Deb'm:
a benchmarking platform for decompilers

by

Jaap van den Bos
Kesava van Gelder
Reijer Klaasse

Student number:	Jaap van den Bos:	850 462 793
	Kesava van Gelder:	852 205 955
	Reijer Klaasse:	852 083 280
Course code:	IB9906	
Thesis committee:	Dr. Ir. Hugo Jonker (examinator),	Open University
	Dr. Ir. Harrie Passier (examinator),	Open University
	Dr. Nico Naus (supervisor),	Open University
	Dr. Tim Steenvoorden (supervisor),	Open University

Open Universiteit
de beste!
www.ou.nl

Deb’m: a benchmarking platform for decompilers

Jaap van den Bos Kesava van Gelder Reijer Klaasse
Open University of The Netherlands

***Abstract*—This paper subscribes itself to the field of decompilation: the discipline of converting low-level code (*eg.* binaries) to higher-level code (*eg.* C source code). Decompilation is vital for source code recovery, binary patching, security analysis, *etc.* Existing decompilers all use their own methods of measuring performance. This makes them often incomparable. This problem has been partially addressed by Kline, creating a framework for testing decompilers, but for now only suitable for the Ghidra decompiler.**

In this paper we present deb’m, a framework suitable to test virtually any decompiler that converts x86(-64) binaries to C code. We do this by making use of smart code markers, so no other information is needed from the decompiler than the decompiled C code. By making use of deb’m, researchers can independently verify the quality of their decompiler. This leads to more comparability between decompilers. To show that deb’m works in practice, we test Ghidra, RetDec and Hex-Rays with deb’m and discuss the results. A framework like deb’m is important, because it provides an independent benchmark for decompilers, indicating the quality of existing decompilers, and aiding the further development of future ones.

I. Introduction

From a programmer’s point of view, a program is a set of human-readable instructions, written in some programming language. Over time a whole range of programming languages has been developed. They were all designed with certain audiences, purposes and constraints in mind. We can study them, as well as the programs written using those languages. If we do the latter, we study source code. This source code needs to be translated into CPU-understandable instructions and put together in so-called binaries. This process is called compilation. These binaries are the CPU’s point of view, as a CPU only understands numbers, stored in a binary form in the computer’s memory.

If we want to know more about a piece of software, studying its source code is the first thing we would like to do. However, not all source code is available. Many companies regard their software’s source codes as trade secrets and treat them as such, and malware creators will not be so kind as to show easily how their products work. Sometimes source code is lost over time for some reason or maybe we do not trust the source code’s publisher. In the absence of source code, studying software means studying binaries.

Studying binaries requires decompilers: software that translates binaries back into a human-readable programming language (the decompiler’s target language). With their help we can (try to) figure out what binaries do and how they do it, which is useful for things like debugging, patching, security analysis, and source code recovery.

Compilation essentially is a one-way process. It is very hard even to distinguish between data sections of a binary and instruction sections. This problem is equivalent to the Halting Problem, meaning it is unsolvable in general [HM80]. Lots of information is lost during compilation, such as symbolic constants and information on identifiers (variable names, function names, labels), though some information may be included as debugging information. A compiler only has to produce a program that is functionally equivalent to the source code. As long as the binary does what it is supposed to do, it does not matter how it is done. This opens doors for optimization of all sorts. A simple example is the inlining of functions, which means effectively copying the function’s code everywhere it is called and omitting the function itself. It saves the overhead of calling, thus speeding execution, though at the price of a larger binary.

Optimization, the loss of information, and the Halting Problem are just a few of the many problems a decompiler faces and these problems can be addressed in different ways, yielding different results when a binary is decompiled. So it is natural that a wide range of decompilers is currently available. They differ in the target language(s) they decompile to, the kind of sources they can handle (some take binaries, some take (Java) byte code), their user interfaces, and their strategies for identifying code fragments such as functions or loops.

We focus on general-use, publicly available decompilers that produce code in the language C. C’s architecture is very close to CPU architectures and therefore, almost any binary can be decompiled into some sort of C, even if it was compiled from another programming language. C is a very popular language for operating systems [WF06, p. 269] and, for the past two decades, has never been below number 2 on the Tiobe index of most popular programming languages [Ti, retrieved May 9, 2024]. It is not surprising therefore, to find a wide range of C decompilers, such as Ghidra [NSA], IDA Pro [IDA] (also referred to as Hex-Rays [YEGPS15]), RetDec [Ret], Smartdec [Sma], Binary Ninja [Nin], JEB [JEB], *etc.*

Given the multitude of decompilers, the question is how to compare them. In many papers, a new decompiler is presented with the emphasis on a certain ability, for example, its ability to decompile without emitting `goto`’s [YEGPS15]. However, this method disregards all other quality indications; there is no clue whether the presented decompiler can find functions or handle data types correctly. Furthermore, the emphasis is on the new decompiler, not on the fairness of the benchmark. The metric comes after the decompiler, where it should be the other way around. A third problem is that these comparisons do not use a standard set of binaries to benchmark, but most simply use their own. This makes it hard to compare the results from different papers.

What we need is a common benchmark: a set of binaries and metrics that can be used to compare decompiler results. Benchmarking enables researchers and developers to easily show that they are on the right track, meaning that their decompiler meets industry standards and indeed exceeds earlier approaches. We know of benchmarks for several computer science fields, such as benchmarks for databases [VM05], cyber-physical systems [ASH13] and networks [HBJS18], but as far as we know, no benchmark for decompilers has been developed yet.

This paper does not present a new decompiler or a new decompiler ability. Our main attention is defining metrics that can be used to describe decompiler quality. Metric*s*, in the plural, because we believe that decompilers must be benchmarked on more abilities. Our platform produces the binaries that comprise the benchmark, saving ground truth information about them. Not only that: it also assesses the decompiler's output, compares it to the ground truth, and computes the metrics. The framework is very versatile: it can be used on any decompiler that supports x86 and/or x64 CPU architectures and emits C. With the use of shell scripts, the platform can invoke any decompiler it can address from the command line, either locally installed decompilers or online available ones. Sample scripts are provided both for Windows and Linux. Written in Java, the platform is OS-independent, which also means we can use decompilers running on any OS. Finally, we have made our platform expandable, making it easy to further develop test metrics.

II. Our framework: deb'm

A schematic overview of our framework is shown in Figure 1. It works roughly as follows. On the forehand, a *producer* part dynamically generates hundreds of C sources and compiles them each with several different flags. This results in a set of sources, binaries, assembly and LLVM-IR representations, wrapped in *containers*. The user runs our program, providing a decompilation script that should take in a binary, and write a decompiled C file to an indicated location. An *assessor* unit feeds the generated artefacts to the decompilation script and compares the result with the ground truth. Finally, a reporting unit generates an overview of the test results. Deb'm delegates the actual work of producing code and assessing the decompiler to so-called feature classes. On the forehand, deb'm allows every feature class to generate input for the ground truth. At runtime, deb'm presents the decompilation result to every feature class, allowing them to calculate their specific metrics. We will now explain the producer and assessor in more detail.

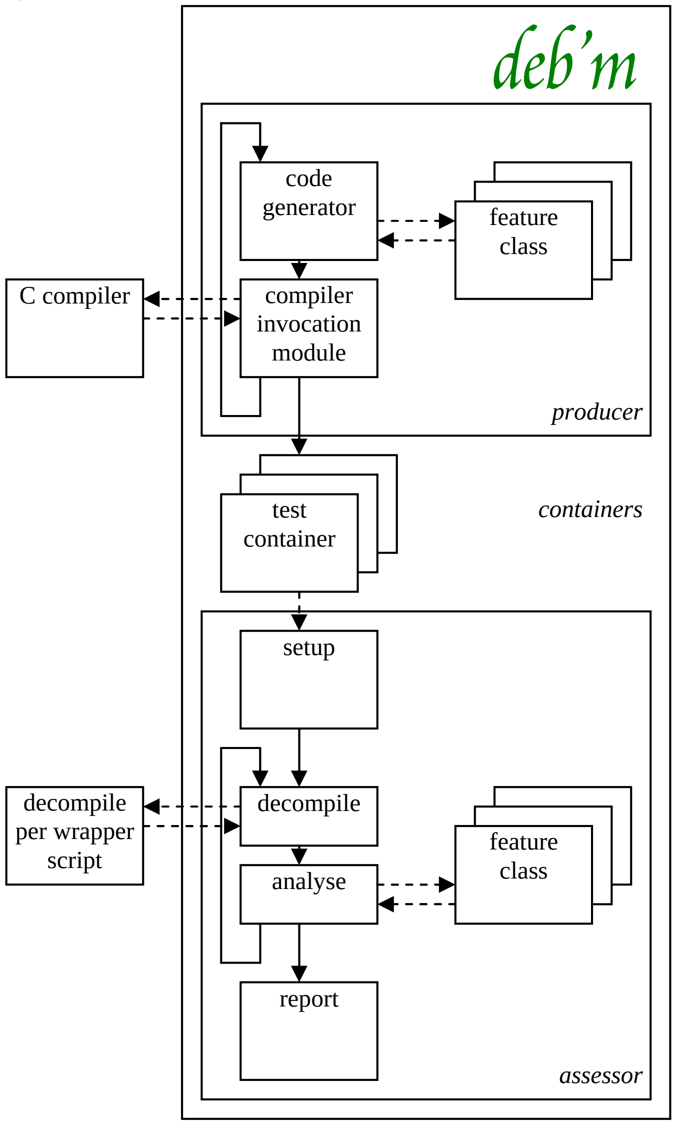
The producer

The producer generates the ground truth and consists of two units: the generator and the compiler.

