DRTTI: Enhancing Ghidra Decompilation of Delphi Applications

Anton Degerfeldt
Umeå University
Umeå, Sweden
anton.degerfeldt@cs.umu.se

Sylvain Fontaine
Umeå University
Umeå, Sweden
sylvain.fontaine@cs.umu.se

Alexandre Bartel
Umeå University
Umeå, Sweden
alexandre.bartel@cs.umu.se

Abstract

We introduce *Drtti*, an automated static analysis plugin for Ghidra that reconstructs high-level class hierarchies, field layouts, and virtual method tables directly from embedded Run-Time Type Information. Additionally, Drtti can import standard-library symbols from the Interactive Delphi Reconstructor (IDR). We also explore lightweight IDR-independent heuristics, such as x87 FPU instruction fingerprints, as conservative hints for locating mathematical code. We evaluate Drtti on a set of 10 Delphi applications covering Delphi 4 and Delphi 8+ VMT layouts. Results indicate that standard Ghidra analysis mapped 57.57% of executable memory to verified data or instructions on average, while Drtti alone reached 61.15% and Drtti with optional IDR symbols reached 87.66%. Drtti provides IDR-independent structural recovery inside Ghidra, while the optional IDR symbols mainly add library naming and data context.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

Keywords

Reverse Engineering, Native Decompilation, Ghidra, Delphi, Run-Time Type Information (RTTI), Object-Oriented Binary Analysis

ACM Reference Format:
Anton Degerfeldt, Sylvain Fontaine, and Alexandre Bartel. 2026. DRTTI: Enhancing Ghidra Decompilation of Delphi Applications. In *2nd Workshop on Software Understanding and Reverse Engineering (SURE '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3847158.3847385

1 Introduction

Understanding legacy Delphi GUI binaries is challenging because relevant program semantics are distributed across VCL form resources, runtime class hierarchies, virtual method tables (VMTs), and compiler-generated run-time type information (RTTI). Although modern decompilers can recover C-like control flow, they often fail to reconstruct object layouts and to connect GUI components to their event handlers. Dynamic analysis is also difficult to automate, since reaching relevant behavior frequently requires GUI interaction.

Delphi binaries nevertheless retain metadata that can support software understanding. Classic Delphi RTTI describes class names, fields, properties, and selected method signatures, while VMTs expose inheritance and dispatch relationships. Together, these artifacts provide an opportunity to recover high-level program structure directly from the executable. QTRE, developed by Wen and Lin [21], demonstrated the value of framework-provided introspection for recovering symbols and callbacks in Qt applications; Delphi offers analogous, but distinct, sources of runtime metadata.

ProdMod2 illustrates this problem. It is a scientific forest growth simulator compiled in 1998 whose source code is no longer available, leaving the executable as the remaining carrier of the scientific model. In this setting, recovering class structures is an intermediate step toward identifying event flows, mathematical helper routines, and formula-heavy regions that drive the simulation.

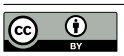
In this work, we present Drtti (Delphi Run-Time Type Information), a plugin for Ghidra 12.0.4 that combines RTTI and VMT analysis to recover class, field, and property names, class hierarchies, and method signatures from Delphi 4 executables. The recovered information is added to Ghidra's program database to improve its representation and decompilation of Delphi applications. Our approach is related to RTTI-based type reconstruction techniques for C++ binaries, such as the IDA Pro plugin developed by Fokin et al. [9] and the *SmartDec* decompiler [8].

We make the following primary contributions to the field of software reverse engineering:

- Drtti, an open-source plugin for Ghidra 12.0.4 designed to reconstruct class hierarchies, virtual method tables, and extract symbolic information from Delphi 4's RTTI system: **https://github.com/software-engineering-and-security/Drtti**
- A systematic approach for assessing decompilation quality. It uses a combination of semantic comparison and quantitative metrics, supported by a set of Delphi test modules.
- A quantitative evaluation of Drtti on a ten-binary Delphi corpus, including the ProdMod2 application for which the source code has been lost and nine source-available applications with documented repositories, commits, and project files. This evaluation measures both executable-memory coverage and Delphi-specific structural recovery, showing that Drtti recovers 8824 binary-level class structures and 24008 fields on the source-available corpus.

In this work, we answer to the following research questions:

RQ1 What types of symbolic information can be recovered from classic Delphi RTTI, as implemented in Delphi executables?
RQ2 To what extent can symbolic information recovered from Delphi RTTI improve the semantic accuracy of Ghidra decompilation output?
RQ3 To what extent can Delphi binary understanding be performed directly inside Ghidra without relying on IDR or a version-specific knowledge base, and what additional context do optional IDR symbols provide?



This work is licensed under a Creative Commons Attribution 4.0 International License.
SURE '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3019-1/2026/11
https://doi.org/10.1145/3847158.3847385

We show that Drtti can recover class, field and property names, as well as class hierarchies, functions, and method signatures directly from the RTTI embedded in an executable. With the added heuristics, the core recovery pipeline no longer requires IDR: VMT discovery, RTTI parsing, event-flow localization, and x87 FPU fingerprinting are performed directly inside Ghidra. IDR remains an optional source of additional RunTime Library (RTL) and VCL names.

Our findings indicate that automated RTTI extraction can recover complete class structures from Delphi executables, transforming low-level pointer expressions into typed object field accessors. In real-world application testing, this technique mapped over 93.54% of target binary memory to data or instructions when combined with standard library symbols. This is a significant improvement over Ghidra 12.0.4's standard suite of analysis tools, which mapped 76.14%. Our case study also demonstrates how this additional information can reduce the need for human-in-the-loop analysis.

2 Related Work

Currently, there exists no decompiler capable of fully reconstructing Delphi code. Dedicated Delphi tools, such as IDR[1] and DeDe[2], can partially reconstruct Delphi code, but have limited utility. These tools were both developed as open-source projects by small development teams and are, as of writing, no longer actively maintained. Due in part to this, they also lack many of the features that researchers have come to expect from modern reverse-engineering software. While industry-standard analysis tools such as Ghidra generally do not support outputting Delphi code, it is still possible to use Ghidra to decompile a Delphi executable and obtain a C-like representation of the program. The problem is that, since Ghidra is designed primarily around C and, to some extent, C++, it does not handle Delphi binaries well out of the box. There are several complicating factors, ranging from the custom proprietary compiler toolchain to the way the Delphi language implements function calls and objects.

Dhrake[3] is a script for Ghidra that aims to bridge the gap between IDR and Ghidra. It reads symbol files produced by IDR to create functions and labels inside Ghidra. Dhrake also generates patches for the signatures of certain library functions to further improve the results. While Dhrake can improve decompilation results, it relies almost entirely on IDR and performs very limited analysis itself. In addition, Dhrake creates structures within Ghidra for the virtual method tables; however, it does not do so for the objects themselves. This means that Dhrake's function-processing capabilities are limited to naming and setting function calling conventions, since the object types passed as parameters are not available within Ghidra.

Utilizing RTTI to recover class, field and property names, as well as class hierarchies, function and method signatures for C++ applications has been studied in works such as Fokin et al. [9] and Sabanal and Yason [18]. Other approaches for type inference include direct virtual method table analysis, as implemented in *SmartDec* [8], as well as a recent plethora of AI-based solutions such as *TYGR* [22], *TypeForge* [20], *DIRTY* [2], *STRIDE* [10], and *XSTRIDE* [14].

Applying RTTI-analysis to Delphi executables within a modern reverse engineering framework such as Ghidra is a novel contribution to the field of software reverse engineering.

```
1 program Greeting;
2 {$APPTYPE CONSOLE}
3 var message: string;
4 begin
5   message := 'Hello World';
6   WriteLn(message);
7 end.
```

Listing 1: Hello World Console Program in Delphi.

3 Background

3.1 Delphi 4

Originally developed as a rapid application development tool for Windows by Borland, Delphi was released in 1995 as a successor to the Turbo Pascal development system. It was released as an ecosystem comprising an integrated development environment (IDE), a component library, and database management frameworks. The language is still in use, and, as of writing, ranks 10th on the TIOBE index[4], a site that tracks programming language popularity.

At its core, the Delphi language is a modified version of Object Pascal, with additional features such as interfaces, properties, and exception handling. Listing 1 contains a simple *"Hello World"*-program written in Delphi that prints "Hello World" to the console from which it is started. This program would be almost identical if it were written in Object Pascal, with the exception of line 2, where a Delphi *compiler directive* specifies that the program is a console application.

3.1.1 Structure of a Delphi Application. The architecture of a Delphi 4 application is heavily based around the *Visual Component Library* (VCL). The VCL framework encapsulates the complexities of the Windows api into a set of reusable components, such as buttons, input fields, etc. A Delphi application consists of a set of so called *Forms*, each corresponding to a window or dialog box in the application.

A Delphi project usually contains a single Project File (.dpr), which serves as the primary entry point. This initializes a global `TApplication` [5] object, manages the creation of the main form, and executes the central message loop, see Listing 2.

The `TApplication` instance acts as the orchestrator for events, dispatching messages from the operating system to specific components. These messages then trigger programmer-defined functions called *event handlers*, redirecting the execution flow. The application's functional layout is divided into discrete *Units* (.pas files) and *Form Files* (.dfm). The former consists of business logic, such as event handlers and data structures, while the latter contains only graphical information, including component sizes, positions, labels, fonts, etc. This strict separation between user interface definitions and procedural logic is akin to the XML-view based approach that

[1] https://github.com/crypto2011/IDR
[2] https://github.com/DeDeProject/DeDe
[3] https://github.com/huettenhain/dhrake
[4] https://www.tiobe.com/tiobe-index/
[5] In Delphi, the names for types are generally prefixed with a `T`, but this is just a convention and is not required by the compiler.

```
1  program Greeting;
2  uses
3    Forms,
4    MainUnit; { the unit for the MainForm }
5  {$R *.RES} { this directive links the project's resource file }
6  begin
7    Application.Initialize;
8    Application.CreateForm(TMainForm, MainForm);
9    Application.Run;
10 end.
```

Listing 2: Project for a windows application in Delphi 4. The unit that defines the `TMainForm` component is in Listing 3.

```
1  unit MainUnit;
2  interface
3  uses
4    Windows, Messages, SysUtils, Classes,
5    Graphics, Controls, Forms, Dialogs, StdCtrls;
6  type
7    TMainForm = class(TForm)
8      Button1: TButton;
9      procedure Button1Click(Sender: TObject);
10   end;
11 var
12   MainForm: TMainForm;
13 implementation
14 {$R *.DFM} { this directive links MainForm's form file }
15 procedure TMainForm.Button1Click(Sender: TObject);
16 begin
17   Close;
18 end;
19 end.
```

Listing 3: Delphi 4 `TMainForm` type used in Listing 2.

is used in traditional Android application development. There are also a number of additional files in a typical Delphi project, such as the resource files (.res), which are embedded directly into the binary on line 5 of Listing 2.

3.1.2 Language Constructs. Delphi is similar in many ways to other statically typed oop languages, especially its ancestor Object Pascal. Herein follows a brief explanation of the language constructs and key concepts important to this work.

Function A self-contained module of code. It is defined by its *signature*, which consists of name, input parameters, and return type. An interesting aspect of Delphi is that there is no `return` keyword; instead, functions return a value of the last expression in the function body. This is commonly done either by creating a variable and assigning the value to it, or by assigning a value to the function name, as is customary in Object Pascal. A feature added by Delphi is that every function implicitly declares a variable called `Result` that can be used for this purpose.

Procedure A function whose signature has no return value.

Class A class serves as an extensible template for creating *objects*. The class encapsulates the state (*fields*) and behavior (*methods*) of an *instance*. A class may *inherit* from another class to gain access to, and extend on, its state and behavior. The class from which a class inherits is commonly called the *parent*, and the classes that inherit from it are called or *child* classes.

Object An object is a discrete instance of a class that is created at runtime. Each object maintains its own identity and state, even if multiple instances of the same class are created.

Method A function, or procedure, that is applied to an instance of a class. Internally, these functions accept a reference to the object

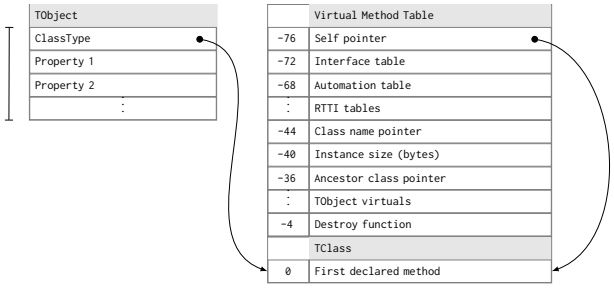


Figure 1: Visualization of virtual method tables in Delphi.

instance as their first parameter. *Virtual* or *dynamic* methods are those that can be overridden by child classes. That is, the behavior may change depending on the type of object to which it is applied. *Abstract* methods are must be overridden by child classes.

Field A field is a storage location for a value of either primitive or composite type, within an object. Fields are identified by their name and the object they belong to, and act as *attributes* of the object.

Property A property defines an attribute of an object, functioning as a logical interface for data access rather than a direct storage location. Unlike a field, which represents a concrete allocation of memory, a property encapsulates data by associating `read` and `write` specifiers with accessors. These accessors may map directly to a backing field or to methods. Properties allow objects to expose attributes as simple variables, even if their values are calculated from the internal state.

