# Lecture Notes on Decompilation

15411: Compiler Design
Maxime Serrano

Lecture 20
October 31, 2013

## 1 Introduction

In this lecture, we consider the problem of doing compilation "backwards" - that is, transforming from a compiled binary into a reasonable representation of its original source. Solving this problem will involve significant consideration of our standard dataflow analyses, as well as a discussion of good selection of internal representations of code.

While the motivation for the existence of compilers is fairly clear, the motivation for the existence of *decompilers* is less so. However, in the modern world there exist *many* legacy systems for which the original source code has been lost, which need bugs fixed in them or to be ported to a more modern architecture. Decompilers facilitate this process greatly. In addition, in malware analysis, generally source is not provided. It is therefore extremely useful to have some way to go from binary to a reasonable approximation of the original code.

## 2 Steps of Decompilation

Roughly, decompilation follows a few steps:

1. Disassembly - transformation from machine code to the assembly equivalent. There are a surprising number of pitfalls here.

2. Lifting and dataflow analysis - transforming the resulting assembly code into a higher-level internal representation, such as our three-operand assembly. One of the tricky parts here is recognizing distinct variables, and detaching variables from registers or addresses. We also recover expressions, function return values and arguments.

3. Control flow analysis - recovering control flow structure information, such as if and while statements, as well as their nesting level.

4. Type analysis - recovering types of variables, functions, and other pieces of data.

# 3 Disassembly

The first step of writing a good decompiler is writing a good disassembler. While the details of individual disassemblers can be extremely complex, the general idea is fairly simple. The mapping between assembly and machine code is in theory one-to-one, so a straight-line translation should be feasible.

However, disassemblers rapidly run into a problem: it is *very difficult* to reliably distinguish code from data.

In order to do so, generally disassemblers will take one of two strategies:

1. Disassemble the sections that are generally filled with code (`.plt`, `.text`, some others) and treat the rest of them as data. One tool that follows this strategy is `objdump`. While this works decently well on code produced by most modern compilers, there exist (or existed!) compilers that place data into these executable sections, causing the disassembler some confusion. Further, any confusingly-aligned instructions will also confuse these disassemblers.

2. Consider the starting address given by the binary's header, and recursively disassemble all code reachable from that address. This approach is frequently defeated by indirect jumps, though most of the disassemblers that use it have additional heuristics that allow them to deal with this. An example tool that follows this strategy is Hex-Ray's Interactive Disassembler.

While disassembly is a difficult problem with many pitfalls, it is not particularly interesting from an implementation perspective for us. Many program "obfuscators" have many steps that are targeted at fooling disassemblers, however, as without correct disassembly it is impossible to carry on the later steps.

# 4 Lifting and Dataflow Analysis

Given correct disassembly, another problem rears its head. As you may have noticed while writing your compilers, doing any form of reasonable analysis on x86_64 is an exercise in futility. The structure of most assembly language does not lend itself well to any kind of sophisticated analysis.

In order to deal with this, decompilers generally do something which closely resembles a backwards form of instruction selection. However, decompilers cannot just tile sequences of assembly instructions with sequences of abstract instructions, as different compilers may produce radically different assembly for the same sequence of abstract instructions.

Further, frequently a single abstract instruction can expand into a very long sequence of "real" instructions, many of which are optimized away by the compiler later on.

There are two primary approaches to dealing with this issue. The first is to simply translate our complex x86_64 into a simpler RISC instruction set. The

tools produced by Zynamics frequently take this approach. The alternative is to translate into an exactly semantics-preserving, perhaps more complicated, instruction set, which has more cross-platform ways of performing analysis on it. This is the approach taken by CMU's BAP research project, as well as by the Hex-Rays decompiler.

The choice of the internal representation can be very important. For our purposes, we'll consider the 3-operand IR that we've been using throughout the semester.

We will summarize the translation from x86_64 to our IR by simply effectively doing instruction selection in reverse. The difficulty here is generally in the design of the IR, which we most likely do not have the time to discuss in detail. Some places to learn about IRs include the BAP website (`bap.ece.cmu.edu`) and the Zynamics paper "REIL: A platform-independent intermediate representation of disassembled code for static code analysis" by Thomas Dullien and Sebastian Porst.

Once we have obtained an IR, we would now like to eliminate as many details about the underlying machine as possible. This is generally done using a form of dataflow analysis, in order to recover variables, expressions and the straight-line statements.

