Towards Ghidra Reverse Engineering Tools for Delphi Executables

Introducing the Dragon to the Greek

Master Thesis

by

Lukas Wenz

[Enrolment number redacted]

in fulfillment of requirements for degree

Master of Science (M.Sc.)

submitted to

Rheinische Friedrich-Wilhelms-Universität Bonn

MA-INF 0401 - Master Thesis

Institute for Computer Science IV

Work group for IT-Security

in degree course

Computer Science (M.Sc.)

First Supervisor: Dr. rer. nat. Jesko Hüttenhain

Crowdstrike GmbH

Second Supervisor: [Identity redacted until confirmation]

University of Bonn

Bonn, August 12, 2025

Abstract

Although Delphi's relevance in legitimate software development has steadily declined, the language continues to play a disproportionate role in the malware ecosystem. More than three decades after its release, Delphi still accounts for approximately 7.4% of all compiled malware globally. This is further amplified by the scarcity of publicly available knowledge about its file format and runtime behaviour, which has hindered the development of effective reverse engineering tools. To date, the only widely used tool for Delphi analysis is the Interactive Delphi Reconstructor, which is designed exclusively for use with IDA Pro and provides full support only for 32-bit binaries.

This thesis presents the first systematic reverse engineering of the Delphi file format in the context of static malware analysis. Based on this analysis, a Ghidra script has been developed that supports both 32-bit and 64-bit binaries compiled with Delphi 2010 through Delphi 12 Athens. The tool leverages metadata embedded by the Delphi compiler to reconstruct symbol name information. In tests on real-world Delphi malware samples, the tool successfully recovers fully qualified function names and return types for between 32% and 54% of functions and parameter names and types for between 27% and 48%. In addition to tooling, the underlying thesis compiles the relevant structural knowledge of the Delphi binary format into a series of compact reference tables, enabling quick lookup and informed analysis.

Contents

1 Introduction 1

2 Related Work 3

3 Static Analysis of Delphi Executables 6
- 3.1 Virtual Method Table . . . 12
- 3.2 Method Definition Table . . . 16
- 3.3 MethodEntryRef . . . 19
- 3.4 MethodEntry . . . 20
- 3.5 ParamEntry . . . 22
- 3.6 RTTI-Objects . . . 23
- 3.7 Format Changes . . . 29

4 Tool Development 32

5 Evaluation 35

6 Conclusion 41

Acronyms 44

References 46

1 Introduction

The field of cybersecurity is perpetually engaged in an arms race between malware authors and IT security specialists. On the one hand, increasingly sophisticated attacks highlight the ever-evolving strategies employed by malicious actors. For instance, cross-platform malware, aims to expand the pool of potential victims [Eck22; Lak24]. According to Mandiant, a U.S.-based IT security company under Google LLC, malware developers may be shifting their focus toward programming languages that facilitate cross-platform compatibility [Eck22]. Alarmingly, even game engines, valued for their inherent cross-platform capabilities, have recently been exploited for malware development [Lak24]. On the other hand, the emergence of large language models (LLMs) has dramatically lowered the barriers to entry for malware creation: Criminally oriented LLMs such as WormGPT, FraudGPT, or DarkBard empower individuals with minimal technical expertise to develop sophisticated malicious software [Poi23]. These tools make advanced coding techniques more accessible, intensifying the challenges in the cybersecurity landscape.

Despite the rapid advancement of technology and the novel challenges it presents, this relentless focus on emerging threats risks neglecting older, yet still significant, problems in the field. One such overlooked issue is the understanding and analysis of Delphi executables. Delphi, a dialect of Object Pascal, produces executable files primarily designed for the Windows operating system [Emb24c]. Originally introduced in 1995, Delphi has seen its usage decline in favour of more modern programming languages [Cas24; Jan24]. For example, as of 2024, the IEEE Spectrum ranks Pascal - the language family to which Delphi belongs - 31st out of 63 programming languages in scientific usage and 37th in employer demand [Cas24]. Similarly, the TIOBE Index, which measures the popularity of programming languages based on online search prevalence, indicates that Delphi/Object Pascal accounts for roughly 1.8% of global relevance, reflecting a loss of nearly 4.1% in the past two decades [Jan24]. These metrics underscore the waning influence of Delphi in legitimate software development.

However, despite Delphi's diminished role in legitimate IT interests, it remains a tool of choice for malware development [Rat24; Wil24]. Notably, as reported by the IT security company CrowdStrike Holdings Inc., the victims of Delphi-based malware are often Spanish- and Portuguese-speaking users of Latin American financial institutions [Rat24]. Globally, statistics from the automated malware unpacking service UnpacMe reveal that Delphi malware accounted for approximately 7.4% of all user-submitted files in 2023 [Wil24]. This persistence raises critical questions about why Delphi remains attractive to malicious actors.

Several factors contribute to the continued use of Delphi in malware creation. First, Delphi compiles to Windows Executables in the Portable Executable (PE) format, enabling native execution by the means of a simple double-click [Emb22]. Given Windows' continued dominance as the most prevalent desktop operating system, this ensures a broad reach for malware [Jos24]. Second,

Delphi's Integrated Development Environment (IDE), licensed by Embarcadero Technologies or freely available for private use, simplifies programming: Drawing from an era of drag-and-drop visual programming, the IDE facilitates code generation with minimal expertise [Emb24b]. Ready-to-use libraries for server communication further streamline the creation of Command-and-Control (C2) malware [Emb14b]. Third, Delphi employs a proprietary method for embedding data into the PE format, much of which remains undocumented over 30 years after its initial release.

The latter point is likely attributable to the aforementioned decreasing relevance of Delphi in legitimate applications, leading the IT security community to deprioritise research into the language. This neglect poses challenges for reverse engineers - IT security specialists tasked with deconstructing compiled code to extract valuable information from software without access to its source code. Reverse engineering is crucial for categorising new malware, identifying Indicators of Compromise (IoC), creating malware-specific signatures for antivirus software, and safeguarding the public through awareness and system updates. However, the lack of focus on Delphi is evident in the limited availability of tools tailored to reverse engineer Delphi binaries. The lack of symbols, such as function names, after decompilation forces reverse engineers to either invest an excessive amount of effort into reverse engineering Delphi binaries or deprioritize them in favour of binaries written in other languages. [Eck22]

This thesis seeks to address this gap by advancing the tools available for analysing Delphi executables. Specifically, it aims to understand the Delphi format, including its metadata storage mechanisms across different versions for both 32-bit and 64-bit architectures. Building on this understanding, the thesis develops reverse engineering knowledge, particularly around symbol recovery and object reconstruction. This theoretical foundation is then translated into practical tools through the creation of a Ghidra script that automates metadata extraction and facilitates symbol reconstruction, simplifying the work of reverse engineers. By making these improvements, the project aims to lower the barrier for analysing Delphi malware, enabling more efficient categorisation and mitigation efforts.

The rest of this thesis is organized as follows: Chapter 2 reviews existing research and tools relevant to the field, establishing the context and highlighting gaps that this work seeks to fill. Chapter 3 details the methodology employed in analysing and enhancing Delphi executable analysis, including the techniques used to extract metadata and reconstruct symbols. Chapter 5 evaluates the effectiveness of the developed tools through practical experiments and case studies. Finally, Chapter 6 summarises the findings, discusses their implications, and outlines potential directions for future research, emphasising the need for continued innovation in reverse engineering tools tailored to niche challenges.

2 Related Work

This chapter reviews the state of research and related work concerning the inner workings of the Delphi programming language, with a particular focus on available resources and reverse engineering tooling options. While numerous other programming languages have received extensive scrutiny in the context of reverse engineering, Delphi appears to have been largely overlooked [Eck22; XZZ12; j4k22]. Despite Delphi's introduction in 1995 and its continued use - notably even in various malware campaigns - there exists a striking lack of academic literature addressing the intricacies of its file structure [Rat24]. To date, the available resources are predominantly limited to programming paradigms and developer guidelines, leaving significant gaps in our theoretical understanding of how Delphi executables interact with the native Windows PE format and the manner in which metadata is stored within these files [Emb24a; Emb25a].

To the best of my knowledge, and after extensive research, no comprehensive or publicly available analysis exists that explains the internal structure of Delphi file formats, their structural components, how these files leverage the Windows PE format for executable construction, or the mechanisms by which they encode and manage metadata. This research is therefore intended to fill this gap by presenting the first detailed systematic insights into these aspects, thereby establishing a foundational framework for further advancements in the field of Delphi reverse engineering tooling.

The lack of theoretical understanding corresponds with the limited practical application of such insights, leading to a shortage of available tooling. Tools such as Binary Ninja and Ghidra are capable of handling the basic structure of these executables. However, these tools do not specifically target the unique aspects of Delphi's file format. As a result, advanced functionalities such as symbol recovery - wherein method names and instance references are re-established during decompilation - tend to be only partially effective or entirely absent [Eck22].

The primary and most well-known practical tool in this niche is the Interactive Delphi Reconstructor (IDR). IDR is an open-source tool published on GitHub, aimed specifically at reverse engineering Delphi executables. It incorporates several disassembly and decompilation features, yet no associated academic papers, theses, or formal publication reports have been identified. Initially released on 25 January 2016, with the most recent update on 9 August 2023, IDR remains the sole public research foundation for Delphi reverse engineering. [cry23]

However, IDR is subject to two significant limitations. Firstly, its full functionality is confined to the analysis of 32-bit Delphi executables [cry23]. While the developer has provided separate versions for 32-bit and 64-bit executables, the latter is explicitly noted as incomplete [cry18]. The project appears to have been discontinued for the 64-bit variant, with the last change recorded on 30 December 2018 [cry18]. This limitation significantly restricts the tool's applicability in modern reverse engineering scenarios where 64-bit executables are prevalent.

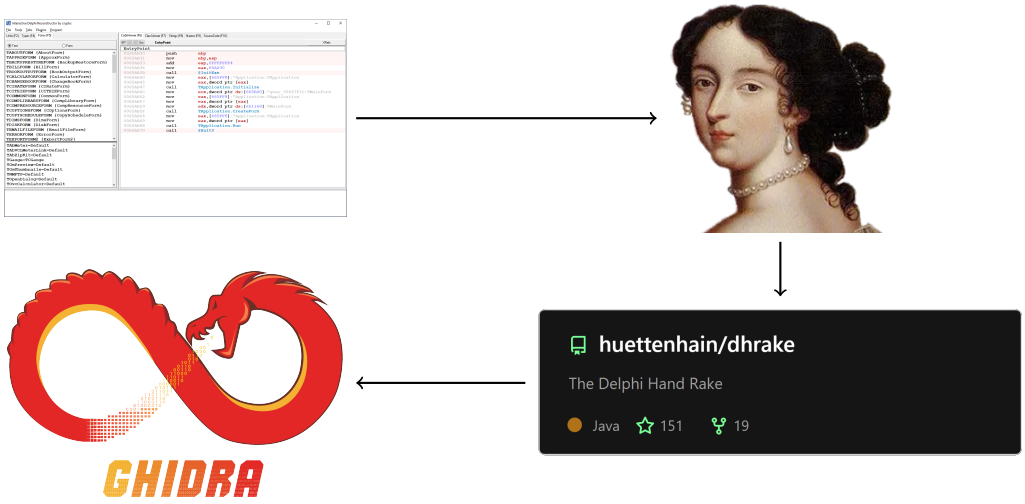

Figure 1: *Dataflow of related work programs for reversing Delphi executables.*

Secondly, the design decisions underpinning IDR dictate that the output data is tailored for use with IDA Pro [Hüt19]. While IDA Pro is widely regarded as the de facto standard tool for professional reverse engineers and malware analysts, its proprietary nature and associated costs can be prohibitive to the masses [Hex25b; Car18]. To address this issue, an alternative tool - a Ghidra script known as the Delphi Hand Rake (dhrake) - has been developed [Hüt23; Hüt19]. Dhrake converts the data generated by IDR into a format that is compatible with Ghidra, thereby enabling users of this open-source reverse engineering platform to benefit from the analyses provided by IDR [Hüt23; Hüt19].

Figure 1 illustrates the data flow between these established reverse engineering tools. When analysing 32-bit Delphi executables through this combined approach, basic symbol recovery is achieved, as evidenced by the extraction of method names and explicit instance references in the decompilation view. Figure 2 provides a comparative example for this. On the left, a typical decompiled entry point for a standard Delphi executable which is designed to interact with a user via visual components is displayed. On the right, the same entry point is shown after processing with IDR and dhrake. In this case, several method names and explicit instance references become visible, highlighting the improvements in symbol recovery. Nevertheless, it is evident that the process remains suboptimal: First, only roughly half the functions have been identified,

Beyond IDR, additional tools for Delphi reverse engineering have been identified through an extensive and systematic search of all GitHub repositories containing more than 100 stars and the keyword *Delphi* as well as repositories combining the keyword *Delphi* with keywords such as *ida*, *idapro*, *ida-pro*, *ghidra*, *binaryninja*, *binaryninja-plugin*, *reverse-engineering*, *static-analysis* and *software-analysis*. In mid-February, 173 repositories were analysed, and four notable tools were identified from this search.

The most recent addition to the Delphi reverse engineering toolkit is `DelphiHelper`. This tool is an IDA Pro plugin written in Python. Developed by the Slovak cybersecurity company ESET, `DelphiHelper` was published on 19 December 2024, with no changes since. It assists in the analysis of x86/x64 executables written in Delphi and provides support for various Delphi-specific features

```
void entry(void)

{
  FUN_00411798(&DAT_005e2cec);
  FUN_005d6be8(*(undefined4 *)
    PTR_DAT_005f2894);
  FUN_005d8bd4(*(undefined4 *)
    PTR_DAT_005f2894,1);
  FUN_005d6c00(*(undefined4 *)
    PTR_DAT_005f2894,&PTR_LAB_005e2af4,
    PTR_DAT_005f275c);
  FUN_005d6d60(*(undefined4 *)
    PTR_DAT_005f2894);
                    /* WARNING:
    Subroutine does not return */
  FUN_00409f8c();
}
```

```
void EntryPoint(void)

{
  FUN_00411798(&DAT_005e2cec);
  TApplication.Initialize(*(undefined4
    *)Application);
  FUN_005d8bd4(*(undefined4 *)
    Application,1);
  Vcl::Forms::TApplication::CreateForm(
    *(undefined4 *)Application,&
    PTR_LAB_005e2af4,gvar_005F275C);
  TApplication.Run(*(undefined4 *)
    Application);
                    /* WARNING:
    Subroutine does not return */
  @Halt0();
}
```

Figure 2: *Ghidra decompilation of a Delphi application's entry point. Left: Before IDR+dhrake; Right: After IDR+dhrake.*

such as parsing Virtual Method Tables (VMTs) or finding entry points of executables. However, as with IDR, there is no accompanying theoretical documentation, leaving all insights embedded within its open-source code. Notably, `DelphiHelper` uses IDR's knowledge base files, further indicating that IDR is the theoretical cornerstone of Delphi reverse engineering research. [ESE24]

The second tool is called `IDA-For-Delphi`. It is another IDA Pro Python script designed to extract all function names from the Delphi Event Constructor, enabling the identification of Visual Component Library (VCL) symbol names. It supports both 32-bit and 64-bit executables, with its initial release on 4 October 2017 and its last update on 27 November 2022. However, its symbol recovery method relies on dynamic analysis, requiring the executable to be run and unpacked at specific breakpoints to restore method names. This limitation contrasts with static analysis methodologies like IDR, `DelphiHelper` or the approach of this thesis, which do not require execution of the target binary. [Col22]

Furthermore, `pythia` is a Python script developed by NCC Group, which extracts portions of VMT information from Delphi executables. Initially released on 20 June 2017, it operates by searching for specific forward offset jumps (`+0x4C`) in code to localize potential VMT positions. Due to this heuristic approach, false positives can occur, and the script applies validity checks to filter them out. The author notes that it has only been tested on 32-bit executables and was intended as a complementary tool rather than a stand-alone solution. The last update was on 25 April 2019, with no further development on its aimed goals since. [NCC19]

Lastly, `delphi_ninja` is a Binary Ninja plugin written in Python, which facilitates Delphi reverse engineering by identifying VMTs and generating their corresponding structures. Similar to IDR, its author states that it only supports 32-bit executables. The initial release was on 17 February 2022, with the last update occurring merely four days later, indicating minimal ongoing development or support. [ImN22]

Ultimately, Delphi reverse engineering remains scarcely supported, with only a handful of tools available - many of which are incomplete according to their authors [cry18; NCC19; ImN22]. Most existing solutions are tied to IDA Pro, a proprietary tool, or are limited to 32-bit executables, restricting their applicability for modern 64-bit cases [ImN22; Col22; ESE24]. Additionally, despite their open-source nature, the intellectual insights behind these tools often remain embedded within their source code rather than being formally documented. This thesis aims to address these gaps by systematically analysing Delphi's file format and exploring ways to enhance the fragmented and underdeveloped tooling landscape.