The *code generator* produces C-sources by asking all of the feature classes for pieces of code. To start, it always generates `int main()`. It keeps adding statements to this function until all feature classes are satisfied with their contribution to the source. The algorithm is recursive in that the feature classes, in turn, can ask the producer for specific code. An example: the producer asks feature class A for a function definition. Feature class A will produce the function info, but in doing so, it must not only consider the function's name, but also its parameters. So, it will choose several parameters, for which it needs certain data types. This may cause feature class *A* to ask the producer for a new or existing data type (say: a `struct`). The producer then passes this request to a feature class that can create `struct`'s. That `struct` will be added to a collection of `struct`'s, so not only its definition will be added to the code only one single time, but it becomes available to all feature classes. The system keeps track of the depth of the recursion, forcing it to end in time. To allow for specific functions when needed, a feature class can also actively inject a function into the source, instead of waiting for the generator to ask for one.

Some feature classes may want to produce functions that are not optimized away by the compiler. To achieve this, a feature class can indicate that a function definition must be put in a separately compiled C-file and then linked to the compiled

Fig. 1. Deb'm framework overview



main source file with linker optimization disabled.

The *compiler invocation module* presents the C code to the compiler making sure the several compiler options are set correctly. Each C source will be compiled for the AMD/x64 and x86 architecture. Per architecture, we compile non-optimized (-O0) and optimized (-O3). At the moment, we only use Clang, targeting Windows, but others could be added, provided that they emit assembly and an intermediate representation, preferably LLVM-IR.

One fundamental concept our framework relies upon, is the concept of a code marker. The function of a code marker is to put useful information in the source code, that is preserved in the compilation and decompilation processes. We use calls to functions put in a separately compiled C-file. These function calls always take a string as the first argument and may take one more argument. The external functions use `printf` to print the string and, when provided, the second argument. The compiler cannot throw the calls to our code marker functions away, because at compile time it does not know what they do. This also means it cannot shift them around in the code, because it could not guarantee program correctness. We start our strings with a unique identifier (UUID), to tell them apart from any other strings that may occur in the binaries. After that comes the name of the feature class that created the marker, followed by the actual information. We end the string with a checksum, ensuring integrity.

To summarize, a code marker is a uniquely identifiable function call, that ensures the integrity of the textual information contained in the first argument.

All the produced files are organized in a folder structure. The root folder holds fifty container folders. Each of the container folders holds seventy-five test folders. Each test folder holds the source code, the binaries, and all intermediate files (such as assembly and LLVM-IR files).

The rationale behind putting so many binaries in one container is a balance between randomness and consistency. If we would present a decompiler with one single binary, the results may be arbitrary, as they would be based upon a sample of one. By creating a set of binaries, we can present decompilers with varying binaries: larger or smaller, with more or fewer functions, various data structures *etc.*. This means that every binary itself has randomized contents and testing one can yield unexpected results, but all the binaries in one container put together have all the different variations we want, and testing the set will produce a consistent result.

The assessor

The assessor consists of three units: a setup unit, an assessment unit, and a reporting unit.

The setup unit randomly chooses a container from the precompiled containers. It does so to prevent a decompiler from adapting its working to one specific container to gain better results. For every binary in the container, it calls the provided decompiler script with the location of the binary and the location to put the decompiled C code. Finally, it checks whether there has been a result from the decompiler, and passes this to the assessment unit.

The assessment unit parses the source C files, the generated LLVM-IR file, and the decompiled C code using ANTLR [PQ95]. It then executes the assessment method of every feature class, passing in the parsed result. Every feature class returns one or more scores for this single binary. The assessor stores these results, along with the compiler flags used to create the binary.

The reporting unit groups all gathered scores by compiler configuration (compiler, optimization level, and architecture). It then asks every feature class for a desired representation of the list of test results. The feature class can set lower and upper bounds for every metric, as well as a target value. The report unit will calculate how near the value that was found after assessing (the actual value) is to the target value. A spot on actual value will result in 100%, an actual value on one of the bounds will result in 0%. This way, feature classes have all the flexibility to implement any metric whatsoever, but the comparison is always easy: the higher the percentage, the closer to the target value, the better the score. Finally, output is generated. By default, this is an HTML file containing a table. The reporting unit can also produce the results in XML format, which facilitates automated processing of the output. For publication purposes, the assessor can also emit TikZ figures and LaTeX tables.

Prerequisites

Having seen all this, we can identify several prerequisites. A decompiler must

- be able to decompile x86 or x64 binaries;
- produce C code;
- be executable from the command line, without requiring user interaction. To communicate with deb'm, a decompiler will most likely need a wrapper script;
- be able to correctly decompile strings as the first argument of a function call. Deb'm needs this to use its code markers.

These prerequisites are relatively general, making deb'm a versatile benchmarking framework.

III. Control flow

In this section, we will discuss how to assess decompilers on their handling of control flow. We start with general remarks on control flow and the challenges that decompilers face. Next, we explain why we choose to primarily focus on loops, how we build loops into our binaries and what we want to measure.

A CPU core reads an instruction from memory, executes it, and moves on to the next instruction. Control flow deals with determining which instruction will be executed next. Programming languages offer different constructs to alter the default (sequential) control flow. More basic ways are `goto`, `if...else`, and function calls. More advanced are `switch`-constructions and iteration constructions (`do...while`, `while`, or `for`). Function calling is reasonably easily recognizable in a binary, as it will involve specific processor instructions. All other constructs will produce one or more jumps in the binary and the decompiler's challenge is to interpret these jumps correctly.

The decompiler must determine what construction in C produced the jump. Was it an `if`-construction or perhaps a loop or a `switch`? All of these can be combined and jumps may even be the result of the `goto`-instruction, which is available in C and may have its use in breaking out of nested loops or handling errors [KR88, p. 66].

Ordinary `if...else` constructs will not pose much of a challenge to a decompiler: the entry point of such a construct is reasonably easily determined, as one can see the computation of a test value and a conditional jump, together making up the `if` statement. The exit point will also be relatively easy: depending on the test result, the next block of instructions will be executed or a jump is made to skip that block.

Much more of a challenge are loops. A loop exists when there is a control flow path that jumps back to a certain instruction that has already been executed. That jump back may be caused by a jump at the end of a loop body, but also by a jump resulting from a `goto` or `continue` statement. Loop bodies may have multiple exit points; not only the normal exit point at the loop continuation test, but also exits caused by `break`, `goto`, `exit()`, or `return`. Combined with the possibility of nested loops, decompilers may have quite a challenge in reconstructing the loop.

Dealing with optimized binaries poses even more challenges. In optimized code, loops may be unrolled. Unrolling means that the loop body, instead of being present once and executed repeatedly, is inserted the required number of times. This saves initializing, updating and testing variables and jumping back to the loop body's start. The execution time saved comes at the price of a bigger binary. If a decompiler does not detect the unrolling, it will emit a larger and less readable decompiler result, containing a block of repetitious instructions. Optimization may also mean that conditional `break` statements are moved around and end up as the loop continuation test in the loop command, or that code after the loop, executed after a conditional `break`, ends up *in* the loop body as part of an `if` block.

In short: loops pose a lot of challenges and that is why we focus on them.

Loop construction

We want to score decompilers by presenting them with a large set of different loops. Loops can differ in many ways. For example: the loop instruction may be one of three (`do`, `while`, `for`), the loop may or may not update a variable every time the body is executed, and a loop may be finite or infinite. We have identified a set of properties and composed a repository containing loops with all kinds of different combinations of these properties. Most of the properties are straightforward, and for reasons of brevity, we only highlight a few.

Loops can be nested. Nesting can be described by trees: the outermost loop is the root of the tree, inner loops are children. Such a tree, which we call a nesting pattern, can have an unlimited number of shapes. We have built a repository containing tree shapes that vary on the number of children per loop and the tree depth. Every time we construct loops in the producer, we select a pattern and set the loop properties for every loop that is a node in the pattern. This ensures that different kinds of nesting patterns will be used. Naturally, we also added a root-node-only pattern, which results in a loop that neither has parents nor children.

Loops also differ in the absence or presence of control flow instructions in the loop body. To add some extra challenge, we have included explicit control flow instructions: `break`, `exit()`, `return` and `continue`. We have also added two types of explicit `goto` statements. One type jumps to a point further away from the first statement following the loop body (representing the use of `goto`'s for error handling), and the other type breaks out of multiple, nested, loops in one go. All these control flow instructions are conditional because otherwise parts of the loop body would be optimized away. After all, code below an unconditional `continue` is unreachable. Loops containing an unconditional `break` (or similar instruction) are no loops at all.