3.1.3 Virtual Method Tables. Every object in Delphi contains a pointer to its class as the first four bytes of its memory representation. The class consists of what is commonly referred to as a VMT. The VMT is an internal data structure used by the compiler and is not intended for direct developer use. It is essentially a list of function pointers to every `virtual` method for the class that the compiler uses to implement *dynamic dispatch*.

In Delphi, additional metadata about the class is placed *before* the list of function pointers. This structure contains basic information about the class, such as its name, the amount of memory required per instance, a reference to the parent class, RTTI information, and so on. The relationship between object, class, and VMT is shown in Figure 1. For the complete layout of the VMT in Delphi 4, see Appendix A.1.

For simplicity, we will adopt the Delphi convention and refer to the entirety of this structure and the list of virtual methods as the VMT.

3.1.4 Run-Time Type Information. RTTI is a specialization of *type introspection* available in some object-oriented programming languages, such as C++, Java, and Delphi. RTTI exposes information about objects and commonly includes at least the class it belongs to. This lets languages implement features like safe type casts and dynamic bindings. It also lies at the core of Java's *reflection* mechanism, which lets developers access and modify classes at runtime.

In Delphi, RTTI is used for the bindings between graphical components and their underlying logic. Names of methods and their parameters can be serialized into the form files that declare the

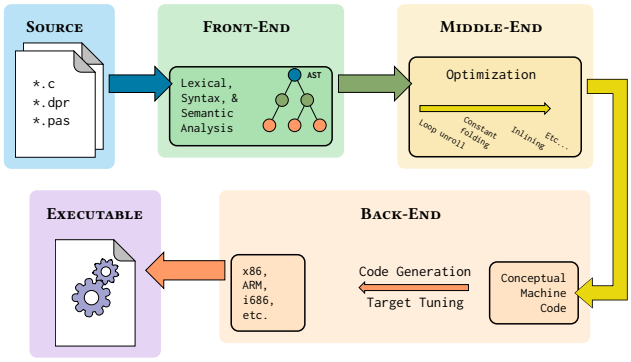


Figure 2: Simplified overview of the compilation process from source code to native executable. Symbols are parsed, turned into an *Abstract Syntax Tree* (AST) and analyzed for correctness. The code can then be optimized and translated to machine code for a specific target architecture and linked into a single native executable.

layout of components, and then loaded back into the application at runtime. This allows for separate compiling of form files and logic units.

Some metadata is always generated for classes in Delphi to enable features such as the `is` keyword, which checks whether an object is an instance of a particular class; however, detailed RTTI is an *"opt-in"* feature. To enable detailed RTTI for an object in Delphi, the developer must declare a `published` section. These sections may contain both field declarations and methods, for which RTTI is generated. RTTI allows other parts of the code to dynamically query an object's fields and properties, and locate published methods.

Internally, the RTTI is stored in several table structures contained in the classes' VMT (see Section 3.1.3). The most important ones for the purpose of this work are the *Method-* and *Dynamic Method* tables, as well as the *Field-* and *Type Info* structures.

3.1.5 Reverse Engineering Challenges. The Delphi programming language is known to produce applications that are relatively challenging to decompile due to several factors.

Native compilation Unlike object-oriented languages, such as Java or C#, which compile to *bytecode* designed to run on a virtual machine, Delphi, like C++, compiles directly to native machine code. Many bytecode languages utilize virtual machines that require extensive metadata and type information to be preserved in the binary for runtime linking and JIT compilation. With native compilation, this is not required, and the majority of symbolic information and high-level abstractions are therefore discarded after the *front-end* phase of compilation process, as presented in Figure 2, has been completed. While Delphi 4 specifically compiles only for Windows x86 architecture, native binaries in general can target multiple CPU architectures and instruction sets. Supporting such a wide variety of formats also contributes to the challenge of maintaining decompilers.

```
1 function Foo(A, B, C, D: Integer): Integer;
2 begin
3   Result := A + B + C + D;
4 end;
5
6 begin
7   ExitCode := Foo(1, 2, 3, 4);
8 end.
```

Listing 4: Abbreviated Delphi code for declaring and calling a function `Foo`.

```
PUSH 4
MOV ECX, 3
MOV EDX, 2
MOV EAX, 1
CALL Foo
```

Listing 5: Assembly for call to `Foo` in Listing 4.

Compiler Optimizations To further complicate matters, the Delphi compiler is quite aggressive in its use of the compiler optimizations, including techniques such as function inlining, loop unrolling, and dead-code elimination by default.

Function Calling Convention Delphi uses the *Borland Fastcall* [1] calling convention, where the first **three** arguments to a function are passed in registers EAX, EDX, and ECX, and the remaining arguments are pushed onto the stack. This process is performed in right-to-left order, as shown in Listings 4 and 5. The generated assembly first pushes the last argument onto the stack and then continues to the left until it reaches the third argument, in this case, the number 3, which is moved into ECX.

This contrasts with other standardized calling conventions, such as *cdecl*, used by the C language, where all function arguments are passed on the stack in left-to-right order.

Another issue one encounters with specialized calling conventions is how to handle large return values. For example, the `Double` is a core Delphi datatype that represents a 64-bit floating-point number. Functions may return `Double` values, but since it is too large to be passed in a 32-bit register, Delphi relies on a technique called *return storage pointers*, where the first argument to the function is a pointer to a location to store the return value. Since object methods usually have a *Self* pointer as their first parameter, referring to the object instance, the arguments must be shifted through the registers in some deterministic way.

For some internal Delphi functions that operate on large floating-point numbers, yet another calling convention is used. The arguments and return values may, in some cases, be passed directly via specialized FPU (*floating-point unit*) registers. An example of this is the function generated by Delphi 4's division operator for floating-point numbers in order to circumvent the famous *Pentium FDIV-bug* [17]. This calling convention makes the patched function behave just like the FDIV instruction.

Event-Driven Architecture Delphi relies heavily on an event-driven architecture and frequently uses function pointers, some of which are modified at runtime based on user interactions. This can make static analysis tools less reliable, as it becomes difficult to identify what functions are being called.

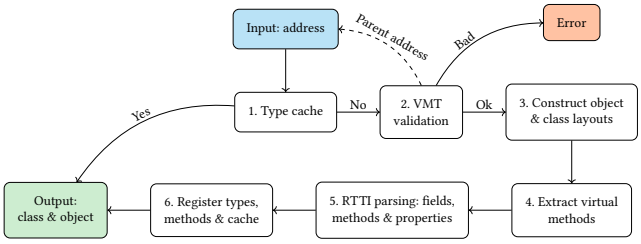


Figure 3: Overview of the Drtti plugin's analysis process for a specific address.

3.2 Ghidra

Drtti is implemented as a Java plugin for Ghidra. It operates on Ghidra's program database after the standard x86 analysis has created an initial view of the binary. The plugin reads memory blocks to locate Delphi metadata and writes the recovered information back as data types, structures, function definitions, and symbols. These annotations make recovered class layouts and member accesses available to Ghidra's decompiler.

Drtti relies on Ghidra's existing instruction decoding and decompilation; it does not modify the processor specification or the intermediate representation. Its contribution is to convert Delphi-specific VMT and RTTI records, which generic analysis may leave as untyped bytes, into the semantic objects used by Ghidra's program database.

3.3 PE Regions Used by Drtti

The 32-bit Delphi binaries in our corpus use the *Portable Executable* (PE) format [16]. Drtti scans executable `CODE` memory for VMT candidates, because the VMT function pointers and the associated compiler-generated metadata are located there in the supported Delphi layouts.

The `.rsrc` section contains serialized VCL form properties. Drtti uses the GUI-resource bindings obtained from these properties when linking recovered components to event handlers. Other PE sections are handled by Ghidra's ordinary analysis and are not specific to Drtti.

4 Drtti Plugin

In this section, we detail some of the inner workings of Drtti. For a practical guide to using Drtti, consult the project repository[6]. Table 1 outlines the core feature set of the Drtti plugin, distinguishing between internal analysis capabilities and features driven by external IDR metadata.

This section describes the process of finding, analyzing, and recovering the RTTI embedded in a Delphi binary, used by Drtti. Potential VMT candidates are located via a heuristic scanning process over addresses in the `CODE` segment of the binary. An overview of the analysis for a single address is shown in Figure 3 and described in detail in the following sections.

The analysis workflow relies on several Delphi-specific heuristics that complement the raw RTTI parsing performed by Drtti. We group them into four families as shown in Table 2. First, detection and layout-selection heuristics identify Delphi binaries and select the expected VMT offsets for the compiler generation under analysis. Second, structural RTTI heuristics validate VMT candidates, recover FieldTables and MethodTables, and distinguish metadata read directly from validated structures from names inferred by cross-reference. Third, event-flow heuristics connect recovered GUI metadata and MethodTable entries to event handlers, narrowing the search from visual components to application logic; this workflow is illustrated in the ProdMod2 case study [7]. Finally, mathematical-code heuristics use rare x87 FPU instruction fingerprints to identify runtime mathematical helpers and formula-heavy application sites.

Category	Feature	Description
RTTI Analysis	Type Injection	Adds primitive and built-in Delphi types to Ghidra's data type manager.
	VMT Discovery	Automated recovery of vmt.
	Structural Modeling	Generates object/class layouts and namespaces, and model class hierarchies.
	Member Resolution	Dynamically types fields, properties, and method references.
	Signature Recovery	Creates and assigns known function signatures.
IDR Integration	Global Variables	Identifies and types known global variables.
	Library Functions	Automates signature assignment for Delphi standard library functions.

Table 1: Functional capabilities of the Drtti plugin.

These fingerprints do not replace IDR symbols, but provide an independent corroborating signal: when both sources point to the same mathematical routine or formula-heavy region, the analyst gains additional confidence in the recovered semantic context.

This separation is important because not all recovered information has the same evidential status. Class names, field names, published methods, and MethodTable entries are read from validated Delphi metadata. Event-handler links are stronger when the same handler name is observed both in GUI metadata and in a MethodTable entry. FPU-based names are semantic fingerprints rather than RTTI facts, and are therefore used either to name small helper-shaped functions or to mark larger application functions as mathematical formula sites.

4.1 Locating & Validating a VMT

Valid VMT are identified first via the *"Self-Pointer"* relationship between `TClass` and VMT at offset -76 (`-0x4c`), as described in Section 3.1.3; this corresponds to step 2 of Figure 3. Further verification can be achieved by checking that every four-byte value in the VMT, except the instance size entry at offset -40 (`-0x28`), is either zero (null) or a valid reference to a location in executable memory. Additionally, every class is expected to have a pointer to a string containing the class name at offset -44 (`-0x2c`).

This executable-memory scan is important in practice. For example, scanning for the canonical self-pointer relation `[vmt - 0x4c] == vmt` on the ProMod2 application produced 311 raw self-pointer

[6] https://github.com/software-engineering-and-security/Drtti

[7] due to lack of space, we document the analysis of ProMod2 in Appendix B

Table 2: Heuristic families used by the Delphi analysis workflow.

Family	Main signal	Output
Detection and layout selection	Delphi compiler/runtime markers, VMT self-pointer relation, version-specific offsets	Candidate Delphi binary and expected RTTI/VMT layout
Structural RTTI recovery	Validated VMTs, FieldTables, MethodTables, TypeInfo records	Class hierarchy, class names, fields, published methods, method signatures
Event-flow recovery	Recovered method names, GUI resource bindings, MethodTable entries	Mapping from GUI actions to handler functions
Mathematical-code localization	Rare x87 FPU instruction fingerprints and calls from recovered handlers	Mathematical RTL helpers and formula-heavy application sites

hits in the CODE block. Adding the class-name validation filter reduced these to 291 valid class VMTs, matching the class count recovered by Drtti. This explains why generic analysis can leave Delphi object metadata as raw bytes even when it is stored in an executable section.

When a valid VMT is found, its *parent* entry at offset -36 is examined; if it is not null, Drtti first attempts to process the parent VMT. This technique allows Drtti to recover the entire class hierarchy from the ground up, starting with `TObject`, and ensures that types for fields, parameters and properties are created before they are needed. To reduce the overhead of this recursive process, an internal cache maps addresses to types that have already been created by Drtti (step 1 of Figure 3).

4.2 Object & Class Layouts

Once the VMT has been identified, empty shells are created for the target object and class structures, as illustrated in step 3 of Figure 3. The size of the object structure is read from the instance size entry in the VMT, and the size of the class structure is determined by finding the number of valid function pointers after offset zero in the VMT. Functions are then created in Ghidra at the locations pointed to by the virtual method entries, and pointers to them are added to the class structure, as shown in step 4 of Figure 3.

4.3 RTTI Extraction

The most important part of the process shown in Figure 3, step 5, is the extraction of RTTI from the VMT. In a Delphi binary, class RTTI is spread across several internal data structures. As Delphi RTTI is opt-in using the `published` keyword, the amount of information available varies greatly from class to class. The name, instance size, and parent class are always present; however, the key RTTI structures: *Method Table*, *Field Table*, and the *Type Info*, may or may not be present. From the first two, Drtti can recover published method names and, in some cases, signatures, as well as the names and locations of fields within an object of a given type.

The exact memory layouts of the internal RTTI structures were never officially documented by Borland or Embarcadero, as they are indirectly exposed through the Delphi standard library api in the *System.TypInfo* module. The layout of these structures is subject to change across versions of Delphi. The structures described herein are derived from manual reverse engineering of Delphi 4 binaries, as well as the unofficial Delphi reference manual *Delphi in a Nutshell* by Lischner [13, Ch. 3]. Full memory layout representations of structures, as they are interpreted in Drtti, as well as their contents, are described in further detail in Appendix A.