3 Static Analysis of Delphi Executables

This chapter focuses on the analysis of the Delphi file format, presenting what is likely the first comprehensive public documentation of its structure and inner workings. By the end of this chapter, the reader will have gained a sound understanding of the different data structures commonly found in Delphi binaries compiled into Microsoft Windows 32-bit and 64-bit executables from versions *Delphi 2010* onwards, up to the latest version available as of March 2025, *Delphi 12 Athens*. In addition to understanding what metadata is typically embedded in these executables, an introduction will be provided on how this information can be leveraged for reverse engineering purposes. A brief comparison with earlier Delphi versions will also be included, given the significant differences between versions prior to *Delphi 2010* and those released afterward. However, this analysis will be strictly limited to Delphi applications compiled as Windows executables, specifically console applications, dynamic libraries, and VCL applications. The latter utilizes the VCL framework designed for building user interfaces on Microsoft Windows [Emb24d].

Throughout this chapter, the details of how Delphi-compiled Windows executables store metadata within the code section (.text section) of the PE file format will be revealed. Through static analysis across six different Delphi versions and a dataset of fourteen Delphi executables, the structure of various internal data elements has been identified, including Delphi's runtime type information (RTTI) objects and Virtual Method Tables (VMTs). The list of analysed executables include both hand-crafted cases and real-world malware samples from families such as *CosmicBanker*, *Culebra*, *Danabot* and *Kiron*. The analysis demonstrates that these binaries contain sufficient embedded metadata to enable pre-execution symbol recovery. This capability is essential for static reverse engineering, as it allows tools to reconstruct method names and signatures without execution, significantly improving the efficiency of reverse engineering workflows [Eck22]. Towards the end of the chapter, a tool designed for recovering symbols from Delphi executables will be introduced, along with an explanation of the methodology used in its development.

To facilitate practical use, the findings of this research have been compiled into various tables, allowing for quick reference and providing valuable insights for reverse engineers working with Delphi executables. This documentation may also serve as a foundation to further expand the tooling landscape dedicated to Delphi reverse engineering. Although this chapter aims to refresh the reader's knowledge of fundamental concepts as they arise, a basic understanding of topics such

as reverse engineering, symbol recovery and the PE format is assumed. Readers who are entirely unfamiliar with these concepts are encouraged to consult the appendix at the end of this thesis, which provides an introduction to these essential topics.

Before delving into the specifics of Delphi's binary format, it is important to first establish a historical context by briefly outlining the key dates relevant to this thesis. Following this, an overview will be provided of the different ways Delphi applications can be compiled, along with a clarification of the scope of analysis.

Delphi was originally developed by Borland Software Corporation and introduced as *Delphi 1* in 1995 as a dialect of Object Pascal [Cor09]. At the time of its initial release, Delphi exclusively produced (16-bit) executable files for the Windows operating system, a focus that has remained central to its development despite later expansions to other platforms, including macOS, Linux, and even mobile operating systems such as Android [Cor09; X24]. However, this work focuses exclusively on Windows-based Delphi applications due to their prevalence and the time constraint of this thesis. Over the years, Delphi has undergone numerous iterations, with many versions bringing significant updates and refinements. With *Delphi 2*, 32-bit architecture was supported for the first time one year after it's initial version [Cor09].

A key aspect of Delphi's history is its change in ownership. Until 2008, Borland Software Corporation was responsible for Delphi's development [Cor09]. From then on, Borland sold the division which oversaw Delphi to Embarcadero Technologies, Inc. [Cor09]. Thus the first Delphi release under Embarcadero was *Delphi 2009* [Cor09]. Under Embarcadero's management, Delphi saw continued updates, most notably for the purpose of this thesis is the introduction of 64-bit Windows support with *Delphi XE2* in 2011 [Gaj17]. In October 2015, Embarcadero itself was acquired by Idera, Inc., which continued to operate the developer tools division under the Embarcadero brand [Bit15]. The first Delphi version released under Idera's ownership was *Delphi 10.1 Berlin*, and the latest version as of March 2025 is *Delphi 12 Athens*, released in 2023 [Sat24].

Regarding the scope of analysis, Delphi applications can be compiled with various high-level settings that affect their file structure. Three of these settings will be discussed here: Firstly, as previously mentioned, Delphi supports multiple application types. This thesis considers user interface applications using the VCL framework (VCL applications), non-graphical console applications, and dynamic libraries, which provide reusable code that can be shared across multiple programs. Other formats, such as Multi-Device Application Packages and DUniX Projects, exist as well but are omitted from this discussion due to time constraints and this thesis' emphasis on the Windows target system. [Emb25b]

Secondly, one of the most fundamental distinctions in Delphi compilation is the target architecture, which can be either 32-bit or 64-bit Intel. The primary difference between these architectures lies in the amount of data that can be processed at once. For this thesis, the most significant implication is whether addresses stored within structures of the Delphi file format are 4 bytes (32 bits) or 8 bytes (64 bits) in length. Regarding which architecture is considered for the format analysis, a fundamental aspect of Windows executables in terms of architecture compatibility must be taken into account: In the broader malware landscape, approximately 90% of malicious software is still compiled as 32-bit executables [Wil24]. Microsoft's strong commitment to backward compatibility

ensures that 32-bit executables can still be run on modern 64-bit Windows systems through the Windows on Windows (WoW64) subsystem [Rad20]. While this design decision maintains the longevity of legacy software, it also has significant implications for cybersecurity, as it allows older 32-bit malware to persist as a viable attack vector. Another potential reason for the prevalence of 32-bit malware is low-effort malware development, where developers make minor modifications to existing malware rather than writing new threats from scratch [Wil24]. Additionally, it is plausible that some threat actors still rely on outdated or legacy development environments which default to producing 32-bit binaries. However, recent trends indicate a growing shift towards 64-bit malware [Wil25]. This development is partly driven by the adoption of modern development tools such as Visual Studio, which default to compiling 64-bit applications [Wil24]. As the proportion of 64-bit malware continues to rise, it becomes increasingly important to account for both architectures in the analysis of Delphi binaries. Accordingly, this thesis examines Delphi's file format across both 32-bit and 64-bit executables.

Another essential compilation setting to consider is the distinction between release and debug modes. In debug mode, additional debugging symbols and metadata are embedded within the executable, facilitating software testing and development. This information is invaluable to developers but is typically stripped from binaries before distribution, particularly in proprietary or security-sensitive applications. In contrast, release mode produces binaries that have been optimized by the compiler and exclude certain metadata, resulting in a more compact executable. Since release-mode binaries are the standard in executables encountered by reverse engineers, this thesis focuses primarily on their analysis. [Emb19]

The following introduces the primary motivation for the structural analysis of the Delphi file format. When examining Delphi executables - compiled using any combination of the settings outlined thus far - even with basic static analysis tools, a key characteristic becomes apparent: the presence of metadata embedded within the file. A simple string extraction already reveals that numerous method names, along with elements of function signatures, are stored in plain ASCII format within the binary - even when compiled in release mode. This behaviour stands in stark contrast to, for instance, stripped C++ binaries, in which function names are typically inaccessible without the use advanced techniques like byte-level pattern matching and signature-based heuristics such as IDA Pro's FLIRT [Hex25a]. The presence of extractable metadata during string analysis implies that these symbolic elements must be stored somewhere within the binary itself. To identify where such data may reside, it is useful to first examine the general format Delphi employs when producing compiled executables. Like most modern Windows-targeted compilers, Delphi utilises the PE format, which is the sole supported format for native code on this platform. This well-documented format offers several mechanisms by which metadata could be stored.

To briefly recall, the PE format is structured into distinct sections, each serving a particular purpose in the organisation of executable content [Bri25]. These sections - such as `.text`, `.data`, `.rdata`, and `.rsrc` - represent logical partitions within the binary file, separating code, initialised and uninitialised data, resources, and other specialised information [Bri25]. Although a range of such sections could, in theory, serve as storage locations for Delphi-specific metadata, many can be discounted due to obsolescence or irrelevance for the purposes of reverse engineering. An example for that is the Common Object File Format (COFF) Symbol Table [Bri25]. Despite its name suggesting a potential location for symbolic metadata, this structure is now largely obsolete and

is generally unused by modern compilers, including the newer versions of Delphi [Bri25]. Other sections, such as `.rsrc`, have also been examined. While it is true that strings occasionally reside in the resource section, they are typically not relevant for the goal of recovering internal program structure or function metadata.

More promising is the `.text` section, which traditionally contains the executable code of a binary [Bri25]. Interestingly, in Delphi executables, especially those generated for graphical applications using the VCL framework, this section often contains more than just machine instructions. Embedded within, one finds a variety of data structures essential to Delphi's runtime environment. Among the most relevant of these are VMTs, which provide metadata and function dispatch mechanisms, which are characteristic of object-oriented constructs.

We will now present our methodology. The first step of the investigation was compiling a 32-bit Delphi executable using the official Delphi IDE. The selected project template was the default graphical user interface application, which by design instantiates a minimal window without implementing any additional logic. This served as a controlled baseline for structural examination.

Manual inspection of the resulting executable's code section revealed several potentially useful patterns, including contiguous sequences of addresses and readable strings corresponding to Delphi internal method names. These initial findings suggested the presence of metadata relevant for tasks such as symbol recovery. In order to validate that these structures could indeed serve such a purpose, a comparison was made against output from IDR, the de facto standard tool for Delphi reverse engineering. This comparison confirmed that the manually discovered structures align with IDR's interpretation, thereby supporting the assumption that the located metadata was semantically meaningful.

At this early stage, two versions of the same binary were effectively in use: One analysed solely within Ghidra, and another whose structure was further enriched by IDR's metadata export and made accessible to Ghidra via the dhrake plugin. The primary analysis was carried out on the former, with findings later confirmed using the latter. This approach facilitated more reliable validation of early hypotheses.

A particularly useful property observed in VCL-based Delphi applications is the predictable structure of their early execution routines. Repeated analysis of several binaries - both synthetic samples and those found in the wild - revealed that the initial call stack is often dominated by a small, consistent set of functions. Among these is the `CreateForm` routine, which dynamically instantiates a form at runtime [Emb14c]. In the Delphi context, a form is a visual container for GUI components and event handlers, akin to a window in other graphical frameworks [Emb14a]. In the initial test binary, a string reference to this function name was discovered near the entry point, and a nearby function was renamed accordingly. Comparison with IDR's analysis confirmed the validity of this identification, serving as an early proof of concept for the methodology, even though the broader conceptual structure remained unclear so far.

The overall strategy was then refined as follows: A chronological, manual walk-through of the code section was conducted, identifying all regions that, despite the section's name, did *not* represent executable instructions. These candidate data structures were examined for patterns, structural repetition, and consistent alignment. Based on these observations, preliminary hypothe-

ses regarding the layout and purpose of these regions were formulated. These hypotheses were then tested against a broader set of binaries, compiled under varying compilation settings, to test their generalisability. In instances where conflicting evidence emerged, the assumptions were either discarded or revised. In addition, both 32-bit and 64-bit variants were analysed whenever applicable, as differences in address size and data layout necessitated careful adaptation of structural models.

Furthermore, throughout this process, heuristic rules and general conventions were employed to guide the interpretation. For example, discrepancies of four bytes between structurally equivalent fields across different architectures typically indicated address values, as pointers are four bytes long in 32-bit and eight bytes long in the 64-bit case. Similarly, recurring small integers - particularly one-byte or two-byte values - found directly before repeating substructures were interpreted as length fields or counts. Where necessary, byte offsets and alignments were examined in context to determine structural boundaries and associations.

In summary, the analysis followed an iterative, trial-and-error methodology grounded in manual analysis. Since there are few instances of executables where a complete ground truth can be established, this process requires comparative evaluation and convention-based reasoning. Findings were continuously refined and re-evaluated given new evidence, enabling the gradual reconstruction of metadata embedded within Delphi executables.

As a final preparatory step before engaging with the format analysis, it is necessary to define the specific reverse engineering task that falls within the scope of this thesis. Clarifying the objectives is critical to understanding which and why certain fields of the extracted data structures are more significant than others.

The principal task under consideration is *symbol recovery*. In the context of reverse engineering, symbol recovery refers to the process of reconstructing high-level, human-readable information about functions, variables, and data structures from a compiled binary to improve the readability and maintainability of subsequent analyses [Pal+24]. This typically involves identifying function names, parameter lists, return types, namespaces, and similar symbolic elements that were present in the original source code but are partially or fully stripped from straightforward access during compilation. Successful symbol recovery allows reverse engineers to restore a semantic view of the program, which can dramatically improve both the efficiency and accuracy of subsequent analyses.

```
1  __classmethod int __fastcall System::TMarshal::ReadInt32(System::TPtrWrapper Ptr,
       NativeInt Ofs = 0);
```

Figure 3: *Example of a function signature for the Delphi internal class method* `ReadInt32` *[Emb13]. The colour scheme is as follows: red denotes return and parameter types, purple represents namespace components, green denotes the function name and blue indicates parameter names.*

We now define the building blocks of a typical function signature in the slightly adapted Embarcadero C++-style. Figure 3 shows an example of such a notation. The *function name* identifies the function or method within the source code; in this instance, the function is named `ReadInt32`. Functions are often enclosed within a *namespace*, which acts as a logical grouping to prevent naming collisions and to provide organisational structure. Here, the function resides in the `System::TMarshal` namespace. The *fully qualified name (FQN)* combines both, therefore becoming `System::TMarshal::ReadInt32`. The *list of function parameters* specifies the input expected by the

function, with each parameter described by a tuple of its type and name; in this case, the function takes a `TPtrWrapper` named `Ptr` - fully qualified as `System::TPtrWrapper` - and an integer whose size matches the CPU architecture named `Ofs`, which defaults to zero. Finally, the `return value` designates the type of data produced by the function, which in this case is a 32-bit signed integer.

The primary advantage of successful symbol recovery lies in the significant acceleration it brings to the reverse engineering process. The reconstruction of symbolic information allows reverse engineers to more rapidly and effectively navigate a binary, gaining contextual understanding from known function names rather than relying solely on low-level machine instructions. This enhanced context facilitates more accurate decompilation and control flow analysis as the relationships between functions and their data become clearer. Symbol recovery is thus especially valuable when examining complex and time-consuming binaries, such as those produced by Delphi.