To present decompilers both with loops that have been unrolled and loops that haven't (we call the latter: preserved loops), we present the compiler with a specific set of loops that are attractive to loop unrolling. These loops only have a very small loop body, and they have few (less than twenty-five) iterations. All the other loops we present to the compiler are not attractive to unroll: they have hundreds of iterations and may have nested loops and dummy statements in them.

We have tried to make loop construction both quite random and quite complete.

The completeness is safeguarded by combining the two repositories containing loop properties and nesting patterns. Every time a loop is needed, the repository containing all the different possible loops is queried and one definition after another is emitted. The same goes for loop nesting patterns. Per container, we have many more loops than there are in the repository, so we are certain that all types of loops and loop nesting patterns are used. As the repositories keep their state between different source files, we ensure that per container all possible loops and all loop patterns are used equally often.

The randomness is safeguarded by randomizing the repositories at the beginning of the production cycle and every time all their elements have been queried. The number of loops per binary, the loop variable values (initial, update and test values), the number of dummy commands and the dummy commands themselves are also chosen randomly.

We put a code marker before each loop, called the before-marker, marking the loop as one of our own, and recording all the properties we put in the source for this loop (among which is a unique ID per loop). We start each loop body with a code marker called the body marker, marking the start of the body and recording its loop ID. Immediately after the loop, we put a code marker called the after-marker, showing where the loop body ends. In sections with dummy commands, code markers that only mark the loop ID can be found.

Loop assessment

Our framework calculates three metrics about loops.

First, we count how many of the loops we put in our binaries are detected as loops. This simple metric gives a bird's-eye view, as a ratio of the number of loops found to the number of loops that could be found.

Second, we count the number of `goto`'s present in the decompiled C code, as it is a very simple but informative metric that has been used before [YEGPS15]. Like counting loops, counting `goto`'s gives a bird's-eye view. Every preserved loop will produce at least one jump in the binary (the one at the end of the loop body, jumping back to its beginning) and clearly, this jump should not result in a `goto` in the decompilation result. As any flow chart can be programmed without `goto`'s [WF06, p. 228], decompilers should even be able to avoid emitting `goto`'s altogether. Therefore, by default, any `goto` found in decompiled C code is unwanted, except for the two `goto`'s we added explicitly. We should not punish a decompiler for keeping these excusable `goto`'s. This is why we also present a `goto`-count that excludes these excusable `goto`'s, showing only the inexcusable ones.

Finally, we present a loop quality metric. Each of our loops that the decompiler identifies will receive a quality score on a scale of 0…10. The score is the sum of scores (0…1) on ten different properties. We first list these properties and explain them in more detail below:

a. loop code is found in any form;
b. the loop is presented as a loop;
c. the loop command is correct;
d. there is no loop leaking;
e. there is no before-marker doubling;
f. there is no after-marker doubling;
g. the first loop body instruction is a body marker;
h. the loop body control flow is correct;
i. the loop variable test is correct;
j. there are no inexcusable `goto`'s.

a) Loop code is found in any form:
This is the most basic test, if it fails, the score for the loop will always be zero. We test this, to make sure that the decompiler found the loop (in whatever decompiled shape or form) at all. This may not be the case, for example, when the loop is in a function that the decompiler doesn't recognize as such. When we go through the decompiled code, every time we find a before-marker, we set the score for this loop to one.

b) The loop is presented as a loop:
This test is done to ensure some sort of loop statement is actually present in the code, showing that there is a loop of any kind. A loop that is not re-rolled by the decompiler will fail this test, as will loops that are constructed using `goto` statements. If this test fails, the score for the loop can never be more than one.

c) The loop command is correct:
This tests whether the loop command (`for`, `do` or `while`) in the decompiled code matches the loop command used in the source code. We treat `for` and `while` as equals as both constructs will only execute the body if the loop variable test holds *true* from the start, whereas `do` loops will always execute the body at least once. We do not distinguish between the different loop commands when assessing loops that have an always *true* test condition. As such a loop will keep on executing the loop body, it doesn't matter if the continuation test is at the top or the bottom of the body.

```c
int main(int argc, char ** argv) {
 while (true){
  printf("body1\n");
  if (getchar()==23){
   break;
  }
  printf("body2\n");
  printf("body3\n");
 }
 printf("done\n");
}
```

Listing 1. Loop leaking example, source

```c
int main(int argc, char ** argv) {
 // note: here starts the first body execution, no loop
      found yet
 printf("body1\n");
 if (getchar() == 23) {
  // note: this is copy of the code following the loop
  printf("done\n");
  return 0;
 }
 printf("body2\n");
 printf("body3\n");

 // note: here starts the second body execution, still
      no loop found yet
 printf("body1\n"); // note: this is a leaked statement
 while (getchar() != 23) {
  printf("body2\n");
  printf("body3\n");
  printf("body1\n");
 }
 printf("done\n");
 return 0;
}
```

Listing 2. Loop leaking example, decompiled result (using retdec), removed decompiler comments and added our own.

d) There is no loop leaking:
Consider Listing 1 and Listing 2, the first being the source, the latter being the decompiler output. Note that the first complete execution of the loop body is unrolled and the code following the loop is duplicated. The second loop execution starts with, what we call, the leaked statement. It is put *before* the loop, rather than in it, as if there was a leak in the loop body. This is unwanted behavior; it is confusing. So, if this behaviour is avoided, we score the loop an extra point.

e) There is no before-marker doubling:
We count the number of before-markers for a loop in the LLVM-IR and compare it to the decompiler output. Sometimes the before-marker gets duplicated in the decompiler's processing of the control flow. This is unwanted behaviour, for it is confusing.

f) There is no after-marker doubling:
We count the number of after-markers for a loop in the LLVM-IR and compare it to the decompiler output. We found that our `break` constructs (that are always conditional) often show up in the decompiled C file as conditional `return`'s. The instructions following the loop are duplicated into a conditional compound statement that ends with a `return`

instruction. This is confusing and thus should be avoided by the decompiler.

g) The first loop body instruction is a body marker: There is no need for things like parameter copy-in-mechanisms at the start of the loop body. Any initialization of the loop variable is done outside the loop body. So, whenever a loop body starts with something other than the body marker, something is wrong.

h) The loop body control flow is correct: The loop body contains at least one code marker (the body marker) and may contain more (used as dummy commands). We check all the code marker ID's, and they should be in ascending order. If they are not, they were somehow shuffled during decompilation.

i) The loop variable test is correct: Not all loops contain loop variables. If not present, we score the loop anyway (because otherwise it might not reach a perfect score). If we do use a loop variable in the source, the continuation test is always put in the loop command and never in a separate conditional `break` construct. We only test `getchar` to produce conditional `break`'s. So, we expect the loop variable continuation test to show up in the loop command in the decompiled code as well. However, we found different results. As shown in Listing 1 and Listing 2, a conditional `break` statement may be transferred from an `if` to the loop command. In this example, the loop does not use a loop variable, so we won't punish the decompiler, but we have also found this when loops do use a loop variable. It is unwanted behaviour, so whenever a `getchar()` (or equivalent) shows up in the continuation test, the decompiler scores no point. Otherwise, we compare the continuation test to the one in the ground truth. The pattern we look for is `<variable_name><operator><numeric>` (where numeric and variable name may be switched). We allow for different numeral notations (hexadecimal, octal, decimal, float; 'F', 'L' and 'U' suffixes).

j) There are no inexcusable `goto`'s: A loop in C results in jumps in the binary. It is the decompiler's job to translate these back to loop constructions and to avoid `goto` statements. We allow the presence of excusable `goto`'s in loops, but any other `goto`'s found will result in not scoring on this subscore.

We end by making some general remarks on the assessing process.

The LLVM-IR code is analysed to determine which loops are included in the binary. They can be distinguished by their unique loop ID's, which are used in every code marker connected to that loop. By analysing the (number of) body-markers for this loop, we can determine whether the loop is unrolled or preserved as a loop. We know the loops' properties by the code markers' information. This mechanism ignores all loops that might be included by other features or by external libraries.

Whenever a loop is encountered in the decompiled C code, we use the loop ID from the code markers to identify the loop, matching the decompiled C code to the C source code and our information on the source code (preserved in the before-marker).

IV. Data structures

We find it interesting to test the recognition of data types because a lot of the semantics of C code are contained in them. Consider, for example, the multiplication expression `a * b`. The meaning of this depends on the types of a and b. If they are both of type `int`, the multiplication can be implemented in x86 by using the `mul` instruction. If, however, a and b are of type float, the relevant instruction is `mulss`. From this, we see that testing datatype recognition is an indication of how well the decompiler understands the instructions.

In general, data types are never directly expressed in machine code, only indirectly. All that is left of data types after compilation is an implementation of the data type semantics. The compiler is free to choose an implementation that matches the program behavior specified by the C standard. For some operations, such as the multiplication mentioned above, there may exist specific hardware instructions. For other operations, such as modular arithmetic, or accessing an element of an array, multiple instructions are required. It would be interesting to know if a decompiler can recognize the idioms used to implement common operations.

