A Framework for Assessing Decompiler Inference Accuracy of Source-Level Program Constructs

Jace Kline
Electrical Engineering and Computer Science
University of Kansas
Lawrence, KS
jace_kline@ku.edu

Dr. Prasad Kulkarni
Electrical Engineering and Computer Science
University of Kansas
Lawrence, KS
prasadk@ku.edu

***Abstract*—Decompilation is the process of reverse engineering a binary program into an equivalent source code representation with the objective to recover high-level program constructs such as functions, variables, data types, and control flow mechanisms. Decompilation is applicable in many contexts, particularly for security analysts attempting to decipher the construction and behavior of malware samples. However, due to the loss of information during compilation, this process is naturally speculative and thus is prone to inaccuracy. This inherent speculation motivates the idea of an evaluation framework for decompilers.**

In this work, we present a novel framework to quantitatively evaluate the inference accuracy of decompilers, regarding functions, variables, and data types. Within our framework, we develop a domain-specific language (DSL) for representing such program information from any "ground truth" or decompiler source. Using our DSL, we implement a strategy for comparing ground truth and decompiler representations of the same program. Subsequently, we extract and present insightful metrics illustrating the accuracy of decompiler inference regarding functions, variables, and data types, over a given set of benchmark programs. We leverage our framework to assess the correctness of the Ghidra decompiler when compared to ground truth information scraped from DWARF debugging information. We perform this assessment over a subset of the GNU Core Utilities (Coreutils) programs and discuss our findings.

I. Introduction

A. Context and Background

In an increasingly digital world, cybersecurity has emerged as a crucial consideration for individuals, companies, and governments trying to protect their information, financial assets, and intellectual property. Of the many digital threats, various forms malware continue to pervade the digital landscape and thus remain a key concern for security analysts. One approach for combating malware involves attempting to deconstruct and reason about the malware itself. Understanding the functionality and behavior of malware samples may aid a security analyst in identifying methods to thwart or disable the malware's effects on a target system and similar systems.

Although simple in concept, the act of reverse engineering and reasoning about malware proves to be a steep challenge. The primary issue is that access to high-level malware source code is almost never available and, thus, any reasoning about the malware must be derived from the malware sample itself. Another issue is that malware authors often leverage obfuscation techniques to mask the intention and behavior of malware samples. To evade antivirus tools using signature-based detection, malware authors may employ techniques such as dead-code insertion, register reassignment, subroutine reordering, instruction substitution, code transposition, and code integration []. To complicate semantic binary code analysis of malware samples, malware authors may leverage compile-time strategies such as stripping and compiler optimizations []. Although we have discussed these obfuscation strategies in the context of malware, these techniques may be also leveraged by developers or companies attempting to dissuade binary code analysis of proprietary software.

Despite the challenge of binary code analysis, there exist many tools that attempt to glean high-level semantic information from binary code samples. A *disassembler* takes binary code as input and produces architecture-specific assembly code as output. Many challenges and considerations exist in the disassembly process - particularly for stripped binary code - such as discerning code from data and locating function boundaries []. One invariant in the disassembly process, however, is that the mapping from assembly instructions to binary instructions and vice-versa is always one-to-one. A *decompiler* takes this reverse mapping process one step further by translating binary code into an equivalent high-level source code representation. The decompilation process is inherently speculative since high-level information such as function boundaries, variables, data types, and high-level control flow mechanisms are lost when a program is compiled. With this, the decompiler must infer enough high-level structure for useful analysis without being overly aggressive and consequently blurring the program's intent. Many decompiler tools are currently in use by the reverse engineering community. Commercial decompiler tools include IDA Pro [] and JEB3 []. Popular open-source decompiler frameworks include Ghidra [], RetDec [], and Radare2 [].

B. Research Problem

Due to the number of decompiler tools as well as the imprecise nature of decompilation, a generalized and extensible quantitative evaluation framework for decompilers is critical. Existing work by Liu and Wang [] proposes an evaluation technique to determine whether recompiled decompiled programs are consistent in behavior to their original binaries. This

technique, although useful, does not offer any insight into the inference accuracy of decompilers with respect to high-level program constructs such as functions, variables, and data types. The inference accuracy of the these high-level constructs are important for analysts to gain an understanding of the analyzed program.

C. Research Objectives

Targeting the current gap in the literature outlined in the previous section, this paper presents a novel framework for quantifying and assessing the accuracy of decompiler tools. To prove our concept, we apply our framework to the Ghidra decompiler and subsequently discuss our findings. The primary objectives achieved by this work are as follows:

1) We define a domain-specific language (DSL), written in Python, for expressing high-level program information such as functions, variables, and data types. This is serves as a language-agnostic medium whereby we can translate program information extracted from a decompiler or a ground-truth source.
2) We extend our DSL to compare program information representations from different sources. A common use case is to compare ground-truth program information to decompiler-inferred program information.
3) Leveraging the comparison logic in (2), we define a set of quantitative metrics to measure the accuracy of function, variable, and data type inference.
4) We develop a translation module in Python that uses DWARF debugging information from a binary program to generate a ground-truth program information representation in our DSL.
5) We utilize the Ghidra Python API to implement a translation module, taking a Ghidra decompilation of a binary program as input and producing a program information representation in our DSL.
6) Using our developed language, metrics, and translation modules, we quantitatively assess the accuracy of the Ghidra decompiler when compared to ground-truth program information obtained from DWARF debugging information. We peform this analysis using the set of GNU Coreutils programs as benchmarks. We present the evaluation results and discuss additional findings and takeaways.

D. Evaluation Summary

We use our evaluation framework to perform an assessment of the Ghidra decompiler (version 10.2) over 105 GNU Core Utilities (version 9.1) benchmark programs compiled with GCC (version 11.1.0). We evaluate Ghidra with no optimizations under three compilation cases of the benchmark programs - (1) stripped, (2) standard (not stripped, no DWARF symbols added), and (3) debug (DWARF symbols included) - to determine how the level of information provided in the binaries affects recovery and inference performance of functions, variables, and data types by Ghidra.

Our function recovery analysis reveals that Ghidra successfully recovers 100% of the 18139 functions under the stripped and standard compilation conditions across all benchmarks. In the debug compilation case, Ghidra successfully identifies all functions but fails to decompile four functions in the *factor* program due to a type resolution error. Upon further analysis, we conclude this is a bug in the Ghidra decompiler.

Analysis of high-level variable recovery shows that the recovery accuracy of variables of primitive data types (char, int, float, pointer) is significantly higher than the recovery accuracy of complex (aggregate) types (array, struct, union), particularly in the stripped and standard compilation cases when no debugging information is present. Overall, we see a partial high-level variable recoveries percentages of 97.1%, 99.2%, and 99.9% for the three compilation cases, respectively. The percentages of exact high-level variable matches for each of the compilation cases are 36.1%, 38.6%, and 99.6%, respectively.

Related to our high-level variable recovery analysis, we perform a ”decomposed” variable recovery analysis. For the decomposition, we recursively decompose each variable into a set of primitive variables as they appear in memory. We then perform the comparison and evaluation similar to in the high-level analysis. We show that the partial recovery percentages for each of the stripped, standard, and debug compilation cases are 73.8%, 92.4%, and 98.0%, respectively. The exact match percentages over the decomposed variables are 24.6%, 25.0%, and 98.0% for each of the compilation cases, respectively. The lower recovery accuracy results in this decomposed analysis are explained by the decomposition of the variables with complex types, namely arrays, that are partially or fully missed in the high-level analysis. These variables, when decomposed, result in an increase in the number of total missed variables. Analysis of decomposed variable recovery by data type shows that int (and char) variables are most accurately inferred, followed by pointer variables, with floating-point (float, double) variables showing the lowest recovery accuracy.