Given these advantages, it becomes clear which specific pieces of information are desirable for effective symbol recovery of functions. Ideally, for each function present within a compiled executable, the following six elements should be obtained:

- *Requirement 1: Function Location*: The address at which the function is defined within the binary.
- *Requirement 2: FQN Recovery*: The FQN of the function, preferably available as a string within the file and ideally accompanied by namespace information.
- *Requirement 3: FQN Mapping*: A bidirectional mapping that unambiguously associates FQNs with their corresponding function addresses, either directly encoded within the file (e.g., via explicit references) or inferable through consistent file structure conventions.
- *Requirement 4: Parameter Types*: A list of parameter tuples comprising parameter types and parameter names, ideally stored as address references and plaintext strings, respectively.
- *Requirement 5: Return Types*: The return value information, in a format analogous to that used for parameters.
- *Requirement 6: Function Signature*: A mapping that associates to each function its corresponding set of parameters and return value.

These criteria establish a framework for evaluating the internal metadata of Delphi binaries: the greater the extent to which the binary preserves and structures such information, the more feasible and complete a subsequent symbol recovery effort will be. With these objectives in mind, the following sections will systematically investigate the storage and organisation of relevant metadata within Delphi executables from versions *Delphi 2010* onwards, up to *Delphi 12 Athens*. A comparison with earlier Delphi versions will follow thereafter.

In the code section of compiled Delphi executables, a variety of internal data structures can be found, often mixed together with actual executable instructions. For the purpose of this thesis, the most relevant structures have been identified as VMTs, RTTI objects, and Method Definition Tables (MDTs).

3.1 Virtual Method Table

The first structure to be discussed is the VMT. These structures are among the most fundamental components of the Delphi binary format, primarily because they act as hubs containing cross-references to other essential structures. Hence they can serve as entry points for reverse engineering efforts such as symbol recovery. The general layout of the VMT, including individual field descriptions, is presented in Table 2, whereas Figure 4 illustrates the same in a schematic view. For completeness, every field is documented, even if some are not directly relevant to the task of symbol recovery. However, fields that are of particular importance to the pursued objective are highlighted in bold. Across all upcoming results, the term *optional* is used to indicate the following: When referring to pointer fields, it denotes that the field was observed to contain either a valid address or a sequence of null bytes. In contrast, for other types of fields - such as padding or complex substructures - the term *optional* implies that the field may be entirely absent from the memory layout, rather than merely filled with null bytes. Regarding nomenclature, the naming conventions used are inspired by Microsoft's PE format. Field names are either directly adopted from outputs generated by IDR or were introduced manually by the author where necessary for clarity and consistency.

Since the results of the format analysis are largely empirical, each field within the following tables is annotated with a *Confidence Level* to indicate the reliability of its interpretation. Four different levels have been defined, as shown in Table 1, ranging from completely unknown structures or contents to information derived from trusted sources. Between these extremes, in case of structural consistency, two different levels of hypothesised results are used, applied based on whether its purpose is rather deducible or not.

Table 1: *Confidence Levels used in format analysis tables including their used numeric values.*

Confidence Level	Description
Unknown **(0)**	The purpose of the field is unknown or cannot be reasonably deduced. The field might have an even more fine-granular structure.
Provisionally Hypothesised **(1)**	A tentative interpretation based on a consistently recurring pattern across all observations, though the exact purpose remains unclear.
Strongly Hypothesised **(2)**	A robust interpretation with a deducible purpose and clear structural coherence, consistent across all observations.
Trusted Reference **(3)**	The structure and purpose of the field is either confirmed by an official source, or derived using the de facto standard tool for Delphi reverse engineering (IDR).

The structure of a VMT is remarkably regular. It contains consecutive, address-sized fields which point to additional structures, data, and functions. There are 22 fields in the 32-bit case, and 25 fields for 64-bit executables. In the former case, a VMT has a size of 88 bytes while it spans 200 bytes in the latter one. The size difference arises not only from architectural differences in address sizes but also because VMTs in the 64-bit case contain an additional 24 bytes of data at the end that are not present in the 32-bit variant. VMTs are automatically generated by the compiler during

the build process and hence not intended to be directly modified by application code. Lastly, each Delphi class is associated with exactly one VMT.

Table 2: *Layout of **VMT**s for versions Delphi 2010 and onwards for 32/64-bit architectures. A field's offset (O) and size (S) information can be seen in the columns of the corresponding letter and architecture number. Addressing is done byte-wise. The column CL depicts the Confidence Level as introduced in Table 1.*

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x0	0x0	4	8	`NextStruct`	Optional pointer to the data structure directly following the VMT, such as user defined functions, InitTable etc.	2
0x4	0x8	4	8	`IntfTable`	Optional pointer to the Interface Table.	3
0x8	0x10	4	8	`AutoTable`	Optional pointer to the Automation Information Table.	3
0xC	0x18	4	8	`InitTable`	Optional pointer to the Instance Initialization Table.	3
0x10	0x20	4	8	**`VmtRtti`**	Optional pointer to the RTTI class, corresponding to the VMT. Possibly an indirect reference.	2
0x14	0x28	4	8	`FieldTable`	Optional pointer to the Field Definition Table.	3
0x18	0x30	4	8	**`MethodTable`**	Optional pointer to the Method Definition table, listing methods of the corresponding class.	3
0x1C	0x38	4	8	`DynamicTable`	Optional pointer to the Dynamic Method Table.	3
0x20	0x40	4	8	`ClassName`	Pointer to the `NameOfVmt` field of the MethodTable, containing a short string of the class name.	3
0x24	0x48	4	8	`InstanceSize`	Cardinal stating the instance size of the VMT when loaded in memory.	3

continued on next page

Table 2 – continued from previous page.

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x28	0x50	4	8	`ParentVmt`	Optional pointer to the VMT of parent class in its inheritance chain. If filled with null bytes, current VMT is a root class.	3
0x2C	0x58	4	8	`EqualsMethod`	Pointer to the entry point of the *Equals* method.	3
0x30	0x60	4	8	`GetHashCodeMethod`	Pointer to the entry point of the *GetHashCode* method.	3
0x34	0x68	4	8	`ToStringMethod`	Pointer to the entry point of the *ToString* method.	3
0x38	0x70	4	8	`SafeCallExceptionMethod`	Optional pointer to the entry point of the *SafeCallException* method.	3
0x3C	0x78	4	8	`AfterConstructionMethod`	Pointer to the entry point of the *AfterConstruction* method.	3
0x40	0x80	4	8	`BeforeDestructionMethod`	Pointer to the entry point of the *BeforeDestruction* method.	3
0x44	0x88	4	8	`DispatchMethod`	Pointer to the entry point of the *Dispatch* method.	3
0x48	0x90	4	8	`DefaultHandlerMethod`	Pointer to the entry point of the *DefaultHandler* method.	3
0x4C	0x98	4	8	`NewInstanceMethod`	Pointer to the entry point of the *NewInstance* method.	3
0x50	0xA0	4	8	`FreeInstanceMethod`	Pointer to the entry point of the *FreeInstance* method.	3
0x54	0xA8	4	8	`DestroyDestructor`	Pointer to the entry point of the *Destroy* destructor method.	3
n.a.	0xB0	n.a.	8	unknown	unknown;	0
n.a.	0xB8	n.a.	8	unknown	unknown;	0
n.a.	0xC0	n.a.	8	unknown	unknown;	0

The first field in the VMT structure is named `NextStruct`. This address points to the data structure that follows immediately after the VMT. Typically, this structure consists of a sequence of addresses

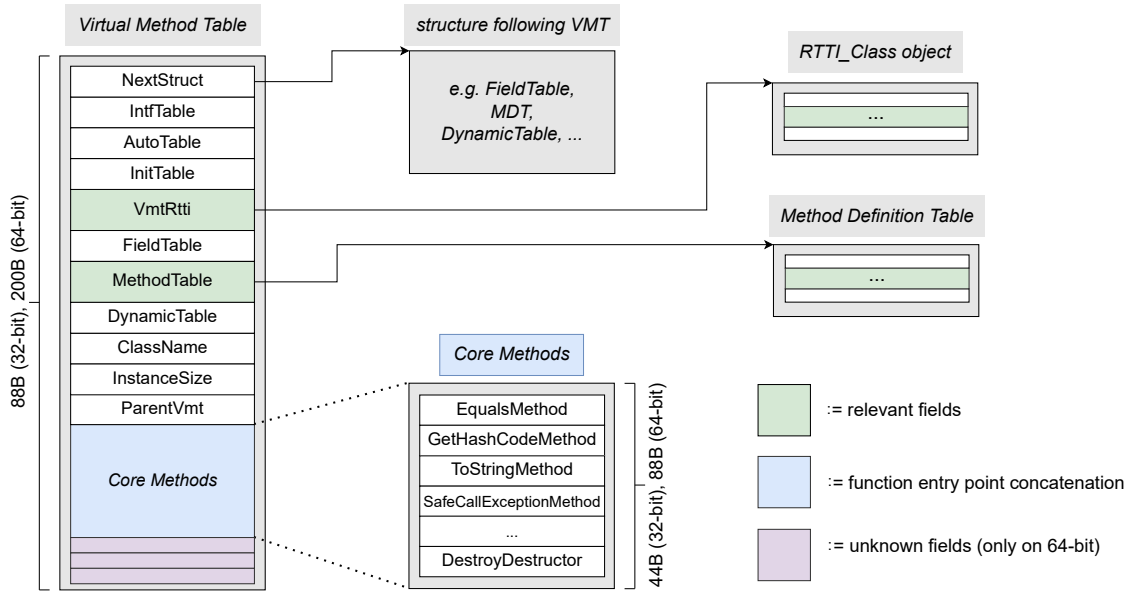

Figure 4: *Abstract illustration of the VMT layout. Fields coloured in green are relevant fields for the symbol recovery task.*

pointing to function entry points, presumably corresponding to user-defined functions. If no user-defined functions are present, the *NextStruct* field points to whichever data structure physically follows the VMT. Empirical observations suggest that the sequence of structures following a VMT mirrors the order in which references are laid out within the VMT itself, provided that the referenced structures actually exist. For instance, if neither user-defined functions nor an Interface Table are present, but an Automation Information Table is, the content of `NextStruct` will reference the latter.

Subsequent to this, eight further address fields are stored within the VMT, each pointing to auxiliary tables related to class metadata. These include the *Interface Table*, *Automation Information Table*, *Instance Initialisation Table*, *Field Definition Table*, *Method Definition Table*, and *Dynamic Method Table*. Among these, the Method Definition Table (MDT) is of particular importance. It contains not only addresses of function definitions but also the symbolic names of these functions as well as their parameter names. Furthermore, FQNs can be reconstructed through references embedded within RTTI class structures, which themselves are accessible via the MDT.

The fifth address field within the VMT, named `VmtRtti`, points to the RTTI class structure associated with the class described by the VMT. As explained later, the referenced RTTI class structure provides information about the namespace of the VMT and its related methods. It therefore plays an essential role in fulfilling *Requirement 2: FQN Recovery*, as namespace information is part of the FQN of a method. However, it has been observed that in some executables this field does not point directly to the RTTI class structure but instead to an intermediary address, which subsequently references the RTTI class. In this thesis, this phenomenon is referred to as an *indirect reference*.

The ninth address field, the `ClassName`, points to a short string containing the class name corresponding to the VMT. While this field enables retrieval of this information, it does not include the entire namespace information and thus is insufficient for constructing FQNs independently.

Following the class name pointer, the VMT stores the `InstanceSize` field, which reflects the size of instances of the class. This field uses the four bytes long `int` data type in 32-bit executables and the eight bytes long `int64` in 64-bit executables. In doing so, the VMT maintains alignment with the corresponding address size throughout the entire data structure. Afterwards, another address field follows, pointing to the VMT of the parent class in the inheritance chain. While this inheritance information can theoretically be used to infer further class relationships, it is of secondary importance for the goals of this work, as namespace reconstruction takes precedence.

Subsequent to this, the VMT structure contains eleven more addresses, each pointing to standard methods inherited by every Delphi class from the universal base class, *TObject*. At offset 88 bytes in 32-bit executables, the fixed structure of the VMT concludes. In 64-bit executables, however, an additional 24 bytes follow. Although the precise purpose of this additional data is not conclusively determined within the scope of this work, it has been consistently observed that it comprises three addresses. These addresses are constant across all VMTs of individual executables, each spaced 16 addresses apart, and they point to memory regions filled with the machine code byte that corresponds to the x86 return instruction (`0xC3`), padded with the machine code byte which represents the x86 `int3` breakpoint instruction (`0xCC`). This pattern suggests a potential defensive programming technique intended to guard against invalid code execution paths.

3.2 Method Definition Table

The second data structure of particular interest in the context of this work is the MDT. This structure plays a central role in the symbol recovery task outlined in this thesis, as it stores - for each method belonging to the class represented by the corresponding VMT - a method's individual entry point, its name, and its return type. Furthermore, the MDT embeds additional RTTI-derived information, including the names and types of its formal parameters. There is at most one MDT per VMT, and it can be accessed via one of the fields directly embedded within the VMT layout.

Compared to the VMT, the general layout of the MDT is noticeably more complex and irregular. This increased complexity is primarily a result of the presence of repeating substructures and a degree of structural nesting. To support a clearer understanding of the MDT's internal composition, Figure 5 presents an abstracted visual representation of its hierarchical structure and its component relationships before Table 3 examines byte-level granularity. Additionally, the figure annotates which categories of information are located within which segments of the MDT.

From a top-down perspective, the MDT can be described in terms of three nested layers of abstraction. On the top level, the layout reveals two principal segments which may, under certain conditions, be spatially separated by a gap containing other data structures. The *upper part* - located at lower memory addresses - serves primarily as a navigation layer, containing a list of addresses that point to the actual content structures of the *lower part* of the MDT. The second level of abstraction concerns the format and semantics structures herein referred to as *MethodEntryRefs* and *MethodEntry*. The former is relatively compact and serves as a pointer to the more elaborate

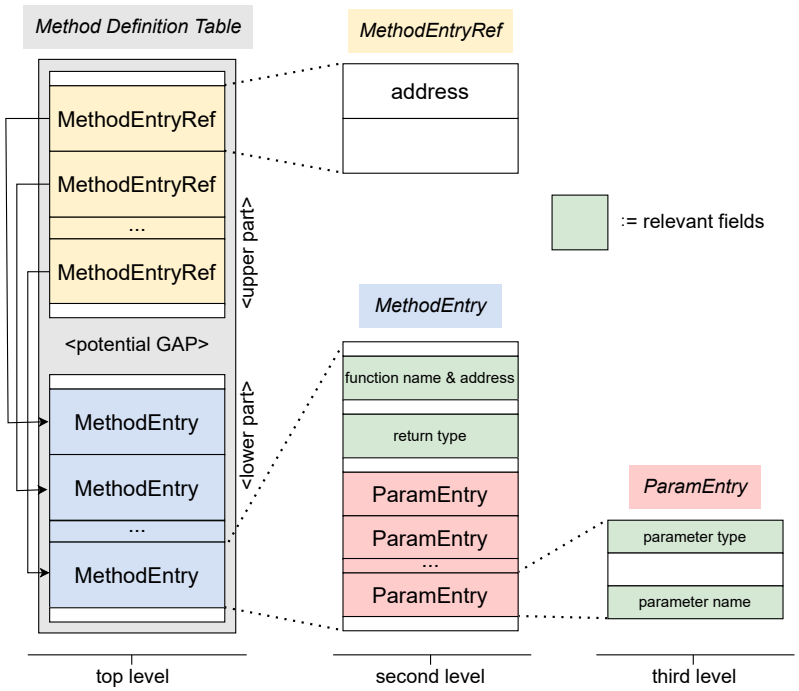

Figure 5: *Abstract illustration of the MDT layout, including its repeating and nested substructures. Parts of substructures coloured in green are relevant fields for the symbol recovery task.*

MethodEntry structure, which stores the actual data associated with the specific method. There exists a one-to-one correspondence between *MethodEntries* and the class methods of the associated VMT. Finally, on the third level of this hierarchy, each *MethodEntry* may optionally contain a non-negative number of embedded *ParamEntry* substructures. Each *ParamEntry* stores a triple of parameter metadata, including both its name and its type.