The categories of data types we test

The data types we test can be divided into four categories:

- **built-in types**, such as `int` and `float`. These are part of the C language, and they are often supported by hardware, in the form of instructions for operations on these types (for example, the `mul` instruction). A decompiler must be able to recognize these types to generate valid C code, because all other types in C are defined in terms of the built-in types;
- **pointers**. A pointer is essentially a regular integer, with the special meaning in C that its value is a memory address. What we test in this case is whether the decompiler can recognize the type of data stored at that memory address;
- **`struct`'s**. A `struct` is a composite type, consisting of one or more instances of other types, for example, two `int`'s and one `float`. An instance of such a `struct` can also be decompiled as three separate variables, but for readability, it is important that decompilers can recognize `struct`'s, especially because they are used very often. There may also be slightly different machine code involved. For example, when given a pointer to a `struct`, accessing a member is done by incrementing the pointer with a fixed member-specific offset. A decompiler may recognize such pointer arithmetic as typical for `struct`'s;
- **arrays**. An array is similar to a `struct` in the sense that it is a composite type, except that it contains only instances of one specific element type, and arrays may have a size determined at runtime. Element addresses are also calculated with pointer arithmetic.

Besides these categories, we also distinguish between global and local scope, because global variables are used differently in machine code compared to local variables. Global variables have fixed memory addresses that are stored in the binary [VOR20], whereas local variables only have addresses relative to the stack pointer. Because of this, there may be a difference between how well a decompiler recognizes data types in those scopes.

It is possible to create all kinds of combinations of type categories, such as an array of `struct`'s, `struct`'s containing pointers, *etc*. We limit ourselves to a specific subset of these combinations that we deem the most important, for two reasons. One is that, as it turns out, decompilers already have a hard time recognizing the categories, so testing complex combinations currently does not yield interesting results. Secondly, we assume that performance on the most important combinations, that we go into below, is indicative of performance on more exotic combinations.

What we test is the following:

- **built-in types**: all of them;
- **`struct`'s**: containing up to five members of built-in type;
- **arrays**: compile time or runtime size, containing `struct`'s or built-in types;
- **pointers**: to any of the previously mentioned types.

We specifically choose to test pointers to arrays, because those often occur in real programs. Every time a program needs a large amount of memory, or simply an amount that is not known at compile time, heap memory is used. Arrays are therefore often allocated on the heap, among which arrays of `struct`'s.

Implementation of test cases

All test cases heavily lean on the use of code markers. The code markers used for data type testing have the following form:

`__CM_printf_ptr("metadata", &var_name);`

The idea behind this is that a decompiler, hopefully, decompiles this to a similar expression that contains the name of a variable. This reveals the name that the decompiler has chosen for the variable. Using its name, we look up the declaration and compare the type that the decompiler has chosen to the original. The original type can be derived from the metadata included in the code marker.

Every test case in the original C code consists of a variable declaration, usage of the variable and a code marker.

The usage is very important, because usage determines the instructions emitted by the compiler, and thus the information the decompiler has to determine the type. Our framework contains a few usage patterns that are randomly chosen. For the built-in types, this comes down to the usage of operators. For `struct`'s, the usage patterns for built-ins are applied to each of the members. For arrays, there are two array-specific usage patterns: looping over all elements in order, and repeated usage of a random index. Within those loops, the same usage patterns are used on the array elements.

```
void __CM_use_memory(void* ptr, unsigned size){
 fwrite(ptr, size, 1, stdout);
 fread(ptr, size, 1, stdin);
}
```

Listing 3. Function that limits compiler assumptions

However, only using a variable is not sufficient. If the usage has a predictable outcome, a compiler may hard code the outcome instead of emitting instructions, reducing the information a decompiler has. To prevent this, we define the function `__CM_use_memory` as follows in Listing 3.

The typical usage of this function is `__CM_use_memory(&var_name, sizeof(var_name))`. This writes the current value of the variable to standard output and reads a new value from standard input. This has three effects, which give sufficient control over optimization:

- The compiler is forced to calculate the value of the variable because the value is sent to the outside world;
- After the function returns, the compiler can make no assumptions about the value of the variable, as the value comes from the outside world;
- The compiler is forced to actually create the variable, in the sense that it allocates memory for it. Without external usage of the memory address, there is no such guarantee, because compilers are free to optimize as long as the program behavior is according to the C standard.

Every usage pattern starts with a call to `__CM_use_memory` to guarantee that the usage is not optimized away.

Assessment

The assessor assesses decompiled code in three stages.

- finding the code markers;
- looking up the type of the variable referenced in each code marker;
- comparing the found types to the original types.

The second stage is the most complicated, due to the usage of names. For example, the declaration `typename t;` alone contains no information about the type of t. It must first be known what the name ”typename” means. Names may be defined in terms of other names, so it is necessary to follow those definitions until definitions containing only built-in types remain.

To ease comparing types, all types are reduced to a normal form with basic built-in types. For example, say there is a `typedef` in scope that defines `myint` as an alternative name for `int`. The `struct` declaration `struct S { myint i,j; }` is then converted to the normal form `struct { int; int; }`, which means a `struct` containing two `int`'s. Note that the normal form is close to, but is not, valid C code.

Assessment is done by assigning, to each pair of an original type and a decompiled type, both in normal form, a similarity score. This score is a weighted average of some sub-scores, that may recursively use the same score function. The score is a number between 0 and 10. It is always 10 when both normal forms are exactly the same. The sub-scores used per category are:

- built-in integer types: the size and signedness;
- built-in floating point types: the size;
- arrays: the size and element type;
- `struct`´s: average of the score per member;
- pointers: the type it refers to.

If a decompiler detects the wrong category, it may still be given points if the result is partially correct. For example, if a pointer is expected, and the decompiled type is an integer, that is partially correct because pointers are integers. This results in a lower score for the decompiler. The same goes for a `struct` containing members that all have the same type, which is detected as an array.

For `struct`'s, members are compared based on their offset in memory compared to the first bit. This means that the member at offset x in the expected `struct`, is compared to the member at offset x in the decompiled `struct`.

`union`'s are also supported as decompiled types. We do not generate them ourselves, but if emitted by a decompiler, the assigned score is the average over the scores per member of the union. This allows decompilers to provide multiple alternatives when the right type is not known.

V. Function identification

When benchmarking a decompiler, function identification is a fundamental thing to look at. A decompiler might use this to distinguish between code and data in the binary. In this section, we show how deb'm scores decompilers on their ability to identify function declarations and function calls. We first mention the challenges for a decompiler regarding this topic and then present our approach to score a decompiler on this.

Challenges of function identification

For a decompiler, function identification is a major hurdle to take. We highlight some important challenges for decompilers regarding this point: function boundaries, unreachable functions, variadic functions and function calls.

One of the most important challenges of function identification is finding its boundaries. This means: finding the address in the binary where the function starts, and where it ends. For an optimized binary, this can be difficult. According to Harris and Miller, when two functions are much like each other, a compiler may choose to share some of their code [HM05]. The matching part is moved somewhere else in the binary so that both functions can branch to it. This makes it difficult for a decompiler since the function is no longer constrained between its start and end address. Another challenging situation for a decompiler is when a function has a conditional return statement before the end of the function. These statements can be hard to distinguish from the actual function end, because the actual function also ends with a return statement.

Unreachable functions form another challenge. To identify functions, some decompilers start at the program's entry point and discover functions from there, using the jump statements. However, there may exist some unreachable functions in the binary, that might be useful to be aware of as a security analyst. To find these functions, a decompiler can not rely on jumps to that function. Instead, it could try to recognize the function prologues specific to the processor architecture.

In C, some functions do not have a fixed amount of parameters. These functions can be called passing in a variable amount of arguments, so-called variadic functions. The function fetches its arguments dynamically. Decompilers might not notice this[1] since this dynamic fetching with `__builtin_va_arg()` produces machine code that may look somewhat like the fetching of arguments of normal functions. This is less of a challenge in x64, where fetching of variadic arguments is more complex, so more easily recognizable.

To check whether this is a real challenge for decompilers, we did a quick test with Ghidra, RetDec and Hex-Rays (Listing 4). Only Hex-Rays correctly reproduced the variadic function in some cases, so this appears to be a challenge.

```c
//Ground truth
int variadicFunction(int x, ...){
  va_list va;
  va_start(va, p1);
  double y = __builtin_va_arg(va, double);
  va_end(va);
  ...
}

//Ghidra decompilation result
int FUN_140001030(int param_1,undefined8 param_2){
  ...
}

//RetDec decompilation result
int64_t function_140001030(int64_t a1, int64_t a2){
  ...
}

//HexRays decompilation result
__int16 __cdecl variadicFunction(int a1, ...){
  ...
}
```

Listing 4. Variadic function and its decompiled versions

Calling a function is such a natural thing in programming languages, that it is one operation in the x86(-64) instruction sets. This should make it easy for a decompiler to recognize them. However, for function calls, compilers often use something called a jump table. This is an array of function pointers. When branching to a function, the pointer is accessed through its index in the jump table. The usage of jump tables forces a decompiler to be able to find and interpret them. There are, among others, two main reasons to use jump tables.

