```
v0 := ld state[748]     // load primes from the trace activation record
      st sp[0], v0      // store primes to interpreter stack
v1 := ld state[764]     // load k from the trace activation record
v2 := i2f(v1)           // convert k from int to double
      st sp[8], v1      // store k to interpreter stack
      st sp[16], 0      // store false to interpreter stack
v3 := ld v0[4]          // load class word for primes
v4 := and v3, -4        // mask out object class tag for primes
v5 := eq v4, Array      // test whether primes is an array
      xf v5             // side exit if v5 is false
v6 := js_Array_set(v0, v2, false)  // call function to set array element
v7 := eq v6, 0          // test return value from call
      xt v7             // side exit if js_Array_set returns false.
```

Figure 3. LIR snippet for sample program. This is the LIR recorded for line 5 of the sample program in Figure 1. The LIR encodes the semantics in SSA form using temporary variables. The LIR also encodes all the stores that the interpreter would do to its data stack. Sometimes these stores can be optimized away as the stack locations are live only on exits to the interpreter. Finally, the LIR records guards and side exits to verify the assumptions made in this recording: that `primes` is an array and that the call to set its element succeeds.

```
mov edx, ebx(748)       // load primes from the trace activation record
mov edi(0), edx         // (*) store primes to interpreter stack
mov esi, ebx(764)       // load k from the trace activation record
mov edi(8), esi         // (*) store k to interpreter stack
mov edi(16), 0          // (*) store false to interpreter stack
mov eax, edx(4)         // (*) load object class word for primes
and eax, -4             // (*) mask out object class tag for primes
cmp eax, Array          // (*) test whether primes is an array
jne side_exit_1         // (*) side exit if primes is not an array
sub esp, 8              // bump stack for call alignment convention
push false              // push last argument for call
push esi                // push first argument for call
call js_Array_set       // call function to set array element
add esp, 8              // clean up extra stack space
mov ecx, ebx            // (*) created by register allocator
test eax, eax           // (*) test return value of js_Array_set
je side_exit_2          // (*) side exit if call failed
...
side_exit_1:
mov ecx, ebp(-4)        // restore ecx
mov esp, ebp            // restore esp
jmp epilog              // jump to ret statement
```

Figure 4. x86 snippet for sample program. This is the x86 code compiled from the LIR snippet in Figure 3. Most LIR instructions compile to a single x86 instruction. Instructions marked with `(*)` would be omitted by an idealized compiler that knew that none of the side exits would ever be taken. The 17 instructions generated by the compiler compare favorably with the 100+ instructions that the interpreter would execute for the same code snippet, including 4 indirect jumps.

i=2. This is the first iteration of the outer loop. The loop on lines 4-5 becomes hot on its second iteration, so TraceMonkey enters recording mode on line 4. In recording mode, TraceMonkey records the code along the trace in a low-level compiler intermediate representation we call *LIR*. The LIR trace encodes all the operations performed and the types of all operands. The LIR trace also encodes *guards*, which are checks that verify that the control flow and types are identical to those observed during trace recording. Thus, on later executions, if and only if all guards are passed, the trace has the required program semantics.

TraceMonkey stops recording when execution returns to the loop header or exits the loop. In this case, execution returns to the loop header on line 4.

After recording is finished, TraceMonkey compiles the trace to native code using the recorded type information for optimization. The result is a native code fragment that can be entered if the interpreter PC and the types of values match those observed when trace recording was started. The first trace in our example, T_{45}, covers lines 4 and 5. This trace can be entered if the PC is at line 4, `i` and `k` are integers, and `primes` is an object. After compiling T_{45}, TraceMonkey returns to the interpreter and loops back to line 1.

i=3. Now the loop header at line 1 has become hot, so TraceMonkey starts recording. When recording reaches line 4, TraceMonkey observes that it has reached an inner loop header that already has a compiled trace, so TraceMonkey attempts to nest the inner loop inside the current trace. The first step is to call the inner trace as a subroutine. This executes the loop on line 4 to completion and then returns to the recorder. TraceMonkey verifies that the call was successful and then records the call to the inner trace as part of the current trace. Recording continues until execution reaches line 1, and at which point TraceMonkey finishes and compiles a trace for the outer loop, T_{16}.