DERIVE: A Tool That Automatically Reverse-Engineers Instruction Encodings

Dawson R. Engler
Computer Systems Laboratory
Stanford University
Stanford, CA
engler@stanford.edu

Wilson C. Hsieh
Department of Computer Science
University of Utah
Salt Lake City, UT
wilson@cs.utah.edu

ABSTRACT

Many binary tools, such as disassemblers, dynamic code generation systems, and executable code rewriters, need to understand how machine instructions are encoded. Unfortunately, specifying such encodings is tedious and error-prone. Users must typically specify thousands of details of instruction layout, such as opcode and field locations values, legal operands, and jump offset encodings. We have built a tool called DERIVE that extracts these details from existing software: the system assembler. Users need only provide the assembly syntax for the instructions for which they want encodings. DERIVE automatically reverse-engineers instruction encoding knowledge from the assembler by feeding it permutations of instructions and doing equation solving on the output.

DERIVE is robust and general. It derives instruction encodings for SPARC, MIPS, Alpha, PowerPC, ARM, and x86. In the last case, it handles variable-sized instructions, large instructions, instruction encodings determined by operand size, and other CISC features. DERIVE is also remarkably simple: it is a factor of 6 smaller than equivalent, more traditional systems. Finally, its declarative specifications eliminate the mis-specification errors that plague previous approaches, such as illegal registers used as operands or incorrect field offsets and sizes. This paper discusses our current DERIVE prototype, explains how it computes instruction encodings, and also discusses the more general implications of the ability to extract functionality from installed software.

1 INTRODUCTION

Many binary tools, such as assemblers, disassemblers, dynamic code generation systems [4, 6, 9, 13, 16], JITs [5, 7, 11], and executable code rewriters [12, 19, 22], need to understand how machine instructions are encoded. Unfortunately, specifying instruction layout with current tools requires looking up and detailing the offsets, sizes, and values of thousands of instruction fields. Unsurprisingly, this process is both error-prone and tedious, especially for CISC machines such as the x86. Currently, system builders specify encodings by hand, use ad hoc home-grown systems [6], or use more principled (but significantly more complex) systems such as the New Jersey Toolkit [17]. While the latter approaches can simplify the task to some degree, they all require that the user correctly specify very low-level information about how each instruction is encoded.

This paper describes a tool called DERIVE that eliminates the need to details of instruction layout. The client supplies a description of the assembly syntax for instructions that they want to encode, and DERIVE returns a specification of every instruction layout. DERIVE is based on a simple observation: virtually all architectures for which a programmer needs binary encodings will already have programs (e.g., assemblers) that contain this information. DERIVE extracts this information, which eliminates the need for clients to specify it. DERIVE feeds permutations of each client-requested instruction to the system's assembler and does simple equation solving on the assembler's output to determine how the instruction is encoded. It then emits a C structure that specifies this encoding, which clients can use to easily build the tools they need. As an example, a machine-independent disassembler can be written in less than two hundred commented lines of C.

DERIVE is general-purpose and portable. It has been used to reverse-engineer encodings on SPARC, MIPS, Alpha, ARM, PowerPC, and x86. In the case of x86, DERIVE handles variable-sized instructions, large instructions (16 bytes), multiple instruction encodings determined by operand size, and other CISC features.

DERIVE is simple. Exploiting an existing code base makes its implementation and input specifications simpler than previous approaches. The current prototype is roughly two thousand lines of C code. In contrast, the New Jersey Machine-Code Toolkit [18, 17] is roughly 12K lines of Icon code. A DERIVE specification consists of a description of an instruction's assembly syntax. Clients specify what instructions they want, rather than how to construct them. This shift from imperative to declarative specification eliminates most of the tedious details required by previous approaches, as well as many of their specification errors.

An important motivation for building DERIVE was our desire to avoid hand-specifying the x86 instruction set. This dis-

Dynamo '00 1/00 Boston, Massachusetts, USA
 ISBN 1-58113-241-7/00/0001...$5.00

taste is one of the main reasons why we have not retargeted two of our JIT systems [6, 7] to the x86, despite repeated requests.

The rest of the paper is as follows. Section 2 discusses related work. Section 3 gives an overview of the DERIVE system and its design. Section 4 discusses the three main solvers in DERIVE: the register solver (Section 4.1), the immediate solver (Section 4.2) and the jump solver (Section 4.3). Section 5 discusses the general approach of automatic reverse-engineering. Section 6 concludes and discusses future work. Finally, the Appendix provides a complete DERIVE specification of the MIPS instruction set.

2 RELATED WORK

The most important precedent to DERIVE is Collberg's work on automatically retargeting compilers [3]. He computes machine specifications by reverse-engineering the knowledge out of C compilers. These machine specifications are used to retarget the BEG back-end generator. Reverse-engineering machine specifications is trickier than reverse-engineering instruction encodings, because the former requires the inference of instruction semantics.

DERIVE can be viewed as similar in spirit to Massalin's "superoptimizer" [15]. Given an instruction (or set of instructions) the superoptimizer searches for cheaper, but functionally equivalent, alternatives by exhaustively testing different instruction combinations on the actual hardware. In a sense, this process can be viewed as extracting knowlege (instruction functionality) buried in an installed base (the hardware).