For constrained systems, it can be slightly more memory efficient. An if-else-if-statement can sometimes be made more efficient using a jump table, see (Listing 5) More commonly, jump tables are used for dynamic library loading. In this case, the external functions are all listed in one array of function pointers. At runtime, the libraries are queried and the correct

[1]We found an issue reporting this behavior in Ghidra that is still open at the time of writing (2024/05/09): https://github.com/NationalSecurityAgency/ghidra/issues/4170

```c
    int value = 2;
    if(value == 0){
      function1();
    }else if(value == 1){
      function2();
    }

    //More efficient: using a jump table
    void (*jmpTable[2]) () = {function1, function2};
    jmpTable[value - 1]();
```

Listing 5. C Jump table versus if-else-if

function addresses can easily be written to the jump table, leaving the rest of the code untouched.

It gets even more difficult when a program branches to a computed pointer, a so-called indirection. This indirection can have multiple layers, implemented by pointers to pointers. In this case, the decompiler needs to have a thorough understanding of the code to resolve which function is actually called.

Our approach

We present how deb'm scores the decompiler on function identification, function start address, return statements, unreachable functions, variadic functions and function calls.

We need to point out on forehand that, for function identification, we do not score optimized binaries. We shortly mention the arguments for this here, they are each explained in the following paragraphs, where we explain the various metrics. In short, the arguments are:

1) Functions might be omitted by inlining them;
2) Start code markers are not guaranteed to be at the exact start of a function;
3) Variadicity in x86 might be optimized away.

Scoring function identification comes with a challenge. To do this, we need to match ground truth functions with decompiled functions. However, function names are not necessarily compiled in or alongside the binary. The solution to this problem is to use the aforementioned code markers. The deb'm producer will start every function with a code marker including the function name. The matching decompiled function is the one that contains this start code marker. This way, we can link every ground truth function to its decompiled counterpart. The score is the recall of all functions in all binaries.

For this approach, compiler optimization would mean a problem. Functions could be inlined in the caller function, making it impossible for the decompiler to find the callee function (argument 1), which impacts the fairness of our benchmark. Is is true that deb'm allows functions to be preserved with optimization by defining them in a separately compiled C file. However, for this to work perfectly, every function would need to be put in a separate file. Otherwise two functions in the same file can possibly call each other, which may lead to the unwanted inlining.

When a function is found, we want to check more specifically whether the decompiler found it at the correct address in the binary. Some decompilers mention the found address in the function name. Others put a comment with the start address just above the function definition or declaration. Some decompilers, like BinaryNinja, completely leave out this information. This makes it hard for deb'm to check the function start address, since we require no more information from the decompiler than the bare C code. We come up with a pragmatic, generic solution, by checking whether the start code marker from the ground truth is the first statement in the decompiled code as well. The trade off is that, using this approach, we can not score optimized binaries, because the start code marker is then no longer guaranteed to be at the exact start of the function body (argument 2).

In practice however, decompilers rarely put the start marker at the exact beginning, because of the nature of a function start in the binary. A compiler automatically adds code before the execution of the first body statement. We call these instructions the function's prologue. A part of this prologue can be decompiled to C code, resulting in statements before the start marker. Because of this, it would not be fair to rigidly check whether the start marker is the first statement. Instead, as long as these statements can be explained as belonging to the function's prologue, deb'm concludes that the function start address is correctly found.

Let us consider an example. The ground truth is listed in Listing 6.

```c
int func_1(int a, int b, int c){
    __CM_printf("marker");
    ...
}
```

Listing 6. Example C code

The decompiled version of this code is shown in the left column of Table I, the x64 assembly is shown in the right column.

For every statement before the start marker, deb'm tries to explain it with one or more corresponding statements from the assembly function prologue.

TABLE I

	Ghidra result	Assembly
1		`_func_1:`
2		`.seh_proc func_1`
3		`pushq %rsi`
4		`.seh_pushreg %rsi`
5	`type4 uVar1;`	**`subq $112, %rsp`**
6	`type8 uVar2;`	**`.seh_stackalloc 112`**
7	`type4 *puVar3;`	
8	`type4 uVar4;`	
9	`type4 uVar5;`	
10	`type4 local_50 [4];`	
11	`type4 local_40 [4];`	
12	`type4 local_30 [4];`	
13	`type4 local_20;`	
14	`type4 local_1c;`	
15	`type4 *local_18;`	
16		`.seh_endprologue`
17		`movq %rcx, %rsi`
18	`local_18 = param_1;`	**`movq %rsi, 88(%rsp)`**
19	`local_1c = param_3;`	**`movq %rdx, 80(%rsp)`**
20	`local_20 = param_2;`	**`movl %r8d, 44(%rsp)`**
21		`movb %r9b, 43(%rsp)`
22	`__CM_printf("marker");`	

The first 11 statements of the Ghidra result, all variable declarations, can be explained by the stack allocation in the assembly. The needed space for the local variables is allocated all at once, in this case 112 bytes. Since we can not verify the data type sizes, we allow Ghidra to produce a maximum of 112 variable declarations for this.

The three variable assignments can be explained from the assembly by what is called register homing. Register homing occurs in x64, where the first four arguments are passed in the first four registers. To be able to use these registers later in the function, the compiler places their values on the stack. deb'm counts every first move operation of the four argument registers to the stack as a homing operation. The argument registers are rcx, rdx, r8 and r9. Since rcx is moved to rsi, and rsi to the stack, deb'm marks this one as a homing operation as well. This results in four homing operations. The decompiler should not produce more variable assignments than there are register homing operations. Ghidra has three of them, so these assignments are all explainable.

There is one more possible explanation for statements before the start marker: unexplainable instructions in the assembly prologue. For each of these instructions, we allow the decompiler to produce one statement as well, marking those as explainable. Unexplainable assembly instructions are those that do not belong to a normal function prologue. deb'm marks the following type of instructions as normal prologue statements:

- Line 1: Function label
- Line 2, 4, 6 and 16: Structured Exception Handling - This kind of operation is used for debugging in the x64 architecture.
- Line 3: Non-volatile register push - Some registers in the x64 architecture are marked non-volatile, which means their value should be preserved over subroutine calls. To make sure the register value does not change, some are temporarily saved on the stack.
- Line 5: Stack allocation
- Line 17: Register moving - Moves from register to register.
- Line 18 - 21: Register homing
- Not in this listing: x86 explicitly pushes the base pointer and sets it to the current stack pointer.

We have mentioned three possible explanations of C statements before the start marker. If there is an unexplained statement, deb'm concludes that the decompiled function starts at a different address than the ground truth function. In this example, all statements are explainable, so deb'm concludes that Ghidra has correctly identified the function start address. The final score is the recall of all correctly identified function starts of all identified functions in all binaries.

The next thing to score the decompiler on is its ability to find return statements. This is relevant for function identification because an intermediate return statement could be interpreted by the decompiler as a function end. In that case, it missed the actual end of the function. The score is the recall of all return statements in all binaries.

To check the recognition of unreachable functions, we made sure to create at least 10 functions per binary that do not get called. As soon as one code marker is found with this function name, we know that the decompiler has found it. Note that not decompiling an unreachable function is not necessarily a bad thing, because it does not impact the behavior of the program. The score is the recall of all unreachable functions in all binaries. It does not say how good the decompiler is, but it serves as an objective metric.

In variadic functions, as we said, the machine code translation of `__builtin_va_arg()` may look like the fetching of normal function arguments, especially in x86. Therefore, producing a variadic function, we give the decompiler a fair chance to recognize it as such. We do this by preceding the argument fetching code with the function code marker. Seeing this, the decompiler can conclude that this fetching is not part of the normal function prologue. To give the decompiler another hint, we make sure that every variadic function is called more than once, with a variable amount of arguments. To check whether a function is decompiled as variadic, we look for the ellipsis at the end of the parameter list. The comparison is simple: when the source function is variadic, the decompiled version should also be variadic. The score is the recall of all variadic functions in all binaries. Scoring a decompiler on recognition of variadicity is impossible on optimized binaries, because in x86, the compiler might simply replace the variadic list with the needed amount of parameters (argument 3).

We check function calls using the code marker based mapping between decompiled and compiled function names. We could check this for every single function call, by adding a code marker just before every call, or even doing the call in the code marker itself. However, since the code can be optimized slightly, even with flag -O0, we can not guarantee that every single function call will be easily identifiable. Instead, for every function x and y, we count the number of calls from x to y in the ground truth. This counting is also done in the decompiled code so that we can compare the results. This way, we do not know for every specific call if it was decompiled. Instead, when *n* calls are missing in the decompiled code, we simply mark the last *n* original function calls as not identified. The score is the F1-score of all function calls in all binaries. The F1-score is the harmonic mean of the recall and the precision.

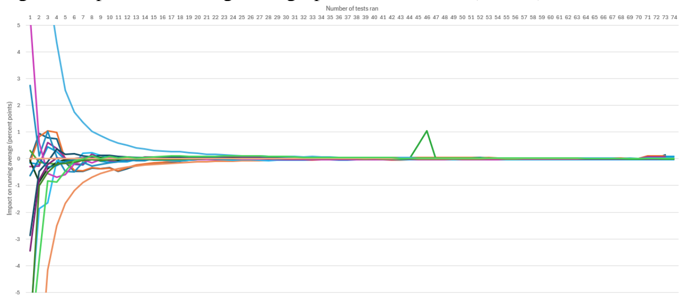
VI. Validation

To validate the correct working of deb'm, we ran five full assessments on Ghidra, RetDec and Hex-Rays. One assessment means invoking the decompiler for $75 \times 4 = 300$ binaries, and comparing the output with our ground truth. Seeing the results, we can point out three things to argue that deb'm produces valid and consistent test results.

