Interprocedural Variable Liveness Analysis
for Function Signature Recovery

MIGUEL ARAUJO AND AHMED BOUGACHA
{maraujo@cs, ahmed.bougacha@sv}.cmu.edu
Carnegie Mellon University
April 30, 2014

Final Project Report

Project webpage `http://cs.cmu.edu/~maraujo/15745/`

Introduction

Compilers translate programs from a high-level source language to a lower-level target language, usually machine assembly. Decompilers reverse this process, to translate machine assembly to a higher-level language. Decompilation provides many challenges, most of them related to the fact that compilation is inherently lossy: high-level source information - such as control flow, function signatures, variables, types, among others - isn't explicitly present in the produced assembly code. In contrast, to produce high-level code, a decompiler has to extract as much of the high-level information contained in the assembly code as possible.

In this project, we focus on function signature recovery, i.e., the process of improving a decompiler's produced high-level code, by recovering function signatures - number and type of parameters and return values - from basic assembly code function constructs.

Our analysis is novel in that it doesn't rely on target machine-specific calling conventions (as defined by the target's Application Binary Interface), which are the basis for translating high-level function calls to low-level machine assembly code, by mapping sequential parameters (or return values) to specific Instruction Set Architecture registers. Indeed, the ABI calling conventions shouldn't be relied upon in the general case, because they are guaranteed to be respected only for calls accross object file / module boundaries. For instance, functions internal to an object may use special "fast" calling conventions, to the compiler's discretion. Also, hand written assembly code doesn't have to respect any convention at all, with the method to pass parameters/return values being defined on a per-function basis.

We base our work on the Dagger decompiler, which produces an LLVM Intermediate Representation output from machine assembly code. We choose to recover function signatures purely at the level of the produced LLVM IR, by analyzing liveness of registers at function boundaries.

We demonstrate an analysis that efficiently provides a conservative function signature for each function found in the input machine assembly code. A conservative but correct signature combines a superset of the parameters and return values: passing or return a useless value isn't ideal from a performance standpoint, but failing to pass/return a used value would compromise correctness.

We run the analysis on a corpus of C and C++ programs, compiled to the X86-64 architecture, and compare the true/original function signatures to the recovered signatures using a variety of criteria.

Function Signature Recovery

A Special Kind of Liveness Analysis

Our analysis is partly based on live variable analysis.

Classical Live Variable Analysis

Live variable analysis determines, at each program point, which variables are live, i.e., may be read before their next write.

SSA and Liveness Analysis

The LLVM IR, which we work on, has Static Single Assignment form, where each instruction produces a value that may not be overwritten afterwards. SSA IRs such as LLVM provide efficient ways to access the def/use chain for each defined value. Thus, an alternative to full-blown Classical Liveness Analysis may be done by analyzing the use chain for the definition we want to determine the liveness of, and combining this with dominance information. For instance, if a definition has no uses, it is dead everywhere.

Register Liveness Analysis

Analysis Domain: Register Variables

Here, we apply liveness analysis to the output of a decompiler. At the very lowest level, decompiler output closely replicates the behavior of the input machine code: most notably, registers defined by the machine's ISA are present as variables in the output, and the code output after translating an

instruction I emulates the behavior of I, such as reads from source registers, and writes to destination registers.

In most modern architectures, registers are also used as the main (and most efficient) way to pass argument or return values between caller and callee. The alternative, as used on older architectures (especially those with few ISA registers), is to use the stack memory to pass function arguments. Stack analysis isn't reliable in the general case, because on most architectures, stack memory isn't segregated from the rest of main memory, i.e., there is no general way to determine whether a stack pointer doesn't alias any other pointer. Because of this, it is usually impossible to determine that a previously spilled register will be reloaded with its original value: for instance, stack buffer overflows, a well-known and simple class of bug, would overwrite spilled registers. However, analyzing the stack as used only for argument passing is feasible; we choose to leave this problem for future work, mainly because we focus on target systems which use an ABI where parameters are preferentially passed in registers (X86-64).

We thus choose to focus on variables representing machine registers as the domain for our liveness analysis. The set of registers for a given machine is fixed, and defined by the machine's ISA.