We perform a data bytes recovery analysis to determine the total percentage of data bytes that are found and missed across all ground truth variables by the decompiler. We discover that the bytes recovery percentages are 61.3%, 80.6%, and 99.5% for each the stripped, standard, and debug compilation cases, respectively.

Lastly, we perform an evaluation of the Ghidra decompiler's array recovery accuracy. We find that, for each the stripped, standard, and debug compilation cases, 36.2%, 71.6%, and 99.5% of ground truth array varnodes overlap with at least one associated decompiler-inferred array varnode, respectively. We find the average size (in bytes) discrepancies between compared ground truth and the decompiler variables to be 458.6, 239.0, and 9.42 for each of the compilation cases, respectively. With respect to the sizes of the ground truth arrays, the average array size error percentages for the array comparisons in each compilation case are 91.2%, 47.5%, and 11.0%, respectively.

Across our analyses, we observe that there is a clear

relationship between the compilation configuration of the benchmark programs and the recovery accuracy of program constructs by the Ghidra decompiler. We find that, with respect to recovery of program constructs, the debug compilation case far outperforms the standard case, which moderately outperforms the stripped case. However, despite the relatively high recovery accuracy of the Ghidra decompiler in the debug case, we futher explore the causes of misses and partial misses in the debug case and find that Ghidra possesses a major limitation in expressing local variables tied to specific lexical scopes. A compiler such as GCC may reuse stack address space for variables associated with non-overlapping and non-nested lexical scopes. This is a problem for the Ghidra decompiler as we observe that all variable declarations are placed at the top level of the function, ultimately preventing these scope-specific variables from being precisely captured. From our manual analysis of the decompiled benchmark programs, we find that this is the cause of the majority of partially missed variables and data bytes in the debug compilation case. This limitation certainly affects the stripped and standard compilation cases as well.

E. Contributions

The three key contributions of this work are as follows:

1) We develop a novel framework for evaluating decompiler tools based on the recovery accuracy of high-level program constructs, including functions, variables, and data types. This framework includes a domain-specific language (DSL), developed in Python, to represent and compare sources of high-level program information and their association with binary-level constructs. In addition, we devise quantitative metrics for expressing recovery accuracy of program constructs.
2) We leverage our framework to perform an in-depth evaluation of the Ghidra decompiler with respect to high-level function, variable, and data type recovery. This evaluation is performed over the GNU Core Utilities programs under three compilation conditions.
3) From our evalution of Ghidra, we discover and discuss the implications of two key issues present in the Ghidra decompiler.

F. Outline

The remainder of this paper is outlined as follows: In section 2, we discuss related research and background concepts useful for the understanding of this work. Next, in section 3, we detail our methodology for developing our evaluation framework. In section 4, we present and discuss the results of applying our evaluation framework to the Ghidra decompiler. We conclude in section 5 with a summary of our results, implications of our work, limitations, and future research directions.

II. Background and Related Work

A. Software Reverse Engineering, Dissassembly, and Decompilation

Software reverse engineering (SRE) is the process of analyzing a software system with the intention to extract design and implementation information, particularly in situations where high-level source code is unavailable []. One common use case for this practice is to understand and deconstruct legacy code present in a software system where the source code has been lost. In this scenario, analysts could use SRE to understand this legacy code, determine its behavior, and ultimately decide how to reuse, patch, or replace the code. Another context for the use of SRE is computer security. Malware, or malicious programs, are nearly always present in binary form without their associated high-level source code. An analyst may use SRE to deconstruct the malware's logic, determine its behavior, and identify approaches to neutralize the malware and harden the host system for prevention of future attacks.

To perform SRE on a binary program, a critical first step is *disassembly*. This process takes binary code as input and produces assembly code as output. A key to this process is that binary and assembly instructions are always mapped one-to-one, and thus the main challenges lie in determining function boundaries and differentiating code, data, and metadata. Factors that contribute to these challenges include the following []:

- Data embedded in code regions
- Variable instruction size (on some architectures)
- Indirect branch instructions (the target of a branch instruction is not statically known)
- Functions without explicit 'CALL' references
- Position independent code sequences
- Manually crafted assembly code

The conversion of binary code to assembly code through disassembly is a desirable starting point in the process of SRE. However, program semantics are still often difficult to interpret and reason about at the assembly code level. This difficulty necessitates an even more speculative process, *decompilation*, that takes a binary program as input and produces a high-level source code representation of the input program's semantics, usually in C. Decompilation, therefore, involves the speculative inference of high-level language concepts such as control flow mechanisms, variables, and data types. Decompiler tools rely heavily on the disassembly process as a first step in their analysis, and therefore the challenges affecting disassembly also naturally affect decompilation. Additional factors that obfuscate the accuracy of decompilation are the following:

- Compiler optimizations
- Stripped debugging information and metadata

With these compounding challenges affecting the decompilation process, it is clear that decompiler tools operate under a great degree of nondeterminism and speculation. This fact highlights the need for a common evaluation framework for decompiler tools.

B. DWARF Debugging Standard

DWARF is a debugging file format used by many compilers and debuggers to support source-level debugging for compiled binary programs []. When specified flags (usually '-g') are present at compilation, DWARF-supporting compilers such as

GCC and Clang will write DWARF debugging information to an output binary program or object file. A resulting binary executable can then be loaded into a DWARF-supporting debugger such as GDB to debug the target binary program with references to line numbers, functions, variables, and types in the source-level program. The DWARF standard is source language agnostic, but generally supports equivalent representations for constructs present in common procedural languages such as C, C++, and Fortran. In addition, DWARF is decoupled from any architecture, processor, or operating system. The generalizability of DWARF debugging information makes it a prime candidate for extracting "ground truth" information about a particular binary program, regardless of the specifics of the source language, architecture, processor, or operating system. DWARF is leveraged in this work to scrape ground-truth information about target binary programs. This information is subsequently used to evaluate the accuracy of the output produced by a target decompiler.

C. Ghidra Reverse Engineering Framework

Ghidra, created and maintained by the National Security Agency Research Directorate, is an extensible software reverse engineering framework that features a disassembler, decompiler, and an integrated scripting environment in both Python and Java []. We use the Ghidra decompiler in this work to demonstrate our decompiler evaluation framework.

D. Related Work

In the 2020 paper *How Far We Have Come: Testing Decompilation Correctness of C Decompilers* by Liu and Wang [], the authors present an approach to determine the correctness of decompilers outputting C source code. They aim to find decompilation errors, recompilation errors, and behavior discrepancies exhibited by decompilers. To evaluate behavioral correctness, they attempt to recompile decompiled binaries (after potential syntax modifications) and use existing dynamic analysis techniques such as fuzzing to find differences in behavior between the recompiled and original programs. The objective of our work differs as we aim to evaluate decompiler inference of high-level structures such as functions, variables, and data types. Accurate inference of high-level structures enables easier understanding of decompiled programs by analysts; however, accurate behavior is also necessary to ensure that the decompiled representation is consistent with the original program. Hence, both of these works evaluate important aspects of decompiler correctness.