1) *The number of tests.* How many binaries must we feed into the decompiler to have a good and stable result? To know this, we tested with 25, 50 and 75 sources, each compiled to 4 binaries, per container. After every

four binaries, we plotted the impact of each added source on the running average. When feeding more binaries no longer affects the average, we have a stable result.

Fig. 2. Impact on running average per extra test run (Ghidra)



We see that during the first 20 tests, the average fluctuates heavily. Between 20 and 50 there is still some fluctuation, even a peek, so to be sure we put 75 tests in each test container. Figure 2 shows the values of Ghidra, the other decompilers show similar values.

2) *Standard deviation.* For every test aspect of every feature class, we calculated the standard deviation of the score over the five assessed containers. These standard deviations are calculated in percentage points, as our metrics are percentages. We calculate it for each combination of metric and decompiler. This results in 78 computed standard deviations, of which 12 are above 1, 6 are above 2 and 1 is above 3. The average standard deviations for Ghidra, RetDec and Hex-Rays are respectively 0.43, 0.50, and 0.89. Given these low averages, we consider the results consistent enough.
3) *Clear winner is (nearly) always a clear winner.* Variation is natural, and running deb'm on one container will never produce the exact same results as on another container. Assessing 5 containers, we compared 3 decompilers on 26 metrics, yielding a total of 390 comparisons. In two cases, RetDec wins from Hex-Rays in one container, while losing in the other. One case is on loop quality scores for unrolled loops. The difference is only 0.55 percentage points between the win and the loss. We do not consider this a significant amount. The other case is the metric for integers type accuracy. This difference is 5.27 percentage points. This is much higher than we would like it to be, but due to time limits, we are at the moment unable to explain this and further research is indicated. However, we think this one inexplicable exception is not enough to overthrow our conclusion that the containers are sufficiently comparable.

VII. Results

After producing all containers, we ran deb'm on container_000, using RetDec[2], Ghidra[3], and Hex-Rays[4]. A sample output, the full HTML-output for container_000, run with Hex-Rays, is shown in Table V in the appendix.

[2]RetDec 5.0, running locally on Windows

[3]Ghidra 11.0, running locally on Windows.

[4]Hex-Rays 8.4.0.240320, hosted online by dogbolt.org.

TABLE II
Control flow metrics

loops found (0%-100%)	**Ghidra**	**RetDec**	**Hex-Rays**
in non-optimized binaries	96.89	60.42	55.62
in optimized binaries	61.62	44.14	34.17
loop quality score (0%-100%)	**Ghidra**	**RetDec**	**Hex-Rays**
in non-optimized binaries	91.28	34.62	56.06
in optimized binaries	48.31	25.81	28.94
total number of goto's emitted (#)	**Ghidra**	**Ret-Rec**	**Hex-Rays**
in non-optimized binaries	12,657	41,007	12,185
in optimized binaries	13,538	42,393	13,204
total number of inexcusable `goto`'s ... emitted (#)	**Ghidra**	**Ret-Rec**	**Hex-Rays**
in non-optimized binaries	9,852	40,053	10,497
in optimized binaries	12,898	41,727	12,821

RetDec did not succeed in decompiling all the binaries; two binaries yielded no results. We do not know why. We discovered that the size of the binaries is a concern, though. During development, RetDec and Ghidra failed when we presented binaries that were too big. They either crashed or their memory usage exploded. As we were more interested in analyzing what was actually produced by the decompilers, we decided to keep the sources - and thus the binaries - small enough. Further research could be done on this subject, but this was out of our scope.

Control flow

The most important metrics are listed in Table II. These metrics are aggregated over CPU architecture. As none of the decompilers was able to re-roll unrolled loops, the table only shows the numbers for preserved loops. Note that the assessor's output, as shown in the appendix, does show a positive loop quality score for unrolled loops. This is because a loop scores its first point not based on the presence of a loop command but on the presence of its before-marker.

For every metric, working with non-optimized binaries produces better results than working with optimized ones. None of the decompilers 'wins' all metrics, underlining the importance of comparing decompilers on multiple metrics.

Ghidra scores best on loop recognition: 96.89% of the loops we put in our binaries are emitted as a loop when non-optimized binaries are analyzed. RetDec and Hex-Rays have significantly lower scores. Ghidra also scores best on the loop quality score, reaching 91.58% for non-optimized binaries.

The total number of `goto`'s shows Hex-Rays doing best, but Ghidra is close by. RetDec is far behind, producing over three times as many `goto`'s as the other two. When excusable `goto`'s are ignored, the results are pretty much the same: Hex-Rays and Ghidra are doing far better than RetDec.

Data structures

The data structure metrics are listed in Table IV.

The results are similar for optimized and unoptimized binaries, but there is a significant difference between architectures. Therefore, we show the results per architecture.

TABLE III
DATA STRUCTURE METRICS

recovered testcases	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	30.82	94.63	15.76
in x86 binaries	57.39	99.91	59.66
Globals detected	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	-	100.00	-
in x86 binaries	-	100.00	-
Locals detected	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	100.00	100.00	100.00
in x86 binaries	99.72	100.00	100.00
Integers detected	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	100.00	100.00	100.00
in x86 binaries	100.00	99.82	63.86
Integers type accuracy	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	100.00	59.27	80.65
in x86 binaries	92.97	42.81	46.74
Floating point numbers detected	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	100.00	0.00	100.00
in x86 binaries	85.33	99.23	50.00
Floating point numbers type accuracy	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	100.00	0.00	100.00
in x86 binaries	85.33	99.23	50.00
Pointers detected	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	54.70	36.07	37.63
in x86 binaries	41.38	33.77	14.97
Pointers type accuracy	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	53.03	39.38	43.43
in x86 binaries	40.43	39.54	19.87
Arrays detected	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	80.80	0.00	100.00
in x86 binaries	83.06	0.00	95.73
Arrays type accuracy	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	59.91	9.05	67.12
in x86 binaries	63.06	8.04	61.63
Structs detected	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	0.00	0.00	0.00
in x86 binaries	0.00	0.00	0.00
Structs type accuracy	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	0.00	0.00	0.00
in x86 binaries	0.54	0.00	4.84

The percentage of recovered test cases indicates how many of the test cases we generated have been assessed. Not all test cases have been assessed because not all code markers could be recovered. In particular, Hex-Rays and Ghidra did not decompile any code markers correctly with x86 optimized binaries, so those have led to no results. The problem is that the second parameter of the code markers is not present, which would normally be the variable. We do not count those as wrong, because without knowing which variable a code marker is for, there really is nothing to assess.

For each type category, there are two kinds of results. The *detected* score is the percentage of assessable cases in which the right type category was detected. The accuracy score is more detailed - it is the type similarity score explained in Section IV.

Ghidra found no global variables at all. Hex-Rays does seem to find some global variables. However, we did not succeed in parsing the corresponding C code. Therefore, those have also led to no results.

Furthermore, none of the three decompilers generated any `struct` types. Still, Ghidra and Hex-Rays have a positive score for `struct` type accuracy, because of `struct`'s that were decompiled as an array, which can be correct in some cases.

Function identification

The results of benchmarking on function identification are listed in Table IV.

TABLE IV
FUNCTION IDENTIFICATION METRICS

Reachable functions	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	100.00	100.00	100.00
in x86 binaries	100.00	96.47	100.00
Function start addresses	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	99.77	90.33	100.00
in x86 binaries	99.98	99.91	100.00
Unwanted prologue statements	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	88.08	72.44	91.58
in x86 binaries	96.68	96.61	96.30
Return statements	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	79.52	78.67	66.37
in x86 binaries	79.60	82.22	81.45
Unreachable functions	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	0.00	0.00	100.00
in x86 binaries	0.00	0.00	100.00
Function calls	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	100.00	89.78	61.73
in x86 binaries	100.00	92.06	100.00
Variadic functions	**Ghidra**	**RetDec**	**Hex-Rays**
in x64 binaries	0.00	0.00	52.04
in x86 binaries	0.00	0.00	0.00

For most metrics, deb'm produces similar scores for all decompilers. We highlight the most remarkable points.

In general, decompilers seem to score better on x86 binaries, especially when it comes to function prologues. This is probably because the instruction set is somewhat simpler.

Finding a normal function is rarely a problem, Hex-Rays and Ghidra do this perfectly.

Ghidra and RetDec actively eliminate unreachable functions, scoring 0 in deb'm. Hex-Rays does not do this by default. We also ran RetDec with the relevant flag turned off, and it scored 100%. Because this is an objective metric, we ran the decompilers with their default configurations.

Only Hex-Rays recognizes functions as variadic, and only in x64 binaries. This is probably because x64 dynamic argument fetching is more complex and thus more easily recognizable.

VIII. Related work

Some studies aim to benchmark decompilers. More often, a decompiler is presented and benchmarked to show its results. From these studies, we highlight the most important ones.