The *Type Info*, represented in Figure 4, is the most complex and most information-rich of the three. It also is the only one that is generated not only for classes, but also for Delphi's primitive and

ClassTypeInfo		
Offset	**Description**	**Type**
`0x0`	Name	pascal string
`Size(Name)`	Class reference	VMT pointer
`Size(Name)+0x04`	Parent type	TypeInfo pointer
`Size(Name)+0x08`	Total property count	unsigned word
`Size(Name)+0x10`	Unit name	pascal string
`Size(Name)+0x10 +Size(UnitName)`	Property count	unsigned word
`Size(Name)+0x12 +Size(UnitName)`	Properties	PropertyInfo[PropertyCount]

Figure 4: In-memory layout of *Type Info* structure for a class.

built-in types. For a class, the *Type Info* Drtti can recover both the names and types of published properties, along with their accessors, whether they are methods or fields. This contributes to the data recovered from the *Field Table* and the *Method Table*.

For primitive and built-in types, the names and sizes are most relevant to Drtti, however, there are exceptions, such as enumerations for which the *Type Info* contains the names of the enumerators. An example of this is the `Boolean` type entry, which contains the names of the two possible values (*"False"* and *"True"*). For further details about the contents of the various *Type Info* structures, see Appendix A.4.

4.4 IDR Integration

After the RTTI has been recovered, one can optionally import additional symbolic information from IDR. To do this, the target binary is first loaded into IDR, for analysis. An *IDC* file containing symbolic information can then be exported from IDR and read by Drtti. The symbolic information from IDR is then merged with the existing data generated by Drtti to provide a more complete picture of the target binary.

4.5 FPU Fingerprinting for Math Runtime Helpers

IDR's knowledge base is useful because many Delphi RTL and VCL routines do not carry RTTI [3, 4]. However, mathematical helper routines form a special case: Delphi 4 emits x87 code for `Extended` floating-point operations, and several transcendental operations have rare, semantic instruction markers [12]. We list them in Table 3. For example, exponentiation uses `F2XM1` and `FSCALE`, logarithms use `FLDLN2` with `FYL2X`, and two-operand arctangent uses `FPATAN`. These instructions contain no absolute addresses, so the fingerprint is insensitive to relocation fixups, unlike byte-level function signatures.

The pass scans each recovered Ghidra function and records the set of x87 mnemonics in its body. A match is classified as a helper

Table 3: x87 fingerprints used for mathematical recovery.

Role	Required x87 markers	Confidence
Ln	FLDLN2, FYL2X	high
Log10	FLDLG2, FYL2X	high
Exp	F2XM1, FSCALE	high
ArcTan	FPATAN	high
Tan	FPTAN	high
SinCos	FSINCOS	high
Sin/Cos	FSIN/FCOS	candidate

when the function is small and contains no calls; otherwise it is classified as an application site using the operation. This distinction is important: helper matches name RTL-like routines, while site matches guide the analyst toward code that implements mathematical formulas.

We explicitly excluded ordinary division from the production fingerprint set. A first attempt to identify a floating-point division helper from FDIV/FDIVP produced two plausible-looking candidates, at 0x004062f0 and 0x0046fcb4.We therefore do not report a corpus-level false-positive rate for this pass: the fingerprints are used as analyst-facing hints rather than automatic semantic labels. Manual inspection showed that these were not the desired RTL helper: one was a decimal scaling routine and the other an application-level geometric predicate. Therefore, the method relies only on rare transcendental instructions, not on common arithmetic opcodes.

5 Methodology

5.1 Experimental Setup

For the hand-written semantic-similarity examples, the Delphi compiler *dcc32* (version 12.0) was used to compile source code. Ghidra version 12.0.4 was used to decompile the resulting executables, and Drtti version 0.3.0 was used to process the RTTI embedded in the executables. The source-available corpus was evaluated separately using the compiler layout selected for each binary, including the Delphi 8+ VMT layout.

5.2 Application Dataset

We evaluate Drtti on a ten-binary Delphi application dataset designed to cover both legacy lost-source reverse engineering and reproducible source-available evaluation. The dataset includes ProdMod2, five DelphiDabbler article demos [5], three official RAD Studio 12 demos [7], and one VCL Styles Utils demo [19]. These binaries cover Delphi 4 and Delphi 8+ VMT layouts, allowing us to evaluate both the original legacy target and the newer layout support added to Drtti.

For inclusion in the quantitative dataset, each source-available target required an official upstream repository, an identified project file, a fixed commit or release, and a reproducible executable.

Per-application measurements and details are reported in Appendix C.

5.3 RQ1: Similarity of Decompiled Code

To gauge the accuracy and efficiency of Drtti, the project repository includes 15 Delphi 4 example targets, comprising 22 Pascal source files, 11 class declarations, and 33 declared routines, including functions, procedures, constructors, and destructors. In the paper, we discuss three representative examples in detail: Rectangle, Button, and Animal. These examples cover the RTTI constructs most relevant to the evaluation, including published properties, method references, virtual methods, and inheritance. The complete example corpus is available in the Drtti repository.

There is no standard reference system for comparing decompiled output to source code. For example, *McCabe Cyclomatic Complexity* [15], used by revng-c [11] primarily measures control flow complexity. Others such as Wang et al. [20], have used LLM to gauge semantic similarity. In this work, we have opted for a manual visual comparison of sections of the decompiled output to Delphi source code, as well as examinations of the data structures that Drtti generates for Ghidra. This approach was deemed the most effective because the output from Ghidra is not necessarily directly comparable to the source code. The complete source code for the example modules is in the Git repository: **https://github.com/software-engineering-and-security/Drtti**.

5.4 RQ2: Symbols Extraction with Drtti

To estimate the amount of symbolic information that Drtti provides, and address 1, we have devised a series of quantitative metrics:

- **Total Symbols**: number of *symbols*, as per Ghidra's *symbol table*. A symbol is any address that is mapped to some string. This includes function names and global data labels.
- **Functions**: number of entrypoints for functions registered in Ghidra's *function manager*. This number excludes *thunks*, which are functions that only pass execution to another function: functions from externally linked Windows DLLs are excluded.
- **Named Functions**: number of **functions** that have been given a name. This is determined as those functions whose name does not follow Ghidra's default naming convention FUN_XXX, where XXX is the address of the function entrypoint.
- **Datatypes**: number of data types in Ghidra's *data type manager*.
- **Classes**: number of **datatypes** that represent Delphi classes.
- **Fields**: total number of fields for all **classes**. Fields are only included if they have been given a type and a name.
- **Strings**: number of string constants within the binary that are known to Ghidra.

Additionally, the proportion of memory within a binary that Ghidra knows to be either a data type or an instruction was measured. Because memory inside the binary must be either data or an instruction, or be unknown, this measurement serves as a proxy for the binary's overall entropy. If more data and instructions are mapped, there is more structure and the overall entropy is lower; conversely, if more memory is unknown, the entropy is greater.

The script used to collect quantitative metrics is in Drtti's Git repository: **https://github.com/software-engineering-and-security/Drtti**.

5.5 RQ3: Symbols from Delphi's Std Library

To evaluate the amount of symbolic information that is provided by Delphi's Standard Library we measure the additional symbols

TPoint (geometry.pas)

```
TPoint = class
private
  FX: Integer;
  FY: Integer;
public
  constructor Create(const X, Y: Integer);
published
  property X: Integer read FX;
  property Y: Integer read FY;
end;
```

Listing 6: TPoint

TRectangle (geometry.pas)

```
TRectangle = class
public
  constructor Create(const AX, AY, BX, BY: Integer);
  destructor Destroy; override;
  function GetWidth: Integer;
  function GetHeight: Integer;
  function CalculateArea: Integer;
published
  UpperLeft: TPoint;
  LowerRight: TPoint;

  property Area: Integer read CalculateArea;
  property Width: Integer read GetWidth;
  property Height: Integer read GetHeight;
end;
```

Listing 7: TRectangle

brought through the IDR databases on the dataset of Delphi programs. The IDR databases contain lists of symbols from the different versions of Delphi's standard library.

6 Results

6.1 RQ1: Similarity of Decompiled Code

This section details the manual comparison of semantic similarity between the output produced by Drtti and the source code of example programs, and is structured as follows. Each section focuses on a particular aspect of the Delphi language.

6.1.1 Published Properties and Methods. The following structures are an excerpt from an example unit that models points and rectangles in Delphi. `TPoint` in Listing 6 represents a point in a two-dimensional plane and consists of two simple published properties, `X` and `Y`. Having them published ensures that RTTI is generated for the properties.

The `TRectangle` class in Listing 7 consists of two `TPoint` objects that define its upper-left and lower-right corners. The rectangle exposes width, height, and area as published properties, calculated from the coordinates of its points.

After importing into Ghidra and running Drtti, without importing symbols from IDR, on an executable that uses the point and rectangle classes, new types were automatically generated from the RTTI and added to Ghidra's *Data Type Manager*. These types represent the classes, the objects themselves, and their virtual method tables. The memory layout of the types for `TPoint` and `TRectangle` objects is shown in Figure 5. These memory layouts are consistent with expectations: each object starts with a pointer to its class, followed by the actual object fields.

Drtti was also able to successfully identify and create symbols for the accessor methods `CalculateArea`, `GetWidth`, and `GetHeight` of the rectangle class as shown in Listing 8. However, it should be noted

TPoint				TRectangle			
Offset	**Length**	**DataType**	**Name**	**Offset**	**Length**	**DataType**	**Name**
0x00	0x4	TPointClass *	tclass	0x0	0x4	TRectangleClass *	tclass
0x04	0x4	Integer	X	0x4	0x4	TPoint *	UpperLeft
0x12	0x4	Integer	Y	0x8	0x4	TPoint *	LowerRight

Figure 5: Memory layout of `TPoint` and `TRectangle` objects in Ghidra's data type manager after Drtti has been run.

Table 4: Symbolic information available after Ghidra's standard analysis and after Drtti analysis on the Rectangle binary.

#	Ghidra	Drtti	
Functions	116	124	+8
Named Functions	1	13	+12
Datatypes	260	306	+46
Classes	0	3	+3
Fields	0	7	+7

that the exact names of the functions are never stored in the binary, since they are not published themselves. Instead, Drtti generated accessor names by prefixing the property name with *Get* or *Set*. For the `Area` property, this results in the decompiled method being called `GetArea` instead of `CalculateArea`.

Symbolic information metrics after Drtti has been run on the Rectangle binary were also calculated and are presented in Table 4. The three classes detected were `TPoint`, `TRectangle`, and `TObject` classes. For the seven fields, six were from the layouts of the `TPoint` and `TRectangle` classes (see Figure 5) and one was the `tclass` field of the `TObject` class. The only named function before Drtti's analysis was the function `entry` that serves as the entrypoint of the program. After Drtti analysis, there were the three published functions in the Delphi source code, the entrypoint, and nine core methods that every object has.

6.1.2 Method References. Function pointers are widely used in Delphi to manage callbacks from graphical components to event handlers[8]. The following Delphi module demonstrates how to model a clickable button, with an `OnClick` handler that could be triggered when the user interacts with it. The `TNotifyEvent` type already exists in the Delphi standard library, but a custom implementation has been included here for completeness, rather than importing the standard one.

Button.pas

```
TNotifyEvent = procedure(Sender: TObject) of object;

TButton = class
private
  FOnClick: TNotifyEvent;
public
  procedure Click;
published
  property OnClick: TNotifyEvent read FOnClick write FOnClick;
end;
```

Note that the declaration of `TNotifyEvent` ends with `of object`, which is a pattern in Delphi that indicates that the function is a method that will be called on an instance. Instead of a regular function pointer, these are stored as *method references* – eight-byte structures consisting of one pointer to the method and one to the object instance.

[8]Embarcadero online Delphi documentation: What Are Events? https://docwiki.embarcadero.com/RADStudio/Sydney/en/What_Are_Events%3F

Delphi

```
function TRectangle.GetHeight: Integer;
begin
  Result := LowerRight.Y - UpperLeft.Y;
end;
```

```
function TRectangle.GetWidth: Integer;
begin
  Result := LowerRight.X - UpperLeft.X;
end;
```

```
function TRectangle.CalculateArea: Integer;
begin
  Result := Width * Height;
end;
```

Ghidra+Drtti

```
Longint TRectangle::GetHeight(TRectangle *this) {
  return this->LowerRight->Y - this->UpperLeft->Y;
}
```

```
Longint TRectangle::GetWidth(TRectangle *this) {
  return this->LowerRight->X - this->UpperLeft->X;
}
```

```
Longint TRectangle::GetArea(TRectangle *this) {
  Longint LVar1;
  Longint LVar2;

  LVar1 = GetWidth(this);
  LVar2 = GetHeight(this);
  return LVar1 * LVar2;
}
```

Listing 8: Comparison of the source code of methods for the `TRectangle` class with Drtti assisted Ghidra decompilation output.