The review *Type Inference on Executables* by Caballero and Lin (2016) provides a comprehensive summary of recent literature on techniques used for variable discovery and type inference. In addition, the authors present various software reverse engineering (SRE) tools and frameworks in terms of their inputs, analysis types, output formats, and use cases. In essence, this work surveys the a set of decompiler tools and characterizes them based on their purported capabilities. The purpose of our work, on the contrary, is to objectively determine the correctness of decompiler tools based on their inference accuracy of high-level program constructs.

III. Methodology

In this section, we discuss the design, construction, and evolution of our decompiler evaluation framework. To achieve this, we identify key objectives that we subsequently address in more detail in the following subsections. These objectives are as follows:

1) Express program information such as functions, variables, data types, and addresses in a common representation.
2) Programmatically capture a "ground truth" representation for a given program.
3) Programmatically scrape program information from decompiler tools, namely Ghidra.
4) Compare two program representations of the same program.
5) Formulate quantitative metrics for evaluating the accuracy of a decompiler.

1) Domain-Specific Language (DSL) for Program Information: In order to make our framework general and reusable, we devise a common domain-specific language (DSL) to represent program information such as functions, variables, data types, and addresses, as well as the relationships between them. This DSL must act as a bridge linking binary-level address information with the source-level structures such as functions, variables, and data types. Combining the information from these two layers of abstraction is, in essence, a mapping between binary-level and source-level structures. The accuracy of this mapping for a given decompiler is precisely the objective of our analysis.

[Figure representing how DSL can be constructed from many sources (DWARF, Ghidra, IDA Pro, etc.)]

The DSL we devised is entirely decoupled from the source of the program information. This allows any ground truth or decompiler source of program information to be translated into this common language and subsequently analyzed or compared with another source of program information. The core of our language is defined in Python and is compatible with Python (Jython or CPython) versions ¿= 2.7. We chose Python because the Ghidra framework supports custom Python scripts for querying and manipulating program information obtained from the disassembler and decompiler. In addition, the Python 'pyelftools' open-source library [] allows scraping DWARF debugging information directly from binary programs. This DWARF information can then be utilized to construct a "ground truth" representation of program information. We discuss this further in the next section.

a) DSL Definitions: In this section, we briefly describe the structure and relationships of the major constructs that comprise our DSL.

At the root of our DSL is the *ProgramInfo* type. The fields of this type include a list of global variables (*Variable* objects) and a list of functions (*Function* objects).

The *Function* type holds information about a function such as the name, the start PC address (*Address* object), the end PC address (*Address* object), a list of parameters (*Variable* objects), a list of local non-parameter variables (*Variable* objects), and the return type (*DataType* object).

The *Variable* type contains information about a source-level global variable, local variable, or parameter. A variable has a name, a data type (*DataType* object), and a list of address "live ranges". We consider a live range (*AddressLiveRange* type) to be the association of a variable's storage address with the PC address range where the storage location is valid for the variable. This "live range" concept allows for the expression of source-level variables that map to multiple underlying storage locations throughout their lifetime. Multiple live ranges may be associated with a single variable when compiler optimizations are present.

The *Address* type represents any absolute or relative location referenced in a binary program. This could include a PC location, variable storage location, or a register. From an implementation perspective, *Address* is the base class with subclasses representing the different types of address constructions based on context. These *Address* subclasses include *AbsoluteAddress*, *RegisterAddress*, *RegisterOffsetAddress*, and *StackAddress*. Each address is associated with an *AddressRegion*. This type is used to manage ordering and comparison logic for addresses that fall within the same region.

The last main construct in our core DSL is *DataType*. This type is represents a source-level data type and is typically associated with a variable or a function return type. *DataType* is the base of a class hierarchy with subclasses representing particular data types. The subclasses include *DataTypeFunctionPrototype*, *DataTypeInt*, *DataTypeFloat*, *DataTypeUndefined*, *DataTypeVoid*, *DataTypePointer*, *DataTypeArray*, *DataTypeStruct*, *DataTypeUnion*. Although these defined types correspond to C-like data types, this language can easily be extended to support other data types present in other high-level programming languages. All data type objects contain a "size" field representing the number of bytes the given data type occupies in memory.

2) Capturing Ground Truth Program Information: With our DSL defined, we need a reliable method to extract "ground truth" information from a program and translate this information into our DSL. This "ground truth" information is intended to be used in a comparison with the program information obtained from a decompiler. Our framework is meant for evaluation and therefore we assume that we have access to the source code of benchmark programs to be used for the evaluation. With this assumption, we consider two options for extracting program information from a given source program.

The first option for extracting ground truth information is to parse the source code's abstract syntax tree (AST) and then use this AST to manually extract functions, variables, and data types. There are two major issues with this approach. First, parsing source code to an AST assumes a particular source programming language which greatly reduces generality. Second, obtaining the AST alone does not offer any binary-level information that allows us to link binary-level addresses with the source-level structures.

The second, more favorable, approach to extracting ground truth program information involves leveraging debugging information optionally included in the binary by the compiler. The primary purpose of debugging information is to link binary-level instructions and addresses with source-level structures. This binary-level to source-level association is precisely what is needed to translate program information into our DSL. Since our framework is developed and targeted at Linux, we choose the DWARF debugging standard as the assumed debugging format for our framework. However, defining a translation module from another debugging format into our DSL is certainly possible and is an idea for future work. The DWARF debugging standard is supported by nearly all major Linux compilers and supports any source programming language (with possible extensions). These properties of the DWARF standard allow it to be used as a "ground truth" source of program information, decoupled from the source language or the compiler.

a) Translating DWARF to the DSL: Starting with a source-level program, we must perform the following steps to extract program information represented in our DSL. First, we compile the source program with the option to include debugging symbols. In our particular analysis we use the GCC compiler specifying the "-g" flag. Many other compilers also offer the option for compilation with the inclusion of DWARF debugging symbols. After we compile the program, we then extract the DWARF debugging information from the resulting binary. We utilize the 'pyelftools' Python library [] to perform this extraction. The extraction results in, among other information, a set of debugging information entries (DIEs). Together, these DIE records provide a description of source-level entities such as functions, variables, and data types in relation to low-level binary information such as PC addresses and storage locations. Each DIE contains the following important features:

- An *offset* uniquely identifying the DIE within its compilation unit. These offsets are how DIEs reference other DIEs.
- A *tag* representing the "class" of the DIE. Example tags include "DW_TAG_subprogram", "DW_TAG_variable", and "DW_TAG_base_type".
- A set of *attributes* specifying tag-specific properties of the DIE. Examples include "DW_AT_name", "DW_AT_size", and "DW_AT_type".

The translation process from the DIE graph into our DSL is, at its core, a process of forming a nested data structure (our DSL's *ProgramInfo* type) from a flattened one (a collection of DWARF DIEs). To tackle this translation, we first define an intermediate representation (IR) language that acts as a "flattened" analog to the constructs present in our DSL. Instead of each IR construct directly containing the fields of other constructs, they instead contain fields that reference the IDs of other constructs through a shared database. The responsibility of the database is to map unique IDs to the flattened constructs. When all the IR constructs have been inserted

into the database, the database then recursively resolves the flattened IR structures into their associated DSL structures, starting from the root *ProgramInfoStub* object, the IR analog to the *ProgramInfo* DSL type. This process is complicated by the fact that some data types, particularly *struct* types, may be recursive or mutually recursive, ultimately creating a cycle in the reference resolver. To address this, we implemented a mechanism whereby each IR node is marked when it is visited. Future attempts to resolve the same IR construct return with the existing object being resolved instead of attempting to resolve the same reference again. With the IR defined and the resolution logic in place, we map the DWARF DIE objects into our "flattened" IR and construct the IR object database. When all the DIEs are processed and translated, we specify the *ProgramInfoStub* node as the root reference and then execute our resolver algorithm to recursively generate the *ProgramInfo* object and subobjects defined in our DSL. Our DWARF translation module consists of about 1000 lines of Python code. The IR and resolver logic adds an additional 600 lines of code.

