDumper.com/reverse/starting-with-a-clean-slate`