TMethodReference

Offset	Length	DataType	Name
0x0	0x4	TNotifyEvent *	method
0x4	0x4	TObject *	instance

TButton

Offset	Length	DataType	Name
0x0	0x4	TButtonClass *	tclass
0x4	0x8	TMethodReference	OnClick

Figure 6: `TButton` and `TMethodReference` layout with Drtti

TAnimal (Animal.pas)

```
8   TAnimal = class
9   protected
10    FName: string;
11  published
12    function GetSound: string; virtual; abstract;
13    property Sound: string read GetSound;
14    property Name: string read FName;
15  end;
```

Listing 9: Animal class

The structures registered by Drtti in Ghidra's data type manager for the `TButton` module are shown in Figure 6. The `TButton` type consists of a class reference and a method reference. The `TNotifyEvent` type is represented using Ghidra's *function definition datatype*, a template for a function with a specific signature, in this case it was defined as:

```
Void TNotifyEvent(TButton *this, TObject *Sender) .
```

6.1.3 Virtual Methods & Inheritance. The following Delphi module demonstrates how a hierarchy of objects can be modeled. The `TAnimal` base class, represented in Listing 9, serves as an abstraction of any animal, and provides a virtual method and a property for retrieving the sound, as well as a constructor and a field for the animal's name.

In this case, a class `TDog`, represented in Listing 10, inherits from `TAnimal` and overrides the `GetSound` method. The implementation simply returns the string `'Woof!'`. The `TDog` class constructor also takes a breed as a parameter, which is exposed as a published property.

As with the `TRectangle` example in Section 6.1.1, the `TAnimal` and `TDog` classes were correctly registered with Drtti in Ghidra's data type manager, with the layouts shown in Figure 7. The function definition data types were created for virtual methods, making the signatures readable directly in the class structures themselves.

Listing 11, presents the code for a simple Delphi application that uses `TAnimal` to print the name, breed, and sound of a `TDog`. Ghidra correctly recognized the function calls and some of the strings that

TDog (Animal.pas)

```
17  TDog = class(TAnimal)
18  private
19    FBreed: string;
20  public
21    constructor Create(AName: string; ABreed: string);
22    function GetSound: string; override;
23    property Breed: string read FBreed;
24  end;
```

Listing 10: Dog class

TAnimal

Offset	Length	DataType	Name
0x0	0x4	TAnimalClass *	tclass
0x4	0x4	char *	Name

TAnimalClass

Offset	Length	DataType	Name
0x0	0x4	char *GetSound(TAnimal *this) *	GetSound

TDog

Offset	Length	DataType	Name
0x0	0x4	TDogClass *	tclass
0x4	0x4	char *	Name
0x8	0x4	char *	Bark

TDogClass

Offset	Length	DataType	Name
0x0	0x4	char *GetSound(TDog *this) *	GetSound

Figure 7: Memory layout of `TAnimal`, and `TDog`, as well as the `TAnimalClass` and `TDogClass` structures, as reported by Ghidra's data type manager with Drtti.

provide context. Important clues, such as the fact that an object is being instantiated and that a virtual method is called, are, however, difficult to see in the Ghidra output. With the Drtti-generated structures in Figure 7 and the correct constructor function signature, the code is significantly more readable.

Note that the `TDog` constructor parameters recovered by Drtti in Listing 11, line 3, do not exactly match the Delphi source code. This is because the Delphi compiler injects additional hidden parameters, including a reference to the class being instantiated and a flag indicating whether memory should be allocated. The flag is necessary to distinguish between a call to a super constructor from within a child constructor and the site of instantiation.

Drtti was unable to recover any additional information about the functions `FUN_00403034` and `FUN_0040395f`, as no RTTI was available for them. Context clues suggest that the former concatenates strings into a buffer, and that the latter is responsible for actually printing the buffer. This was confirmed by decompiling the binary with IDR and importing the symbols with Drtti.

Delphi

```
var
  Pet: TAnimal;
begin
  Pet := TDog.Create('Fido', 'Golden Retriever');
  Writeln(Pet.Name, ' the ', TDog(Pet).Breed, ' says ', Pet.Sound);
end.
```

Ghidra

```
void entry(void) {
  ...
  puVar2 = FUN_00403ea0(&PTR_LAB_00403e10, '\x01', &DAT_00404098, "Golden
  ↪  Retriever");
  (**(code **)*puVar2)(puVar2,local_14);
  puVar1 = local_14[0];
  puVar3 = FUN_00403034(PTR_DAT_004050b0, puVar2[1]);
  puVar3 = FUN_00403034(puVar3, " the ");
  puVar3 = FUN_00403034(puVar3, puVar2[2]);
  puVar3 = FUN_00403034(puVar3, " says ");
  puVar3 = FUN_00403034(puVar3, puVar1);
  FUN_0040395f(puVar3);
  ...
}
```

Ghidra+Drtti

```
void entry(void) {
  ...
  dog = TDog::Create(&TClass::TDogClass, true, "Fido", "Golden Retriever");
  pcVar2 = (*dog->tclass->GetSound)(dog);
  pcVar1 = FUN_00403034(PTR_DAT_004050b0, dog->Name);
  pcVar1 = FUN_00403034(pcVar1, " the ");
  pcVar1 = FUN_00403034(pcVar1, dog->Breed);
  pcVar1 = FUN_00403034(pcVar1, " says ");
  pcVar2 = FUN_00403034(pcVar1, pcVar2);
  FUN_0040395f(pcVar2);
  ...
}
```

Listing 11: Comparison of the Delphi source code of an example that uses `TAnimal` to print out the name, breed and sound of a `TDog`, with the relevant section of the corresponding Ghidra output before and after Drtti has been run. Limited manual re work, i.e. the removal of redundant type casts and the naming of the variable dog, has been done to improve readability.

6.2 RQ2: Symbols Extraction with Drtti

Table 5: Compact quantitative results on Delphi dataset. Counts are totals over all binaries; coverage values are unweighted averages. Functions exclude thunks.

Metric	Ghidra	Drtti	Drtti+IDR
Total symbols	628754	658285	2620399
Functions	75959	86227	103546
Named functions	459	3409	52065
Datatypes	10544	48948	50543
Classes	0	8824	8824
Fields	0	24008	24008
Exec. data coverage	11.02%	14.32%	40.82%
Exec. instruction coverage	46.55%	46.84%	46.84%
Exec. total coverage	57.57%	61.15%	87.66%

Table 5 shows that the main benefit of Drtti is structural rather than simply additional instruction discovery. Standard Ghidra recovers no Delphi class structures on this corpus, whereas Drtti recovers 8824 binary-level class structures and 24008 fields. Average executable-memory coverage increases from 57.57% to 61.15% with Drtti alone. Optional IDR import further increases named functions and data coverage, but it does not change the recovered class or field counts and does not increase instruction coverage. Thus, for RQ2, recovered RTTI helps reverse engineering mainly by turning anonymous binary regions into a navigable object model, while IDR adds complementary runtime and library naming context.

6.3 RQ3: Symbols from Delphi's Std Library

The gap in Figure 8 reflects the complementary roles of Drtti and IDR. Drtti increases the function set by following Delphi-specific structures such as VMTs and method tables, which expose application-level entry points that standard Ghidra analysis may leave unstructured. The optional IDR import adds a further increase because its knowledge base identifies additional Delphi runtime and library routines. Across the ten-binary corpus, the function count increases from 86227 with Drtti alone to 103546 with Drtti+IDR. This makes IDR valuable for runtime and library coverage, while the recovered class and field structures still come from Drtti's RTTI-based analysis.

7 Discussion

Information Recovery Capabilities of Delphi RTTI

The evaluation confirms that Drtti can reliably extract class layouts, field offsets, explicit property accessors, method signatures, and VMT. However, this recovery is constrained by Delphi's *"opt-in"* approach to RTTI. The compiler primarily emits metadata to support the runtime component bindings within the VCL. For other parts of the program, RTTI will only be available if there if the developer explicitly constructs the classes with it in mind.

Symbolic recovery is limited to components declared with the `published` section. Core procedural logic residing inside private methods or standalone subroutines will likely lack accompanying RTTI records. In the `ProdMod2` case study, the localized forest growth calculation algorithms lacked RTTI for the most part.

Furthermore, as observed with the `TSimForm` button handler methods, the recovered names are entirely dependent on the original developer's naming conventions. Generic or non-descriptive source identifiers will clearly limit the usefulness of recovered metadata.

Impact of RTTI on Decompilation Semantic Accuracy

While Drtti does not replace algorithmic decompilation, it can provide the analyst with valuable information that contextualizes the instructions. By injecting typed object structures and explicit class scopes before control-flow analysis, arbitrary pointer arithmetic can be transformed into meaningful expressions for objects, such as method calls or field accesses.

In the audit of the legacy `ProdMod2` system, this transition significantly reduced analysis duration. Instead of manually mapping every GUI component and its linkages to trace the event loop, the parsed metadata exposed the application architecture. This allowed the analyst to locate callbacks of interest in an event-driven system quicker and more easily.

The fact that Ghidra cannot output Delphi code directly does not necessarily pose a problem. The core concepts of the Delphi language can mostly be modeled within Ghidra's decompilation engine. The programs presented in Section 5.3 provide a concrete example of this, highlighting how closely Ghidra's decompiled output matches the source code.

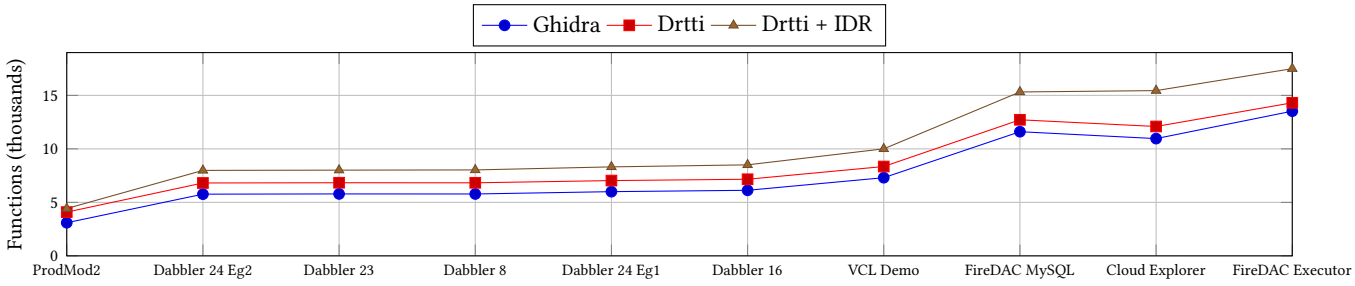


Figure 8: Functions available in Ghidra before and after Drtti, with and without optional IDR symbols on the dataset

However, the evaluation exposed clear boundaries to this semantic improvement, driven by the architectural constraints of the underlying decompiler:

- **Calling Convention Mismatches:** Ghidra sometimes struggles with the *Borland Fastcall* convention, especially with *Self* and return-value storage pointer parameters. Optimization anomalies or FPU register usage can also cause Ghidra to misidentify parameters. Because neither the RTTI nor the information exported from IDR explicitly records register assignments, Drtti is currently unable to resolve these issues.
- **Control Flow Disruption:** Delphi 4's exception handling, which uses *Structured Exception Handling* (SEH), can scramble Ghidra's control flow graphs, leading to split or merged functions. While Drtti can hint at function boundaries via method tables, fully resolving this issue requires improvements to Ghidra's exception-handling parser.
- **Type Propagation Failures:** Ghidra propagates types via data-flow analysis. If an intermediate P-code instruction performs an opaque bitwise operation or an unresolved indirect call, the type binding can be broken. The register reverts to a raw pointer (e.g., `undefined4 *`), forcing the analyst to manually resolve the issue.

IDR-Independent Recovery and Optional Library Symbols

IDR is no longer a prerequisite for the core Delphi recovery workflow. Drtti recovers the application object model directly from the binary: both Drtti configurations recover 291 classes and 2269 fields on ProdMod2, while standard Ghidra recovers no Delphi class structures. The FPU fingerprinting pass follows the same principle for mathematical code: it uses rare x87 instruction patterns rather than external byte signatures or an IDR knowledge base.

The set comparison further shows that Drtti is not merely a subset of IDR. Across the source-available corpus, 69.3–74.0% of Drtti class names also appear in the IDR-derived exports, but the Jaccard similarity remains only 21.1–50.0%. Thus, Drtti recovers a substantial shared structural core independently, while IDR mainly contributes additional library naming context. As shown in Table 13, Drtti is not merely a subset of IDR.

IDR remains useful, but its role is complementary. Its version-specific knowledge base provides additional RTL and VCL names, which explains the increase in named functions in Table 7. However, it does not increase instruction coverage beyond RTTI-informed analysis in Table 6. The contribution is therefore not simply adding IDR symbols to Ghidra, but moving the central Delphi recovery steps into a Ghidra-native, knowledge-base-independent workflow.

This distinction also matters for reproducibility: IDR is a standalone Windows-oriented tool, whereas Drtti runs inside Ghidra's cross-platform analysis environment.

Robustness and Deployment Considerations

Another consideration for the utility of Drtti and RTTI analysis in general, for security and software auditing workflows, is its resilience against binary hardening techniques. As mentioned, the compiler strips symbols and prunes unreferenced objects and methods from final compilation units by default to control artifact sizes. Any component that is bound to the core VCL architecture, or explicitly declared under a `published` header, must, however, be accessible. Delphi's runtime environment requires that string identifiers for methods and fields, and VMT be present so that GUI layouts can be constructed at runtime. Consequently, even in aggressively optimized or production-stripped binaries, metadata about visual components will always be available to Drtti.

Against commercial binary packers and obfuscators, such as UPX[9] and Themida[10], static extraction frameworks naturally require a preparatory unpacking phase. If an executable's sections are encrypted or compressed on disk, Drtti cannot locate any. Given that it has been unpacked beforehand, the packing process itself is unlikely to introduce any additional issues.