[DWARF parsing figure: DIEs -¿ IR -¿ DSL]

3) Capturing Decompiler Program Information: In addition to capturing a ground-truth program representation in our DSL, we must construct a DSL representation of the program information obtained from a decompiler we wish to evaluate. Depending on the decompiler and the structure of its output, this process may take many forms, often involving querying APIs exposed by the decompiler framework. In all cases however, this shall involve defining a translation module from the decompiler output to the structures defined in the DSL. Hence, our framework can be employed on any decompiler assuming a translation module implementation.

a) Translating Ghidra Decompiler Output to the DSL: For our analysis of the Ghidra decompiler, we utilize the Ghidra scripting API to programmatically scrape and process information about the decompilation of target binary programs. The Ghidra scripting environment exposes its own collection of data structures and functions from which we obtain our information. Since the Ghidra scripting environment supports Python, we directly import and leverage our "flattened" IR (described in the previous section) and our DSL constructs to carry out the translation.

The strategy employed for the Ghidra translation is similar to that of our DWARF translation algorithm described in the previous section. We utilize the Ghidra API to obtain particular information about functions, variables, data types, and associated addresses gathered during the decompilation. Of particular use to our translation logic is the *DecompInterface* object exposed by the Ghidra API. This interface supports decompiling functions one at a time. Information inferred by each function's decompilation is used to update Ghidra's internal representation of the program information. By decompiling each of the functions extracted from Ghidra's disassembly analysis, we attempt to form a complete decompiled interpretation of the entire input program.

We use the same IR defined for the DWARF translation to accumulate flattened records corresponding to these program constructs in a database. From here, we run the same resolution algorithm on the IR constructs database to generate the root *ProgramInfo* object in our DSL. The Ghidra-specific translation logic is implemented in roughly 900 lines of Python code.

4) Comparison of "Ground Truth" and Decompiler Program Information: After converting both the ground-truth and decompiler program information into our DSL representation, we next formulate and implement a strategy to compare the two resulting *ProgramInfo* objects. To achieve this, we create an extension of our DSL that defines data structures and functions for capturing comparison information at different layers.

a) Data Type Comparison: Given two *DataType* objects and an offset between their start locations, we devise a method to capture nuanced information about the comparison of the data types.

We define the *metatype* of a data type to be general "class" of the given data type. These metatypes include *INT*, *FLOAT*, *POINTER*, *ARRAY*, *STRUCT*, *UNION*, *UNDEFINED*, *VOID*, and *FUNCTION_PROTOTYPE*. We consider *INT*, *FLOAT*, *POINTER*, *UNDEFINED*, and *VOID* to be *primitive metatypes* since they cannot be decomposed further. *ARRAY*, *STRUCT*, and *UNION* are considered *complex metatypes* since these types are formed via the composition or aggregation of different members or subtypes. We consider the 'char' data type to be of the *INT* metatype with size equal to one byte.

[Figure: Ariste type lattice]

A *primitive type lattice* [] is used to hierarchially relate primitive data types based on their metatype, size, and signedness (if applicable). More general types are located higher in the lattice while more specific types are located closer to the leaves. A type lattice may be used to determine whether two primitive data types are equivalent or share a common parent type.

[Figure: Subset relationship example(s)]

We next define a *subset* relationship between two data types. For a given complex data type X and another data type Y with a given offset (possibly 0) between the location of X and Y in memory, Y is considered a *subset* type of X if Y is equivalent to a "portion" of X, consistent with the offset between X and Y. For example, if X is an array, any sub-array or element of X such that elements are aligned and the element types are equivalent to X is considered a subset of X. If X is a struct or union, any sub-struct or member with proper alignment and equal constituent elements is considered a subset of X.

Suppose we have two *DataType* objects X (ground truth) and Y (decompiler) with offset k from the start of X to the start of Y. The goal is to compute the *data type comparison level* for the given comparison. The possible values for the comparison level are as follows, from lowest equality to highest equality:

* *NO_MATCH*: No relationship could be found between X and Y. * *SUBSET*: Y is a subset type of the complex type X. * *PRIMITIVE_COMMON_ANCESTOR*: In the primitive type lattice, primitive types X and Y share a common ancestor

type. * *MATCH*: All properties of X and Y match including metatype, size, and subtypes (if applicable).

We first check the equality of X and Y. If X and Y are equal, we assign the *MATCH* comparison code. In the case that X and Y are both primitive types, we attempt to compute their shared ancestor in the primitive type lattice. If a common ancestor could be found, we assign *PRIMITIVE_COMMON_ANCESTOR*. If X is a complex type, we employ an algorithm to determine whether Y is a subset of X at offset k by recursively descending into constituent portions of X starting at offset k (sub-structs, sub-arrays, elements, members) and checking for equality with Y. If a subset relationship is found, we assign the *SUBSET* compare level. In all other cases, we assign the *NO_MATCH* compare level.

b) Variable Comparison: There are two main contexts where variable comparison occurs. The first context is at the top level, where the set of ground-truth global variables is compared to the set of decompiler global variables. The second context for variable comparison is within the context of a function when we compare parameters or local variables between the ground-truth and the decompiler. In either case, comparing sets of variables starts with the decomposition of each *Variable* object from the DSL into a set of *Varnode* objects in our extended DSL.

A *Varnode* ties a *Variable* to a specific storage location and the range of PC addresses indicating when variable lives at that location. The varnodes for a given variable are directly computed from the variable's live ranges discussed previously. In unoptimized binaries, it is the case that a single *Variable* shall decompose into a single *Varnode*.

With each variable decomposed into its associated varnodes, we next partition the varnodes from each the ground-truth and the decompiler based on the "address space" in which they reside. These address spaces include the *absolute* address space, the *stack* address space, and the *register offset* address space (for a given register). The *stack* address space is a special case of the *register offset* address space where the offset register is the base pointer which points to the base of the current stack frame.

For the set of varnodes in each address space, we first order them based on their offset within the address space. Next, we attempt to find overlaps between varnodes from the two sources based on their location and size. If an overlap occurs between two varnodes, we compute a data type comparison taking into account the offset between the start locations of the two varnodes. The data type comparison approach is described in the previous section.

Based on the overlap status and data type comparison of a ground-truth varnode X, one of the following *varnode comparison levels* will be assigned:

- *NO_MATCH*: X is not overlapped with any varnodes from the other source.
- *OVERLAP*: X overlaps with one or more varnodes from the other space, but the data type comparisons are level *NO_MATCH*.
- *SUBSET*: X overlaps with one or more varnodes and each of its compared varnodes has data type comparison level equal to *SUBSET*. In other words, the compared varnode(s) make up a portion of X.
- *ALIGNED*: For some varnode Y from the other source, X and Y share the same location and size in memory; however, the data types of X and Y do not match. The data types comparison could have any compare level less than *MATCH*.
- *MATCH*: For some varnode Y from the other source, X and Y share the same location and size in memory, and their data types match exactly.

The inference of variables with complex data types including structs, arrays, and unions proves to be a major challenge for decompilers. Recognizing this, we develop an approach to compare the sets of ground truth and decompiler variables (varnodes) in their most "decomposed" forms. An analysis of this sort helps to recognize how well a decompiler infers the primitive constituent components of complex variables. Furthermore, this allows us to recognize the aggressiveness and accuracy of complex variable synthesis from more primitive components.