# 5 Control Flow Analysis

Having reached this stage, we now have a reasonable control flow graph, with "real" variables in it. At this point, we *could* produce C code which is semantically equivalent to the original machine code. However, this is frequently undesirable. Few programs are written with as much abuse of the `goto` keyword as this approach would entail. Most control flow graphs are generated by *structured* programs, using `if`, `for` and `while`. It is then desirable for the decompiler to attempt to recover this original structure and arrive at a fair approximation of the original code.

This form of analysis relies largely on graph transformations. A primary element of this analysis relies on considering *dominator* nodes. Given a start node $a$, a node $b$ is said to dominate a node $c$ if every path from $a$ to $c$ in the graph passes through $b$. The *immediate dominator* of $c$ is the node $b$ such that for every node $d$, if $d$ dominates $c$, then either $d = b$ or $d$ dominates $b$.

## 5.1 Structuring Loops

We will consider three primary different classes of loops. While other loops may appear in decompiled code, analysis of these more complex loops is more difficult. Further reading can be found in the paper "A Structuring Algorithm for Decompilation" by Cristina Cifuentes. Our three primary classes are as follows:

1. *While loops*: the node at the start of the loop is a conditional, and the latching node is unconditional.

2. *Repeat loops*: the latching node is conditional.

3. *Endless loops*: both the latching and the start nodes are unconditional.

The *latching node* here is the node with the back-edge to the start node. We note that there are at most *one of these* per loop in our language, as `break` and `continue` do not exist.

In order to do so, we will consider *intervals* on a digraph. If $h$ is a node in $G$, the interval $I(h)$ is the maximal subgraph in which $h$ is the only entry node and in which all closed paths contain $h$. It is a theorem that there exists a set $\{h_1, ...h_k\}$ of header nodes such that the set $\{I(h_1), ...I(h_k)\}$ is a partition of the graph, and further there exists an algorithm to find this partition.

We then define the sequence of *derived graphs* of $G$ as follows:

1. $G^1 = G$.

2. $G^{n+1}$ is the graph formed by contracting every interval of $G^n$ into a single node.

This procedure eventually reaches a fixed point, at which point the resulting graph is *irreducible*.

Note that for any interval $I(h)$, there exists a loop rooted at $h$ if there is a back-edge to $h$ from some node $z \in I(h)$. One way to find such a node is to simply perform DFS on the interval. Then, in order to find the nodes in the loop, we define $h$ as being part of the loop and then proceed by noting that a node $k$ is in the loop if and only if its immediate dominator is in the loop.

The algorithm for finding loops in the graph then proceeds as follows. Compute the derived graphs of $G$ until you reach the fixed point, and find the loops in each derived graph. Note that if any node is found to be the latching node for *two* loops, one of these loops will need to be labeled with a goto instead. While there do exist algorithms that can recover more complex structures, this is not one of them.

## 5.2 Structuring Ifs

An *if* statement is a 2-way conditional branch with a common end node. The final end node is referred to as the *follow* node and is immediately dominated by the header node.

First, compute a post-ordering of the graph, and traverse it in that order. This guarantees that we will analyze inner nested ifs before outer ones.

We now find if statements as follows:

1. For every conditional node $a$, find the set of nodes immediately dominated by $a$.

2. Produce $G'$ from $G$ by reversing all the arrows. Filter out nodes from the set above that do not dominate $a$ in $G'$.

3. Find the closest node to $a$ in the resulting set, by considering the one with the highest post-order number.

The resulting node is the *follow* node of $a$.

# 6 Type Analysis

Given control flow and some idea of which variables are which, it is frequently useful to be able to determine what the *types* of various variables are. While it may be correct to produce a result where every variable is of type `void *`, no one actually writes programs that way. Therefore, we would like to be able to assign variables and functions their types, as well as hopefully recover structure layout.

A compiler has significant advantages over a decompiler in this respect. The compiler knows which sections of a structure are padding, and which are actually useful; it also knows which things a function can take or accept. A compiler notices that the functions below are different, and so compiles them separately; a decompiler may not be able to notice that these functions accept different types without some more sophisticated analysis. In particular, on a 32-bit machine, these functions will produce *identical* assembly.

```
struct s1 { int a; };
int s1_get(struct s1 *s) { return s->a; }
struct s2 { struct s1 *a; };
struct s1 *s2_get(struct s2 *s) { return s->a; }
```

Given this problem, how does type analysis work?

# 7 Other Issues

Other issues that haven't been discussed here include doing things like automatically detecting vulnerabilities, detecting and possibly collapsing aliases, recovering scoping information, extracting inlined functions, or dealing with tail call optimizations. Many of these problems (and, in fact, many of the things discussed above!) do not have satisfactory solutions, and remain open research problems. For one, CMU's CyLab contains a group actively doing research on these topics.