Delphi Compatibility The current implementation of Drtti is engineered around Delphi 4 and 8 VMT layouts. Supporting additional version requires to manually verify the different formats of the VMT and adapt Drtti to handle the various offsets and potentially new or different fields of each VMT version.

Incomplete RTTI Extraction Drtti currently does not process a few internal structures containing some RTTI. The ones that are known to exist, but have not yet been implemented, are described in Appendix A, Section A.1 and Section A.4. Due to the nature of the missing data and the lack of official documentation on the subject, it is difficult to estimate how much there is left to process and what impact it would have on Drtti's RTTI capabilities.

Serialized Forms A potential resource within Delphi executables that Drtti could benefit from processing is the `.rsrc` segment of the binary. Among other things, this block holds serialized GUI layout data. Component properties like position, size, and crucially, event-handlers bindings. For instance, a button component entry in the resource block explicitly pairs the text token `OnClick` with the

[9] https://github.com/upx/upx
[10] https://www.oreans.com/Themida.php

name of a designated handler method, such as `TMainForm.Framen-period1Click`. Future versions of Drtti could integrate a dedicated `.rsrc` parsing subsystem. By correlating the strings found in the resource blocks with the function pointers and types extracted from VMT structures, one could construct a lookup table of components and their handlers. This could reduce the amount of exploratory data-flow tracking, such as the string searches in the ProdMod2 case study, that the analyst is required to perform.

Sample size Drtti has only been tested on a limited number of Delphi programs. A wider array of sample programs of different scales compiled with different versions would increase the accuracy of the results and help determine the value of Drtti.

Manual comparison of semantic similarity As mentioned in Section 5.3, a manual comparison of semantic similarity was used to determine the accuracy of Drtti's RTTI recovery. Future work could explore using natural language processing (NLP) or AI techniques to automatically measure similarity.

8 Conclusion

This paper investigates the challenges of reverse engineering Delphi applications and explored how RTTI can be leveraged to improve decompilation results.

Results indicate that a variety of Delphi constructs can be recovered: Drtti has been shown to reliably provide Ghidra with class names, class hierarchies, virtual method tables, object sizes, field offsets, and method signatures, all generated directly from the RTTI.

We show that this information can be used to significantly improve decompilation results of a modern reverse engineering tool that is not designed specifically for Delphi, Ghidra. With the help of Drtti, opaque pointer arithmetic expressions are successfully transformed into typed object field accessors and method calls.

Across the ten-binary evaluation corpus, standard Ghidra analysis mapped 57.57% of executable memory to known data or instructions on average, while Drtti alone reached 61.15% and Drtti with optional IDR symbols reached 87.66%. These results show that Drtti provides IDR-independent structural recovery inside Ghidra, while optional IDR symbols mainly add library naming and data context rather than forming a dependency of the recovery pipeline.

Open Science

The implementation is available at this link: **https://github.com/software-engineering-and-security/Drtti**

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

This paper was edited for grammar using Claude AI.

References

[1] Borland®. 1998. *Object Pascal Language Guide* (delphi 4™ ed.). Borland International, Inc., 100 Borland Way, Scotts Valley, CA 95067-0001.

[2] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting Decompiler Output with Learned Variable Names and Types. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4327–4343.

[3] crypto2011. 2019. Interactive Delphi Reconstructor. https://github.com/crypto2011/IDR. Accessed 2026-06-23.

[4] crypto2011. 2019. KBBUILDER. https://github.com/crypto2011/KBBUILDER. Accessed 2026-06-23.

[5] DelphiDabbler. 2026. article-demos: Demo Code Relating to DelphiDabbler Articles. https://github.com/delphidabbler/article-demos. Accessed 2026-06-26.

[6] Per Magnus Ekö. 1985. *En produktionsmodell för skog i Sverige, baserad på bestånd från riks-skogstaxeringens provytor*. Ph. D. Dissertation.

[7] Embarcadero Technologies. 2026. RAD Studio 12 Demos. https://github.com/Embarcadero/RADStudio12Demos. Accessed 2026-06-28.

[8] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *2011 18th Working Conference on Reverse Engineering*. 347–356. doi:10.1109/WCRE.2011.49

[9] Alexander Fokin, Katerina Troshina, and Alexander Chernov. 2010. Reconstruction of class hierarchies for decompilation of C++ programs. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 240–243.

[10] Harrison Green, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2024. STRIDE: Simple type recognition in decompiled executables. *arXiv preprint arXiv:2407.02733* (2024).

[11] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan) *(ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 637–651. doi:10.1145/3320269.3384766

[12] Intel Corporation. 2026. *Intel 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference*. Accessed 2026-06-23.

[13] Ray Lischner. 2000. *Delphi in a Nutshell* (first edition ed.). O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472-9902.

[14] Sam L. Thomas Lukas Seidel and Konrad Rieck. 2026. Practical Type Inference: High-Throughput Recovery of Real-World Structures and Function Signatures . In *17th ACM Conference on Data and Applications Security and Privacy (CODASPY)*. 1–12.

[15] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[16] Microsoft. 2026. PE Format. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format. Accessed 2026-06-29.

[17] Vaughan Pratt. 1995. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–107.

[18] Paul Vincent Sabanal and Mark Vincent Yason. 2007. Reversing C++. *Black Hat DC* (2007).

[19] VCL Styles Utils Contributors. 2026. VCL Styles Utils. https://github.com/RRUZ/vcl-styles-utils. Accessed 2026-06-27.

[20] Yanzhong Wang, Ruigang Liang, Yilin Li, Peiwei Hu, Kai Chen, and Bolun Zhang. 2025. TypeForge: Synthesizing and Selecting Best-Fit Composite Data Types for Stripped Binaries . In *2025 IEEE Symposium on Security and Privacy (SP)*. 1–18. doi:10.1109/SP61157.2025.00193

[21] Haohuang Wen and Zhiqiang Lin. 2023. Egg hunt in Tesla infotainment: a first look at reverse engineering of Qt binaries . In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 224, 18 pages.

[22] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, Mayur Naik, Yan Shoshitaishvili, Ruoyu Wang, and Aravind Machiry. 2024. TYGR: Type Inference on Stripped Binaries using Graph Neural Networks. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4283–4300.

Appendices

A Delphi 4 – Key Internal Structures

A.1 Virtual Method Table

The virtual method table is an important structure in the implementation of object orientation in Delphi. It is used to store information about the methods that an object may have, as well as some basic information about the class, such as its name, parent class, base methods common for all objects. It also contains references to various RTTI information that is used at runtime. For the layout of the ones that are processed by Drtti, see Sections A.2-A.4.

VMT (76 bytes)			
Offset Drtti	**Description**	**Type**	**Processed by**
-0x4C	Class self reference	pointer	✓
-0x48	InterfaceTable	pointer	×
-0x44	AutomationTable	pointer	×
-0x40	InitializationTable	pointer	×
-0x3C	TypeInfo	pointer	✓
-0x38	FieldInfo	pointer	✓
-0x34	MethodTable	pointer	✓
-0x30	DynamicMethodTable	pointer	×
-0x2C	Name	pointer to string	✓
-0x28	Instance size	integer	✓
-0x24	Ancestor class	pointer	✓
-0x20	SafeCallException	function pointer	✓
-0x1C	AfterConstruction	function pointer	✓
-0x18	BeforeDestruction	function pointer	✓
-0x14	Dispatch	function pointer	✓
-0x10	DefaultHandler	function pointer	✓
-0x0C	NewInstance	function pointer	✓
-0x08	FreeInstance	function pointer	✓
-0x04	Destroy	function pointer	✓
0x0	Virtual methods	function pointer[...]	✓

A.2 Field Table

The published field table starts with a 2-byte unsigned integer containing a count of the number of fields, followed by a 4-byte pointer to a class table. The class table lists all the classes used by the published fields. Each field contains an index into this table. The class table starts with a 2-byte count followed by a list of class references where each class reference is 4 bytes. A class reference is just a pointer to the class's VMT. Following the class table reference is a series of field definition entries.

FieldTable (6 + 4 × Count bytes)		
Offset	**Description**	**Type**
0x0	Count	unsigned word
0x2	Class table	pointer
0x6	Fields	FieldEntry[Count]

Each field definition consists of an unsigned 4-byte integer offset, and a 2-byte unsigned integer into the index into the class table, followed by the field name as a length-prefixed pascal style string.

FieldEntry (6 + size(Name) bytes)		
Offset	**Description**	**Type**
0x0	Offset within object	unsigned integer
0x4	Class table offset	unsigned word
0x6	Name	pascal string

A.3 Method Table

The method table starts with a 2-byte unsigned integer containing a count of the number of methods, followed by a series of method table entries detailing the methods of a class.

MethodTable		
Offset	**Description**	**Type**
0x0	Count	unsigned word
0x2	Method records	MethodEntry[Count]

Each method entry starts with an unsigned 2-byte integer, specifying the size of the method record, followed by the method entrypoint (4 bytes), and then followed by the method name as a length-prefixed pascal style string. The record size usually is 6 + size(name), but some method records contain information about parameters. The record size information lets one know when this data is present as the reported size will be larger than it needs to be to store the entry size, method address, and method name. If the size is 4 bytes larger than it needs to be, that means the method takes no parameters (except for the object reference). If the size is more than 6 bytes larger than it needs to be, the information for the methods parameters are stored following the sixth byte. Each **ParameterInfo** structure has a pointer to a **TypeInfo** structure, see Section A.4, for the parameters type, followed by the parameters name as a pascal style string, followed by a trailing zero-byte.

MethodEntry		
Offset	**Description**	**Type**
0x0	Entry size	unsigned word
0x2	Entrypoint	pointer
0x6	Name	pascal string
0x6+size(Name)	Parameters	ParameterInfo[...]

A.4 TypeInfo

The **TypeInfo** structure is a so-called tagged union. The value of the first byte of each structure is used to determine what type of structure it is. There are eighteen types of **TypeInfo** structures in Delphi 4, and each has a different layout. The following table shows the values with their corresponding type names, and whether they are currently supported by Drtti or not.

Value	Name	Drtti support	Section
0	Unknown	×	
1	Integer	✓	Section A.5
2	Char	✓	Section A.5
3	Enumeration	✓	Section A.6
4	Float	✓	Section A.7
5	String	✓	Section A.8
6	Set	×	
7	Class	✓	Section A.11
8	Method	✓	Section A.12
9	Wide Char	✓	Section A.5
10	Long String	✓	Section A.8
11	Wide String	✓	Section A.8
12	Variant	✓	Section A.9
13	Array	✓	Section A.10
14	Record	×	
15	Interface	×	
16	Int64	✓	Section A.5
17	Dynamic Array	×	

A.5 Ordinal

Every ordered number (words or symbols) store type information with the following layout. First, the name of the type is stored as a pascal string, and then the ordinal type is stored as a byte, followed by min and max values that can be represented by that type.

OrdinalTypeInfo		
Offset	**Description**	**Type**
`0x0`	Name	Pascal string
`Size(Name)`	Ordinal type	Byte
`Size(Name)+0x1`	Min value	4-byte integer
`Size(Name)+0x5`	Max value	4-byte integer

The ordinal type is one of:

0: `otSByte` – A signed byte.
1: `otUByte` – An unsigned byte.
2: `otSWord` – A signed word (2-bytes).
3: `otUWord` – An unsigned word (2-bytes).
4: `otSLong` – A signed long (4-bytes).
5: `otULong` – An unsigned long (4-bytes).

The ordinal type signifies the number of bytes that are required to store the datatype, however, the final datatype representation will depend on whether the type info is for a character, number or enumeration.

A.6 Enumeration

The type info for enumerations is identical to that of ordinals, except for an additional list of enumeration entry names that follow the max value. For most types, the number of entries is simply $max - min + 1$, however, for certain types that exist for compatibility with other languages this does not hold.

As an example, the `Boolean` datatype in Delphi defines zero as false and one as true. This is represented internally by an enumeration with two entries, "false" and "true" and corresponding min and max values 0 and 1. The `WordBool` datatype exists so that Delphi can be compatible with languages in which zero is false and any non-zero value is true, in this case the number of entries in the enumeration is still only two ("false" and "true") but the min and max values are the limits of a signed integer (0x80000000 and 0x7FFFFFFF), so the calculation of $max - min + 1$ does not work and a special case must be made.

EnumerationTypeInfo		
Offset	**Description**	**Type**
`0x0`	Name	Pascal string
`Size(Name)`	Underlying ordinal type	Byte
`Size(Name)+0x1`	Min value	4-byte integer
`Size(Name)+0x5`	Max value	4-byte integer
`Size(Name)+0x9`	Enumeration names	Pascal strings[max-min+1]

A.7 Float

Floating point datatypes store their name, as well as a float type which signifies how the float should be interpreted.

FloatTypeInfo		
Offset	**Description**	**Type**
`0x0`	Name	Pascal string
`Size(Name)`	Float type	Byte

The float type is one of:

0: `ftSingle` – `Single` type.
1: `ftDouble` – `Double` type.
2: `ftExtended` – `Extended` type.
3: `ftComp` – `Computational` type.
4: `ftCurr` – `Currency` type.

A.8 String

The type information for strings is minimal. For pascal strings, the maximum length is stored as a byte, for other types of strings there is no RTTI at all.

StringTypeInfo		
Offset	**Description**	**Type**
`0x0`	Max length	Byte

A.9 Variant

Variants represent a union of types, and thus since there is no way to know which type the variant contains, the amount of RTTI is minimal. In fact, the only type information that is stored is the length, in bytes, of the largest type.