[Figure: Example of "decomposing" complex varnode]

We first implement an approach to recursively strip away the "complex layers" of a varnode to its most primitive decomposition. This primitive decomposition produces a set of one or more primitive varnodes. For example, an array of elements is broken down into a set of its elements (decomposed recursively). A struct is broken down into a set of varnodes associated with each of its members (decomposed recursively). Unions present a special case since the members share a common, overlapping region of memory. Hence, to decompose a union, we transform it into an *UNDEFINED* primitive type with the same size as the union.

We apply this primitive decomposition to each varnode in the sets of ground truth and decompiler varnodes. With the two sets of decomposed varnodes, we leverage the same variable comparison approach described previously to compare the varnodes in these sets. The resulting comparison information is treated as a separate analysis from the unaltered varnode sets.

c) Function Comparison: The first step in function comparison is to determine whether each ground-truth function is found by the decompiler. We first order the functions from each source by the start PC address of the function. Next, we attempt to match the functions from the two sources based on start address. Any functions from the ground-truth that are not matched by a decompiler function are considered "missed". Functions the are found by the decompiler but absent from the ground-truth are considered "extraneous". For any missed functions, we consider its associated parameters, local variables, and data types to also be "missed".

For each "matched" function based on start PC address, we compute and store information including the return type comparison, parameter comparisons, and local variable com-

parisons. These sub-comparisons leverage the data type and variable comparison techniques described previously.

5) Quantitative Evaluation Metrics: In this section, we define quantitative metrics for evaluating the accuracy of the a given decompiler when compared to a ground-truth source. We rely on the function, variable, and data type comparison information discussed previously to extract these metrics. In the following sub-sections, we define sets of metrics that associated with tables seen in our evaluation results section.

a) Functions: This set of metrics outlines the function identification performance of the decompiler.

- *Ground truth functions*: The number of functions present in the ground truth program representation.
- *Functions found*: The number of functions from the ground truth set that are identified by the decompiler.
- *Functions missed*: The number of functions from the ground truth set that are not identified by the decompiler.
- *Functions recovery fraction*: The fraction of ground truth functions found by the decompiler divided by the number of ground truth functions.

b) Varnodes: Recall that a *Varnode* is defined to be a source-level *Variable* tied to a single storage location for a range of PC addresses. In analyses of unoptimized binaries, the mapping of variables to varnodes is one to one. This set of metrics illustrates the decompiler's accuracy in recovering varnodes.

- *Ground truth varnodes*: The total number of varnodes present in the ground truth source. This includes varnodes associated with global and local variables from all functions.
- *Varnodes matched @ level=LEVEL*: Each ground truth varnode is associated with a *varnode comparison level* (*NO_MATCH*, *OVERLAP*, *SUBSET*, *ALIGNED*, *MATCH*) during the comparison with the set of decompiler varnodes. This metric specifies the number of ground truth varnodes that are matched at the specified level.
- *Varnodes average comparison score*: For each *varnode comparison level*, we first linearly assign an integer representing the strength of the varnode comparison (*NO_MATCH* = 0, *OVERLAP* = 1, *SUBSET* = 2, *ALIGNED* = 3, *MATCH* = 4). We then normalize these scores to fall within the range zero to one. Then, for each ground truth varnode, we compute this normalized score. We take the average score over all ground truth varnodes to obtain the resulting metric. This metric approximates how well, on average, the decompiler infers the ground truth varnodes.
- *Varnodes fraction partially recovered*: The fraction of ground truth varnodes with a match level greater than *NO_MATCH*.
- *Varnodes fraction exactly recovered*: The fraction of ground truth varnodes with a match level equal to *MATCH*.

We repeat this varnode analysis for the decomposed (primitive) set of varnodes resulting from recursively decomposing each of the high-level varnodes into its most primitive set of varnodes. We also repeat our analysis of the original set of varnodes filtered by metatype. The metatypes considered are *INT*, *FLOAT*, *POINTER*, *ARRAY*, *STRUCT*, and *UNION*. Lastly, we repeat the analysis of the decomposed varnodes when filtered by metatype. For this metatype analysis over the decomposed varnodes, we only consider the primitive metatypes *INT*, *FLOAT*, and *POINTER* since the varnodes are guaranteed to be primitive.

c) Data Bytes: These metrics look at the total number of data bytes from all variables recovered by the decompiler when compared to the ground truth source.

- *Ground truth data bytes*: The total number of data bytes captured from the ground truth source, derived from all global and local variables.
- *Bytes found*: The total number of data bytes recovered by the decompiler that overlap with data bytes found in the ground truth.
- *Bytes missed*: The number of data bytes present in the ground truth that were not recovered by the decompiler.
- *Bytes recovery fraction*: The fraction of ground truth data bytes found by the decompiler divided by the total number of ground truth bytes.

d) Array Comparisons: In this set of metrics, we aim to evaluate the accuracy of the array inference performed by the decompiler. We examine each array comparison made during the comparison of the ground truth with the decompiler and observe the discrepancies in length, size (bytes), dimensions, and element type. The following metrics are presented:

- *Ground truth varnodes (metatype=ARRAY)*: The number of ground truth varnodes with metatype of ARRAY.
- *Array comparisons*: The number of array comparisons made when comparing the ground truth with the decompiler. The decompiler may infer 0 or more array varnodes for each given ground truth array varnode.
- *Array varnodes inferred as array*: This measures how many ground truth array varnodes are compared to at least 1 decompiler-inferred array varnode.
- *Array varnodes inferred as array fraction*: Equivalent to *Array varnodes inferred as array* divided by *Ground truth varnodes (metatype=ARRAY)*. This expresses the fraction of ground truth array varnodes that are associated with at least one decompiler array inference.
- *Array length (elements) average error*: For each array comparison, we find the absolute difference in the number of elements inferred by the decompiler as compared to the ground truth. We then average these differences over all array comparisons to arrive at this metric.
- *Array length (elements) average error ratio*: For each array comparison, we first find the absolute difference in the number of elements inferred by the decompiler as compared to the ground truth. We then divide this error by the length of the ground truth array to get the error as

a ratio of the array size. The average of these ratios over all array comparisons produces this metric.
- *Array size (bytes) average error*: This metric is similar to *Array length (elements) average error* but measures the error in bytes instead of number of elements.
- *Array size (bytes) average error ratio*: This metric is similar to *Array length (elements) average error ratio* but computes the error in bytes instead of array elements.
- *Array dimension match score*: This metric is the number of array comparisons where the decompiler inferred the correct number of dimensions divided by the total number of array comparisons.
- *Array average element type comparison score*: Each *data type comparison level* is first mapped to an integer as follows: *NO_MATCH* = 0, *SUBSET* = 1, *PRIMITIVE_COMMON_ANCESTOR* = 2, *MATCH* = 3. We then normalize these values such that the range is scaled from 0 to 1. We refer to this as the *data type comparison score*. Then, for each array comparison, we compute the *data type comparison score* and subsequently average the scores across all array comparisons to generate this metric.

IV. Evaluation

To demonstrate our evaluation framework, we target the Ghidra decompiler (version 10.2). We use the GNU Core Utilities programs (version 9.1) as our set of benchmarks. For each of the benchmark programs, we evaluate the accuracy of Ghidra decompilation with the program compiled in three ways: (1) stripped, (2) standard (not stripped, no debugging symbols), and (3) DWARF debug symbols included. We use the results from each of these cases to discern how the amount of information included in the binary affects the Ghidra decompiler's inference accuracy. To limit the scope of our analysis, we only consider unoptimized binaries. We use the GCC compiler (version 11.1.0) to compile the benchmark programs. The architecture and operating system of the testing machine are x86-64 and Ubuntu Linux (version 20.04), respectively.