The New Jersey Machine-Code Toolkit generates routines to manipulate machine-code, based on a specification language called SLED (Specification Language for Encoding and Decoding). SLED specifications are exact descriptions of instruction layout; an example description of the bitfields in one instruction type reads as follows:

```
fields of itoken (32)
  op 30:31 rd 25:29 op3 19:24 rs1 14:18  i 13:13
  simm13 0:12 disp30 0:29 op2 22:24 imm22 0:21
  a 29:29 cond 25:28 disp22 0:21 asi 5:12 rs2 0:4
  opf 5:13 fd 25:29 cd 25:29 fs1 14:18 fs2 0:4
```

In contrast, DERIVE's specifications are at a much higher level (see Appendix A). As a result, DERIVE completely eliminates many sources of errors noted in imperative specifications [8]: conflicting values specified for the same bit, instruction operands that are not used in the actual encoding, wrong values for specific operands, field offsets and field sizes, and reversed operands. Additionally, DERIVE's use of the assembler naturally catches specification errors, such as the wrong number of operands and illegal register names.

In addition, DERIVE is much smaller, because it bootstraps functionality off of an existing installed base. It is roughly 2K lines of commented C code, compared to the NJ Toolkit's 12K lines of high-level Icon code.

3 OVERVIEW

The most important parts of the DERIVE system are the following three machine-independent pieces:

1. A preprocessor that consumes easy-to-write client specifications of instructions and emits a vector of C structures that specifies the instructions whose encodings we wish to derive. We describe these specifications later in this section.
2. The DERIVE solver, which solves for each encoding and then emits a C structure that describes each different encoding. The solver is heart of the system. Section 4 describes how it works in some detail.
3. An encoder generator that consumes encoding specifications and emits functions (or macros) that will generate these encodings.

In this paper we concentrate on the implementation of the DERIVE solver.

DERIVE treats the assembler's implementation as a black box with a well-defined interface. The interface is the symbolic instruction set accepted by the assembler. DERIVE reverse-engineers instruction encodings from the assembler by feeding instruction permutations to it and examining the resultant executable.

Instruction encodings have several features that make it ideal for automatically computing them. First, encodings are *stateless*, in that the same instruction and operands always give the same encoding. Thus, we do not need to search all possible instruction sequences but can instead just examine each instruction in isolation. Second, instruction operands are relatively independent. For a given instruction, the location and value of the bits used to specify a particular operand value is typically the same, irrespective of the values of the other operands. As a result, representing and searching the operand space is efficient. We can solve for each operand in isolation, while holding the others fixed, rather than solving for all possible combinations. Finally, instruction encodings are simple. For example, an immediate value tends to be encoded with the same bit pattern as its value. The only (minor) transformation done by existing machines is the sign-extension of immediates or the truncation of jump offsets.

3.1 Input and Output

Clients specify what instructions they want encodings by giving a yacc-like specification of instructions. The specification contains productions that contain a list of instructions to generate and a description of the assembly syntax for the instructions. Operands can be registers, immediates or labels. Registers are collected in a named list. Immediates are specified by type (current types are signed and unsigned integers) and an optional size hint. Labels include an optional size hint.

An example format to generate single-operand instructions on the x86 is:

```
/* list of registers */
regs = (%eax, %ebp, %ebx, %ecx, %edi, %edx, %esi, %esp);

/* partial list of single operand instructions */
dec, inc, neg, not, pop, /* .. */ --> &op& r_1:regs;
```

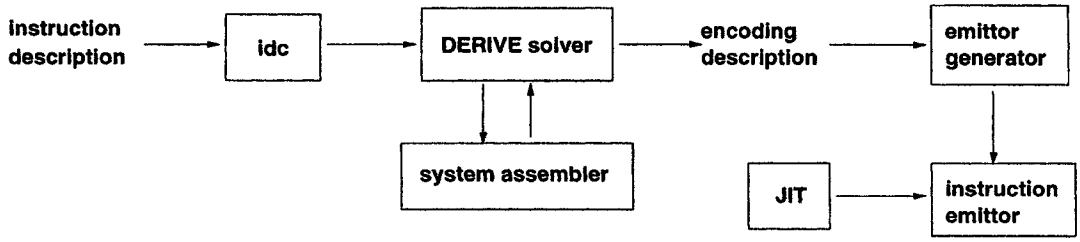

Figure 1: How information flows through our tool chain. We have built the tools along the primary horizontal axis.

Each instruction has the assembly syntax "&op& r_1". &op& indicates where the instruction occurs. The operand specification says that the formatting substring "r_1" is replaced by members of the list of registers reg.

A more complex x86 production is for instructions that take five operands: one source register, and four operands to specify a memory location, composed of an immediate displacement, a base register, an index register, and a scale factor:

```
%{
unsigned pow2(unsigned x) { return 1 << x; }
%}

base_regs = (%bx, %bp); index_regs = (%si, %di);

xchg, xadd, add, adc, sub, cmp, and, or, xor, /* ... */
 --> &op& r_1:regs, disp (r_base:regs,
        r_index:index_regs,scale:pow2);
```

DERIVE solves for each operand and then emits one or more C data structures that specify each instruction encoding. A given instruction may have several operand-dependent encodings. For example, the x86 has multiple instruction encodings, depending on whether an instruction's immediate value fits in 8 or 32 bits. Figure 4 gives the structure definition. Figure 5 gives a sample instance for the dec instruction in the first specification above. Appendix A gives a complete specification of the MIPS instruction set.

3.2 Assumptions

The DERIVE design makes several assumptions. First, if we view the assembler as a transformation $A : S \rightarrow B$ from symbolic instructions S to binary instructions B, DERIVE assumes that such a transformation is linear. In other words, $A(s_1 s_2) = A(s_1)A(s_2)$. In order to satisfy this assumption, assemblers cannot perform program transformations: they cannot add, remove, or reorder instructions. If native assemblers do not satisfy this requirement, DERIVE requires that the user specify how to turn off such features. For example, on the MIPS we must explicitly tell DERIVE how to turn off instruction reordering. (Note that we could use brute force to turn off optimization by surrounding each instruction with labels.)