VariantTypeInfo		
Offset	**Description**	**Type**
`0x0`	Length	Unsigned word

A.10 Array

The type info for arrays contains the element size, type and total number of elements. For multi-dimensional arrays, the type info also contains the number of dimensions, as well as the types used to for every dimension. Delphi commonly uses the `Set` type to allow arrays to have other indices than zero to the size of the array.

ArrayTypeInfo		
Offset	**Description**	**Type**
`0x0`	Element size	Unsigned integer
`0x04`	Element count	Unsigned integer
`0x08`	Element type	TypeInfo pointer
`0x12`	Dimension count	Unsigned byte
`0x13`	Dimension type vector	TypeInfo pointer[DimCount]

A.11 Class

The type info for a class contains the name of the class, a reference to the class, as well as a pointer to the type info for the parent class. The main contents of the class type info however is the RTTI for the published properties of the class. The total property count is the number of properties introduced by a specific class, as well as any properties introduced by the parent class. A list of the published properties declared in a specific the class is attached to the end of the class type info.

ClassTypeInfo		
Offset	**Description**	**Type**
`0x0`	Name	pascal string
`Size(Name)`	Class reference	VMT pointer
`Size(Name)+0x04`	Parent type	TypeInfo pointer
`Size(Name)+0x08`	Total property count	unsigned word
`Size(Name)+0x10`	Unit name	pascal string
`Size(Name)+0x10 +Size(UnitName)`	Property count	unsigned word
`Size(Name)+0x12 +Size(UnitName)`	Properties	PropertyInfo[PropertyCount]

Every property entry includes the type and name of the property as well as the *accessor*. There is also additional metadata about how the property is stored in memory, the index (if it is an indexed property accessor), and a default value (if the property has a default value).

PropertyInfo		
Offset	**Description**	**Type**
`0x0`	Type	TypeInfo pointer
`0x04`	Getter	4-byte integer
`0x08`	Setter	4-byte integer
`0x12`	Storage specifier	4-byte integer
`0x16`	Index value	Unsigned integer
`0x20`	Default value	4-byte integer
`0x24`	Name index	Unsigned word
`0x26`	Name	Pascal string

The **Getter** and **Setter** fields can contain different data depending on what they reference. To determine the type in the getter and setter fields, one must check the value of the highest byte. If it is 0xFF, then field refers directly to a field and the lowest three bytes should be interpreted as an offset into the object instance. If the highest byte is 0xFE, then the field represents a virtual method and the lowest two bytes should be interpreted as the index of a virtual method for the class. If the highest byte is neither then the getter or setter should be interpreted as the entrypoint of a function.

A.12 Method

Method type info is generated for named function pointer types such as:

```
TFoo = procedure(X: Integer);
TBar = function(): String of object; .
```

The type info for a method stores its name, the kind of method, the number of parameters, the return type name (if any), and metadata about the parameter types.

MethodTypeInfo		
Offset	**Description**	**Type**
`0x0`	Name	Pascal string
`Size(Name)+0x4`	Method type	Byte
`Size(Name)+0x5`	Parameter count	Byte
`Size(Name)+0x6`	Return type name	Pascal string
`Size(Name)+0x6 +Size(ReturnType)`	Parameter types	ParamTypeInfo[ParameterCount]

The method type is one of:

Value	Name	Description
0	`mkProcedure`	Function without a return value, callable on an object instance.
1	`mkFunction`	Function with a return value, callable on an object instance.
2	`mkConstructor`	Constructor that creates a new object of a class.
3	`mkDestructor`	Destructor that destroys an object of a class.
4	`mkClassProcedure`	Static function without a return value, callable on the class, without an object instance.
5	`mkClassFunction`	Static function with a return value, callable on the class, without an object instance.
6	`mkClassConstructor`	Constructor that is called once when the class is first initialized.
7	`mkClassDestructor`	Destructor that is called when the class is destroyed.
8	`mkOperatorOverload`	Overloaded operator.
9	`mkSafeProcedure`	Same as `mkProcedure` but effectively surrounded by try-catch.
10	`mkSafeFunction`	Same as `mkFunction` but effectively surrounded by try-catch.

It is as of writing not confirmed which of these value can be generated by a Delphi 4 compiler and which were introduced later, although preliminary testing suggests that it is likely that only values 0 through 5 are applicable.

Each parameter contains the name type of the parameter and the name of the parameter itself as pascal style strings, as well as a single byte containing a set of bitwise flags.

ParamTypeInfo		
Offset	**Description**	**Type**
`0x0`	Flags	Byte
`0x1`	Name	Pascal string
`0x1+Size(Name)`	Type name	Pascal string

The flag byte for a parameter contains information about modifiers such as `const` and `var`:

Table 6: Percentage of the ProdMod2 binary mapped to known data or instructions in Ghidra. IDR improves data naming but does not increase instruction coverage over RTTI-informed analysis.

Memory	Type	Ghidra	Drtti	Drtti + IDR
Executable	Data	7.26%	19.76%	21.52%
Executable	Instruction	68.87%	72.03%	72.03%
Executable	Total	76.14%	91.78%	93.54%
Global	Data	59.42%	64.71%	65.45%
Global	Instruction	29.13%	30.46%	30.46%
Global	Total	88.55%	95.17%	95.91%

Table 7: Amount of symbolic information available in Ghidra for the ProdMod2 binary before and after Drtti, with and without imported IDR symbols. Functions exclude thunks. Named functions are functions whose name does not follow Ghidra's default `FUN_` naming convention.

#	Ghidra	Drtti	Drtti + IDR
Total symbols	16835	21965	26770
Functions	3096	4089	4425
Named functions	1	760	4048
Datatypes	709	2358	2358
Classes	0	291	291
Fields	0	2269	2269
Strings	1970	4948	5267

Value	Name	Description
`0b1`	`var`	Parameter is passed by reference, not by value.
`0b10`	`const`	Parameter is a constant.
`0b100`	`array`	Parameter is a dynamic array. texttt
`0b1000`	`pointer`	Parameter is a pointer.
`0b10000`	`reference`	Parameter is a reference (pointer to non-null object).
`0b100000`	`out`	Parameter is passed by reference, and is where a value of the type should be stored. The initial value before calling the function is discarded.

B ProdMod2

B.1 Methodology

A case study on *ProdMod2*, a forest growth simulation program that was written in Delphi 4 by the Swedish forestry research institute Skogforsk, based on the production model by Ekö [6]. The executable was compiled in 1998 for Windows NT, and the source code has been lost.

The goal of the study was to use Drtti to discover where the logic that advances the simulation forward is located, and to recover formulas used to calculate the next state.

The analysis of ProdMod2 utilized a combination of static and dynamic analysis. The application was run under a debugger in a QEMU[11] vm that simulates a Windows NT environment to examine

[11] https://www.qemu.org/

Table 8: Amount of available symbolic information in Ghidra with and without the Drtti plugin, as well as with optional IDR symbols, for the ProdMod2 binary. The number of functions excludes thunks.

# IDR Symbols	Ghidra	Drtti	Drtti with
Total symbols	16835	21965	26770
Functions	3096	4089	4425
Named Functions	1	760	4048
Datatypes	709	2358	2358
Classes	0	291	291
Fields	0	2269	2269
Strings	1970	4948	5267

Table 9: Percentage of the ProdMod2 binary mapped to known data or instructions in Ghidra, with Drtti, and with optional IDR symbols.

Memory	Type	Ghidra	Drtti	Drtti with IDR Symbols
Executable	Data	7.26%	19.76%	21.52%
	Instruction	68.87%	72.03%	72.03%
	Total	76.14%	91.78%	93.54%
Global	Data	59.42%	64.71%	65.45%
	Instruction	29.13%	30.46%	30.46%
	Total	88.55%	95.17%	95.91%

the runtime behavior and inform the static analysis. Ghidra was used to decompile the binary using Drtti with and without IDR symbols included, and quantitative metrics (see Section 5.4), were collected.

The formula recovery process followed a specific sequence of steps that were repeated in a loop until the desired information was extracted:

(1) Visual inspection of the GUI to identify an interactive component of interest.
(2) Search for strings related to the component.
(3) Explore references to the string.
(4) Attempt to identify the handler function for the component using the text and the references.
(5) Verify the handler identification using a debugger.
(6) Examine functions called from the handler under a debugger to correlate the values visible in the GUI with the parameters passed to those functions.

B.2 Overview

The amount of symbolic information available in Ghidra before and after Drtti has been run on the ProdMod2 binary is shown in Table 8. For all type information, Drtti performed the same with and without IDR symbols, and no standard Ghidra analysis generated anything close to the class structures. The function data shows a 32% increase using only RTTI extraction and a 43% increase when using both RTTI and IDR symbols, compared to Ghidra. Before using Drtti, the only known named function was the entrypoint,

which Ghidra names `entry`. The IDR symbol import resulted in a 530% increase in the number of named functions, compared to relying only on RTTI. The number of known strings increased by 251% and 267%, respectively, when using Drtti with pure RTTI and IDR symbols, compared to Ghidra's analysis.

The total memory footprint of the ProdMod2 binary was 1.72MB, of which 0.73MB is marked as *executable memory*, that is, memory containing both code and data, according to Ghidra. Table 9 shows the proportion of the binary that is mapped to some known data type or to instructions in Ghidra before and after applying Drtti and importing IDR symbols. The percentage that Ghidra knows to be instructions increased by 4.6% relative to a standard Ghidra analysis when using Drtti at both the executable and global memory scale. Since essentially all program logic is contained in the executable memory, this was to be expected. Data coverage, the amount of executable memory mapped to some known data type, increased by 270% and 292%, respectively, when using Drtti with and without IDR. In global memory, the increase was less noticeable at 9% and 10%, respectively.

From Table 9, one can see that using Drtti with IDR only reduced the overall program memory entropy – that is, the amount of completely unknown binary data – by 1.9% compared to relying solely on embedded RTTI. The additional data from IDR also did not improve instruction coverage at all, indicating that Drtti already adequately assists Ghidra's own analysis engine with instruction locations using only RTTI. This could indicate that the 336 additional functions recovered when running Drtti with IDR, seen in Table 8, are small. It could also be that Ghidra's data flow analysis lost track of the boundaries for some functions, so they effectively merged. Once Drtti informs Ghidra about these boundaries, the data flow may be corrected, resulting in additional functions without actually mapping any more memory as instructions.

An example of the difference between unmapped and mapped memory is shown in Figure 9. When Ghidra is unsure whether to interpret a memory location as data or an instruction, it simply displays each byte as its hexadecimal value. Once the location is identified as a function entry point, the signature is displayed together with a list of references to the function (the `XREF` part), and the names of instructions and registers are shown.

B.3 Entrypoint Analysis

A common place to start any re work is the entrypoint of the binary. Listings 12, 13, and 14 show the decompiled entrypoint of the binary using Ghidra's standard analysis, Drtti, and Drtti with imported IDR symbols.

The number of default functions and untyped variable names in the entrypoint was 25, 15, and 0, respectively, for Ghidra, Drtti, and Drtti with IDR. For the IDR supported analysis, the actual names of some of the `PTR_DAT_*` variables were still unknown. However, Drtti did recover their types and determined that they were global variables, and thus they were renamed `gvar_*` to convey that information.

In both Listing 13 and Listing 14, the `TApplication` type has been added to the data type manager. There are, however, two main changes introduced by the IDR symbols. The first is that they allowed Drtti to apply method signatures for which there was

Ghidra

```
1  void entry(void) {
2  undefined *puVar1;
3  undefined4 uVar2;
4  undefined4 extraout_ECX;
5
6  FUN_00406ab4(&DAT_004b23d8);
7  if (*PTR_DAT_004b81b8 == 0x2e) {
8    *PTR_DAT_004b81b8 = 0x2c;
9  }
10 uVar2 = FUN_00444ea0(&PTR_LAB_0049d534, '\x01', *PTR_DAT_004b802c);
11 puVar1 = PTR_DAT_004b7fb4;
12 *PTR_DAT_004b7fb4 = uVar2;
13 FUN_004490f8(*PTR_DAT_004b7fb4, puVar1, extraout_ECX);
14 (**(code **)(**(int **)PTR_DAT_004b7fb4 + 0x84))();
15 FUN_0044c14c();
16 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_004af9bc, PTR_DAT_004b7e04);
17 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_004974e0, PTR_DAT_004b7e0c);
18 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_0049772c, PTR_DAT_004b7e34);
19 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_0049d340, PTR_DAT_004b7eb4);
20 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_0048cb5c, PTR_DAT_004b7cf0);
21 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_00497074, PTR_DAT_004b8060);
22 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_0049da84, PTR_DAT_004b7f48);
23 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_00497df4, PTR_DAT_004b81d8);
24 FUN_0044c164(*PTR_DAT_004b802c, &PTR_LAB_004a02c4, PTR_DAT_004b8088);
25 FUN_0044c1e4(*PTR_DAT_004b802c);
26 FUN_00403980();
27 }
```

Listing 12: The decompiled entrypoint of the ProdMod2 binary using Ghidra's standard analysis.