A. Setup

Prior to evaluation, we compile the 105 Coreutils benchmark programs with three compilation configurations: (1) stripped, (2) standard (not stripped, no debugging symbols), and (3) DWARF debug symbols included. For each program, we first extract the ground truth information from the binary with DWARF symbols included via our DWARF translation module. We then use our Ghidra translation module to extract the Ghidra decompilation information from the binaries compiled under each of the compilation configurations. At this point, all program information from the DWARF and Ghidra sources are represented as *ProgramInfo* objects in our DSL.

Next, for each program, we perform a comparison of the program information scraped from DWARF (from the "debug" binary including DWARF symbols) with the information obtained from the Ghidra decompilation of the programs under each of the compilation configurations. The information from these comparisons are expressed in the form of objects which contain comparison information about functions, variables, and data types compared between the DWARF and Ghidra sources.

With the comparisons computed for each program and compilation configuration, we use these comparisons to compute high-level metrics that summarize the performance of the Ghidra decompiler with respect to the given benchmarks and compilation configurations (stripped, standard, and debug).

B. Function Recovery

Tables XX, YY, and ZZ in the appendix present function recovery metrics of each benchmark program under the three compilation configurations. Table AA shows the summarization of the recovery statistics accumulated over all benchmark programs. We find that over the 18139 functions present in the ground truth, the stripped and standard compilation cases produce 100% function recovery while the debug case fails to recover four functions, resulting in a 99.9% recovery rate. Upon examination of table ZZ in the appendix, we find that all four functions missed are from the *factor* program.

To determine the cause of the missed functions, we further investigate the Ghidra decompilation of *factor* and find that each of the missed functions results in a decompilation error, "Low-level Error: Unsupported data-type for ResolveUnion". This indicates that an error occurred when attempting to resolve a union data type within the decompilation of these functions. Since this error only occurs in the debug compilation case, it is clear that Ghidra's parsing and interpretation of DWARF information contributes to this error. This same union data type causing the error is successfully captured and represented in our ground truth program information and, thus, this is likely a bug within Ghidra's resolution logic.

In summary, we see that Ghidra successfully finds all functions for all compilation configurations. However, in the debug case, Ghidra's attempt to interpret and utilize DWARF information to resolve a union data type in the *factor* program results in a decompiler error for four functions. This error indicates a bug in Ghidra's DWARF parsing or union resolution logic.

C. High-Level Variable (Varnode) Recovery

To evaluate the variable (varnode) recovery accuracy of the Ghidra decompiler, we first measure the inference performance of high-level varnodes, including varnodes with complex and aggregate types such as arrays, structs, and unions. We further measure the varnode inference accuracy by metatype to decipher which of the metatypes are most and least accurately inferred by the decompiler. This analysis is performed under each compilation configuration (stripped, standard, and debug).

Tables XX, YY, and ZZ in the appendix show the inference of high-level varnodes for each benchmark compiled with each of the compilation configurations. This data is summarized in table AA. We find that Ghidra at least partially infers 97.2%, 99.3%, and 99.6% and precisely infers 36.1%, 38.6%, and

99.7% of high-level varnodes for each for the stripped, standard, and debug compilation cases, respectively. In addition, the varnode comparison scores for each compilation case are 0.788, 0.816, and 0.998, respectively. These metrics indicate that the standard compilation case slightly outperforms the stripped case in varnode inference while the debug compilation case results in significant improvements over both the stripped and standard cases, particularly in exact varnode recovery.

In tables XX-XX, YY-YY, and ZZ-ZZ, we show the inference performance of high-level varnodes for each benchmark, broken down by the metatype of the ground truth varnodes, and for all compilation configurations. We summarize this information in table BB. From the stripped and standard compilation cases, we observe that varnodes with metatype *INT* are most accurately recovered when considering varnode comparison score, fraction partially recovered, and fraction exactly recovered. In the stripped case, the inference of *ARRAY* varnodes shows the worst performance with a varnode comparison score of 0.315. In the standard case, varnodes with metatype *STRUCT* are least accurately recovered with a varnode comparison score of 0.560, followed closely by *ARRAY* and *UNION*. We see that, for both the stripped and standard compilation cases, the complex (aggregate) metatypes, *ARRAY*, *STRUCT*, and *UNION*, show the lowest recovery accuracy with respect to varnode comparison score. Among the primitive metatypes, *FLOAT* shows the worst recovery metrics for these two compilation cases.

The debug compilation case demonstrates high relative recovery accuracy across varnodes of all metatypes when compared to the stripped and standard cases. Of the primitive metatypes, varnodes of the *FLOAT* metatype are perfectly recovered while varnodes of the *INT* and *POINTER* metatypes show exact recovery percentages of 99.8% and 99.9%, respectively. The complex (aggregate) metatypes, on average, display slightly lower recovery metrics than the primitive metatypes in the debug compilation case. The *ARRAY* metatype reveals the worst varnode comparison score at 0.986. The *UNION* metatype demonstrates the lowest exact match percentage at 87.5%.

D. Decomposed Variable (Varnode) Recovery

In this section, we repeat a similar varnode recovery analysis over all varnodes; however, we first recursively decompose each varnode into a set of primitive varnodes (see section XX). We perform this analysis over all benchmarks for each of the three compilation cases.

Similar to the high-level varnode analysis, we show the inference of the decomposed varnodes for each benchmark and for each compilation configuration in appendix tables XX, YY, and ZZ. Table AA summarizes this information. Naturally, we expect to see lower recovery metrics compared to the high-level varnode analysis since each complex varnode is now analyzed as a set of its constituent parts. Hence, a single "missed" high-level varnode is translated into a set of primitive varnodes, each "missed" in this analysis. We find this hypothesis to hold true across all compilation cases as each the varnode comparison score, varnodes fraction partially recovered, and varnodes fraction exactly recovered show lower values than in the high-level analysis. We see that the decomposed varnode comparison scores for the strip, standard, and debug compilation cases are 0.586, 0.703, and 0.980, respectively. The varnodes fraction partially recovered are 73.8%, 92.5%, and 98.0% while the varnodes fraction exactly recovered are 24.7%, 25.0%, and 98.0% across the compilation cases, respectively. Interestingly, in the stripped compilation case, we find that the number of "missed" decomposed varnodes (139937) exceeds the number of "exactly matched" decomposed varnodes (131719). This is largely due to the quantity of high-level *ARRAY* and *STRUCT* varnodes that are missed in the stripped case.

We split the decomposed varnodes by metatype and show these results in tables XX-XX, YY-YY, and ZZ-ZZ. We present the summary of these results over each compilation case in table BB. The table shows that the stripped and standard compilation cases demostrate the poorest inference performance in terms of varnode comparison score for varnodes of metatype *FLOAT*. However, we find that the percentage of "missed" *INT* varnodes is worse than that of *FLOAT* in the standard and debug compilation cases, and is nearly the same in the stripped case. This may be explained by the prevalence of integer (or character) arrays in the Coreutils benchmark programs when compared to other array types. Recovery accuracy of the *POINTER* metatype is comparable to the *INT* metatype across the three compilation cases.