We start with *A Framework for Assessing Decompiler Inference Accuracy of Source-Level Program Constructs* [Kli22] by Kline. In this thesis, Kline aims to test a decompiler on several main source-level program constructs. He translates the ground truth to a domain-specific language (DSL), using the C-code and DWARF debug information. Unlike deb'm, he also relies on information about the decompilation result using the Ghidra Scripting API. After converting this result to the same DSL, it is compared to the ground truth. This is done using four quantitative metrics describing functions, varnodes, data bytes, and array comparisons. Only unoptimized binaries are tested. Although decompilers are mentioned in general, the thesis focuses on Ghidra. Deb'm is more flexible, supporting a range of decompilers and also assessing optimized binaries.

Another decompiler testing framework, D-Helix [ZKW+24] is presented by Zou et al. They test decompilers by recompiling every decompiled function separately and comparing the compiled IR to the lifted IR from the decompiler using symbolic differentiation. Although the design is generic, they estimate that supporting a new decompiler takes around 40 days. It also requires a decompiler to deliver some sort of IR, alongside C code. With deb'm, analyzing a new decompiler only requires writing a wrapper script.

Katz et al. present a decompiler based on deep learning: TraFix [KOGY19]. It automatically learns a decompilation algorithm based on a given compiler. The correctness of TraFix is tested by generating C code from a varying subset of the C language. This code is compiled using Clang [cla07] and GCC [gcc87]. The verification is done by recompiling the decompilation result and comparing the original low-level code with the recompiled version. This approach ensures correctness, but not readability. A recompiled binary may be equal to the compiled one, but the decompiled code may be unreadable. Readability is relevant and therefore, several metrics in deb'm focus on the quality of code instead of functional equivalence.

Coda [FCL+19] is a decompiler presented by Fu et al. It is based on neural networks to do the decompilation task. The binaries used as training data are generated and compiled with Clang [cla07], disabling optimization. Based on the binary, Coda predicts the next token in a higher level PL.

To evaluate Coda, they use binaries and assembly code from real-world applications with various difficulties. The correctness of decompilation is measured by something they call Token Accuracy. This is the percentage of predicted tokens that match the ground truth tokens.

Another relevant decompiler to mention is the *Byte-Equivalent Decompiler (BED)* [SRN+18], made by Schulte et al. This decompiler is compared to the then industry-leading Hex-Rays decompiler [YEGPS15]. They do this comparison using two main metrics: readability (split into 12 metrics) and byte similarity. Readability focuses on the number of `goto`'s, casts, variables, scopes, live ranges, dead assignments, macros, `typedef`'s, characters, lines, abstract syntax trees (ASTs), and matched literals. Byte similarity measures how many of the original instructions are present in the recompiled binary. The problem is that byte-equivalence on the statement level is a boolean value. When, on average, the byte-similarity is 81%, this only means that 81% of the lines are fully correct. The other 19% are only checked for readability but can be complete nonsense.

Harrand et al. also present a decompiler benchmarking platform in their paper *Java Decompiler Diversity* [HSVMB19]. They check decompilers on (1) syntactic correctness, which means the decompiler should emit recompilable code; (2) semantic equivalence modulo input [LAS14] and (3) syntactic similarity. They use real-world code to test decompilers. Using real-world code best reflects decompilers' challenges in practice. For deb'm, however, we chose to have the most grip on the binaries tested.

Another decompiler worth mentioning is the *C-Decompiler* [CQH+13] by Chen et al. They aim to produce highly readable code by recognizing as many library functions as possible. To evaluate the C-Decompiler, they use four evaluation criteria. Firstly, function analysis. They measure the number of true/-false positives/negatives to score the function identification capabilities. Secondly, they measure the variable expansion rate. This is the rate of extra variables compared to the original code. The third criterion focuses on reduction percentage, which relates the number of lines in assembly code to the number of lines in the C code. Finally: Cyclomatic Complexity, which is the number of linearly independent paths through a program's source code. Although they claim to address variable and function identification, they only seem to evaluate the latter.

Shaila et al. present a disassembler called DisCo in their paper *DisCo - Combining Disassemblers for Improved Performance*[SDF+21]. They combine the results of multiple existing decompilers to create a powerful disassembler. As a ground truth, they took 88 IoT malware programs from Github. The binaries they fed into the disassemblers were all stripped and non-obfuscated. Their primary metric for evaluating DisCo is the CFS; the rate of correctly identified function starts.

Liu et al. have created the decompiler correctness tester, presented in *Testing Decompilation Correctness of C Decompilers* [LW20]. They use CSmith [CSm] to generate source code, and compile it with GCC. They recognize that decompiled code is generally not recompilable. To be able to measure the semantic correctness, they try to extract separate functions from the result and compile them in a new source with only a main function and the extracted function. This way, parts of the decompilation results are recompilable.

IX. Conclusion

We have shown that it is possible to create a set of binaries that can be used as a benchmark. We have also shown that it is possible to analyze decompiler output in an automated way and produce multiple metrics that are useful in the comparison of decompiler quality. Our platform uses multiple metrics as we intended and we succeeded in making it truly versatile and expandable.

We have used our platform to compare three decompilers; two of them ran offline and one online. We have shown the differences in their qualities. It was also verified that it does not matter which of our containers is used: for every metric, a similar score was given.

We found that decompilers handled non-optimized binaries better than optimized ones. We also found that no decompiler 'won' all the metrics, underlining the importance of comparing decompilers on multiple metrics. However, on many metrics, Ghidra turned out to be the better decompiler.

We set the framework basics of deb'm and implemented three feature classes. But we consider it worthwhile expanding deb'm. In the future, expansions like these could be considered:

- adding new metrics, scoring decompilers on things like:
  - the quality of mathematical expressions;
  - the ability to recognize library functions;
  - the ability to discard all code automatically added by the compiler, such as all code executed before `int main();`
  - the amount of assembly emitted in the decompiler output;
  - the readability of the decompiler output;
  - the function equivalence of the ground truth and the decompilation result;
- adding more compilers to the producer;
- adding more CPU architectures to the producer (such as ARM);
- producing binaries for different operating systems.

References

[ASH13] Saurabh Amin, Galina A. Schwartz, and Alefiya Hussain. In quest of benchmarking security risks to cyber-physical systems. *IEEE Network*, 27(1):19–24, 2013.

[cla07] clang: a c language family frontend for llvm. 2007. https://clang.llvm.org/.

[CQH+13] Gengbiao Chen, Zhengwei Qi, Shiqiu Huang, Kangqi Ni, Yudi Zheng, Walter Binder, and Haibing Guan. A refined decompiler to generate c code with high readability. *Software: Practice and Experience*, 43(11):1337–1358, 2013.

[CSm] CSmith. https://github.com/csmith-project/csmith.

[FCL+19] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32, 2019.

[gcc87] Gcc, the gnu compiler collection. 1987. https://gcc.gnu.org/.

[HBJS18] Yasir Hamid, Veeran Ranganathan Balasaraswathi, Ludovic Journaux, and Muthukumarasamy Sugumaran. Benchmark datasets for network intrusion detection: A review. *Int. J. Netw. Secur.*, 20(4):645–654, 2018.

[HM80] R.N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[HM05] Laune Harris and Barton Miller. Practical analysis of stripped binary code. *SIGARCH Computer Architecture News*, 33:63–68, 12 2005.

[HSVMB19] Nicolas Harrand, César Soto-Valero, Martin Monperrus, and Benoit Baudry. Java decompiler diversity and its application to meta-decompilation. In *2019 19th International working conference on source code analysis and manipulation (SCAM)*, pages 92–102. IEEE, 2019.

[IDA] Pro IDA. https://hex-rays.com/IDA-pro/.

[JEB] JEB. https://www.pnfsoftware.com/jeb/.

[Kli22] Jace Kline. *A Framework for Assessing Decompiler Inference Accuracy of Source-Level Program Constructs*. PhD thesis, University of Kansas, 2022.

[KOGY19] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall P T R, 33rd printing edition, 1988.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[LW20] Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487, 2020.

[Nin] Binary Ninja. https://binary.ninja.

[NSA] NSA. https://ghidra-sre.org/.

[PQ95] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[Ret] RetDec. https://github.com/avast/retdec.

[SDF+21] Sri Shaila, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. Disco: Combining disassemblers for improved performance. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 148–161, New York, NY, USA, 2021. Association for Computing Machinery.

[Sma] SmartDec. https://github.com/smartdec/smartdec.

[SRN+18] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[Ti] Tiobe-index. https://www.tiobe.com/tiobe-index/.

[VM05] M. Vieira and H. Madeira. Towards a security benchmark for database management systems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 592–601, 2005.

[VOR20] F. Verbeek, P. Olivier, and B. Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *18th International Conference on Software Engineering and Formal Methods (SEFM 2020)*, 2020.

[WF06] David A. Watt and William Findlay. *Programming language design comcepts*. John Wiley & Sons, Ltd, 2006.

[YEGPS15] Khaled Yakdan, Sebastien Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. 2015. http://dx.doi.org/10.14722/ndss.2015.23185.

[ZKW+24] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. D-helix: A generic decompiler testing framework using symbolic differentiation. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

Appendix

TABLE V
HTML output for container 0, run with Hex-Rays.