We now turn to the top-level layout of the MDT, which is summarised in Table 3. The format begins with a two-byte field whose purpose remains unidentified. Interestingly, across all observed binaries, this field consistently contains null bytes, suggesting a potential role as a marker or, more plausibly, as a reserved flags field that defaults to the standard value `0x0000`. The consistent presence of this field at the very beginning of the MDT rather supports the latter interpretation.

Table 3: *Layout of **MDT**s (top level view) for versions Delphi 2010 and onwards for 32/64-bit architectures. A field's offset (O) and size (S) information can be seen in the columns of the corresponding letter and architecture number. The column CL depicts the Confidence Level introduced by Table 1.*

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x0	0x0	2	2	unknown	Likely a reserved field filled with null bytes.	0

continued on next page

Table 3 – continued from previous page.

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x2	0x2	2	2	`NumOfMethodEntryRefs`	Number $m \geq 0$ of `MethodEntryRefs` structures following this field.	2
0x4	0x4	8	12	$\texttt{MethodEntryRef}_1$	1st `MethodEntryRef` substructure, pointing to 1st `MethodEntry` substructure. Does not exist if $m = 0$.	2
...	...	...	...	...	...	...
n.a.[1]	n.a.[2]	8	12	$\texttt{MethodEntryRef}_m$	m-th `MethodEntryRef` structure. Does not exist if $m = 0$.	2
n.a.[3]	n.a.[4]	2	2	unknown	Likely flags of unknown purpose. (*After this field, a gap of dynamic length may occur until the following field.*)	0
n.a.	n.a.	len(S)	len(S)	`NameOfVmt`	Pascal string representing the VMT's name.	2
n.a.	n.a.	dyn	dyn	$\textbf{MethodEntry}_{1,\ldots,m}$	m consecutive `MethodEntry` substructures, each containing data of the 1st,...,m-th function. Does not exist if $m = 0$.	2
n.a.	n.a.	dyn	dyn	`MethodTablePadding`	0 to 4 null bytes of unknown logic.	1

Immediately following this is a two-byte field labelled `NumOfMethodEntryRefs`, which specifies the number of *MethodEntryRef* substructures present in the upper part of the MDT. This is followed by a contiguous sequence of the actual `MethodEntryRef` substructures, making the MDT fundamentally dynamic in size. Concluding the upper segment of the MDT is another two-byte field of unknown purpose. Unlike the initial two byte-null field, this latter field has been observed to contain various low hexadecimal values. The diversity of these values suggests that this field may function as a set of flags as well, although its exact semantics remain speculative at this stage. Despite the dynamic nature of the upper part, the fixed size of the *MethodEntryRef* structures allows for explicit byte offsets to be defined up to this point in the format specification tables.

[1] Field offset can be calculated as: $(m-1) \times 8 + 4$.
[2] Field offset can be calculated as: $(m-1) \times 12 + 4$.
[3] Field offset can be calculated as: $m \times 8 + 4$.
[4] Field offset can be calculated as: $m \times 12 + 4$.

The lower part of the MDT begins with a field named `NameOfVmt`, which firstly introduces a structural pattern inherited from Delphi's ancestor language, Pascal. Specifically, Delphi encodes variable-length ASCII strings using *Pascal Strings*. These strings consist of a one-byte length prefix followed by the corresponding number of ASCII characters. Formally, a Pascal String S is defined as a tuple of the form

$$S = n, \text{char}_1, \text{char}_2, ..., \text{char}_n,$$

where the first byte n encodes the length of the string content. Also, we define $\text{len}(S)$ as the length of a Pascal String S in bytes, including the first byte n. The maximum length of such strings is 256 bytes, composed of one byte for the length and up to 255 bytes for the actual character data. Hence, Pascal Strings are not null-terminated; their boundaries are instead determined by the embedded length prefix.

However, in some executables, the lower segment of the MDT is not immediately contiguous with the upper part. It may be separated by other data structures such as a Dynamic Method Table. This architectural choice implies that the `NameOfVmt` field cannot generally be accessed via a fixed offset relative to the start of the MDT. Consequently, in Table 3 the offset column is annotated as "n.a." (not applicable) for such fields. Notably, the content of `NameOfVmt` is still directly accessible via the `ClassName` field embedded in the corresponding VMT, as outlined earlier in Table 2.

Following the `NameOfVmt` field is a sequence of *MethodEntry* substructures. Their number always matches that of the *MethodEntryRef* entries due to the one-to-one mapping. Each *MethodEntry* encodes the primary metadata required for symbolic function reconstruction. Due to their structural complexity, their layout will be examined in greater detail after the full presentation of the MDT's top-level view has been completed.

The use of Pascal Strings in both the `NameOfVmt` field and within the *MethodEntry* structures introduces further variability in size. Since the lengths of these fields depend on their respective content, the layout tables for these parts of the MDT cannot provide reliable fixed offsets. The same rationale applies to future format tables. Similar cases will not be restated in detail to avoid redundancy.

At the end of the MDT lies a sequence of null bytes, whose length ranges between zero and four. Although this suggests an alignment strategy based on four-byte boundaries, counterexamples have been observed that disprove this assumption. As such, the purpose of this trailing padding remains unclear. Finally, it is worth noting that if the *NumOfMethodEntryRefs* field holds the value zero, no second-level or third-level substructures are present at all. This condition represents an empty MDT corresponding to a VMT that does not define any methods.

3.3 MethodEntryRef

On the second structural level of the MDT, we begin with the relatively compact layout of the *MethodEntryRef* substructure, summarised in Table 4. These substructures are either 8 or 12 bytes in length, depending on the target architecture, and serve a singular but essential purpose: each one references the location of its associated *MethodEntry* structure, which resides later in the MDT's layout.

Table 4: *Layout of the **MethodEntryRef** substructure for versions Delphi 2010 and onwards for 32/64-bit architectures. A field's offset (O) and size (S) information can be seen in the columns of the corresponding letter and architecture number. The column CL depicts the Confidence Level using the abbreviations introduced by Table 1.*

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x0	0x0	4	8	`AddrOfMethodEntry`	Pointer to the corresponding `MethodEntry` structure.	2
0x4	0x8	4	4	unknown	Likely flags of unknown purpose.	0

The first field of a *MethodEntryRef* is `AddrOfMethodEntry`, which contains the address of the corresponding *MethodEntry*. This direct link is critical for the correct parsing and traversal of the MDT. Following this, there are four bytes of currently unidentified data. Although no interpretable pattern has yet emerged from the analysis of this field, its fixed position and consistent presence suggest that it likely holds auxiliary flag information or meta-data.

3.4 MethodEntry

Subsequent to the *MethodEntryRef*, we encounter the structurally more elaborate *MethodEntry* substructure, as shown in Table 5. This component contains several fields of immediate relevance for the identification, reconstruction, and analysis of class methods within a compiled Delphi executable. Moreover, it hosts a list of third-level substructures which contain vital information about a method's parameters.

Table 5: *Layout of a **MethodEntry** substructure for versions Delphi 2010 and onwards for 32/64-bit architectures. A field's offset (O) and size (S) information can be seen in the columns of the corresponding letter and architecture number. The column CL depicts the Confidence Level introduced by Table 1.*

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x0	0x0	2	2	`LenOfMethodEntry`	Cardinal stating the length of this `MethodEntry` including this 2B long field in bytes.	2
0x2	0x2	4	8	**`FunctionDefinition`**	Pointer to the entry point of a function definition.	2
0x6	0xA	len(S)	len(S)	**`NameOfFunction`**	Pascal String of the function name represented by the previous field.	2

continued on next page

Table 5 – continued from previous page.

O'32	O'64	S'32	S'64	Fieldname	Description	CL
n.a.	n.a.	2	2	unknown	Byte sequence `03h 00h`.	0
n.a.	n.a.	4	8	**`ReturnType`**	Optional pointer to an RTTI class object representing the function's return type.	2
n.a.	n.a.	2	2	unknown	unknown	0
n.a.	n.a.	1	1	`NumOfParamEntries`	Number of `ParamEntry` substructures $k \geq 1$ following this field.	2
n.a.	n.a.	1	1	unknown	unknown	0
n.a.	n.a.	dyn	dyn	**`ParamEntry`**$_1$	1st `ParamEntry` substructure. For class methods, it is mandatory; its fields are `"Self"` and a pointer to the VMT's RTTI class (indirect reference).	2
n.a.	n.a.	dyn	dyn	**`ParamEntry`**$_{2,\ldots,k}$	Optional additional `ParamEntry` substructures. Only exist for $k > 1$. *In this case, each* `ParamEntry`$_i$ *for* $i \in \{2,\ldots,k\}$ *is preceded by 3 bytes of unknown data!*	2
n.a.	n.a.	4	4	`Separator`	Separating bytes `02h 00h 02h 00h`, marking the end of the structure, potentially with subsequent null byte padding.	1

The first field is called `LenOfMethodEntry`, which denotes the total size of the *MethodEntry* substructure, including the length of this field itself. This design reflects a third distinct approach used by Delphi to encode structural lengths within its binary format. Fixed-size structures, such as the VMT, represent the first category, while repetition counters - e.g. `NumOfMethodEntryRefs` in the top-level layout of the MDT - constitute the second. By contrast, the `LenOfMethodEntry` field exemplifies a self-describing length field, enabling dynamic parsing based on in-structure metadata.

Immediately following this field is an address pointing to the entry point of the function described by this *MethodEntry*. Conveniently, the subsequent field is a Pascal String encoding the function's name (excluding its namespace). This proximity of address and name allows for a reliable association between executable code and its symbolic identifier. As all reachable VMTs contain entries for their defined methods, and all *MethodEntry* substructures contain both name and address information, this mechanism alone satisfies *Requirement 1: Function Location* and

contributes substantially toward meeting *Requirement 2: FQN Recovery* and *Requirement 3: FQN Mapping*.

Following the function name field is a two-byte segment that has consistently contained the byte sequence `0x03 0x00` across all analysed executables. The absence of variation either implies a conventional marker value, possibly denoting the end of the name field, or alternatively functioning as another standard flag field.

The next field is an address referencing the RTTI class object representing the return type of the function. This reference directly supports *Requirement 5: Return Types* and partially fulfils *Requirement 6: Function Signature* by enabling the recovery of the function's return type. If the pointer is filled with null bytes, it can be inferred that the function does not return a value.

The subsequent four bytes are largely of uncertain purpose. However, the third byte within consistently encodes the number of *ParamEntry* substructures, and thus by extension, the number of parameters associated with the method. Given this contextual proximity, it is likely that the surrounding three bytes encode additional flags or metadata, although their precise function remains unknown.

The *ParamEntry* list follows, comprising a series of third-level substructures whose format will be addressed separately after the second-level view is fully discussed. Each *ParamEntry* combines RTTI references to a parameter's type with a cleartext name, and thus plays a central role in enabling complete symbol recovery.

Two aspects of the *ParamEntry* substructure demand particular emphasis: First, in the case of class methods, the first parameter invariably represents the "Self" object. This characteristic is typical of object-oriented programming. In such cases, the first *ParamEntry* references the RTTI of the class from which the method originates. This constitutes an alternative pathway for deriving the FQN, thus contributing towards *Requirement 2: FQN Recovery* in an alternative way. The distinction between this approach and using the VMT's `VmtRtti` field lies in their way of accessing the RTTI object: the *ParamEntry* makes use of an *indirect reference*, whereas the VMT does so directly. Second, when multiple parameters are present, the *ParamEntry* layout adopts an unusual structure. Beginning with the second *ParamEntry*, each subsequent substructure is preceded by a three-byte prefix - presumed to be flag information. These three-byte blocks are notably absent from the first *ParamEntry*, and likewise omitted entirely if the method takes zero or only one parameter.

Across all analysed samples, every *MethodEntry* concludes with the four-byte sequence `0x02 0x00 0x02 0x00`. This consistent pattern is believed to function as a separator, marking the end of the substructure. Additionally, in some instances, this sequence is followed by null-byte padding, potentially to satisfy alignment constraints.

3.5 ParamEntry

On the final layer of the MDT, there is the *ParamEntry* substructure. Each *ParamEntry* represents a single parameter of a method defined by the surrounding *MethodEntry*, and it encapsulates essential information for the semantic reconstruction of function interfaces. The *ParamEntry* substructure consists of three fields and its layout is shown in Table 6.

Table 6: *Layout of the **ParamEntry** substructure for versions Delphi 2010 and onwards for 32/64-bit architectures. A field's offset (O) and size (S) information can be seen in the columns of the corresponding letter and architecture number. The column CL depicts the Confidence Level introduced by Table 1.*

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x0	0x0	4	8	`AddrOfRtti`	Optional pointer to the corresponding RTTI object (indirect reference).	1
0x4	0x8	2	2	unknown	unknown	0
0x6	0xA	len(S)	len(S)	`NameOfParam`	A Pascal String stating the name of the parameter.	1

The first field is a pointer to the RTTI object that defines the type of the parameter, which is vital for fulfilling *Requirement 4: Parameter Types*. Following this are two bytes whose purpose remain unidentified. However, their consistent presence and specific position across all examined binaries strongly suggest their function as a flag or metadata field. The final field is a Pascal String containing the parameter's name. Paired with the information about parameter types, the presence of these readable parameter names fulfils *Requirement 4: Parameter Types*. Due to the structured way in which the information can be accessed, the mapping involving MDT's second-level return type and third-level parameter tuples can be established both individually and in relation to the corresponding second-level function, thereby fulfilling *Requirement 6: Function Signature*.

Notably, the order of the *ParamEntry* substructures within the *MethodEntry* corresponds exactly to the parameter order in the original function definition, as confirmed by cross-referencing with official Delphi documentation.

With the introduction of this final substructure, the structural analysis of the MDT concludes. The information encoded across its three hierarchical levels - from second-level function names, their entry points and return types, to third-level parameter names and types - clearly illustrates that the MDT format constitutes a rich and systematically organised source of symbolic information. What remains to be examined, however, is the format and the content of the RTTI objects, which are referenced by several preceding fields, including one within the *ParamEntry* substructure.

3.6 RTTI-Objects

Across all analysed executables, RTTI objects appear to be the most frequently occurring structural elements. They serve as the cornerstone of Delphi's internal mechanism for managing type information at runtime, enabling reflective access to type metadata during execution. Delphi executables contain a variety of RTTI object types, each presumably conforming to a distinct internal format and serving a specific semantic purpose. For clarity, this thesis adopts the term *RTTI object type* to refer to the different kinds of RTTI structures encountered. Each such object begins with a single-byte magic number, which uniquely identifies its type and, by extension,

determines the structure of the data that follows. A total of twenty-one distinct RTTI object types have been identified, which are enumerated in Table 7. Their names are based on labels extracted from executables that were previously processed using the IDR tool.

Table 7: *List of the different **RTTI object types** for versions Delphi 2010 and onwards for 32/64-bit architectures. The descriptions only address contents between the typical header and footer of all RTTI objects.*

Magic Byte	RTTI object type	Description	CL
`0x01`	`RTTI_Integer`	Contains 9 bytes of unknown purpose.	1
`0x02`	`RTTI_Char`	Contains 9 bytes of unknown purpose.	1
`0x03`	`RTTI_Enumeration`	Concatenation of Pascal Strings for the enumeration's attributes' value names and namespace, preceded by 9 bytes of unknown purpose and potentially size information and an address of indirect self reference.	1
`0x04`	`RTTI_Float`	Likely contains just one byte of unknown purpose.	0
`0x05`	`RTTI_ShortString`	Likely contains just one byte of unknown purpose.	0
`0x06`	`RTTI_Set`	Contains addresses pointing to RTTI objects, preceded by likely 1 byte of size information. Followed by 3 bytes of unknown data.	0
`0x07`	**`RTTI_Class`**	Contains data like parent class, namespace, property information. Its layout is shown in detail in Table 8.	1
`0x08`	`RTTI_Method`	Contains multiple addresses pointing to RTTI objects with corresponding names and likely flag information. Not to be mixed up with MDTs.	0
`0x09`	`RTTI_WChar`	Contains 9 bytes of unknown purpose.	1
`0x0A`	`RTTI_AnsiString`	Contains two bytes of unknown purpose.	1
`0x0B`	`RTTI_WideString`	No additional fields other than the standard RTTI object header and footer.	1
`0x0C`	`RTTI_Variant`	No additional fields other than the standard RTTI object header and footer.	1
`0x0D`	`RTTI_Array`	Contains 8 bytes of unknown data, an address to an RTTI object, followed by 5 bytes of unknown data on 32-bit and 9 bytes on 64-bit.	1

continued on next page

Table 7 – continued from previous page.