E. Data Bytes Recovery

Following from our varnode inference analysis, we next assess the accuracy of the Ghidra decompiler with regards to the total number of data bytes recovered across all varnodes. This analysis provides an important perspective on data recovery as the size of an improperly inferred varnode may result in a wide range in the number of misinferred bytes. For example, a large array and a single character are each represented by a varnode, but the quantity of data present in the array is much greater than that of a character. Hence, it is important to capture this nuanced view of data recovery.

In tables XX, YY, and ZZ, we show the data bytes recovery metrics for each of the benchmark programs under each compilation case. We summarize the data bytes recovery for each of the compilation cases in table AA. We see that Ghidra recovers 61.3%, 80.6%, and 99.5% of data bytes in the stripped, standard, and debug compilation cases, respectively.

F. Array Inference Accuracy

The last major analysis we perform targets the array inference accuracy of the Ghidra decompiler. We aim to measure metrics regarding the total number of arrays inferred, the length and size discrepancies of compared arrays, and the similarity of element types of compared arrays. We perform this analysis across the Coreutils benchmarks and for each compilation configuration, resuling in tables XX, YY, and ZZ

located in the appendix. This information is summarized in table AA, broken down by compilation configuration.

Across all benchmarks, there are 2138 ground truth arrays present. For each the stripped, standard, and debug compilation cases, the number of ground truth arrays recognized as arrays by the decompiler are 774 (36.2%), 1530 (71.6%), and 2128 (99.5%), respectively. We see that the numbers of array comparisons for each compilation case are greater than these metrics indicating that Ghidra infers some ground truth arrays to be more than one array.

From the array comparisons, we observe that the average absolute differential in array length (number of elements) for the stripped, standard, and debug compilation cases are 134.7, 151.2, and 9.4, respectively. When scaling these errors with respect to the length of the ground truth arrays in the comparisons, the error ratios are 2.84, 5.44, and 0.11 for the compilation cases, respectively. This reveals that, in the debug case for example, the lengths of decompiler-inferred arrays are off by an average of 9.4 elements and roughly 11% (greater or less than) of the size of the ground truth arrays they are compared to. These metrics, however, fail to capture whether the decompiler-inferred array has element types of the correct length. Thus, a similar analysis on the size (number of bytes) errors yields errors and error ratios of 458.6 (0.91), 239 (0.47), and 9.41 (0.11) for each compilation case, respectively. This, for example, shows that arrays inferred in the standard compilation case have an average absolute byte differential of 239 and a relative error of 47% compared to the size of the ground truth array they are compared to.

In this analysis, we also capture a measure of the array dimension match score for each compilation case. This metric measures the fraction of array comparisons where the decompiler-inferred array has the same dimensionality (one-dimensional, two-dimensional, etc.) as the ground truth array. The stripped and standard compilation cases display dimensionality match ratios of greater than 97.4%, while the debug case shows 100% dimensionality inference accuracy.

The last portion of our array recovery analysis focuses on the element type inference accuracy of the decompiler-inferred arrays when compared to the element types of the ground truth arrays. We compute a data type comparison score between the element types from each array comparison and average these across all array comparisons derived from our benchmark programs. This data type comparison score is similar in concept to the varnode comparison score and is described in section XX. We find that decompiler-inferred arrays in the stripped, standard, and debug compilation cases show 0.781, 0.670, and 0.999 average element type comparison scores, respectively. The better performance demonstrated in the stripped case compared to the standard case appears to be a data artifact resulting from fewer array comparisons present in the stripped analysis.

G. Debug Compilation Case Discussion

Upon examination of our results thus far, the reader may wonder why the debug compilation case does not produce 100% recovery for varnodes and data bytes across all benchmarks. The same DWARF debugging information used to generate the ground truth program information is also provided to the Ghidra decompiler in this case and therefore, theoretically, Ghidra should be able to precisely capture the same program information.

We manually investigate this phenomenon over our benchmark programs and find that the cause of these recovery inaccuracies stems from the Ghidra decompiler's inflexibility in expressing local variables tied to lexical scopes. We find that the Ghidra decompiler output only lists variable declarations at the top level of the function and does not support declarations of local variables within lexical scopes. Instead, Ghidra attempts to move the declaration of these scope-specific variables to the top level of the function. Often, this behavior does not negatively influence the variable recovery of the given function. However, there are cases where multiple exclusive (not overlapping or nested) lexical scopes contain variable declarations. In many of these cases, the compiler recognizes the exclusivity of the lexical scopes and assigns the scope-specific variables to shared space on the stack since the variables shall never be instantiated simultaneously. The size of the shared region allocated by the decompiler is equivalent to the size of the largest variable in the set of scope-specific variables that share the region. In essence, this is equivalent to an implicit union formed by the compiler. The DWARF debugging standard and our DSL both possess the ability to express these overlapping scope-specific variables, but the Ghidra decompiler does not. From our observations, we find that Ghidra greedily captures and declares scope-specific variables at the top level of the function based on the order in which it recovers the variables. In the debug compilation case (utilizing DWARF information), Ghidra appears to only consider the first scope-specific variable mapped to a given address on the stack based on the order of the variables in the list of debugging information entries (DIEs) parsed from DWARF. The subsequent scope-specific variables associated with the given address are simply ignored, causing Ghidra to potentially miss several varnodes and data bytes. We consider this to be a shortcoming and an area of future improvement for the Ghidra decompiler.

V. Conclusion

A. Summary of Methodology

To develop our decompiler evaluation framework, we outline and execute the following objectives:

1) Express program information such as functions, variables, data types, and addresses in a common representation.
2) Programmatically capture a "ground truth" representation for a given program.
3) Programmatically scrape program information from decompiler tools, namely Ghidra.
4) Compare two program representations of the same program.

5) Formulate quantitative metrics for evaluating the accuracy of a decompiler.

We devise and implement a common domain-specific language (DSL) for expressing the association of high-level program information such as functions, variables, and data types, with binary-level constructs such as addresses and storage locations. With our DSL, we develop a parser for extracting DWARF debugging information from binary programs and representing this information in our DSL. This information is to be used as a ground truth source of program information in comparisons with decompiler representations. Next, we leverage the Ghidra Python API to develop a translator module, taking Ghidra decompilation output as our input and translating the information into our DSL. With our parsing modules constructed for both our ground truth and decompiler soures, we extend our DSL to support the comparison of two sources of program information parsed from a ground truth source and a decompiler source. We subsequently develop quantitative metrics for assessing and summarizing comparisons of program information sources.

B. Summary of Results

We utilize our developed framework to assess the recovery performance of the Ghidra decompiler (version 10.2) over the 105 GNU Core Utilities (version 9.1) benchmark programs. Using the GCC compiler (version 11.1.0), we compile the benchmarks with no optimizations under three separate compilation configurations: (1) stripped, (2) standard (not stripped, no DWARF symbols added), (3) debug (DWARF symbols included).

Our function recovery analysis reveals that Ghidra recovers 100% of the 18139 functions across all benchmarks in the stripped and standard compilation cases. In the debug case, we find four missed functions in total, all present in the *factor* benchmark program. We discover that the missed functions are all caused by a decompiler error resulting from a failure in resolving a union data type. We conclude that this is a bug in the Ghidra decompiler.

In our high-level varnode analysis, we find that the recovery accuracy of primitive (*INT*, *FLOAT*, *POINTER*) metatypes is greater than that of the complex (aggregate) metatypes (*ARRAY*, *STRUCT*, *UNION*) across all compilation cases. This finding follows from the fact that inferring complex varnodes involves an extra layer of speculation and inference involving the synthesis of low-level varnodes. In all compilation cases, the *ARRAY* metatype displays the greatest number of "missed" varnodes.