Drtti

```
1  void entry(void) {
2  undefined *puVar1;
3  undefined4 uVar2;
4  undefined4 extraout_ECX;
5
6  FUN_00406ab4(&DAT_004b23d8);
7  if (*PTR_DAT_004b81b8 == 0x2e) {
8    *PTR_DAT_004b81b8 = 0x2c;
9  }
10 uVar2 = FUN_00444ea0(&TSplashForm::TSplashFormClass, '\x01',
   ↪  *PTR_DAT_004b802c);
11 puVar1 = PTR_DAT_004b7fb4;
12 *PTR_DAT_004b7fb4 = uVar2;
13 FUN_004490f8(*PTR_DAT_004b7fb4, puVar1, extraout_ECX);
14 (**(code **)(**(int **)PTR_DAT_004b7fb4 + 0x84))();
15 FUN_0044c14c();
16 FUN_0044c164(*PTR_DAT_004b802c, &TMainForm::TMainFormClass,
   ↪  PTR_DAT_004b7e04);
17 FUN_0044c164(*PTR_DAT_004b802c, &TAboutBox::TAboutBoxClass,
   ↪  PTR_DAT_004b7e0c);
18 FUN_0044c164(*PTR_DAT_004b802c,
   ↪  &TProgEgenskaperForm::TProgEgenskaperFormClass, PTR_DAT_004b7e34);
19 FUN_0044c164(*PTR_DAT_004b802c, &TOpenForm1::TOpenForm1Class,
   ↪  PTR_DAT_004b7eb4);
20 FUN_0044c164(*PTR_DAT_004b802c, &TGallraForm::TGallraFormClass,
   ↪  PTR_DAT_004b7cf0);
21 FUN_0044c164(*PTR_DAT_004b802c,
   ↪  &TDiagramEgenskapForm::TDiagramEgenskapFormClass, PTR_DAT_004b8060);
22 FUN_0044c164(*PTR_DAT_004b802c, &TClipBoarForm::TClipBoarFormClass,
   ↪  PTR_DAT_004b7f48);
23 FUN_0044c164(*PTR_DAT_004b802c, &TSIForm::TSIFormClass, PTR_DAT_004b81d8);
24 FUN_0044c164(*PTR_DAT_004b802c, &TFormNaringA::TFormNaringAClass,
   ↪  PTR_DAT_004b8088);
25 FUN_0044c1e4(*PTR_DAT_004b802c);
26 FUN_00403980();
27 }
```

Listing 13: The decompiled entrypoint of the ProdMod2 binary after running Drtti.

no RTTI. The function with the generic name `FUN_0044c164`, for example, was resolved to the Delphi method `CreateForm(InstanceClass: TComponentClass; var Reference)`.

Secondly, the name and type of some global variables were recovered. The importance of knowing the type is evident in lines 6-9, where the IDR symbolic information reveals that `0x2e` and

Ghidra

```
0x4b075c   55   ??   55h U
0x4b075d   8b   ??   8Bh
0x4b075e   ec   ??   ECh
0x4b075f   33   ??   33h 3
0x4b0760   c9   ??   C9h
0x4b0761   51   ??   51h Q
...
```

Drtti

```
void __register FileOpen1Execute(void)
TMainForm::FileOpen1Execute          XREF[1]: 004b00fd(*)
0x4b075c   55       PUSH    EBP
0x4b075d   8b ec    MOV     EBP,ESP
0x4b075f   33 c9    XOR     ECX,ECX
0x4b0761   51       PUSH    ECX
...
```

Figure 9: Example showing the difference in Ghidra's assembly listing between memory that has been mapped to instructions using Drtti and the same memory location when only Ghidra's default analysis is used.

Drtti + IDR

```
1  void EntryPoint(void) {
2  TCustomForm *pTVar1;
3
4  InitExe();
5  if (*DecimalSeparator == '.') {
6    *DecimalSeparator = ',';
7  }
8  pTVar1 = TCustomForm::Create(&TSplashForm::TSplashFormClass, true,
   ↪  *Application);
9  *gvar_004B7FB4 = pTVar1;
10 (*(*gvar_004B7FB4)->tclass->Update)(*gvar_004B7FB4);
11 TApplication::Initialize(*Application);
12 TApplication::CreateForm(*Application, &TMainForm::TMainFormClass,
   ↪  gvar_004B7E04);
13 TApplication::CreateForm(*Application, &TAboutBox::TAboutBoxClass,
   ↪  gvar_004B7E0C);
14 TApplication::CreateForm(*Application,
   ↪  &TProgEgenskaperForm::TProgEgenskaperFormClass, gvar_004B7E34);
15 TApplication::CreateForm(*Application, &TOpenForm1::TOpenForm1Class,
   ↪  gvar_004B7EB4);
16 TApplication::CreateForm(*Application, &TGallraForm::TGallraFormClass,
   ↪  gvar_004B7CF0);
17 TApplication::CreateForm(*Application,
   ↪  &TDiagramEgenskapForm::TDiagramEgenskapFormClass, gvar_004B8060);
18 TApplication::CreateForm(*Application, &TClipBoarForm::TClipBoarFormClass,
   ↪  gvar_004B7F48);
19 TApplication::CreateForm(*Application, &TSIForm::TSIFormClass,
   ↪  gvar_004B81D8);
20 TApplication::CreateForm(*Application, &TFormNaringA::TFormNaringAClass,
   ↪  gvar_004B8088);
21 TApplication::Run(*Application);
22 Halt0();
23 }
```

Listing 14: The decompiled entrypoint of the ProdMod2 binary after running Drtti with imported IDR symbolic information.

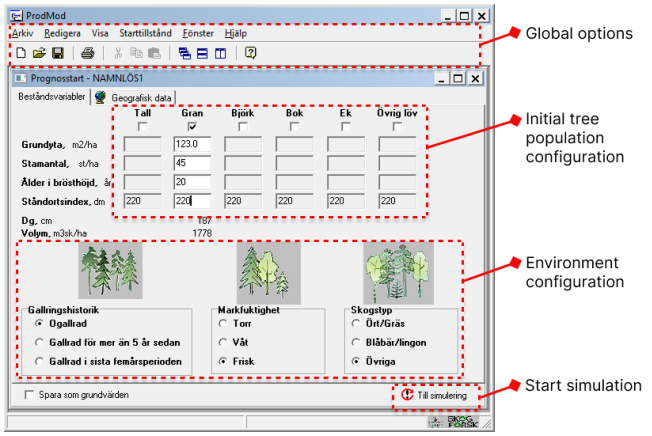


Figure 10: Anatomy of the initial window the user is presented with when starting the application. Important components are outlined in red.

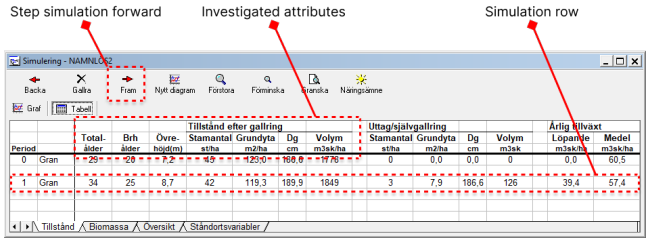


Figure 11: The main simulation window.

`0x2c` should be interpreted as `'.'` and `','`, respectively. Knowing that the `Application` variable was of the type `TApplication` also allowed Ghidra's data flow analysis to perform more accurate type propagation.

Combined, this method and type information allowed Ghidra to create a more comprehensible representation of the entrypoint.

B.4 Simulation Formula Recovery

This section describes the process of recovering formulas for some values of calculated by the ProdMod2 application with the help of the Drtti plugin.

After opening the binary in Ghidra and examining the entrypoint, as described in Section B.3, the application was started inside a vm. Looking at the main application window, shown in Figure 10, the button that starts a simulation has the text `'Till Simulering'`. A search for that string within Ghidra yielded no matches when the standard analysis was run. After running Drtti, it was discovered that this button was part of the `TMainForm` resource block. This form class was known from the entrypoint analysis, see line 12 of Listing 14.

Examination of the methods that Drtti has recovered for `TMainForm` yielded the method `TMainForm::Tillsimulering1Click`, which has a very similar name. A debugger was used to verify that this method is called when a user presses the button. This then opens the main simulation form window, shown in Figure 11, where the simulation runs and where it can be advanced. Another string search for the label on the forward button, `'Fram'`, revealed that the class responsible for this window is called `TSimForm`.

To identify where the values for each simulation row are stored in memory and which calculations transition them from one state to the next, the next step was to examine which methods were called when the button was pressed. Looking at the names of

Drtti + IDR

```
void TMainForm::Framenperiod1Click(TMainForm *this, TObject *Sender) {
Boolean BVar1;
FUN_004b1210(this, Sender);
BVar1 = IsClass((*Screen)->FActiveForm, &TSimForm::TSimFormClass);
if (BVar1 == False) {
  BVar1 = IsClass((*Screen)->FActiveForm, &TDiagramForm::TDiagramFormClass);
  if (BVar1 != False) {
    FUN_004a3188(*(TForm **)&(*Screen)->FActiveForm[1].field_0x8);
  }
} else {
  FUN_004a3188((*Screen)->FActiveForm);
}
return;
}
```

Listing 15: The decompiled `TMainForm::Framenperiod1Click` function in the ProdMod2 binary after running Drtti with imported IDR symbols.

`TSimForm` did not yield any promising results. The event-handlers had generic names that did not convey what the method does, such as `ToolButton1Click`. However, looking through the methods for `TMainForm` an additional time revealed yet another method with a recognizable name: `TMainForm::Framenperiod1Click`. Using the debugger again, it was discovered that this method was not called when the user pressed the forward button in the simulation. Instead, it was discovered that one of the menus at the top of the GUI contains an option named `'Fram en period'`, which steps the simulation forward if it has been opened. This method was then decompiled, and the output is shown in Listing 15. The function name could be recovered using Drtti alone; however, the `IsClass` function was provided by IDR. The logic seemed to indicate that the function `FUN_004a3188` was called if either the simulation form or a diagram form is open. Using Ghidra's reference manager, it was discovered that this function was also called from the event handler `TSimForm::ToolButton17Click`. With the debugger, it was confirmed that this was the event handler for the `'Fram'` button.

Once the function likely responsible for advancing the simulation was identified, the recovery process became increasingly manual. For program logic that does not interact directly with the GUI, there is no need to generate RTTI, and thus there were very few structural symbols for Drtti to recover. In this region, IDR symbols provided useful names for some Delphi runtime math routines, while the FPU fingerprinting pass provided an independent signal for formula-heavy code. In particular, rare transcendental x87 markers helped identify logarithmic and exponential computation sites. By contrast, ordinary floating-point division was excluded from the final fingerprint set: an FDIV/FDIVP-based experiment produced plausible-looking false positives and was therefore not reliable enough to use as a semantic signature. This provided additional context for the analyst, but the remaining formula recovery still required manual inspection and dynamic validation.

After further re work starting with `FUN_004a3188`, the formulas for the attributes *Total-ålder*, *Brh-ålder*, *Grundyta*, *Stamantal*, and *Dg* were partially recovered. The entire simulation state change relies on intermediate values calculated from tree and environment-specific constants that were, for the most part, identified; however, they have been omitted from this paper as they are not directly relevant to the problem at hand.

In summary, the recovery of formulas for attributes like *Grundyta* and *Stamantal* demonstrates how Drtti can speed up re work using RTTI. By mapping the architecture of classes like `TMainForm` and `TSimForm`, and providing the names of event handlers, the search for the relevant sections of the binary was made easier.

C Source-Available Delphi Corpus Detailed Results

This appendix reports the per-binary quantitative measurements for the source-available Delphi corpus. All binaries listed here have an official upstream source repository, an exact commit, an identified project file, and a reproducible executable in the local evaluation dataset. The corpus is built from DelphiDabbler article demos, Embarcadero RAD Studio 12 demos, and the VCL Styles Utils demo repository [5, 7, 19].

Table 10: Detailed per-binary results for the five Dabbler applications. Each project block contains two compact tables: symbolic information available in Ghidra, then memory mapped to known data or instructions. Functions exclude thunks; named functions are functions whose name does not follow Ghidra's default `FUN_` naming convention.

Dabbler 8

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	45982	47740	197370
Functions	5784	6828	8032
Named functions	55	300	3841
Datatypes	1077	5166	5339
Classes	0	943	943
Fields	0	2435	2435
Strings	4932	4330	28590

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	10.55%	13.22%	43.17%
Exec.	Instr.	43.17%	43.12%	43.12%
Exec.	Total	53.72%	56.34%	86.28%
Global	Data	20.09%	22.45%	48.91%
Global	Instr.	38.16%	38.12%	38.12%
Global	Total	58.25%	60.57%	87.03%

Dabbler 16

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	48218	49991	208399
Functions	6128	7160	8507
Named functions	54	298	4150
Datatypes	1072	5150	5321
Classes	0	939	939
Fields	0	2417	2417
Strings	5257	4641	30257

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	10.74%	13.24%	43.28%
Exec.	Instr.	43.29%	43.26%	43.26%
Exec.	Total	54.03%	56.50%	86.54%
Global	Data	20.96%	23.16%	49.48%
Global	Instr.	37.95%	37.91%	37.91%
Global	Total	58.90%	61.07%	87.39%

Dabbler 23

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	46044	47775	197280
Functions	5789	6834	8012
Named functions	54	298	3828
Datatypes	1072	5160	5331
Classes	0	943	943
Fields	0	2423	2423
Strings	4999	4376	28576

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	10.80%	13.41%	43.35%
Exec.	Instr.	43.20%	43.16%	43.16%
Exec.	Total	54.00%	56.57%	86.51%
Global	Data	20.29%	22.60%	49.07%
Global	Instr.	38.21%	38.17%	38.17%
Global	Total	58.50%	60.77%	87.24%

Dabbler 24 Eg1

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	47403	49188	203794
Functions	6001	7036	8317
Named functions	54	300	4028
Datatypes	1255	5347	5518
Classes	0	943	943
Fields	0	2412	2412
Strings	5159	4552	29726

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	10.70%	13.29%	43.26%
Exec.	Instr.	43.28%	43.24%	43.24%
Exec.	Total	53.97%	56.53%	86.49%
Global	Data	20.11%	22.40%	48.92%
Global	Instr.	38.31%	38.27%	38.27%
Global	Total	58.42%	60.68%	87.19%

Dabbler 24 Eg2

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	45675	47407	196406
Functions	5767	6812	7988
Named functions	54	299	3811
Datatypes	1256	5337	5508
Classes	0	944	944
Fields	0	2428	2428
Strings	4985	4398	28484

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	10.70%	13.38%	43.25%
Exec.	Instr.	43.02%	42.98%	42.98%
Exec.	Total	53.71%	56.36%	86.23%
Global	Data	20.21%	22.59%	48.99%
Global	Instr.	38.03%	38.00%	38.00%
Global	Total	58.24%	60.59%	86.98%

Table 11: Detailed per-binary results for the source-available Delphi applications FireDAC MySQL, FireDAC Executor, VCL Styles and Cloud Explorer. Each project block contains two compact tables: symbolic information available in Ghidra, then memory mapped to known data or instructions. Functions exclude thunks; named functions are functions whose name does not follow Ghidra's default `FUN_` naming convention.