Since architectures have relatively small numbers of registers, DERIVE can try each register value in register operands. For immediates and branch targets, though, exhaustively enumerating the values is infeasible. Thus, DERIVE's second assumption is that immediates of the same size can be encoded in the same instruction. This allows it to exhaustively enumerate all possible instruction encodings with $O(n)$ tests (rather than $O(2^n)$): one for each of the n bits of an immediate field.

The implementation of DERIVE makes the assumption that each field can be solved for independently, while holding the others fixed. That is, each field is non-overlapping. This restriction is important for efficiency, and we know of no machine that violates it.

The main client specification error the DERIVE implementation leaves open is it cannot guarantee that the given specification covers the entire instruction set. For example, if the specification omits machine registers, DERIVE will not compute their encodings. However, such errors are "fail stop" in that the instruction specification will mark these registers as illegal; encoders can then include explicit legality checks.

4 IMPLEMENTATION

DERIVE decomposes instructions into an "opcode mask" and an arbitrary number of operand fields. An opcode mask contains the sequence of 1's that must be set in an encoding to specify a given opcode. Operands can be registers, immediates, or labels. An operand field contains the sequence of 1's that must be set to specify a given operand value. Register fields are specified as a sequence of masks, one for each legal register value, and an offset that the mask must be shifted to. Immediates and label fields are generally represented as a size and an offset. The size gives the maximum immediate size (in bits) that can be encoded; the offset, the location to which to shift it.

DERIVE is composed of three solvers, each specialized to derive a specific operand type. These solvers are quite general: they handle variable-sized instructions, arbitrarily large instructions, and instructions that take an arbitrary number of operands. Each solver uses the assembler to compute instruction encodings. DERIVE emits code into an assembler file, runs the assembler, and finds the code in the resulting executable. We explicitly emit fenceposts using assembler data directives, so that we can locate where instructions begin and end.

The solvers all share the same basic idea. Each one locates a field and determines its size by emitting one instruction for each legal operand value (while holding all other field values fixed) and finding all bits that change in the binary encoding. Those bits that change belong to the field. Its

offset is given by the lowest changing bit; its size by the difference between its lowest and highest bit. A specific operand's value exactly equals the value of these bits when it is used. All other bits belong to other fields, or to the instruction mask. After all fields have been solved for, the instruction mask is set to the remaining unclaimed bits (i.e., those that are set to 1 in every emitted instruction).

The two main differences between the solvers is how they iterate over legal operand values, and what encoding schemes they can handle. The register solver derives each register field encoding by simply iterating over all legal operand registers and recording each register's unique field bit pattern. In contrast, the immediate solver cannot enumerate all operand values. Fortunately, immediate encodings are size-, rather than value-dependent: an instruction that can encode one n bit immediate value can encode any n-bit immediate. Thus, it need only derive an encoding for each size (measured in bits), rather than each value. Finally, the jump solver derives each address field's encoding in an instruction. Like immediates, jump fields are encoded as offsets or addresses in instructions. The main difference between this solver and the immediate solver is that operand values are generated differently. To see the encoding of a given immediate value, the immediate solver can simply emit it, while the jump solver must emit labels. A consequence is that it is prohibitively expensive to solve for large offsets or addresses. We exploit sign-extension of negative jump offsets and jumps to unresolved labels to find the size of a field efficiently.

4.1 The Register Solver

The register solver is DERIVE's most basic solver, and is recursively called by all of the other solvers. It has two tasks. First, it finds the current instruction's opcode mask. Second, it finds both the location and size of each register operand field, along with all 1's that must be set for each legal register for this field.[1]

The solver performs these two tasks simultaneously. It first claims all bits for the opcode mask. As it solves for each register field, it reclaims all the field's bits from the opcode mask. At the end, exactly those bits that remains set to 1 for every emitted instruction will belong to the opcode mask. In the degenerate case of an instruction that takes no operands, the solver need only emit a single instruction: the value of this is the opcode mask.

Figure 2 gives pseudo-code for the register solver. It derives a specific field as follows:

1. Iterate over all legal registers in the field. For each register, create a copy of the the instruction format string. Replace the current operand with the current register, and all other operands with their first legal value. Then emit the instruction and read it back in.

2. To derive the opcode mask, the solver finds all bits set to 1 in all emitted instructions. It initially sets all of the opcode mask bits to 1:

```
op_mask = ~0;
```

[1]Register fields can be non-contiguous sequences of bits. We refer to them as "fields" only for linguistic convenience.

```
# solve for instruction 'inst', specify encoding
# in 'ispec'
void register_solve(ispec, inst)
   # op_mask will have 1's for every bit that
   # is always set to 1: start with all bits set
   op_mask = ~0;

   # solve for each register operand field
   foreach op in inst.reg_operands
      # emit one instruction for each legal op
      # reg, holding all other operands fixed.
      foreach r in op.regs
         # get the instruction format string
         i = inst.format_string;
         # replace r in the instruction.
         rewrite(i, op.field_name, op.register[r]);
         # set first instance of all other fields.
         foreach op' in (inst.reg_operands - op)
            rewrite(i, op'.field_name, op'.regs[0])
         # emit and read back in.
         inst[r] = emit(i);
         # reduce inst mask
         op_mask = op_mask & inst[r];
      end;
      # solve for current field.  core idea: only
      # bits that change values can belong to field.

      # record all 1's that never change
      const_bits = ~0;
      foreach r in op.regs
         const_bits = inst[r] & const_bits;

      # get field by set a 1 in all bits that change
      field = 0;
      foreach r in op.regs
         field = field | (inst[r] & ~const_bits);

      # the field's size is bounded by its most and
      # least significant bits.
      nbits = msb(field) - lsb(field) + 1;

      # field's offset = its least significant bit.
      offset = lsb(field);

      # for each register, record a mask containing
      # all 1 bits we need to set.
      foreach r in op.regs
          ispec[op].rmask[r]
               = (inst[r] & field) >> offset;

      # initialize the other parts of spec.
      ispec[op].type     = REGISTER;
      ispec[op].encoding = IDENT;
      ispec[op].offset   = offset;
      ispec[op].nbits    = nbits;
   end;

   # op_mask now has 1's in all positions we
   # must set to specify this instruction.
   ispec.i_mask = op_mask;
end;
```