Magic Byte	RTTI object type	Description	CL
0x0E	RTTI_Record	Contains multiple addresses pointing to RTTI objects with corresponding names and likely flag information.	0
0x0F	RTTI_Interface	Contains an address (to the base class for all interfaces), followed by 17 bytes of unknown data, a Pascal String for the namespace and 4 bytes of likely flag data.	1
0x10	RTTI_Int64	Contains 16 bytes of unknown purpose.	1
0x11	RTTI_DynArray	Contains 12 bytes on 32-bit or 16 bytes on 64-bit of unknown purpose as well as (duplicate) addresses to RTTI objects and namespace as Pascal String.	1
0x12	RTTI_UString	No additional fields other than the standard RTTI object header and footer.	1
0x13	RTTI_ClassRef	Contains an address pointing to a specific RTTI_Class object.	2
0x14	RTTI_Pointer	Contains an address pointing to another RTTI object, mostly RTTI_Record or RTTI_DynArray or RTTI_Pointer.	1
0x15	RTTI_Procedure	Contains an address to an unknown structure, which consist out of multiple addresses to RTTI objects, flag data and Pascal Strings.	1

Although the precise semantic roles of many RTTI object types remain to be conclusively determined - primarily due to time constraints - initial hypotheses regarding their purpose, as well as observed structural characteristics are included alongside each entry in Table 7. Nonetheless, the overall framing of RTTI objects appears consistent: Each begins with the aforementioned one-byte type indicator, whose value is typically observed to fall between `0x01` and `0x15`. This is followed by a Pascal String that encodes the name of the RTTI object. Thereafter, the structure may optionally include additional data of variable length. In some object types, such as *UString*, no additional data has ever been observed. This data often includes null bytes, `0xFF`-padded regions that may serve as bit masks, or addresses pointing to other RTTI structures, suggesting a system of interlinked metadata. Regardless of their internal variability, all RTTI objects may conclude in a null-byte padding, mostly preceded by `0x01` or `0x02` flag data.

With symbol recovery as the primary focus of this analysis, special attention is devoted to those RTTI object types that are directly referenced in the VMT and MDT structures. This focus narrows the field to a single object type of particular relevance: the *RTTI class object*. This object, consistently associated with the magic byte `0x07`, is detailed in Table 8. Also, Figure 6 presents an abstracted

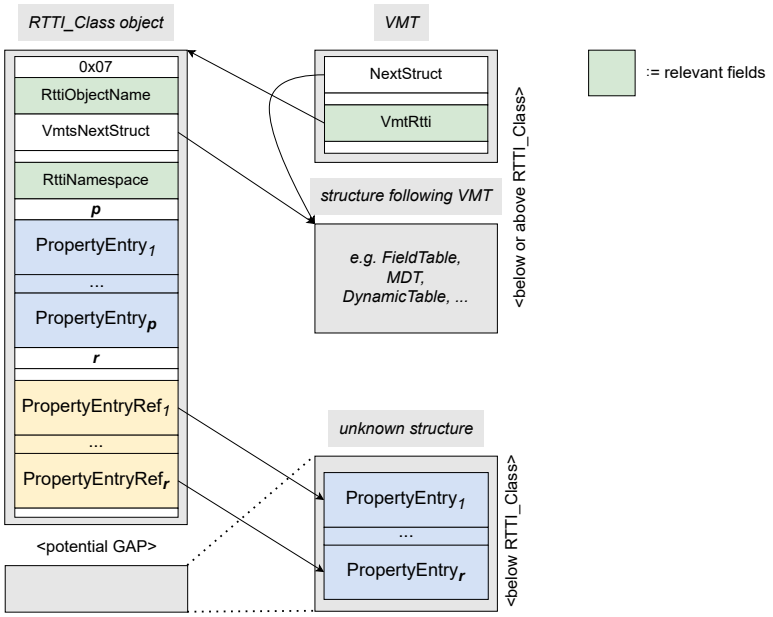

Figure 6: *Abstract illustration of the RTTI_Class layout, including its repeating* `PropertyEntry` *structures as well as its relationship to its corresponding VMT. Parts of substructures coloured in green are relevant fields for the symbol recovery task.*

visual to support a clearer understanding of the relationship between a VMT and its class object as well as the layout of its repeating substructures.

Table 8: *Layout of the* ***RTTI Class*** *structure for versions Delphi 2010 and onwards for 32/64-bit architectures. A field's offset (O) and size (S) information can be seen in the columns of the corresponding letter and architecture number. The column CL depicts the Confidence Level as introduced in Table 1.*

O'32	O'64	S'32	S'64	Fieldname	Description	CL
0x0	0x0	1	1	`RttiObjectType`	Magic byte declaring the type of RTTI object (`0x07` for RTTI_Class objects).	2
0x1	0x1	len(S)	len(S)	**`RttiObjectName`**	Name of the RTTI_Class object.	0
n.a.	n.a.	4	8	`VmtsNextStruct`	Optional pointer to the structure following the corresponding VMT (same as VMT's `NextStruct` field).	2

continued on next page

Table 8 – continued from previous page.

O'32	O'64	S'32	S'64	Fieldname	Description	CL
n.a.	n.a.	4	8	`DirectAncestorClass`	Optional pointer to the parent class in the RTTI object's inheritance chain (indirect reference).	2
n.a.	n.a.	2	2	unknown	unknown; likely flags	0
n.a.	n.a.	len(S)	len(S)	**`RttiNamespace`**	The *namespace* of the RTTI class as a Pascal String.	2
n.a.	n.a.	2	2	`NumOfPropertyEntries`	Number of `PropertyEntryRef` substructures $p \geq 0$ following this field.	2
n.a.	n.a.	n.a.[5]	n.a.[6]	$\texttt{PropertyEntry}_1,\ldots,p$	Optional concatenation of *PropertyEntry* substructures embedded inside the RTTI class structure. Each *PropertyEntry* substructure contains the following in order: Up to three addresses (which may be replaced with unknown data), 2 bytes of unknown data, another optional address, 8 bytes of unknown data, concluded by a Pascal String. The addresses may point to code or RTTI objects. The strings embody the names of the class's properties. These substructures only exist if $p > 0$.	1
n.a.	n.a.	1	1	`NumOfPropertyEntryRefs`	Number of `PropertyEntryRef` substructures $r \geq 0$ following the *next* field.	2
n.a.	n.a.	2	2	`RefSeparator`	Two separating bytes `0x0002`.	1

continued on next page

Table 8 – continued from previous page.

O'32	O'64	S'32	S'64	Fieldname	Description	CL
n.a.	n.a.	$r \times 7$	$r \times 11$	`PropertyEntryRef`$_1, ..., r$	Optional concatenation of *PropertyEntryRef* substructures, containing an address and three bytes of flag information each, which is mostly `0x020002`. The addresses of these substructures point to specific `PropertyEntry` substructures following this RTTI class structure. These substructures only exist if $r > 0$.	2
n.a.	n.a.	dyn	dyn	`Padding`	Optional null byte padding.	2

Like all RTTI objects, the *RTTI class object* begins with a type-identifying magic byte and a Pascal String denoting the object's name, encoded within the fields `RttiObjectType` and `RttiObjectName` respectively. In this case, the *class* object is uniquely associated with the magic byte `0x07`. Immediately following this header, the structure contains two addresses: The first is a direct link to the structure that succeeds the corresponding VMT, and it thus mirrors the value held in the VMT's `NextStruct` field. The second address is an indirect reference to the class's direct ancestor, if one exists, and thereby encodes class inheritance information. Two bytes of currently undetermined purpose follow this pair of addresses.

Beyond these, the RTTI class object reveals a final and essential component necessary for the task of symbol recovery - one that addresses the two remaining requirements not yet satisfied by the previously discussed structures. Specifically, these are *Requirement 2: FQN Recovery* and *Requirement 3: FQN Mapping*, both concerned with the reconstruction of fully qualified function names. Although information about function names and their classes is already available, the enclosing namespace had not yet been fully recoverable. This missing piece is now provided by the `RttiNamespace` field, which encodes the namespace of the RTTI class object itself as a Pascal String. By concatenating the class name (retrieved from the RTTI object name) with this namespace, the complete namespace of any function defined within the class can be constructed. This, in combination with the function name found in the MDT's specific *MethodEntry*, yields the function's FQN, thereby fulfilling *Requirement 2: FQN Recovery*. Moreover, the extraction of a function's FQN can be systematically repeated across all methods, because it is possible to deterministically navigate from any given *MethodEntry* to the corresponding RTTI class object. This direct mapping capability ensures that *Requirement 3: FQN Mapping* is also met. Hence, all previously defined pieces of information necessary for effective symbol recovery from Delphi binaries are now deterministically available, at least under the assumptions presented in this chapter.

[5]Size can be calculated as: $p \times 26 + \text{len}(S)$.
[6]Size can be calculated as: $p \times 42 + \text{len}(S)$.

Following the `RttiNamespace` field, two optional concatenations of RTTI-related substructures may appear. Each of these is preceded by a size field that specifies the number of contained substructures. The first of these is a list of *PropertyEntry* substructures. These are fixed in length - 30 bytes on 32-bit systems and 42 bytes on 64-bit - and are composed of a mixture of addresses and bytes whose exact semantics remain partially undetermined. Their structure typically includes up to three optional addresses, followed by two bytes of unknown purpose, another optional address, eight further bytes of unknown data, concluded by a Pascal string. Some of these addresses appear to point to other RTTI structures, whose names often match the names of corresponding class members like properties. In other cases, the addresses may link to code segments. Occasionally, however, the values at these positions do not correspond to valid addresses at all, and their function remains entirely unclear. Interestingly, *PropertyEntry* concatenations are not exclusively embedded within the RTTI class structure, as Figure 6 highlights. They may also be found outside the structure, either immediately following it or at an offset beneath it. Despite the variance in location, no clear difference in semantic purpose between the two groups of *PropertyEntry* structures has been observed thus far.

The second and final list within the RTTI class object is composed of *PropertyEntryRef* substructures. This sequence is prefaced by a field that contains the number of these substructures, labelled `NumOfPropertyEntryRefs`. However, this second concatenation is separated from the actual list of entries by two intermediate bytes, which are consistently filled with `0x0002` across all observations. Thus, they may serve as markers analogous to the `Separator` field of the MDT *MethodEntry*. Nevertheless, the assumption of a default flag value appears to be a plausible alternative. Each *PropertyEntryRef* structure contains a single address alongside three bytes of what is likely flag or type indicator data - mostly the byte sequence `0x020002`. The address itself points to a *PropertyEntry* structure located outside of the RTTI class structure, creating an indirection analogous to the one observed in the relationship between *MethodEntryRef* and *MethodEntry* substructures in the MDT. The structure concludes with an optional null-byte padding sequence.

3.7 Format Changes

As indicated in the captions of the previous layout tables, the analysed format structures remain unchanged across all Delphi versions from *Delphi 2010* up to and including *Delphi 12*. This finding results from a systematic examination of format evolution across successive compiler generations. The methodological framework of this analysis comprised four sequential phases: the collection of suitable binaries, comparative structural analysis, a compatibility cross-check using IDR, and a contextual historical evaluation.

The initial phase involved gathering a representative corpus of executables compiled with various Delphi versions. This proved challenging due to the declining popularity of Delphi in contemporary software development and the limited availability of older installation packages [Jan24; Cas24]. At present, only versions of *Delphi 10.3* and later are publicly accessible via the Embarcadero website, which poses a significant limitation for efforts involving legacy binaries [Emb25c]. This restriction is due to the fact that the compiler version in Delphi cannot be set independently within the programming environment but is instead intrinsically tied to the installed version of *RAD Studio*, Delphi's official IDE. As Embarcadero no longer offers downloads of RAD Studio 10.2 or

earlier, and given that this thesis commenced in early 2025, manual compilation of executables using Delphi versions prior to *Delphi 10.3* is no longer readily feasible.

To extend coverage to earlier versions, publicly available malware repositories, such as `malshare[.]com`, were consulted, given that numerous malware families are known to have been authored in Delphi [Wil24]. Executables from families such as *Kiron*, *Cosmic Banker*, *Culebra*, *DanaBot*, *DarkGate*, *Amadey*, *Rekram*, *GRenam*, and *TotalSpy* were identified and analysed. Version identification was facilitated by the Detect It Easy (DIE) tool. Based on its output, 64-bit versions of Kiron and CosmicBanker malware samples were attributed to *Delphi 11* and *Delphi 10.3*, Culebra samples for both 32-bit and 64-bit to *Delphi XE2* through *Delphi XE4*, DanaBot samples to *Delphi 10.2*, DarkGate to *Delphi 2* and *Delphi 3*, Amadey to *Delphi 6*, Rekram to *Delphi 2006*, GRenam to *Delphi 2006*, and TotalSpy to *Delphi 5*. These binaries formed the empirical basis for the ensuing structural analysis, whose results are summarised together with historical release dates of various Delphi versions in Table 9.

Table 9: *Summary of historic Delphi versions, their release years (abbreviated with a preceding apostrophe), and the identified format. "16" refers to the 16-bit-only format of Delphi 1, likely different but unconfirmed due to a lack of analysable samples. "o" denotes the older internal structure, while "n" marks the current format for 32-bit executables; "n/n" indicates that both 32- and 64-bit executables share this format, introduced with 64-bit support in Delphi XE2.*

Version	1	2	3	4	5	6	7	8	2005	2006	2008	2009	2010	XE
Release	’95	’96	’97	’98	’99	’01	’02	’03	’04	’05	’07	’08	’09	’10
Format	16	o	o	o	o	o	o	o	o	o	o	o	n	n
Version	XE2	XE3	XE4	XE5	XE6	XE7	XE8	10	10.1	10.2	10.3	10.4	11	12
Release	’11	’12	’13	’13	’14	’14	’15	’15	’16	’17	’18	’20	’21	’23
Format	n/n	n/n	n/n	n/n	n/n	n/n	n/n	n/n	n/n	n/n	n/n	n/n	n/n	n/n

In the second phase, a comparative structural analysis was conducted to identify correspondences and divergences relative to the format layout produced by *Delphi 12*. It was found that all Kiron, CosmicBanker, DanaBot and Culebra samples conformed structurally to the modern format specification. Given that the Culebra samples are associated with the time frame of *Delphi XE2* to *Delphi XE4* (circa 2011–2013), this suggests that no significant alterations were made to the relevant format structures between *Delphi XE2* and *Delphi 12* - at least insofar as standard VCL applications in release mode are concerned.

A markedly different picture emerged for executables compiled with older versions such as *Delphi 2006* (GRenam) and *Delphi 2* (DarkGate). These exhibited a series of structural deviations across various structures. In these versions, VMTs themselves are considerably smaller (76 bytes on 32-bit) and lack entries for the *Equals*, *GetHashCode*, and *ToString* class methods, which are not merely NULL but entirely absent. MDTs are only present when the VMT corresponds to a custom user created form, as the presence of defaulting names for Delphi forms implies. In addition, the format of the MDTs is far less complex when compared to the one of *Delphi 2010* and onwards. An MDT consists of a concatenation of method entries, preceded by information about their number. Each of these substructures comprises information about its own length, an address pointing to a function’s entry point and a Pascal string representing its name (not an FQN). This layout is provided in detail in Table 10.