Description	Value
Decompiler name	hexrays

Description (unit)	Architecture	Compiler	Optimization	Min score	Actual score	Max score	Target score	% min/max	# tests
Number of loops found as loop - all loops (#)	x64	clang	N	0	528	3,250	3,250	**16.25**	75
	x64	clang	Y	0	278	3,218	3,218	**8.64**	75
	x86	clang	N	0	3,087	3,250	3,250	**94.98**	75
	x86	clang	Y	0	1,674	3,218	3,218	**52.02**	75
Number of loops found as loop - only preserved loops (#)	x64	clang	N	0	528	3,250	3,250	**16.25**	75
	x64	clang	Y	0	278	2,856	2,856	**9.73**	73
	x86	clang	N	0	3,087	3,250	3,250	**94.98**	75
	x86	clang	Y	0	1,674	2,856	2,856	**58.61**	73
Number of loops found as loop - only unrolled loops (#)	x64	clang	Y	0	0	362	362	**0.00**	63
	x86	clang	Y	0	0	362	362	**0.00**	63
Loop quality score - all loops (school mark)	x64	clang	N	0.0	2.1	10.0	10.0	**21.00**	75
	x64	clang	Y	0.0	0.9	10.0	10.0	**9.56**	75
	x86	clang	N	0.0	9.1	10.0	10.0	**91.13**	75
	x86	clang	Y	0.0	3.9	10.0	10.0	**39.94**	75
Loop quality score - preserved loops (school mark)	x64	clang	N	0.0	2.1	10.0	10.0	**21.00**	75
	x64	clang	Y	0.0	1.1	10.0	10.0	**11.11**	73
	x86	clang	N	0.0	9.1	10.0	10.0	**91.13**	75
	x86	clang	Y	0.0	4.6	10.0	10.0	**46.77**	73
Loop quality score - unrolled loops (school mark)	x64	clang	Y	0.0	0.3	10.0	10.0	**3.66**	63
	x86	clang	Y	0.0	1.0	10.0	10.0	**10.00**	63
Total number of goto's found (#)	x64	clang	N	0	5,019		0		75
	x64	clang	Y	0	5,180		0		75
	x86	clang	N	0	7,166		0		75
	x86	clang	Y	0	8,024		0		75
Total number of inexcusable goto's found (#)	x64	clang	N	0	4,822		0		75
	x64	clang	Y	0	5,139		0		75
	x86	clang	N	0	5,675		0		75
	x86	clang	Y	0	7,682		0		75
Data structures: recovered testcases ()	x64	clang	N	0	680	2,158	2,158	**31.51**	75
	x64	clang	Y	0	0	2,158	2,158	**0.00**	75
	x86	clang	N	0	1,311	2,158	2,158	**60.75**	75
	x86	clang	Y	0	1,264	2,158	2,158	**58.57**	75
Data structures: Globals detected ()	x64	clang	N	0	0	0	0	**100.00**	75
	x64	clang	Y	0	0	0	0	**100.00**	75
	x86	clang	N	0	0	0	0	**100.00**	75
	x86	clang	Y	0	0	0	0	**100.00**	75
Data structures: Locals detected ()	x64	clang	N	0	680	680	680	**100.00**	75
	x64	clang	Y	0	0	0	0	**100.00**	75
	x86	clang	N	0	1,311	1,311	1,311	**100.00**	75
	x86	clang	Y	0	1,264	1,264	1,264	**100.00**	75
Data structures: Integers detected ()	x64	clang	N	0	93	93	93	**100.00**	75
	x64	clang	Y	0	0	0	0	**100.00**	75
	x86	clang	N	0	112	184	184	**60.87**	75
	x86	clang	Y	0	123	184	184	**66.85**	75
Data structures: Integers type accuracy ()	x64	clang	N	0.0	8.0	10.0	10.0	**80.65**	93
	x86	clang	N	0.0	3.8	10.0	10.0	**38.21**	184
	x86	clang	Y	0.0	5.5	10.0	10.0	**55.27**	184

Data structures: Floating point numbers detected ()	x64	clang	N	0	21	21	21	**100.00**	75
	x64	clang	Y	0	0	0	0	**100.00**	75
	x86	clang	N	0	21	40	40	**52.50**	75
	x86	clang	Y	0	19	40	40	**47.50**	75
Data structures: Floating point numbers type accuracy ()	x64	clang	N	0.0	10.0	10.0	10.0	**100.00**	21
	x86	clang	N	0.0	5.2	10.0	10.0	**52.50**	40
	x86	clang	Y	0.0	4.7	10.0	10.0	**47.50**	40
Data structures: Pointers detected ()	x64	clang	N	0	111	295	295	**37.63**	75
	x64	clang	Y	0	0	0	0	**100.00**	75
	x86	clang	N	0	23	579	579	**3.97**	75
	x86	clang	Y	0	151	583	583	**25.90**	75
Data structures: Pointers type accuracy ()	x64	clang	N	0.0	4.3	10.0	10.0	**43.43**	295
	x86	clang	N	0.0	1.3	10.0	10.0	**13.73**	579
	x86	clang	Y	0.0	2.5	10.0	10.0	**25.97**	583
Data structures: Arrays detected ()	x64	clang	N	0	168	168	168	**100.00**	75
	x64	clang	Y	0	0	0	0	**100.00**	75
	x86	clang	N	0	306	318	318	**96.23**	75
	x86	clang	Y	0	254	267	267	**95.13**	75
Data structures: Arrays type accuracy ()	x64	clang	N	0.0	6.7	10.0	10.0	**67.12**	168
	x86	clang	N	0.0	6.2	10.0	10.0	**62.94**	318
	x86	clang	Y	0.0	6.0	10.0	10.0	**60.06**	267
Data structures: Structs detected ()	x64	clang	N	0	0	103	103	**0.00**	75
	x64	clang	Y	0	0	0	0	**100.00**	75
	x86	clang	N	0	0	190	190	**0.00**	75
	x86	clang	Y	0	0	190	190	**0.00**	75
Data structures: Structs type accuracy ()	x64	clang	N	0.0	0.0	10.0	10.0	**0.00**	103
	x86	clang	N	0.0	0.4	10.0	10.0	**4.69**	190
	x86	clang	Y	0.0	0.5	10.0	10.0	**5.00**	190
Reachable functions (recall)	x64	clang	N	0.000	1.000	1.000	1.000	**100.00**	5,998
	x86	clang	N	0.000	1.000	1.000	1.000	**100.00**	5,998
Function start addresses (recall)	x64	clang	N	0.000	1.000	1.000	1.000	**100.00**	5,998
	x86	clang	N	0.000	1.000	1.000	1.000	**100.00**	5,998
Unwanted prologue statements (%)	x64	clang	N	0.0	5,493.1	5,998.0	5,998.0	**91.58**	5,998
	x86	clang	N	0.0	5,776.0	5,998.0	5,998.0	**96.30**	5,998
Return statements (recall)	x64	clang	N	0.0	3,221.0	4,853.0	4,853.0	**66.37**	5,998
	x86	clang	N	0.0	5,753.0	4,853.0	4,853.0	**81.45**	5,998
Unreachable functions (recall)	x64	clang	N	0.000	1.000	1.000	1.000	**100.00**	750
	x86	clang	N	0.000	1.000	1.000	1.000	**100.00**	750
Function calls (F1-score)	x64	clang	N	0.000	0.617	1.000	1.000	**61.73**	8,069
	x86	clang	N	0.000	1.000	1.000	1.000	**100.00**	8,069
Variadic functions (F1-score)	x64	clang	N	0.000	0.520	1.000	1.000	**52.04**	5,998
	x86	clang	N	0.000	0.000	1.000	1.000	**0.00**	5,998
Number of decompiled files produced (#)	x64	clang	N	0	75	75	75	**100.00**	75
	x64	clang	Y	0	75	75	75	**100.00**	75
	x86	clang	N	0	75	75	75	**100.00**	75
	x86	clang	Y	0	75	75	75	**100.00**	75
Number of parser crashes (#)	x64	clang	N	0	0	75	0	**100.00**	75
	x64	clang	Y	0	0	75	0	**100.00**	75
	x86	clang	N	0	0	75	0	**100.00**	75
	x86	clang	Y	0	0	75	0	**100.00**	75
Parser errors (errors per total number of lines)	x64	clang	N	0	6,417	563,269	0	**98.86**	75
	x64	clang	Y	0	5,100	590,968	0	**99.14**	75
	x86	clang	N	0	3,525	773,073	0	**99.54**	75
	x86	clang	Y	0	3,525	924,734	0	**99.62**	75

No aggregation over compiler configuration has taken place.

Most metrics' target scores are the upper bounds, but some are the lower bounds, such as the number of unwanted goto's. As the actual values are quite far from the target, the percentage is low, indicating a bad score.

The last three metrics are not discussed in the main paper as they are only very rough indicators and fairly self-explanatory. We noticed that sometimes a decompiler isn't able to produce any output and considered it useful to report. The same goes for parses crashes (ANTLR hardly crashes, but we have seen it happen occasionally) or generated parser errors (which happens from time to time). These latter two metrics are an indication of the general quality of the decompiled code, measuring how well they conform to the C grammar.