Figure 2: Pseudo-code for register solver.

Then, while iterating over each register field, it incrementally reduces the mask by bitwise "anding" it with each binary instruction:

```
op_mask = op_mask & inst[r];
```

This process will leave the mask with 1's in every position that has a 1 set for all instruction instances.

3. To derive the current register field, the solver examines the emitted instruction stream, looking for all bits that change between 0 and 1. Such bits belong to the field, since all other operand values were fixed, and all values for this field were enumerated.

   To find these bits, the the solver first creates a mask with 1's in all positions that are 1 in every emitted instruction:

   ```
   # record all 1's that never change
   const_bits = ~0;
   foreach r in op.regs
      const_bits = inst[r] & const_bits;
   ```

   It then finds all bits that do change by bitwise anding each instruction with the complement of this mask and logically summing the result:

   ```
   field = 0;
   foreach r in op.regs
      field = field | (inst[r] & ~const_bits);
   ```

   At the end, the field mask has a 1's set for every position in the field. The size of the field is bounded by the most and least significant bits set in this computed mask. The field's offset is given by its least significant bit.

4. To derive the 1's that must be set to encode each register in the current field, the solver again iterates over all legal registers, bitwise anding each instruction with the computed field mask:

   ```
   # for each register, record a mask
   # containing all 1 bits we need to set.
   foreach r in op.regs
      ispec[op].rmask[r]
         = (inst[r] & field) >> offset;
   ```

   Setting these bits to 1 will specify the current register.

A limitation of the current solver is that it assumes that changing register values will not change the instruction size. This assumption is just an implementation simplification, and not a fundamental limitation. We do not make it for other field types.

4.2 The Immediate Solver

DERIVE's immediate solver follows the same template as the register solver: it cycles each operand value over legal values, while holding the other operands fixed. However, unlike register operands, it is impractical to enumerate all legal values for an immediate operand: an n bit immediate field would require 2^n permutations. Fortunately, on all machines we know of, immediates of the same size are encoded in the same way. I.e., if an immediate x is encoded in an instruction of size i, then all other immediates of that size can be encoded in i. Thus, given an immediate field of size n bits, this assumption lets the solver exhaust all possible encodings with only $O(n)$ tests (i.e., one for each size). Note that there can be multiple legal encodings, depending on the size of the immediate. For example, the x86 typically provides special "short" instruction forms for 8-bit immediates, and longer instruction encodings for larger ones. The solver will find each different encoding and and emit a specification for each truly different one. The solver makes the additional assumption that immediates are encoded literally in the instruction stream. While we could add support to derive fancier encoding schemes, we are not aware of a machine that needs it.

At a high level, the solver finds each immediate field's encoding by iterating down and solving for each field size from the maximum number of bits (n) down to 1 bit. To solve for m bit immediates, it chooses two m bit constants with complementary 1s and 0s in their low $m-1$ bits. It then emits two instructions, one with each immediate as an operand. A bitwise xor of the two instructions will set exactly the low $m-1$ bits in the immediate field, giving both the field's offset and size. Before the solver attempts to derive an encoding it first checks if a previous (larger) encoding matches what the assembler does for immediates of this size. If it matches, DERIVE skips down to the next size; if not, it invokes the full solver.

Figure 3 gives detailed pseudo-code for the immediate solver. At a high level, it works as follows:

1. Chose a random $m-1$ bit value and create two m-bit constants, $v0$ and $v1$, by setting $v0$ to the value and $v1$ to its complement. Then set the mth bit in each:

   ```
   x  = (1 << (m - 1));
   v0 = random() % x;
   v1 = ~v0      % x;
   v0 = v0 | x;          # set mth bit
   v1 = v1 | x;
   ```

   By emitting instructions with these two constants, we force the low $m-1$ bits of the immediate fields to have complementary bit values, while all other bits remain constant. The solver creates two format strings, the first with the immediate operand replaced with $v0$, the second with $v1$. These strings are fed to the register solver, which derives all register field encodings and a partial opcode mask:

   ```
   inst0 = inst1 = inst;
   rewrite(inst0.fmt, op.field_name, v0);
   rewrite(inst1.fmt, op.field_name, v1);
   register_solve(s0, inst0);
   register_solve(s1, inst1);
   ```