Table 10: *Layout of the MDT structure for Delphi versions Delphi 2 to Delphi 2009 (32-bit only, since 64-bit is not applicable). The column* CL *indicates the Confidence Level as defined in Table 1.*

Offset	Size	Field	Description	CL
0x0	2	`NumOfOldMethodEntries`	Number of $m'(\geq 0)$ *OldMethodEntries*.	2
0x2	n.a.[7]	$\texttt{OldMethodEntry}_{1\ldots m'}$	m' consecutive *OldMethodEntry* substructures, each containing 2 bytes stating its length, an address pointing to a function's entry point and a Pascal String denoting the function's name.	2

Another difference can also be observed in the RTTI class structures. The general shape of the structure corresponds broadly to the specification in Table 8. Similarly to the VMT though, it misses some of its fields. Specifically, all three fields related to the *PropertyEntryRef* substructures as well as the concluding null byte padding are totally absent. Furthermore, the `RttiNamespace` field appears to reference only the specific module name rather than the full namespace hierarchy used in modern versions (e.g. *System* instead of *System.SysUtils*). Lastly, the meaning and function of the `DirectAncestorClass` field likewise remain ambiguous, as it is uncertain whether the inheritance model in earlier Delphi versions exhibited the same hierarchy and semantics as it does today.

Given these structural deviations, it is of interest to investigate the extent to which the previously defined requirements for symbol recovery remain applicable to earlier Delphi versions, specifically those spanning from *Delphi 2* to *Delphi 2009*. Starting from a valid VMT, it is possible to identify and extract the associated MDTs - provided such structures exist in the given binary - as well as the corresponding RTTI class information. Within the MDTs, the method entries appear in the form of address-name pairs, where each function's entry point is directly mapped to its name as a Pascal string. This allows for the reliable extraction of symbolic labels for at least a subset of methods and, as such, fulfils *Requirement 1: Function Location*, while also partially satisfying *Requirement 2: FQN Recovery* and *Requirement 3: FQN Mapping*. However, the available RTTI class objects in these earlier Delphi versions do not preserve complete namespace information. The field that, in modern versions, would encode the fully qualified namespace appears to contain only the local unit or library name, thereby failing to meet *Requirement 2: FQN Recovery* and, by consequence, also *Requirement 3: FQN Mapping*, which presupposes a correct and complete class hierarchy. Furthermore, the older MDTs lack embedded substructures or metadata from which return types or parameter lists could be inferred. As a result, none of the function signature information that is recoverable in modern binaries can be extracted through these tables alone. This shortcoming directly contravenes *Requirement 4: Parameter Types*, *Requirement 5: Return Types*, and *Requirement 6: Function Signature*. Hence, the mechanisms proposed and evaluated in this thesis prove insufficient for symbol recovery in Delphi binaries compiled with versions up to and including *Delphi 2009*. An alternative means of acquiring the missing semantic information - either through emulation, dynamic analysis, or heuristic pattern matching - remains an open challenge and was not examined in detail due to the time constraints of this study.

[7] Size can be calculated as: $m' \times (6 + \text{len}(S))$

As a third step, a compatibility analysis was conducted using the IDR tool. IDR can be used to analyse Delphi executables according to knowledge bases which are tailored to specific Delphi versions. The tool supports manual and automatic detection of Delphi versions, ranging from *Delphi 2* to *Delphi XE4*. The core idea underlying this step is as follows: if a binary compiled with Delphi version α exhibits a format structure that is incompatible with the analysis mode manually configured for version β - such that IDR's analysis fails - then it can be inferred that the internal format of Delphi version α differs from that of version β. It was observed that the *TotalSpy* sample could not be analysed successfully under any IDR configuration corresponding to *Delphi 2010* or later. In contrast, viable results were obtained using any version from *Delphi 2* through *Delphi 2009*. The *GRenam* sample exhibited identical behaviour. These findings further indicate a substantial structural change in the binary format around the transition to *Delphi 2010*.

The final phase involved contextualising these technical findings in relation to Delphi's historical development. The alignment of the observed differences and IDR's compatibility analysis with the release of *Delphi 2010* strongly suggests that significant changes to the binary structure were introduced shortly after Embarcadero Technologies, Inc. took over Delphi development from Borland Software Corporation. These structural modifications may be attributable to the adoption of *extended RTTI mechanisms*, which was featured in *Delphi 2010*'s release notes [Emb10]. However, a definitive confirmation remains elusive due to the absence of binaries compiled with *Delphi 2009* or *Delphi 2010*.

For the sake of completeness, it is important to note that throughout Delphi's long history, support for 32-bit- and 64-bit-compilation was introduced at specific milestones. Hence, the first real fundamental structural break occurred with the transition from *Delphi 1* (16-bit) to *Delphi 2* (32-bit). However, due to the unavailability of representative *Delphi 1* executables, this earlier discontinuity could not be examined within the scope of this study. Moreover, support for 64-bit binaries was only introduced with *Delphi XE2*, which is why Table 9 refers to the 64-bit format only from this version onwards.

To sum up, this chapter presents the first comprehensive public documentation of Delphi's binary format for Windows executables (32-bit and 64-bit) from *Delphi 2010* through *Delphi 12 Athens*. Focusing on VCL GUI programs, console apps and DLLs, it shows how substantial metadata are embedded directly in the `.text` section, mainly for the purpose of enabling symbol recovery. Three core structures have been reconstructed:

- VMTs: Fixed-size tables (88 bytes on 32-bit, 200 bytes on 64-bit) containing pointers to auxiliary structures (e.g. RTTI objects, MDTs).
- MDTs: Hierarchical nested structures consisting out of *MethodEntries* (data blocks containing information like function address, name, return type and a count of parameters), *MethodEntryRefs* (pointing to entries) and *ParamEntries* (parameter-type RTTI pointer plus name).
- RTTI objects: Twenty-one RTTI types have been identified, of which the RTTI_Class (magic byte `0x07`) provides namespace information (pivotal for recovering FQNs).

Together, these findings demonstrate that Delphi executables retain rich symbolic information - class names, namespaces, method signatures and parameter details - that can be systematically extracted to satisfy all requirements for static symbol recovery.

4 Tool Development

Having established the relevant structures and fields essential for theoretical symbol name recovery in Delphi executables, this chapter now focuses on the design decisions underlying the development of a tool that enables such recovery in practice. By the end of this chapter, the reader will understand the rationale for choosing Ghidra as the primary tooling environment, the heuristics involved in locating VMTs within the binary - most notably the scanning of the `.text` section for fixed forward jump distances supplemented by sanity checks to minimise false positives - as well as further internal mechanisms relevant to the implementation of the tool.

As established in the preceding chapter, all requirements for effective symbol name recovery are met for Delphi versions *Delphi 2010* through *Delphi 12 Athens*. However, due to the lack of compliance with the defined requirements in analysed samples of earlier Delphi versions and due to time constraints, this chapter focuses exclusively on executables compiled with said versions.

In order to ensure the tool's practical relevance and usability for reverse engineering, it must be implemented as a script or plugin for an established reverse engineering framework. Such tools include *IDA Pro*, *Ghidra*, and *Binary Ninja*. Given the scope and constraints of this thesis, the implementation had to be confined to a single framework. IDA Pro was excluded from consideration due to licensing costs associated with scripting and script execution. Binary Ninja was similarly ruled out, as access to its API is also bound to financial restrictions. By contrast, Ghidra provides an open-source alternative that offers a comprehensive and free plugin and scripting interface. In addition, the update released in February 2025 introduced native support for *CPython 3* through *PyGhidra*, along with integration for Visual Studio Code, hence significantly enhancing the development workflow for custom scripts [Nat25]. For these reasons, Ghidra was selected as the tooling environment of choice for the implementation of the symbol name recovery tool presented in this thesis.

The first objective of the developed tool is the reliable identification of VMTs, as they represent the structural hub for all relevant file format artefacts discussed in Chapter 3. Various approaches to identifying VMTs have been explored in related work, each with its own advantages and limitations. For example, `DelphiHelper` relies on pre-existing knowledge bases derived from the IDR decompiler, the de facto standard tool for analysing Delphi binaries [ESE24]. It further leverages the presence of VCL-specific constructs, notably by scanning the `.text` section for the occurrence of the `CreateForm` string, which is a very typical symbol name at the entry point of Delphi executables [ESE24]. While effective in many cases, this approach fails to generalise across Delphi binaries that do not employ the VCL framework, such as console applications. Another tool, `IDA-For-Delphi`, pursues a dynamic analysis strategy, placing breakpoints during runtime to detect VMTs [Col22]. However, as one principal design goal of the present work is to enable fully static analysis without executing the binary, such methods are not suitable for adoption in this context.

A third example is `pythia`, which claims to detect VMTs in 32-bit executables by scanning for forward jumps of precisely `0x4C` bytes [NCC19]. However, this offset does not correspond to any regular or predictable jump pattern based on the current understanding of VMT structures in binaries of versions *Delphi 2010* and onwards. Moreover, the references found in the documentation

of `pythia` suggest that it was developed primarily for *Delphi 2005*, possibly explaining the difference in forward reference offsets.

Despite these limitations, the underlying concept of scanning for uniform forward jump distances proves instructive, as modern Delphi executables exhibit a high degree of structural regularity in the layout and alignment of VMTs. This makes a heuristic approach based on forward jump detection particularly viable. As established in the previous chapter, the `.text` section contains the structures themselves and the first field of a VMT, namely `NextStruct`, contains a reference to the end of the VMT. Also, the size of a VMT remains constant for a given architecture. Because of these characteristics, the proposed tool implements a static scanning mechanism over the `.text` section, seeking forward references of 88 bytes for 32-bit executables and 200 bytes for 64-bit ones. Each matching offset is treated as a *candidate* for being the start of a VMT.

This initial candidate set is then refined using a series of sanity checks designed to minimise the rate of false positives. The main principle behind these checks is to validate whether the values of specific fields of a VMT candidate fall within sensible ranges, particularly in terms of addressing. Fields that *must* contain pointers to other parts of the `.text` section are particularly well-suited for validation, since their absence or misalignment would indicate an invalid structure. *Optional* fields are less effective for validation because their absence does not inherently disqualify a structure from being a valid VMT. However, their presence can still be informative. A good example is the *NextStruct* and *MethodTable* fields: Though optional, they can contribute additional confidence when populated with plausible values, as these structures have been observed to always reside at higher addresses relative to their corresponding VMTs. These characteristics form the first set of sanity checks.

Another particularly robust component of the sanity checks focuses on the final section of currently known VMT fields, which includes references to a set of functions such as *Equals*, *GetHashCode*, and others. Out of the eleven documented function references, only one field is optional, namely *SafeCallExceptionMethod*. To minimize chances of coincidentally aligned valid addresses, a VMT candidate is discarded if at least one of these ten fields point outside the `.text` section.

Although this approach is not entirely unerring, it offers several advantages and aims to mitigate its limitations: Firstly, it operates purely through static analysis. Secondly, it ensures exhaustive coverage of all VMTs within the executable, thanks to their consistent layout. Lastly, any misleading VMT candidates introduced by the heuristic are attempted to be filtered out through a set of targeted sanity checks.

After applying the sanity checks, the tool operates on a reduced set of candidate VMTs. Starting at each of these addresses, it traverses the data structures as described in the previous chapter to locate the relevant fields and attempts to extract meaningful information. This traversal, data access, extraction and architecture identification are all implemented using Ghidra's API. Finally, the tool attempts to apply all recovered symbol names to their corresponding functions using Ghidra's API. In addition, it performs a basic form of symbol type recovery: For instance, it creates pointer data types based on the fully qualified names (FQNs) of recovered RTTI class names which represent return and parameter types. Where a direct mapping exists between recovered

type names and Ghidra's built-in types - such as `Boolean`, `void`, or `Integer` - the tool assigns the appropriate native type instead.

5 Evaluation

This chapter evaluates the efficacy of the symbol recovery tool for Delphi executables as presented in Chapter 4. The principal method involves comparing counters implemented directly within the recovery script against decompilation metadata retrieved via the Ghidra API. The evaluation is conducted on a heterogeneous dataset comprising real-world software, malware samples, and handcrafted Delphi executables. Following the discussion of methodology and dataset composition, the reader will be presented with insights into the impact of Ghidra's auto-analysis feature and the tool's capabilities and limitations, including the central finding that nearly half of all symbols within real-world Delphi malware samples could be successfully recovered.

The central evaluation methodology hinges on Ghidra's API, particularly its ability to enumerate function definitions in decompiled binaries. The symbol recovery tool internally maintains counters for function names, return types, parameter names and parameter types. These counters tally the number of successfully resolved symbols during execution. Upon completion, they are each divided by the total number of functions reported by Ghidra. Assuming accurate decompilation on Ghidra's side, this ratio is used as a proxy for the tool's accuracy of functions, defined as the proportion of functions for which symbolic information was recovered and applied. Separate accuracy values are calculated for each relevant component of a function signature, including names, return types, parameter types, and their respective namespaces, where applicable.

The evaluation was performed on a total of seventeen Delphi executables, comprising six handcrafted binaries, two legitimate software binaries, and nine real-world malware samples. The handcrafted executables were analysed in both 32-bit and 64-bit formats. The two legitimate samples - *Acrylic DNS Proxy* and the current release of *RAD Studio* - are only available in 32-bit form. The malware corpus spans both 32-bit and 64-bit architectures and includes samples from families such as *Kiron*, *CosmicBanker*, *DanaBot* and *Culebra*. The legitimate software samples were identified using DIE signatures, and the same methodology was applied to determine the Delphi version of the malware samples. The handcrafted binaries include "*VclMinimum*", the default GUI project template in RAD Studio; "*Concat*", a console application with a deliberate inclusion of theproblematic `LStrCatN` function often encountered in Delphi reverse engineering contexts; and "*VclEnhanced*", a more feature-rich variant of *VclMinimum* incorporating extensive interaction with VCL components. Overall, the evaluation dataset covers at least seven different Delphi compiler versions. The associated Table 11 outlines each sample in detail, including their types of origin, the malware hashes, the detected Delphi versions, and a binary classification of the tool's success per executable.