FireDAC MySQL

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	93806	97482	372240
Functions	11604	12718	15319
Named functions	57	314	7416
Datatypes	1160	5574	5763
Classes	0	1017	1017
Fields	0	2649	2649
Strings	10190	9512	51688

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	10.81%	12.61%	40.15%
Exec.	Instr.	47.67%	47.65%	47.65%
Exec.	Total	58.47%	60.25%	87.80%
Global	Data	24.34%	25.85%	49.05%
Global	Instr.	40.16%	40.14%	40.14%
Global	Total	64.50%	65.99%	89.19%

FireDAC Executor

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	131224	136061	523353
Functions	13524	14307	17497
Named functions	9	212	8200
Datatypes	631	3932	4103
Classes	0	814	814
Fields	0	1811	1811
Strings	20574	20021	73838

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	15.91%	17.32%	43.54%
Exec.	Instr.	46.33%	46.32%	46.32%
Exec.	Total	62.24%	63.64%	89.86%
Global	Data	23.37%	24.65%	48.34%
Global	Instr.	41.85%	41.85%	41.85%
Global	Total	65.23%	66.49%	90.18%

VCL Styles

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	61562	63608	255471
Functions	7306	8350	10003
Named functions	55	304	4838
Datatypes	1112	5239	5420
Classes	0	963	963
Fields	0	2463	2463
Strings	6398	5759	37194

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	10.38%	12.54%	41.59%
Exec.	Instr.	45.34%	45.30%	45.30%
Exec.	Total	55.73%	57.84%	86.89%
Global	Data	23.22%	25.04%	49.55%
Global	Instr.	38.27%	38.23%	38.23%
Global	Total	61.49%	63.28%	87.79%

Cloud Explorer

#	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Total symbols	92005	97068	439316
Functions	10960	12093	15446
Named functions	66	324	7905
Datatypes	1200	5685	5882
Classes	0	1027	1027
Fields	0	2701	2701
Strings	12022	11356	63140

Mem.	Type	Ghidra	Ghidra+Drtti	Ghidra+Drtti+IDR
Exec.	Data	12.32%	14.44%	45.11%
Exec.	Instr.	41.33%	41.30%	41.30%
Exec.	Total	53.65%	55.74%	86.41%
Global	Data	22.75%	24.58%	51.12%
Global	Instr.	35.77%	35.74%	35.74%
Global	Total	58.52%	60.33%	86.87%

Table 12: Official source provenance for the source-available corpus. Only these targets are included in the source-corpus metrics. Binary size is the on-disk executable size in the local evaluation dataset.

Target	Source	Size (MiB)	Commit	Project file
DD Art. 08	DelphiDabbler demos [5]	2.21	`2df38c3`	`article-08/ExeType.dpr`
DD Art. 16	DelphiDabbler demos [5]	2.35	`2df38c3`	`article-16/CustomLVDemo.dpr`
DD Art. 23	DelphiDabbler demos [5]	2.20	`2df38c3`	`article-23/Article23.dpr`
DD Art. 24-1	DelphiDabbler demos [5]	2.28	`2df38c3`	`article-24/Eg1/Article24Eg1.dpr`
DD Art. 24-2	DelphiDabbler demos [5]	2.20	`2df38c3`	`article-24/Eg2/Article24Eg2.dpr`
Cloud Explorer	RAD Studio 12 demos [7]	4.90	`e2a4d59`	`CloudAPI/CloudExplorer/CloudExplorer.dpr`
FireDAC MySQL	RAD Studio 12 demos [7]	4.61	`e2a4d59`	`FireDAC/.../MySQL/GettingStarted.dpr`
FireDAC Executor	RAD Studio 12 demos [7]	6.13	`e2a4d59`	`FireDAC/Tool/Executor/FDExecutor.dpr`
VCL Styles	VCL Styles Utils [19]	3.07	`5f3aaea`	`Vcl Styles Basic Demo/Demo.dpr`

Table 13: Normalized class-name set overlap between IDR exports and the IDR-independent Drtti configuration on the source-available corpus.

Corpus	Bins.	Drtti	IDR	Common	Jaccard
DelphiDabbler	5	2236	2496	1554	47.0–50.0%
RAD Studio 12	3	1382	3444	977	21.1–30.4%
VCL Styles	1	470	661	329	41.0%

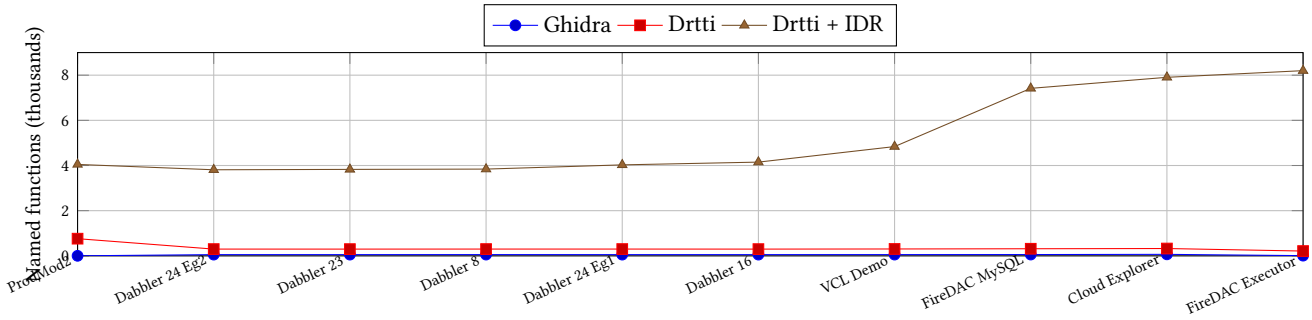


Figure 12: Named functions available in Ghidra before and after Drtti, with and without optional IDR symbols. Values are plotted in thousands.

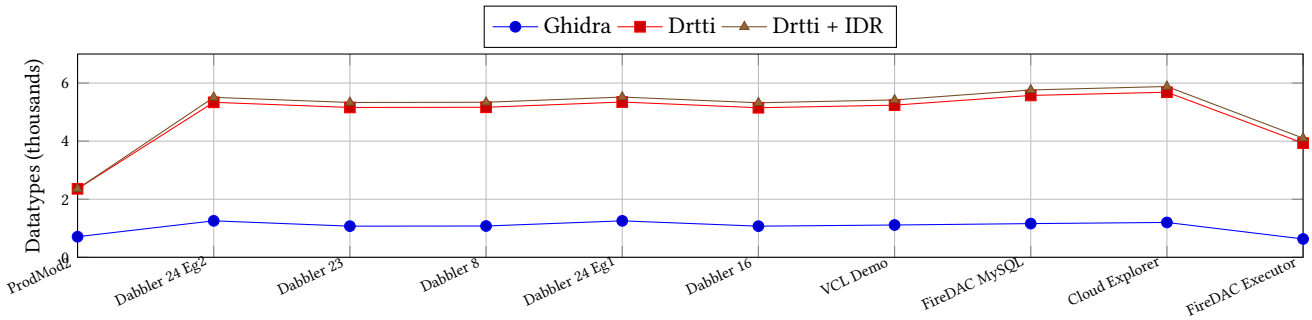


Figure 13: Datatypes available in Ghidra before and after Drtti, with and without optional IDR symbols. Values are plotted in thousands.

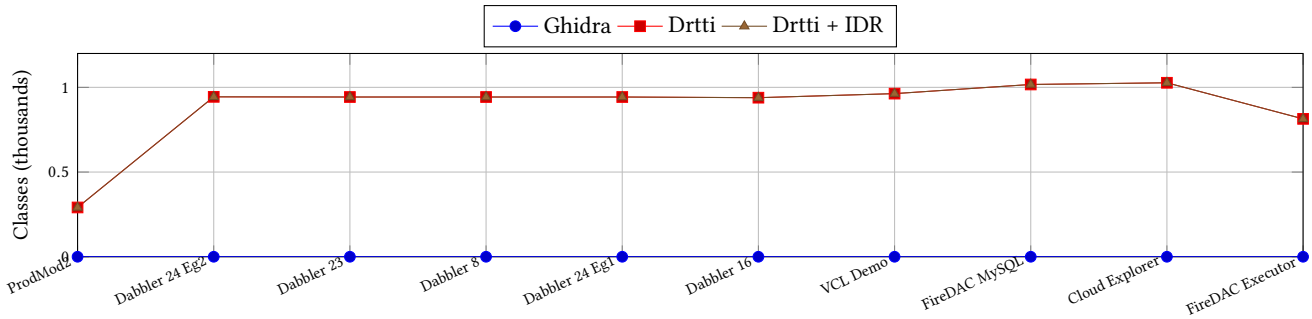


Figure 14: Classes recovered in Ghidra before and after Drtti, with and without optional IDR symbols. Values are plotted in thousands.

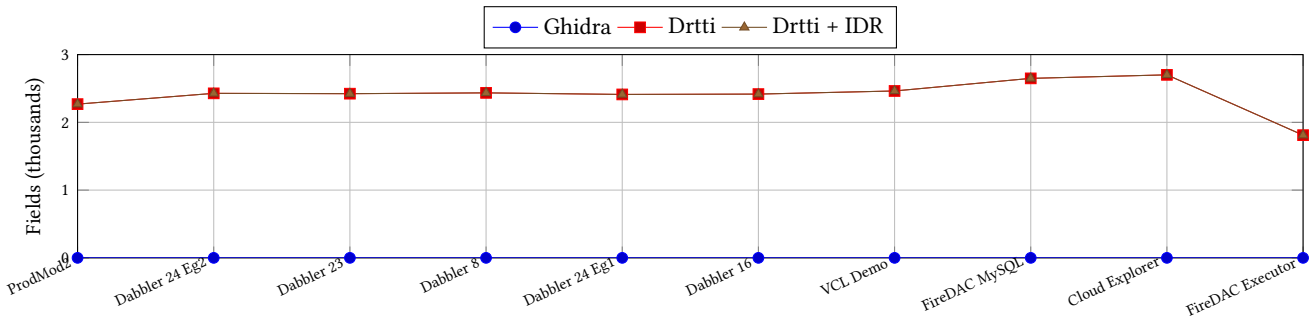


Figure 15: Fields recovered in Ghidra before and after Drtti, with and without optional IDR symbols. Values are plotted in thousands.

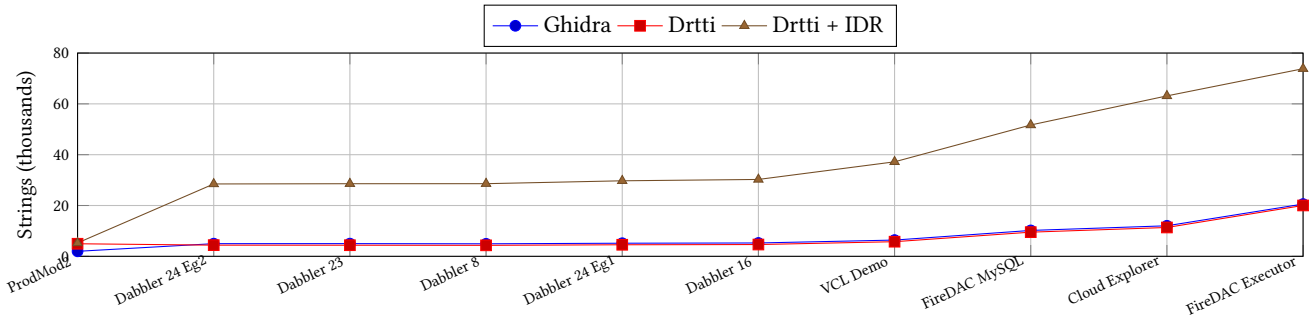


Figure 16: Strings available in Ghidra before and after Drtti, with and without optional IDR symbols. Values are plotted in thousands.

These appendix figures show that the contribution of Drtti and optional IDR symbols differs by symbol type. Class and field recovery comes from Drtti: standard Ghidra recovers no Delphi class structures, while Drtti and Drtti+IDR recover the same class and field counts. By contrast, the largest IDR gains appear in functions, named functions, and strings, reflecting the import of external Delphi runtime and library context.