   The two specifications s0 and s1 should have the same register masks, and the same size in bytes. The only difference between the two should be the opcode mask, which will differ by exactly the bits that differ in v0 and v1.

```
# a simplified immediate solver: only solves for
# a single immediate field), and returns list of
# specs for each different instruction encoding.
list specs immed_solve(inst, op)
    # solve for each op size (measured in bits)
    foreach m = op.bits to 0
        # make two constants with mth bit set to 1
        # and lower bits alternating
        x = 1 << (m - 1);
        v0 = random() % x;
        v1 = ~v0      % x;
        v0 = v0 | x;      # set m-1th bit
        v1 = v1 | x;

        # use register solver to derive both inst
        # and register masks.  only the mask can
        # differ (by exactly the low m-1 bits in
        # v0 and v1).
        inst0 = inst1 = inst;
        rewrite(inst0.fmt, op.field_name, v0);
        rewrite(inst1.fmt, op.field_name, v1);
        register_solve(s0, inst0);
        register_solve(s1, inst1);
        assert(s0.nbytes == s1.nbytes);
        assert(s0.rmask  == s1.rmask);

        # set the low m-1 bits in field to 1.
        field_bits = s0.imask ^ s1.imask;
        # field's offset = its least significant bit
        field_offset = lsb(field_bits);
        # add back mth field bit: all field bits = 1
        field_bits |= 1 << (m + field_offset - 1);
        # fix inst mask
        s0.imask = s0.imask & ~field_bits;

        # check some assumptions
        field_mask = (1 << m) - 1;
        x = (s0.imask >> field_offset) & field_mask;
        assert(x == v0);
        x = (s1.imask >> field_offset) & field_mask;
        assert(x == v1);
        assert(imask == (imask1 & ~field_bits));
        assert(msb(field_bits) == (field_offset+m));

        # initialize the other parts of spec.
        s0[op].type     = IMMED;
        s0[op].encoding = IDENT;
        s0[op].offset   = offset;
        s0[op].nbits    = m; # by def'n

        # append spec to list if it is a different
        # encoding.  In reality we skip duplicate
        # encodings rather than filtering them.
        spec' = last(specl);
        if (spec' && spec'.nbytes == s0.nbytes &&
            spec'.offset == s0.offset)
               assert(spec'.nbits == s0.nbits+1);
        else
               append(specl, s0);
    end;
    return specl;   # return list of all specs.
end;
```

Figure 3: Pseudo-code for simplified immediate solver.

2. The solver finds the immediate field by first xoring the two instruction masks. Since only the lower $m-1$ bits of the field differ, this act sets 1s exactly in the location of these bits and 0s everywhere else. This result's least significant bit gives the field's offset. The solver then corrects the opcode mask by removing all field bits from it.

   ```
   # set the low m-1 bits in field to 1.
   field_bits = s0.imask ^ s1.imask;
   # field's offset = its least significant bit
   field_offset = lsb(field_bits);
   # add back mth field bit: all field bits = 1
   field_bits |= 1 << (m + field_offset - 1);
   # fix inst mask
   s0.imask = s0.imask & ~field_bits;
   ```

3. The solver checks to see if a previous encoding matches this one. Since we work our way down from larger immediates to smaller immediates, we may have already figured out the encoding for the field. If we have, the current encoding is rejected; otherwise, it is added to the list of derived encodings. Note that we must evaluate the encoding for each immediate size, because certain architectures (x86) have some instruction encodings that vary with the immediate size.

Our actual solver is more general than this sketch, and handles an arbitrary number of immediate operands. It also allows the user to specify specially encoded fields, such as the power-of-two scale field on the x86.

4.3 The Jump Solver

Machine instructions encode jump destinations as immediates (either as direct addresses or relative offsets). As a result, our jump solver closely mirrors the preceding immediate solver. It has two main differences.

First, jump immediates tend to be transformed before being encoded. For example, relative jump offsets are computed by subtracting the current program counter value (or sometimes its succeeding value) from the target address. Additionally, on machines that force 4-byte alignment of jump targets, all targets will have two trailing 0s, which are removed during encoding. The jump solver must derive these transformations. Fortunately, as discussed below, there appear to be a small set of common transformations, which our solver iterates through until it finds a match.

Second, many instructions (such as branches) do not allow direct specification of target addresses; instead, operands must be labels. Thus, creating an n-bit immediate offset requires emitting a label that is (roughly) 2^n instructions away. For large immediate values, this technique is impractical. Below, we discuss how to solve this problem by exploiting sign extension.

At a high level, the jump solver must handle two independently varying instruction options: (1) relative vs. absolute jumps and (2) jumps that take immediates vs. those that only take labels. The former options have different encoding schemes; the latter options, different solver schemes. This section concentrates only on the case of relative jumps that take only labels as operands. The two cases of jumps that accept immediate operands are simpler versions of this one.

The remaining case of absolute jumps that only take labels does not seem to occur in practice; for completeness, we briefly sketch how to handle such a situation at the end of this section.

As a first step, the solver classifies a jump as either relative or absolute by emitting two consecutive jumps to the same target and comparing the emitted code values. Absolute jumps will have identical bits, since both instructions encode the same target address. Relative jumps will differ, since they are different distances from the target and thus will have different offset values.

Given an instruction which holds a relative jump, the solver must find the jump target field's offset, size, and encoding. It's core technique mirrors that of the previous two solvers: choose operands which will trigger different bit settings in the encoded instruction, and use these differences to find a field's location and size. First, the solver finds out the base for the field's offset. For example, on the x86 the offset is from the current instruction; on the SPARC it is from the next instruction; on the ARM it is the instruction that occurs two instructions after the current instruction.

The solver then finds a field's offset by emitting two forward jumps whose target differs by one instruction. These two instructions will differ by a single bit, which is the lowest bit in the target field. The solver obtains the field's offset by performing a bitwise xor of these instructions, then finding their least significant bit.

The solver's main challenge is exactly finding the field's size. Generating the code to hit large offset values is impractical (e.g., an offset 2^n instructions away). Fortunately, on existing machines, the jump instruction's target field is sign-extended (to allow both forward and backward jumps). The solver discovers a field's size by exploiting this fact. It emits one backward and one forward label reference. The latter will set all high field bits to one, the latter sets the lowest field bits. The field's size is found by performing a bitwise xor and then looking for the most significant bit.