Decompiling Register Accesses

The Dagger decompiler initially produces functions which have a single parameter, a pointer to a structure. This structure represents the register set of the target machine. Inside a translated function, local variables are used to represent machine registers. At function boundaries (entry, exit blocks, and function call instructions), all variables are saved/reloaded to/from the corresponding register's entry in the register set structure. The concrete client for the analysis described in this paper is a later transformation, which would promote elements of the register set structure to full-blown parameters/return values. This would improve the quality of the decompiled output code, by making it higher level.

Formal Parameters

Detecting potential Argument Registers

Formal parameter detection is based on global register liveness analysis. At their simplest, function arguments are just values that are defined outside the callee function (in the caller), and are used inside the callee function. We established that the means of communicating values between caller and callee is mostly registers. Thus, if a register is used but has not been defined since the entry of the function, it is bound to have been defined in the caller

(or earlier, for instance the caller's caller: we treat this case without any special consideration: if the register was defined in the caller's caller, the value was just passed from the latter, to the former, through the second function).

In the decompiled IR produced by Dagger, the initial register access in a function goes through the register set structure. Loads from the structure are thus initial values of registers. These are the variables that we want to analyze the liveness thereof. Registers that are initially loaded and aren't immediately dead are good candidates for promotion to formal parameters.

Distinguishing between Argument Register Uses

There are however different cases where an initially loaded register might be used.

Callee-save Registers First, callee-save registers, as mandated by the ABI, need to be saved (and restored when returning) by the callee if they are modified within it. The standard mechanism is to save the register on the stack at the callee function's entry block. We use a simple heuristic to detect these cases: a use of an initially loaded register that is saved through a pointer derived off of the value of the ISA's preferred stack pointer register is not considered as an argument use.

"False Positive" Uses Another case to consider is code sequences that are what we call "false positive" uses. For instance, a very common pattern for zeroing a register is to XOR (Bitwise Exclusive-Or) it with iself. For all intents and purposes, such an instruction would appear as a legitimate use of the register. However, simple algebraic transformations, as provided by LLVM's `instcombine` and `instsimplify` passes, replace these "false positive" uses with "true negatives"; in this case, a constant 0.

"True Positive" Uses Finally, real uses of the initially loaded register might appear in the code. If all such uses agree on a type, i.e., they are all casts to the same type, the latter type is used as the type of the newly discovered formal parameter.

Return Type

The other component of function signatures is the return type. Return value detection is done using a partially interprocedural register liveness analysis. Analoguously to function arguments, returned values are values defined inside the callee function but used outside, in the caller. Again, focusing on

registers as the value transfer vector, detecting return values of a function f is reducible to detection of live registers right after calls to f.

The union of the live register set after each call to a function f is used as the candidate set for return values. From this set, we exclude registers known to not have been modified by the function. Again, this also includes heuristics, mainly dismissing loads from a stack-pointer-derived-pointer as modifications of the register. We run the analysis top-down, starting at the entrypoint in the call graph.

Of note is the fact that we group returned values in a structure, this being the canonical way to return multiple values in LLVM IR. However, this makes no conceptual difference to multiple separate return values.

Expected Failure Points

The algorithm, in its current incarnation, will fail in a few different cases, some of which arguably impossible to handle in the general case.

Unused argument/return value If a register argument or return value is never used, it doesn't exist from our standpoint. If it's not used, the compiler might not even generate code for it, so, even before the decompilation process, at the compiled assembly code level, the argument/return value isn't present. Arguably, this isn't a failure of the recovery algorithm per se: if the argument/return value isn't used, it might as well not exist.

"False positive" extra parameter If the earlier algebraic simplifications failed to eliminate "false positive" uses of a candidate register argument, the promotion to function parameter will proceed. This is a somewhat general class of failures; usually, more aggressive simplifications are the solution (maybe even going as far as dedicated transformations for known patterns.)

"Hidden" extra parameters There's a few situations where extra parameters might appear in the compiled assembly, as a result of the process of lowering data types natively unsuitable for the target architecture. One interesting case where this occurs is C functions returning structures. In some cases, ABIs specify that the pointer to the returned structure memory be provided by the caller function to the callee, as an argument, to eliminate the overhead of copying temporary structures. This manifests itself as an extra parameter, not present in the original source code. Similarly, when passing values larger than the size of a single General Purpose Register, several parameter registers might be used, where there was only one original high-level source code parameter.