Table 11: *Overview of the set of executables used for evaluation, including their origin, architecture, Delphi version, and whether the tool was able to recover symbol names.*

Executable	Type	Architecture	Delphi Version	Result
VclMinimum-32	crafted	32-bit	Delphi 12 Athens	success
VclMinimum-64	crafted	64-bit	Delphi 12 Athens	success
Concat-32	crafted	32-bit	Delphi 12 Athens	success
Concat-64	crafted	64-bit	Delphi 12 Athens	success
VclEnhanced-32	crafted	32-bit	Delphi 12 Athens	success
VclEnhanced-64	crafted	64-bit	Delphi 12 Athens	success
RAD Studio	legitimate	32-bit	Delphi 10.3 Rio	success
Acrylic DNS Proxy	legitimate	32-bit	Delphi 7	fail, old format
Kiron[a]	malware	64-bit	Delphi 11 Alexandria	success
CosmicBanker[b]	malware	64-bit	Delphi 10.3 Rio	success
DanaBot[c]	malware	32-bit	Delphi 10.2 Tokyo	success
DanaBot[d]	malware	32-bit	Delphi 10.2 Tokyo	success
DanaBot[e]	malware	32-bit	Delphi 10.2 Tokyo	success
DanaBot[f]	malware	32-bit	Delphi 10.2 Tokyo	success
Culebra[g]	malware	32-bit	Delphi XE2-XE4	success
Culebra[h]	malware	32-bit	unknown	success, if unpacked
Culebra[i]	malware	64-bit	unknown	fail, packed

[a] SHA-256: `77bac49729638d6b0fc0dc5ae60b5cca55d45ecc93342d49ff1d93e36d9ffa9c`
[b] SHA-256: `5b257d4393aa3c464edf5201f7507c263d698aa7e555c3d8d77270165ba00a5e`
[c] SHA-256: `88d82df599a0e7cd21f8e98b717c7365544800d3c466ad8deb8020d2ed488b55`
[d] SHA-256: `52e071839a2713f39d2c718b2961f17be85a8646fcf4d461ba06973f5237bca0`
[e] SHA-256: `3e6f4bcce39757b1869e132238d9fe73a019ab155e10ac2f4802f67486713640`
[f] SHA-256: `28d446af80937f858098c4496863123c26f74ed3ca6ccc0d6fb3682a7ff64156`
[g] SHA-256: `e8eae2afa57ff0b348342fa4ba5b37165ecf07458e859418ed19d5f00df02271`
[h] SHA-256: `bd5c8d44b1ec1a4ba712aa7e853f89d934bd459a2e502d4fce83b4be9cc90ca3`
[i] SHA-256: `f8711e4437e5576ed3e016a3fe160f9b1f44d4c82a214f430e5c5b6c4e5ac6f7`

Table 11 shows that, out of the seventeen executables listed in the table, fifteen exhibited at least partial symbol recovery. One of the samples, a 32-bit compiled executable belonging to the Culebra malware family, was packed and required unpacking prior to successful evaluation. This was achieved via the publicly available *UnpacMe* service provided by *Open Analysis*, which identified *VMProtect* as the packing mechanism. However, another sample[1] from the same family, this time compiled as a 64-bit binary, could not be successfully unpacked using the same service. Due to time constraints, manual unpacking was deprioritised and was excluded from the table for better readability. This case highlights a limitation of the tool: the packing process often modifies section names in PE files, rendering the tool unable to locate the original *.text* section from which it is designed to extract symbol information.

[1] SHA-256: `f8711e4437e5576ed3e016a3fe160f9b1f44d4c82a214f430e5c5b6c4e5ac6f7`, unidentified version due to packing

Table 12: *Effect of different modes of operation ("Mode") on results across various metrics and executables. The first column indicates whether the executable has not been analysed ("!a"), has been auto-analysed by Ghidra ("a"), or has been auto-analysed and subsequently processed once by the tool ("a+t"). The remaining columns, in order, report the following: the number of recovered VMTs; the number of functions identified by Ghidra before and right before the end of tool execution; the number of recovered function names; the number of functions for which the FQN has been recovered; the number of recovered return types; and the number of functions for which all parameter types and names have been recovered (="sets").*

Executable	Mode	#VMTs	#pre funcs	#post funcs	#recov. func'names	#recov. FQNs	#recov. ret.	#recov. param-sets
VclMinimum-32	!a	817	52	4903	4851	4851	4851	4378
	a	817	6878	9796	4846	4846	4846	4373
	a+t	817	9799	9799	4846	4846	4846	4373
VclMinimum-64	!a	840	52	5059	5007	5007	5007	4547
	a	840	8762	10956	5007	5007	5007	4547
	a+t	840	10970	10970	5007	5007	5007	4547
Kiron[a]	!a	2589	66	13280	13214	13214	13214	11727
	a	2589	18112	24533	13214	13214	13214	11727
	a+t	2589	24622	24622	13214	13214	13214	11727
Culebra[b]	!a	1116	85	5205	5121	5121	5121	4585
	a	1116	10415	12924	5105	5105	5105	4575
	a+t	1116	12939	12939	5105	5105	5105	4575

[a] SHA-256: `77bac49729638d6b0fc0dc5ae60b5cca55d45ecc93342d49ff1d93e36d9ffa9c`
[b] SHA-256: `e8eae2afa57ff0b348342fa4ba5b37165ecf07458e859418ed19d5f00df02271`

The second reason for failure concerns Delphi versions older than *Delphi 2010*. *Acrylic DNS Proxy*, for instance, was compiled using *Delphi 7*. As outlined in Section 3.7, such legacy versions encode type and class metadata differently from modern Delphi binaries. Since the symbol recovery tool depends on the current file format, versions compiled using *Delphi 2009* or earlier are currently out of scope. This limitation also applies to various other malware samples, including those of the families *GRenam*[2], *Rekram*[3], *Amadey*[4], and *DarkGate*[5], which are also not included in the evaluation tables for the sake of visual clarity.

Next, to investigate whether Ghidra's auto-analysis feature influences recovery accuracy, each executable was evaluated in three configurations: immediately after a fresh import, after running auto-analysis, and finally after running auto-analysis followed by executing the tool once. The number of total functions was recorded both at the start and end of the tool's execution, revealing how Ghidra's API and internal states affect these values. Table 12 illustrates the impact of these configurations on various numerical metrics across a selection of four Delphi executables from the dataset.

[2] SHA-256: `fd00f7c92f577fa6b32f627416d490f69f84cb8c84fc8970e22b6ee5e559378c`, *Delphi 2006*
[3] SHA-256: `c6057052e628c82f999936fd7add770c285ef402106ddb8c67d2276869a012f0`, *Delphi 2006*
[4] SHA-256: `84b0d1229c5954319112c0960b74ed3e405b7cda7020c78bbfe9147dd90355c8`, *Delphi 6*, *Delphi 7* or *Delphi 2005*
[5] SHA-256: `8847f8a1621525079a8e31c05d0a6140ffa756bdff0cc9a57c31c3b3c604848c`, *Delphi 2* or *Delphi 3*

An examination of the results reveals that the combination of the pre-execution function count and the number of recovered function names closely aligns with the post-execution total, with minor discrepancies likely due to overlapping detections. Moreover, runs that included auto-analysis consistently produced stable recovery values, independent of subsequent executions. Moreover, the tool performs robustly regardless of whether auto-analysis is enabled, and in some cases, even achieves marginally better results without it. Nevertheless, auto-analysis was enabled during the main evaluation runs, as it provides a more realistic function baseline and avoids artificially inflated accuracy values due to undercounting total functions.

Another observation is the consistent recovery of function names, namespaces, and return types per run. Whenever a function's *MethodEntry* structure yields a valid name, its enclosing RTTI class structures are also reliably available, enabling accurate namespace resolution. Likewise, recovery of additional namespaces, resulting in FQNs of functions, has proven to be error-free across all observations if a function name is successfully recovered. Interestingly, the values for function FQNs and return types are also identical across all successful runs. While this could suggest an inherent property of the tooling's inner workings, manual inspection revealed that this alignment is presumably coincidental. Therefore, for the sake of space and clarity, the columns for the accuracy values of function names, their namespaces, and return types are merged in subsequent tables.

Lastly, the parameter *set* recovery metric reflects the number of functions for which *all* parameter tuples - comprising both the data type *and* parameter name - were successfully recovered. Across different configurations, this metric remains largely stable. While certain cases - such as `VclMinimum-32` and `Culebra` - show minor variations, typically involving up to 10 additional functions, these fluctuations are minimal and do not significantly affect the evaluation's conclusions.

Table 13 presents the results for all executables for which a successful run of the symbol recovery tool was performed. For each binary, it states the measured accuracy for recovered function names (FQNs), return types, and parameter sets.

Starting with the handcrafted executables, it is immediately apparent that the *Concat* executables - compiled for both 32-bit and 64-bit architectures - yield the worst results by a significant margin, with accuracy values for all metrics remaining below 10%. In contrast, both *VclMinimum* and *VclEnhanced* reach accuracy values of approximately 45-50% for recovered function FQNs and return types, and 41-45% for parameter sets. Given that the primary difference between *Concat* and the other two samples lies in the presence of graphical functionality (notably the use of the VCL framework), these results suggest that Delphi compilers tend to include more metadata related to the VMTs of graphical libraries such as the VCL. Besides comparable accuracy values, the *VclMinimum* and *VclEnhanced* samples also exhibit strikingly similar numbers of recovered functions per architecture. This observation points to the possibility thatonce the VCL framework is introduced, the Delphi compiler appears to embed the bulk of relevant metadata, and further graphical additions do not substantially alter the recovery landscape. Interestingly, the *VclMinimum* and *VclEnhanced* samples contain around ten times more functions than console applications. This highlights the significant overhead introduced by graphical components.

At the same time, the number of functions detected in the *RAD Studio* sample is roughly 71% lower than in *VclMinimum*. Despite including visual components, it performs the worst among all

Table 13: *Results for successful runs after a clean import and auto-analysis of the executables. The columns are, in order: the name of the analysed executable, the total number of functions it contains, the number of recovered fully qualified function names, the number of functions for which all parameter types and names have been recovered (="sets"), and the accuracy values for function FQNs and return types, and parameter sets. The column for recovered return types has been omitted, as their count matches that of the recovered function FQNs, in order to save space.*

Executable	#funcs	#rec_func / #rec_ret	#rec_param	Acc_func / Acc_ret	Acc_param
VclMinimum-32	9796	4846	4373	49,47 %	44,64%
VclMinimum-64	10956	5007	4547	45,70 %	41,50%
Concat-32	733	66	53	9,00 %	7,23%
Concat-64	831	70	57	8,42 %	6,86%
VclEnhanced-32	9699	4847	4374	49,97 %	45,10%
VclEnhanced-64	10966	5007	4547	45,66 %	41,46%
RADStudio	2873	896	703	31,19 %	24,47%
Kiron	24533	13214	11727	53,86 %	47,80%
CosmicBanker	16033	8695	7513	54,23 %	46,86%
DanaBot	5295	2803	2481	52,94 %	46,86%
DanaBot	5738	3055	2679	53,24 %	46,69%
DanaBot	13737	5341	4717	38,88 %	34,34%
DanaBot	10763	5576	4926	51,81 %	45,77%
Culebra	12924	5105	4575	39,50 %	35,40%
Culebra, packed	9053	2857	2472	31,56 %	27,31%

such samples, with only 31.19% of function FQNs and return types, and 24.47% of parameter sets recovered.

Turning to the malware samples, two stand out: *Kiron* and *CosmicBanker*, with over 24,500 and 16,000 functions respectively. These samples achieve the best overall results, with around 54% accuracy for function FQNs and return types, and 46-48% for parameter sets. Three of the *DanaBot* samples follow closely, each attaining roughly between 52% and 53% accuracy for function FQNs and roughly 46-47% for parameter sets. However, one particular *DanaBot* sample with the highest total number of functions shows noticeably lower results, with only 39% accuracy for function names and 34% for parameter sets. This serves to caution against interpreting a high function count as an indicator of improved recovery performance. Another instructive case is that of the not packed *Culebra* sample. It was compiled with one of the oldest Delphi versions that still uses today's file format. For this executable, accuracy scores between 35% and 40% have been achieved. In contrast, its packed variant exhibited an 8% drop in accuracy after unpacking.

It is also worth highlighting that both `RAD Studio` and `CosmicBanker` are compiled with Delphi 10.3 Rio, yet exhibit vastly different results. This suggests that symbol recovery performance is

```
void entry(void)

{
  FUN_00411798(&DAT_005e2cec);
  Vcl::Forms::TApplication::Initialize(*
    (TApplication **)PTR_DAT_005f2894);
  FUN_005d8bd4(*(undefined4 *)
    PTR_DAT_005f2894,1);
  Vcl::Forms::TApplication::CreateForm(*
    (TApplication **)PTR_DAT_005f2894,(
    TComponentClass *)&PTR_LAB_005e2af4
    ,(TApplication *)PTR_DAT_005f275c);
  Vcl::Forms::TApplication::Run(*(
    TApplication **)PTR_DAT_005f2894);
                    /* WARNING:
    Subroutine does not return */
  FUN_00409f8c();
}
```

```
void EntryPoint(void)

{
  FUN_00411798(&DAT_005e2cec);
  TApplication.Initialize(*(undefined4
    *)Application);
  FUN_005d8bd4(*(undefined4 *)
    Application,1);
  Vcl::Forms::TApplication::CreateForm(
    *(undefined4 *)Application,&
    PTR_LAB_005e2af4,gvar_005F275C);
  TApplication.Run(*(undefined4 *)
    Application);
                    /* WARNING:
    Subroutine does not return */
  @Halt0();
}
```

Figure 7: *Ghidra decompilation of a typical Delphi application's entry point. Left: This thesis' tool; Right: After IDR+dhrake.*

more strongly influenced by the nature and content of the original source code rather than by compiler version alone. Furthermore, across the dataset, no clear trend emerges that would link lower tool performance directly to older compiler versions. This is likely due to the limited version variety in the evaluation dataset, as most samples are associated with only the latest few versions of Delphi.

A brief discussion on architecture: while the handcrafted 64-bit samples consistently perform slightly worse than their 32-bit counterparts, this is not a generalisable trend. On the contrary, the best-performing binaries in the entire dataset, namely *Kiron* and *CosmicBanker*, are both 64-bit malware samples. This suggests that architecture alone is not a primary limiting factor.

However, some caveats must be acknowledged. Firstly, this evaluation measures transformation coverage, not semantic correctness. Given the lack of original source code - especially for the malware samples - there is no scalable and time-efficient method to assess whether the recovered names semantically match their original intentions. In cases where manual validation was feasible, no severe discrepancies were identified, but this remains a fundamental limitation inherent to all decompilation-based recovery efforts.

Secondly, it should be noted that Ghidra's own decompiler behaviour can lead to small inconsistencies in the function counts it reports. For example, the number of total functions reported for the *Kiron* sample deviated by around 90 between two separate analysis runs under otherwise identical conditions. While this variation corresponds to less than 1% of the total functions, and thus has negligible influence on the reported accuracies, it is an artefact worth mentioning.

As a final evaluation of the tool's efficacy, a comparison to the de facto standard toolchain for Ghidra reverse engineering is conducted. Figure 7 presents the results of decompilation after executing both tools on the same entry point of the same Delphi executable, which is already shown

```
bool System::TObject::InheritsFrom(
    TObject *Self, TClass *AClass)

{
...
}

```

```
undefined4 TObject.InheritsFrom(int
    param_1, int param_2)

{
...
}

```

Figure 8: *Ghidra decompilation of a typical Delphi application's function definition. Left: This thesis' tool; Right: After IDR+dhrake.*

in Chapter 2. On the left, the results produced by this thesis' tool are shown; on the right, the decompilation output of IDR+dhrake is displayed. Following execution, it becomes evident that both tools identify approximately the same number of functions. However, the results produced by this thesis' tool are more consistent and more precise with respect to FQN and parameter type information. An examination of a function's signature, as illustrated in Figure 8, reveals that this thesis' tool successfully recovers all symbolic data. By contrast, the approach of IDR+dhrake appears to rely on unspecific, generic Ghidra data types for return types and parameter types, and fails to retrieve parameter names or the function's FQN.

In summary, the presented tool is capable of recovering non-parameter-related symbols from real-world Delphi malware samples with an average accuracy of 47%, and parameter-related symbols with 39%. Unlike many related approaches, the tool supports both 32-bit and 64-bit binaries. However, two main limitations remain. First, it struggles with console applications, likely due to their limited reliance on GUI-related libraries from which much of the extractable RTTI is derived. Secondly, support for file formats pre-*Delphi-2010* has not yet been developed. However, while the scope of this evaluation is limited by the relatively small and version-skewed dataset, the results demonstrate the practical viability of symbol recovery in contemporary Delphi malware.

6 Conclusion

This thesis set out to address the lack of effective reverse engineering tooling for Delphi executables. To this end, foundational research into the structure of Delphi's compiled format was conducted. The principal theoretical contribution lies in the identification and analysis of the layout and interrelation of three core structures: the Virtual Method Table (VMT), Method Definition Table (MDT), and runtime type information (RTTI) class structures, all of which are commonly embedded in compiled Delphi programs of versions *Delphi 2010* and later. Among these, the VMT emerges as the central anchor, referencing both MDT and RTTI data. Taken together, these structures encode extensive symbolic information, including method names and entry points, class hierarchies, namespaces, return and parameter types, instance sizes, property definitions, and more.