Finally, the jump solver checks whether the offset is transformed. Currently it only checks whether the offset has trailing zeros stripped (i.e., the offset is a word offset rather than a byte offset). This checks covers all of the cases that we know of; additional ones could be easily added if necessary.

If we needed to derive jumps on a hypothetical machine that has forward-only relative jumps, or absolute jumps that only take labels, the immediate field will not be sign-extended, and this last trick will not work. We know of two alternative approaches to decode jumps on such a machine:

1. Exploit separate compilation. Using an undefined label as an operand forces the assembler to leave space for the largest field value. Comparing the field's value before and after linking gives its size and encoding.

2. Fall back on manual assistance and have the client annotate the field with its maximum size. This single annotation is not unreasonable, since derive still frees the client from specifying the rest of the instruction set.

5 AUTOMATIC REVERSE ENGINEERING

DERIVE provides an example of how to exploit a general observation: most programs that one will ever write have already been written, or at the least have significant portions of their functionality embedded in existing code. For example, if one writes a C compiler, a Unix operating system, or a debugger, many instances already exist. Thus, techniques to automatically reverse engineer functionality from pre-existing code bases could dramatically ease software construction. This paper can be read as an example of how to pull an implementation's knowledge (instruction encodings) out of a black-box implementation (an assembler) via a well-defined specification (assembly language). In a sense, automatic reverse engineering (AutoRev) can be viewed as a robust way to do code reuse in the presence of an uncooperative code base.

A related area of machine learning research is called "inductive logic programming" [1]. Its goals are similar to ours: to build tools that can automatically reason about other programs. Researchers in the field have developed methods for deriving Prolog programs from logic statements about program inputs and outputs. DERIVE attacks a slightly different problem: the goal is not generate a program, but to extract specific knowledge from an existing program, so that other tools can use that knowledge.

If code already exists, why not use it directly? In many practical cases, existing implementations are not in a form that lend themselves to code reuse.[2] A system may not have source code, it's implementation may be correct but unmaintainable, or its code's interfaces or implementation may be inappropriate and require too much manual modification to use. This latter situation is true for binary encoders: while there are public domain assemblers with source code, reusing their code to generate efficient encoders appears to be too time-consuming to be worthwhile. Whatever the specific reasons, it is an empirical fact that programmers frequently re-implement significant portions of existing code, often for good reason. Further, many worthwhile projects are not done because the effort required would be too great. Practical AutoRev techniques would allow these projects to be done.

As an example, current linker technology tends to be a couple of decades old. Most linkers have no support for link-time type checking, stripping of functions that are not called, code packing for better instruction cache performance, etc. Unfortunately, these limits will likely persist because the effort to rewrite linkers for existing architectures is simply too great. In fact, several language-level design decisions in C++ were made because the language committee decided they could not count on improving the deployed linker state of the art (but could count on widespread deployment of new compilers!) [20]. From our experiences with DERIVE, it appears possible to cheaply bootstrap a new generation of linkers using the existing set. Linkers need only a few specific types of machine-dependent information, derivable by feeding appropriate inputs to existing assemblers and linkers.

There are a number of challenges to AutoRev. Theoretically, it can produce a new table-driven version of any (terminat-

[2]With the fortunate exception of reusing system libraries, traditional code reuse has proven surprisingly expensive and difficult.

ing) implementation by simply running the implementation on every input and recording its output. Of course, in many cases such an approach is impractical: the input space may be too large to search, or the table itself may be too large to store. To a first order, whether AutoRev is practical is determined by how much of an implementation's input space an AutoRev system needs to explore. Our AutoRev strategy involves searching relatively few points in the input space and inferring the values of the rest from these. The correctness challenge to this approach is finding all "inflection points" in the implementation: if one is missed, the new implementation will behave incorrectly.

AutoRev provides several interesting possibilities beyond automatic software construction. For example:

1. Automated regression test generation. AutoRev can be used to automatically test a new implementation of some program X against an existing X'.

   The AutoRev system can generate test cases by searching for implementation inflection points and then checking that these cases match in both implementations. Inflection points can be found by ripping code apart either at the source level (using a compiler) or at the executable level (using an executable analyzer), and have the AutoRev system mutate inputs until it covers these inflections. The AutoRev system can generate test cases by searching for implementation inflection points and then checking that these cases match in both implementations.

2. Automatic software rewriting. An interesting use of AutoRev is as an automatic way to "throw the first one away" [2]. For example, one could write an ad hoc (but correct) binary encoder, and then use DERIVE to extract a clean representation.

6 CONCLUSION

We have presented a system, DERIVE, that reverse-engineers instruction encodings from pre-existing software (the system assembler). By doing so, it frees users from having to specify (or, frequently, mis-specify) the plethora of details of instruction layout. They need only list the instructions and operands they want encodings for. DERIVE extracts these details from the system assembler. DERIVE successfully reverse engineers instruction encodings on the SPARC, MIPS, Alpha, ARM, PowerPC, and x86. In the last case, it handles variable-sized instructions, large instructions (16 bytes), multiple instruction encodings determined by operand size and other CISC features. DERIVE is significantly smaller (and simpler) other instruction encoding systems. Further, its declarative specifications eliminate the mis-specification errors that plague other approaches (e.g., illegal registers used as operands, the wrong size, offset, and value for instruction fields).