Our decomposed (primitive) varnode analysis demonstrates that Ghidra is least effective at inferring floating-point (metatype *FLOAT*) decomposed varnodes over the benchmark programs in the stripped and standard compilation cases. However, we see that Ghidra completely misses a larger fraction of decomposed varnodes with metatype *INT*. This is explained by the larger incidence of integer arrays in the Coreutils benchmark programs, which are more likely to be missed or only partially recovered as demonstrated in our high-level varnode analysis. We show that decomposed varnodes of metatype *POINTER* are recovered comparably to those of metatype *INT*.

In our analysis of data bytes recovery summarized across all benchmarks, we find that the Ghidra decompiler shows 61.3% recovery in the stripped compilation case, 80.6% recovery in the standard case, and 99.5% recovery in the debug case.

Our array inference analysis illustrates that the compilation configuration of our benchmark programs has a significant impact on both array recovery and the inference accuracy of the arrays that are recovered. We find that, for each the stripped, standard, and debug compilation cases, 36.2%, 71.6%, and 99.5% of ground truth array varnodes overlap with at least one associated decompiler-inferred array varnode, respectively. We find the average size error ratio of the decompiler-inferred arrays with respect to the ground truth arrays to be 0.91, 0.47, and 0.11 for the compilation cases, respectively.

The function, variable, data bytes, and data type recovery analyses show clear recovery accuracy differentials between the three compilation cases. In general, we find that the debug case (DWARF symbols included) performs the best by a large margin, followed by the standard case which slightly outperforms the stripped case. Despite the decent recovery performance in the debug case, we seek an explanation for the decompiler still failing to capture a portion of the ground truth information, particularly varnodes and data bytes. We find that the Ghidra decompiler is limited in its ability to express overlapping stack variables gathered from non-overlapping, non-nested lexical scopes within the same parent function. This scenario arises when the compiler recoginizes the exclusivity of lexical scopes within a function and subsequently assigns scope-specific variables from these lexical scopes to the same address or region on the stack.

C. Limitations

The primary limitation of our framework in its current state is the lack of support for comparing and evaluating program information gathered from optimized binary programs. Our DSL supports the expression of program information from optimized binaries, but the comparison logic assumes certain properties about the program information to reduce the complexity of the analysis. Namely, we assume that each high-level variable to be associated with a single storage location in memory for the purposes of comparison. In addition, we assume that the program counter (PC) "live range" of the variable is the entire PC range of the parent function for local variables and the entire program for global variables. In optimized binaries, these assumptions do not always hold. For example, optimizations may result in a single high-level variable being stored across a combination of stack locations and registers depending on the current instruction. In essence, optimizations introduce an additional temporal dimension that drastically increases the complexity of the analysis. Each live range of each variable would need to be considered, then a set of comparison "snapshots" would need to be performed based

on the overlaps of the variable live ranges. An aggregation of these "snapshot" comparisons shall then be performed in such a way to evaluate the recovery of each of the high-level variables. Our current framework is built with this type of analysis in mind, but the scope of this work only considers the case of unoptimized binaries. Future work shall include the extension of the framework to support the evaluation of optimized binaries.

Another assumption in our analysis is that only non-parameter variables with stack and absolute (global) addresses are considered for comparison. This includes heap-allocated data which must be referenced by a pointer accessible from the current function. Our language and framework support the ability to represent register and register offset locations which shall be useful in future optimized analysis.

Another limitation in this work is our exclusive support for the DWARF debugging standard for extracting ground truth program information. However, as discussed previously, our framework can easily be extended to support the implementation of parsers for other debugging formats.

Regarding decompiler evaluation, our framework excels at assessing the recovery and inference of high-level program constructs. However, our framework lacks any form of behavioral analysis. Existing work by Liu and Wang [] showcases an approach to evaluating the behavioral correctness of decompiler outputs. A full decompiler analysis shall combine our structural analysis with the behavioral analysis demonstrated by this existing work.

The final noteworthy limitation in our work is that we use our framework to assess only the Ghidra decompiler. We consider our framework to be the primary contribution of this research and therefore leave the analysis and comparison of other decompilers for future work.

D. Future Work

As discussed in the previous section, a major future work objective shall be to extend our framework to support optimized binaries. In addition, we shall use our framework to assess and compare the recovery performance of decompilers beyond Ghidra.

In our function recovery analysis, recall that the Ghidra decompiler fails to decompile four functions within the *factor* program only in the case where DWARF debugging symbols are included. We conclude from the error messages returned that the decompilation errors for these functions result from Ghidra's inability to resolve a particular union data type present in the program. Since this error does not occur for the other compilation cases of the *factor* program, we gather that the DWARF information scraped by Ghidra contributes to this error. With this observation, we recognize that a useful obfuscation strategy for binary programs may, instead of stripping all debugging symbols, be to include misleading and contradictory debugging information. Reverse engineering tools and decompilers analyzing a binary program with misleading debugging symbols included may produce incorrect outputs or potentially crash based on this erroneous information. This is certainly an area worthy of future research. In addition, the union resolution issue observed in our analysis shall be patched in the Ghidra framework.

In our assessment of the Ghidra decompiler, we observe that Ghidra does not successfully capture all ground truth variables and data bytes even in the case the DWARF debugging information is present. Upon further investigation, we discover this shortcoming is due to Ghidra's inability to express local variable declarations at the lexical scope level. Instead, Ghidra forces all local variables to be declared at the top level of the given function. This causes Ghidra to partially miss cases where the same stack address region is used by the compiler to store local variables declared in non-overlapping, non-nested lexical scopes within the same function. An area of future work shall be to modify the Ghidra decompiler to support the expression of more flexible local variable constructs that are not required to be declared at the top level of a function.

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

APPENDIX

[Table 1: FUNCTIONS (strip)] [Table 2: FUNCTIONS (standard)] [Table 3: FUNCTIONS (debug)]

[Table AA: VARNODES (strip)] For X in INT, FLOAT, POINTER, ARRAY, STRUCT, UNION... [Tables XX-YY: VARNODES (metatype=X) (strip)]

[Table AA: VARNODES (standard)] For X in INT, FLOAT, POINTER, ARRAY, STRUCT, UNION... [Tables XX-YY: VARNODES (metatype=X) (standard)]

[Table AA: VARNODES (debug)] For X in INT, FLOAT, POINTER, ARRAY, STRUCT, UNION... [Tables XX-YY: VARNODES (metatype=X) (debug)]

[Table AA: VARNODES (decomposed) (strip)] For X in INT, FLOAT, POINTER, ARRAY, STRUCT, UNION... [Tables XX-YY: VARNODES (decomposed) (metatype=X) (strip)]

[Table AA: VARNODES (decomposed) (standard)] For X in INT, FLOAT, POINTER, ARRAY, STRUCT, UNION... [Tables XX-YY: VARNODES (decomposed) (metatype=X) (standard)]

[Table AA: VARNODES (decomposed) (debug)] For X in INT, FLOAT, POINTER, ARRAY, STRUCT, UNION...

[Tables XX-YY: VARNODES (decomposed) (metatype=X) (debug)]

[Table YY: BYTES (strip)] [Table YY: BYTES (standard)] [Table YY: BYTES (debug)]

[Table ZZ: ARRAY COMPARISONS (strip)] [Table ZZ: ARRAY COMPARISONS (standard)] [Table ZZ: ARRAY COMPARISONS (debug)]