How Symbol Resolution Works

rtc-draper edited this page on Apr 12, 2014 · 2 revisions

The purpose of this document is to outline the manner in which LLVM performs symbolic resolution during disassembly. At the time of this writing, symbolic disassembly can be achieved for X86 and ARM targets.

Setup for Symbolic Disassembly

The first step to performing symbolic disassembly is creating an MCDisassembler object. Next, one must set up this object for disassembly by invoking its '''setupForSymbolicDisassembly''' function with four parameters: an OpInfoCallback, a SymbolLookupCallback, a DisInfo block, and an MCContext.

The OpInfoCallback is documented as follows:

```
/**
 * The type for the operand information call back function.  This is called to
 * get the symbolic information for an operand of an instruction.  Typically
 * this is from the relocation information, symbol table, etc.  That block of
 * information is saved when the disassembler context is created and passed to
 * the call back in the DisInfo parameter.  The instruction containing operand
 * is at the PC parameter.  For some instruction sets, there can be more than
 * one operand with symbolic information.  To determine the symbolic operand
 * information for each operand, the bytes for the specific operand in the
 * instruction are specified by the Offset parameter and its byte widith is the
 * size parameter.  For instructions sets with fixed widths and one symbolic
 * operand per instruction, the Offset parameter will be zero and Size parameter
 * will be the instruction width.  The information is returned in TagBuf and is
 * Triple specific with its specific information defined by the value of
 * TagType for that Triple.  If symbolic information is returned the function
 * returns 1, otherwise it returns 0.
 */
typedef int (*LLVMOpInfoCallback)(void *DisInfo, uint64_t PC,
                                  uint64_t Offset, uint64_t Size,
                                  int TagType, void *TagBuf);
```

We are currently unsure of how to utilize this callback because we do not know how to leverage it differently than SymbolLookupCallback. As such, we just have it return 0 for now.

Next, there is the SymbolLookupCallback. It is documented as follows:

```
/**
 * The type for the symbol lookup function.  This may be called by the
 * disassembler for things like adding a comment for a PC plus a constant
 * offset load instruction to use a symbol name instead of a load address value.
 * It is passed the block information is saved when the disassembler context is
 * created and the ReferenceValue to look up as a symbol.  If no symbol is found
 * for the ReferenceValue NULL is returned.  The ReferenceType of the
 * instruction is passed indirectly as is the PC of the instruction in
 * ReferencePC.  If the output reference can be determined its type is returned
 * indirectly in ReferenceType along with ReferenceName if any, or that is set
 * to NULL.
 */
typedef const char *(*LLVMSymbolLookupCallback)(void *DisInfo,
                                                uint64_t ReferenceValue,
                                                uint64_t *ReferenceType,
                                                uint64_t ReferencePC,
                                                const char **ReferenceName);
```

Implementing a symbol lookup callback is simple once the undocumented mysteries surrounding its use have been unraveled.

Here are some things you should know.

1) ReferenceType determines the semantics of ReferenceValue. For example, if ReferenceType == LLVMDisassembler_ReferenceType_In_Branch, then ReferenceValue will be the offset of the branch address from the start of the section we are currently disassembling. However, if ReferenceType == LLVMDisassembler_ReferenceType_InOut_None, then ReferenceValue is simply the value of the operand whose symbol we are trying to resolve.

2) The LLVM disassembly process will only attempt symbolic resolution using the SymbolLookup callback if resolution using the OpInfo callback fails to resolve a symbol.

With this in mind, we know that when ReferenceType == LLVMDisassembler_ReferenceType_In_Branch, we can call a function to find the symbol located at the address [(.text address) + ReferenceValue].

It is important to note that this only applies to resolution of statically-linked symbols. The LLVM API does not currently resolve calls to dynamically-linked functions; in other words, the LLVM API makes no connection between a call to a dynamically-linked function and the symbol corresponding to that call. In order to resolve dynamically-linked symbols, we will need to add our own logic to leverage the API to perform dynamic symbol resolution based on the type of the binary we are disassembling (i.e. ELF-64, COFF...)

Pages 8

Find a page...

+ Add a custom sidebar

Clone this wiki locally

```
https://github.com/draperlaboratory/fracture.wiki.git
```