We intend to extend DERIVE's techniques to reverse engineer object code file formats, including debugging and linkage information. Using this information, we hope to easily construct a set of interesting tools, including AutoRev-derived versions of ATOM [19], dynamic linking libraries [10], object-level sandboxers [21], executable optimizers, and linkers. By eliminating much of machine-specific effort needed to build such tools, useful, but niche ones such as ATOM [19], could enjoy wide-spread cross-architecture deployment, and wide-spread but primitive tools such as linkers could enjoy a much needed boost of functionality.

This domain has two allures. First, it represents one of the best cases for AutoRev: such encodings are relatively stateless, and various independence assumptions can be exploited to make AutoRev fast. Second, the area is plagued by the need to repeatedly reimplement functionality contained in existing software. Binary utilities cannot always be used directly. For some systems, it is too expensive to call existing programs: dynamic code generation systems cannot afford the time to call an assembler. In other cases, the software has an inappropriate form and must be rewritten from scratch. For example, one common "trick" that commercial companies use to discourage third-party vendors is to have proprietary symbol table layouts, which change on every software release [14]. The cost of manually reverse-engineering these formats has forced some implementors to avoid object-level modifications [14], in spite of the strong advantages for such an approach.

Future AutoRev research will continue developing developing methods both to robustly extract functionality through software interfaces and and to check that such functionality covers all cases. While AutoRev is not applicable to all domains, our initial experiences with DERIVE demonstrate that for some it provides a powerful alternative to traditional software construction approaches.

7 REFERENCES

[1] F. Bergadano and D. Gunetti. *Inductive Logic Programming.* MIT Press, 1996.

[2] Frederick P. Brooks, Jr. *The Mythical Man-Month.* Addison Wesley, 20th anniversary edition, 1995.

[3] Christian Collberg. Reverse interpretation + mutation analysis = automatic retargeting. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[4] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 1996.

[5] P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of 11th POPL*, pages 297–302, Salt Lake City, UT, January 1984.

[6] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170, Philadelphia, PA, USA, May 1996. *http://www.pdos.lcs.mit.edu/~engler/vcode.html.*

[7] D.R. Engler and M.F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1996*, pages 53–59, Stanford, CA, USA, August 1996.

[8] Mary Fernández and Norman Ramsey. Automatic checking of instruction specifications. In *1997 International Conference on Software Engineering*, 1997.

[9] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. Annotation-directed run-time specialization in C. In *Symposium on Partial Evaluation and Semantics-Based Program Manip ulation (Amsterdam, The Netherlands)*, pages 163–178, New York, June 1997. ACM.

[10] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software: Practice and Experience*, 24(4):375–390, April 1991.

[11] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of PLDI '94*, pages 326–335, Orlando, Florida, June 1994.

[12] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *SPE*, 24(2):197–218, feb 1994.

[13] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996.

[14] Steve Lucco. Personal communication. Use of undocumented proprietary formats as a technique to impede third-party additions, August 1997.

[15] Henry Massalin. Superoptimizer — a look at the smallest program. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, October 1987.

[16] Massimiliano Poletto, Wilson Hsieh, Dawson Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.

[17] N. Ramsey and M. F. Fernández. The New Jersey machine-code toolkit. In *1995 Winter USENIX*, December 1995.

[18] Norman Ramsey and Mary Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, to appear.

[19] Amitabh Srivastava and Alan Eustace. Atom - a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[20] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, Reading, MA, 1994.

[21] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 1993.

[22] D.W. Wall. Systems for late code modification. *CODE 91 Workshop on Code Generation*, 1991.