Building on these theoretical insights, a symbol recovery tool was developed. The tool operates by scanning the `.text` section of a given PE executable - Delphi's default compilation target on Windows - for indicative patterns of Delphi's highly regular VMT layout. Through the application of a series of sanity checks, false positives are minimised. Each candidate VMT is then deterministically traversed in accordance with the structural relationships established in this thesis, to locate and extract symbolic metadata. This information is subsequently applied to the executable using the Ghidra API, resulting in annotated functions, namespaces, parameter lists, classes, and data types that significantly enhance the readability and interpretability of the decompilation.

To the best of the author's knowledge, no prior academic work has addressed this subject. On the practical side, the only widely recognised tool is the Interactive Delphi Reconstructor (IDR), a tool specifically designed for combined usage with IDA Pro. According to its author, the tool is incomplete for 64-bit binaries. An extensive survey of community-driven tools revealed a small number of alternatives, most of which are either based on IDR, limited to IDA Pro, restricted to 32-bit binaries, or designed solely for the recovery of VMT information.

When comparing the tool presented in this thesis to IDR, both achieve similar rates of method name recovery. However, the present tool distinguishes itself in several important respects. Firstly, it supports both 32-bit and 64-bit binaries, yielding comparable results across architectures. Secondly, it reconstructs fully qualified names (FQNs) for all recovered functions, rather than merely partial identifiers. Thirdly, it creates proper class and namespace objects within Ghidra. Lastly, and most notably, it correctly annotates function parameters and return types - even when these are compound RTTI classes.

In general, an evaluation conducted on a curated set of real-world Delphi malware samples demonstrate the tool's effectiveness. On average, it recovers the fully qualified names and return types for 47% of all functions, and for 39% of all functions, the entirety of a function's corresponding parameter information (combining types and names) has been successfully recovered. Nonetheless, certain limitations remain. Most notably, the tool cannot operate on packed binaries, a common characteristic of malware samples. However, unpacking is generally considered a preliminary step in reverse engineering and is thus assumed to occur prior to the tool's application. Furthermore, the tool performs best on executables that make use of graphical libraries such as the Visual Component Library (VCL); in contrast, console-based applications yield significantly lower recovery rates, with success rates dropping to below 10% in such cases.

The most significant limitation, however, lies in the tool's scope. The focus of this thesis has been on Delphi versions from *Delphi 2010* to *Delphi 12 Athens*, covering approximately sixteen years of latest development. Due to fundamental differences in the structure of earlier versions, as outlined in previous chapters, the theoretical models developed here are not directly applicable to older executables, and the tool is currently unable to process them.

Future work should primarily aim to extend the tool's applicability to older Delphi versions. This requires a deeper understanding of legacy formats, as well as access to suitable executables. However, two major challenges impede progress in this direction. First, early versions of Delphi appear to embed significantly less symbolic metadata in their compiled output. Second, executables compiled with older versions are increasingly difficult to obtain, particularly since the current vendor, Idera Inc., no longer provides official access to legacy Delphi compilers.

Beyond version support, further refinement of the current theoretical model remains a promising direction. Several fields within the VMT, MDT and RTTI structures remain insufficiently understood. Future research into these yet unexplored areas could unlock new mechanisms for symbol recovery and thereby further enhance the capabilities of reverse engineering tools for Delphi software.

Another topic for future research is to investigate why Delphi executables contain the metadata described in this thesis in the first place. If the metadata is required at runtime, for instance, for generating stack traces, it is likely to be difficult to tamper with due to the closed-source nature of the Delphi compilation process. However, if it is not required, an adversary could potentially circumvent the analysis technique described in this thesis in specific ways. One conceivable approach would be to populate fields that contain pointers to relevant structures such as MDT with random addresses, most likely resulting in failed traversal and data extraction. A possible countermeasure to this adversarial tactic would be the construction of a signature database for Delphi functions. Since Delphi employs a single compiler model, it would be feasible to collect signatures for every function across the finite set of compiler versions. Notably, the insights developed in this thesis can be utilised to obtain a partial mapping of functions to their respective fully qualified names (FQNs) for versions of *Delphi 2010* and later. In this context, examining which Delphi libraries embed what quantity of metadata during compilation could also prove to be a valuable future research.

Ultimately, the tool developed in this thesis constitutes a much-needed entry point into the tooling landscape for a programming ecosystem that, despite its age, still remains relevant across domains after 30 years. It brings reverse engineers what is arguably the most valuable asset in their workflow: a more time-efficient workflow. Likewise, this thesis serves as a reminder to the scientific community not to overlook past technologies, and to occasionally turn back from the pull of prevailing trends in order to address neglected but enduring challenges.

Acronyms

C2

Command-and-Control. 2

COFF

Common Object File Format. 8

DHRAKE

Delphi Hand Rake. 4, 9, 40, 41

DIE

Detect It Easy. 30, 35

FQN

fully qualified name. 10, 11, 15, 16, 22, 28, 30–32, 34, 38, 39, 41, 43

IDE

Integrated Development Environment. 2, 9, 29

IDR

Interactive Delphi Reconstructor. 3–5, 9, 12, 24, 29, 32, 33, 40–42

IoC

Indicators of Compromise. 2

LLM

large language model. 1

MDT

Method Definition Table. 11, 15–23, 25, 28–32, 41, 43

PE

Portable Executable. 1–3, 7, 8, 12

RTTI

runtime type information. 6, 11, 15, 16, 22–25, 28, 29, 31, 32, 34, 41–43

VCL

Visual Component Library. 5–7, 9, 30, 32, 33, 35, 38, 42

VMT

Virtual Method Table. 5, 6, 9, 11–17, 19, 21, 22, 25, 26, 28, 30–34, 38, 41–43

WoW64

Windows on Windows. 8

References

[Bit15] Bitter, Rob: *Thoma Bravo Announces Sale of Embarcadero to Idera, Inc.* Archived version of the original Business Wire news article on June 27, 2022. 2015. URL: https://web.archive.org/web/20220627160854/https://www.businesswire.com/news/home/20151007006216/en/Thoma-Bravo-Announces-Sale-of-Embarcadero-to-Idera-Inc. (visited on 03/29/2025).

[Bri25] Bridge, Karl et al.: *PE Format - Microsoft Learn*. 2025. URL: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 02/24/2025).

[Car18] Carlos Garcia Prado, Jon Erickson: *Solving Ad-hoc Problems with Hex-Rays API*. Archived version of the original Fireeye blog post on January 21, 2022. 2018. URL: https://web.archive.org/web/20220121164818/https://www.fireeye.com/blog/threat-research/2018/04/solving-ad-hoc-problems-with-hex-rays-api.html (visited on 02/19/2025).

[Cas24] Cass, Stephen: *The Top Programming Languages 2024*. 2024. URL: https://spectrum.ieee.org/top-programming-languages-2024 (visited on 01/27/2025).

[Col22] Coldzer0: *IDA-For-Delphi GitHub Page*. 2022. URL: https://github.com/Coldzer0/IDA-For-Delphi (visited on 02/19/2025).

[Cor09] Cornelius, David: *Delphi History*. 2009. URL: https://corneliusconcepts.tech/delphi-history (visited on 03/29/2025).

[cry18] crypto2011: *IDR64 GitHub Page*. 2018. URL: https://github.com/crypto2011/IDR64 (visited on 02/19/2025).

[cry23] crypto2011: *IDR GitHub Page*. 2023. URL: https://github.com/crypto2011/IDR (visited on 02/19/2025).

[Eck22] Eckels, Stephen: *Ready, Set, Go — Golang Internals and Symbol Recovery*. 2022. URL: https://cloud.google.com/blog/topics/threat-intelligence/golang-internals-symbol-recovery/?hl=en (visited on 01/27/2025).

[Emb10] Embarcadero Technologies Inc.: *What's New in Delphi and C++Builder 2010*. Archived version of the original Embarcadero Delphi release notes on June 27, 2013. 2010. URL: https://web.archive.org/web/20130627154733/http://docwiki.embarcadero.com/RADStudio/2010/en/What's_New_in_Delphi_and_C%2B%2BBuilder_2010#Delphi_Compiler_Changes (visited on 05/14/2025).

[Emb13] Embarcadero Technologies Inc.: *System.TMarshal.ReadInt32*. 2013. URL: https://docwiki.embarcadero.com/Libraries/Sydney/en/System.TMarshal.ReadInt32 (visited on 04/27/2025).

[Emb14a] Embarcadero Technologies Inc.: *FMX.Forms.TForm*. 2014. URL: https://docwiki.embarcadero.com/Libraries/Athens/en/FMX.Forms.TForm (visited on 04/26/2025).

[Emb14b] Embarcadero Technologies Inc.: *System.Net*. 2014. URL: https://docwiki.embarcadero.com/Libraries/Sydney/en/System.Net (visited on 01/27/2025).

[Emb14c] Embarcadero Technologies Inc.: *Vcl.Forms.TApplication.CreateForm*. 2014. URL: https://docwiki.embarcadero.com/Libraries/Athens/en/Vcl.Forms.TApplication.CreateForm (visited on 04/26/2025).

[Emb19] Embarcadero Technologies Inc.: *Build Configurations Overview*. 2019. URL: https://docwiki.embarcadero.com/RADStudio/Athens/en/Build_Configurations_Overview (visited on 03/29/2025).

[Emb22] Embarcadero Technologies Inc.: *PE (portable executable) header flags (Delphi)*. 2022. URL: https://docwiki.embarcadero.com/RADStudio/Athens/en/PE_(portable_executable)_header_flags_(Delphi) (visited on 01/27/2025).

[Emb24a] Embarcadero Technologies Inc.: *Delphi Documentation Main Page*. 2024. URL: https://docwiki.embarcadero.com/Libraries/Athens/en/Main_Page (visited on 02/19/2025).

[Emb24b] Embarcadero Technologies Inc.: *Delphi Product Editions*. 2024. URL: https://www.embarcadero.com/products/delphi/product-editions (visited on 01/27/2025).

[Emb24c] Embarcadero Technologies Inc.: *Supported Target Platforms*. 2024. URL: https://docwiki.embarcadero.com/RADStudio/Athens/en/Supported_Target_Platforms (visited on 01/27/2025).

[Emb24d] Embarcadero Technologies Inc.: *VCL*. 2024. URL: https://docwiki.embarcadero.com/RADStudio/Athens/en/VCL (visited on 03/29/2025).

[Emb25a] Embarcadero Technologies Inc.: *Learn Delphi Main Page*. 2025. URL: https://learndelphi.org/ (visited on 02/19/2025).

[Emb25b] Embarcadero Technologies Inc.: *RAD Studio*. Information available inside the tool. 2025. URL: https://www.embarcadero.com/products/rad-studio (visited on 03/29/2025).

[Emb25c] Embarcadero Technologies Inc.: *RAD Studio Previous Versions*. 2025. URL: https://www.embarcadero.com/products/rad-studio/previous-versions (visited on 05/14/2025).

[ESE24] ESET Research: *DelphiHelper GitHub Page*. 2024. URL: https://github.com/eset/DelphiHelper (visited on 02/19/2025).

[Gaj17] Gajic, Zarko: *Delphi History from Pascal to Embarcadero Delphi XE 2*. 2017. URL: https://www.thoughtco.com/history-of-delphi-1056847 (visited on 03/29/2025).

[Hex25a] Hex-Rays: *FLIRT*. 2025. URL: https://docs.hex-rays.com/user-guide/signatures/flirt (visited on 04/27/2025).

[Hex25b] Hex-Rays: *IDA Pro Pricing Page*. 2025. URL: https://hex-rays.com/pricing (visited on 02/19/2025).

[Hüt19] Hüttenhain, Jesko: *Reverse Engineering Delphi Binaries in Ghidra with Dhrake*. 2019. URL: https://blag.nullteilerfrei.de/2019/12/23/reverse-engineering-delphi-binaries-in-ghidra-with-dhrake/ (visited on 01/23/2024).

[Hüt23] Hüttenhain, Jesko: *Dhrake GitHub Page*. 2023. URL: https://github.com/huettenhain/dhrake/ (visited on 01/23/2024).

[ImN22] ImNotAVirus: *Delphi VMT Analyzer (v0.1.2) GitHub Page*. 2022. URL: https://github.com/ImNotAVirus/delphi_ninja (visited on 02/19/2025).

[j4k22] j4k0xb: *webcrack GitHub Page*. 2022. URL: https://github.com/j4k0xb/webcrack (visited on 02/19/2025).

[Jan24] Jansen, Paul: *TIOBE Index for December 2024*. 2024. URL: https://www.tiobe.com/tiobe-index/ (visited on 01/27/2025).

[Jos24] Joshi, Sagar: *40 Operating System Statistics: Who Will Dominate 2025?* 2024. URL: https://learn.g2.com/operating-system-statistics (visited on 01/27/2025).

[Lak24] Lakshmanan, Ravie: *Cybercriminals Exploit Popular Game Engine Godot to Distribute Cross-Platform Malware*. 2024. URL: https://thehackernews.com/2024/11/cybercriminals-exploit-popular-game.html (visited on 01/27/2025).

[Nat25] National Security Agency (Official GitHub Organization Account): *Ghidra 11.3 Change History (February 2025)*. 2025. URL: https://github.com/NationalSecurityAgency/ghidra/blob/Ghidra_11.4_build/Ghidra/Configurations/Public_Release/src/global/docs/ChangeHistory.md#ghidra-113-change-history-february-2025 (visited on 07/06/2025).

[NCC19] NCC Group Plc: *pythia GitHub Page*. 2019. URL: https://github.com/nccgroup/pythia (visited on 02/19/2025).

[Pal+24] Pal, Kuntal Kumar et al.: ""Len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning". In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, pp. 4069–4087.

[Poi23] Poireault, Kevin: *The Dark Side of Generative AI: Five Malicious LLMs Found on the Dark Web*. 2023. URL: https://www.infosecurityeurope.com/en-gb/blog/threat-vectors/generative-ai-dark-web-bots.html (visited on 01/27/2025).

[Rad20] Radich Quinn et al.: *Running 32-bit Applications - Microsoft Learn*. 2020. URL: https://learn.microsoft.com/en-us/windows/win32/winprog64/running-32-bit-applications (visited on 03/29/2025).

[Rat24] Ratto, Kevin: *Still Alive: Updates for Well-Known Latin America eCrime Malware Identified in 2023*. 2024. URL: https://www.crowdstrike.com/en-us/blog/latin-america-malware-update/ (visited on 01/27/2025).

[Sat24] Sathsara, Isuru: *A Brief Overview of Delphi Language*. 2024. URL: https://medium.com/@isuru.dex/a-brief-overview-of-delphi-language-527f294c9558 (visited on 03/29/2025).

[Wil24] Wilson, Sean: *Malware Trends: Yearly 2023*. 2024. URL: https://blog.unpac.me/2024/01/30/malware-trends-yearly/ (visited on 01/27/2025).

[Wil25] Wilson, Sean: *Malware Trends: Yearly 2024*. 2025. URL: https://blog.unpac.me/2025/02/20/malware-trends-yearly-review-2024-2/ (visited on 03/29/2025).

[X24] X., Serge: *Delphi Programming Language*. 2024. URL: https://www.softacom.com/wiki/development/delphi-programming-language/ (visited on 03/29/2025).

[XZZ12] Xu, Wei ; Zhang, Fangfang ; Zhu, Sencun: "The power of obfuscation techniques in malicious JavaScript code: A measurement study". In: *2012 7th International Conference on Malicious and Unwanted Software*. IEEE. 2012, pp. 9–16.