It should be possible to transform all variants of these hidden parameters to the correct high-level construct. This is left for future work.

Experiments

Experimental Setup

We applied the variable liveness analysis technique to a corpus of common C and C++ algorithms (Table 1). Most of these algorithms use different C++ data structures (such as vectors and maps), so the overall call graph to be analyzed is not trivial. While the goal of this evaluation was to validate the method developed, the use of real implementations not written for this specific purpose allows us to interact with less frequent language structures and to better gauge the limitations of the current approach.

Algorithm	**# functions**	**# arguments**
2sat	3	2, 2, 0
Addition Chain	1	3
Bezout Theorem	1	2
Dijkstra	3	3
Min-cost Max-flow	2	3, 4
Number of Divisors	1	1
Tarjan	1	2

Table 1: Algorithms used for benchmarking.

The algorithms were compiled to the target architecture and decompiled using our liveness analysis. Subsequently, a signature file was created representing the original function signature. A benchmarking tool was developed that would read the decompiled IR, the compiled binary and the function signature and would try to match the different recovered function arguments to the ones expected.

The benchmarking process can be summarized as follows:

1. the signature file is read so we know what functions are to be analyzed (external libraries are often linked and those functions need to be ignored).
2. the symbols table of the executable is used to translate function names to function addresses (using the “nm” tool)
3. functions in the decompiled IR are represented by their address and are mapped back to their corresponding name using the above information

4. We check whether the correct number of arguments was recovered and several heuristics are use to match the original function arguments to the recovered arguments.

Experimental Evaluation

Table 2 summarizes the results of our algorithm.

Algorithm	Function Name	# original args.	# recovered args.
2sat	SCC	2	2
	addClause	4	4
	isSatisfiable	0	7
Addition Chain	addchain	3	4
Bezout Theorem	find_bezout	2	3
Dijkstra	dijkstra	3	3
Min-cost Max-flow	dijkstra	3	3
	mcmf	4	3
Number of Divisors	num_divisors	1	3
Tarjan	tarjan	2	2

Table 2: Results after function signature recovery.

We must note that, in most scenarios, not only was the correct number of arguments recovered but a correct matching of the types was made between the LLVM IR data type and the function signature data type (e.g. C++ integers were usually mapped to LLVM IR i32 arguments). In order to do this, we had to consider both pointer arguments (e.g. i8* or i32*) and 64-bits integers to be possible pointers in the original function signature. This is because it is not always possible for the decompiler to successfully distinguish between a 64-bit integer and an array pointer.

We are happy to report that all errors found were either expected failure points or language features we had not initially considered, leading us to believe that the liveness analysis implementation is fully functional and apparently bug-free. A few details that should be considered for further improvements:

- Even though the *addClause* function in the 2sat algorithm had the correct number of arguments recovered, we were not able to match 2 of them as C++ integers were decompiled to i64 registers and not i32 as expected. The variable type sizes is an issue that is difficult to overcome unless we consider all possible sizes an integer argument might have.

- The *isSatisfiable* function in the 2sat algorithm had not arguments but 7 were recovered. We believe this is due to the fact the the function was very small and was most likely inlined. Function inlining is something we overlooked during the initial design that is obvious in retrospect.
- The extra argument found in the Bezout Theorem algorithm is simply the address of the structure being returned and was an expected failure point.
- The missing parameter in the Min-Cost Max-flow algorithm was being passed as a reference. Even though we did not consider variables passed as reference initially, we are still surprised that the argument was not found at all and further analysis would be required.
- The *num_divisors* functions was fairly small and we believe it was not called in enough different contexts so that we could prune the extra parameters we recovered.

Conclusion

We present a technique for recovering function signatures on decompiled machine code. Using register liveness analysis on function boundaries, we are able to accurately recreate the signatures of the original functions. We implement our technique on the Dagger decompiler, itsef based on the LLVM compiler framework. We run several tests on real world programs, comparing the real/original/source function signature, with the recovered signature, after compilation, decompilation, and our analysis. We are aware of several shortcomings in our recovery algorithm; all of which are good candidates for future work.

Distribution of total credit

We believe total credit should be distributed in equal parts amonst the two participants.