A MIPS DESCRIPTION

Following is a complete specification of the MIPS instruction set for DERIVE.

```
%{
char *directive = ".set noreorder\n.set nomacro\n"
        ".set noat\n";
%}

regs = ( $0, $1, $2, $3, $4, $5, $6,
  $7, $8, $9, $10, $11, $12, $13, $14, $15,
  $16, $17, $18, $19, $20, $21, $22, $23,
  $24, $25, $26, $27, $28, $29, $30, $31 );

fregs = ( $f0, $f1, $f2, $f3, $f4, $f5, $f6,
  $f7, $f8, $f9, $f10, $f11, $f12, $f13, $f14,
  $f15, $f15, $f16, $f17, $f18, $f19, $f20,
  $f21, $f22, $f24, $f25, $f26, $f27, $f28,
  $f29, $f30, $f31 );

fcond = ( $fcc0, $fcc1, $fcc2,
  $fcc3, $fcc4, $fcc5, $fcc6, $fcc7 );

nop, sync, tlbr, tlbwi, tlbwr, tlbp, eret --> &op&;

movf, movt --> &op& r_d:regs, r_s:regs, r_c:fcond;

jr, jalr, mfhi, mthi, mflo, mtlo --> &op& r_s:regs;

jalr, tge, tgeu, tlt, tltu, teq, tne, mfc0, dmfc0,
cfc0, mtc0, dmtc0, ctc0, cfc1, mtc1,
mfc2, cfc2, mtc2, ctc2, mult, multu, dmult, dmultu
 --> &op& r_d:regs, r_s:regs;

sll, sra, srl, dsll, dsrl, dsra, dsll32,
dsrl32, dsra32 --> &op& r_d:regs, r_w:regs, imm;

sllv, srlv, srav, movz, movn, dsllv, dsrlv, dsrav,
add, addu, sub, subu, and, or,
xor, nor, slt, sltu, dadd, daddu, dsub, dsubu -->
 &op& r_d:regs, r_w:regs, r_s:regs;

div, divu, ddiv, ddivu -->
 &op& " $0", r_w:regs, r_s:regs;

break, syscall --> &op& imm;

j, jal --> &op& &label&;

bltz, bgez, bltzl, bgezl, bltzal, bgezal, bltzall,
bgezall, bnezl, blezl, bgtzl, blez, bgtz -->
 &op& r_s:regs, &ref& &label&;

tgei, tgeiu, tlti, tltiu, teqi, tnei -->
 &op& r_s:regs, imm;

beq, bne, beql, bnel -->
 &op& r_s:regs, r_t:regs, &label&;

addi, addiu, slti, sltiu, andi, ori, xori,
lui --> &op& r_d:regs, &ref& imm;
```

```
daddi, daddiu --> &op& r_d:regs, r_w:regs,
 &ref& imm;

bc0f, bc0t, bc0fl, bc0tl, bc2f, bc2t, bc2fl,
bc2tl: branch;
branch -> &op& &label&;

bc1f, bc1t, bc1fl, bc1tl: branch, branchc ;
branchc -> &op& r_s:fcond, &label&;

add.s, add.d, sub.s, sub.d, mul.s, mul.d, div.s,
 div.d --> &op& r_d:fregs, r_w:fregs, r_s:fregs;

mfc1, dmfc1, dmtc1, ctc1 -->
 &op& r_t:regs, r_s:fregs;

sqrt.s, sqrt.d, abs.s, abs.d, mov.s, mov.d, neg.s,
neg.d, round.l.s, round.l.d, trunc.l.s, trunc.l.d,
ceil.l.s, ceil.l.d, floor.l.s, floor.l.d,
round.w.s, round.w.d, trunc.w.s, trunc.w.d,
ceil.w.s, ceil.w.d, floor.w.s, floor.w.d, recip.s,
recip.d, rsqrt.s, rsqrt.d, cvt.s.d, cvt.s.w,
cvt.s.l, cvt.d.s, cvt.d.w, cvt.d.l, cvt.w.s,
cvt.w.d, cvt.l.s, cvt.l.d
 --> &op& r_d:fregs, r_s:fregs;

movf.s, movt.s, movf.d, movt.d -->
 &op& r_d:fregs, r_w:fregs, r_s:fcond;

movz.s, movz.d, movn.s, movn.d -->
 &op& r_d:fregs, r_w:fregs, r_s:regs;

c.f.s, c.f.d, c.un.s, c.un.d, c.eq.s, c.eq.d,
c.ueq.s, c.ueq.d, c.olt.s, c.olt.d, c.ult.s,
c.ult.d, c.ole.s, c.ole.d, c.ule.s, c.ule.d,
c.sf.s, c.sf.d, c.seq.s, c.seq.d, c.ngl.s,
c.ngl.d, c.lt.s, c.lt.d, c.nge.s, c.nge.d,
c.le.s, c.le.d, c.ngt.s, c.ngt.d -->
 &op& r_d:fcond, r_w:fregs, r_s:fregs;

lwxc1, ldxc1, swxc1, sdxc1 -->
 &op& r_d:fregs, r_w:regs ( r_s:regs );

pref --> &op& r_h:hints, imm (r_w:regs);
prefx --> &op& r_h:hints, r_w:regs ( r_s:regs );
hints = ( 0, 1, 4, 5, 6, 7 );

madd.s, madd.d, msub.s, msub.d, nmadd.s, nmadd.d,
nmsub.s, nmsub.d -->
 &op& r_d:fregs, r_r:fregs, r_s:fregs, r_t:fregs;

ldl, ldr, lb, lh, lwl, lw, lbu, lhu, lwr, lwu, sb,
sh, swl, sw, sdl, sdr, swr, ll, lwc2, lld, ldc2,
ld, sc, swc2, scd, sdc2, sd -->
 &op& r_d:regs, &ref& imm (r_w:regs);

l.s, l.d, s.s, swc1, s.d, sdc1 -->
 &op& r_d:fregs, imm (r_w:regs);

cache --> &op& r_d:cache_ops, imm (r_w:regs);
cache_ops = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
 10, 11, 13, 15, 16, 17, 18, 19, 20, 21, 23,
 24, 25, 27, 30, 31 );
```

```
struct inst_spec {
  char *inst;                                    /* instruction name */
  char *fmt;                                      /* assembly format */
  short n_ops;                                      /* operand count */
  unsigned nbytes;                                /* instruction size */
  char *namesuffix;                             /* name used in emittor */
  unsigned char op_mask[MAX_BYTES];                      /* opcode mask */

  struct operand {
    char *sym_op;                              /* symbolic name of operand. */
    enum op_type { REG, IMMED, LBL_R, LBL_A } type;          /* op type */
    t_type encoding;                             /* how is operand encoded? */
    unsigned lo;                  /* lowest legal value of field. */
    int nbits;                                  /* number of bits in field. */
    unsigned   mask[MAX_REGS];                             /* register mask */
    int offset;                                   /* offset in instruction */
    sign_type signed_field;                    /* signed or unsigned field? */
    int relative_offset;                /* where relative jumps start
                                           from: offset from end of jump */
    int wants_ref;                     /* generate an emittor to set this */
    int index;                                        /* operand index */
  } ops[MAX_OPS];
};
```

Figure 4: The DERIVE encoding structure. It includes the instruction, formatting string used to generate the instruction, the instruction mask, and a list of operand specifications. These specifications can be easily converted to other languages.

```
{ "decl", "&op& r_1", 1, 1, "",
    { { 0x48, } },
    { { "r_1", REG, IDENT, 0, 3,
        {0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,
         ~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,
         ~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,
         ~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,~0,
         ~0,~0,~0,~0,~0,~0,}, 0, 0, 0, 0},
    },
}
```

Figure 5: DERIVE-emitted specification for one form of the x86 `decl` instruction.