Software Model Checking without Source Code*

Sagar Chaki James Ivers
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA
(chaki|jivers)@sei.cmu.edu

Abstract

We present a framework, called AIR, for verifying safety properties of assembly language programs via software model checking. AIR extends the applicability of predicate abstraction and counterexample guided abstraction refinement to the automated verification of low-level software. By working at the assembly level, AIR allows verification of programs for which source code is unavailable–such as legacy and COTS software–and programs that use features–such as pointers, structures, and object-orientation–that are problematic for source-level software verification tools. In addition, AIR makes no assumptions about the underlying compiler technology. We have implemented a prototype of AIR and present encouraging results on several non-trivial examples.

1 Introduction

Over the past decade, there has been considerable advancement in the theory and practice of automated formal software verification. One of the most promising paradigms to emerge in this area is software model checking (SMC) [3] – a combination of counterexample guided abstraction refinement [11] with predicate abstraction [18]. SMC verifies that a program P satisfies a specification ϕ iteratively, as follows:

1. **(Abstraction)** Construct a conservative model M from P via predicate abstraction. Go to Step 2.
2. **(Verification)** Model check $M \models \phi$. If this is the case, then terminate with result $P \models \phi$. Otherwise let CE be a counterexample to $M \models \phi$ returned by the model checker. Go to Step 3.
3. **(Validation)** Check whether CE corresponds to some concrete behavior of P. If this is the case, then we obtain a real counterexample and terminate with $P \not\models \phi$. Otherwise, CE is a spurious counterexample. Go to Step 4.
4. **(Refinement)** Construct a more precise model M' that does not admit CE as a behavior and repeat from Step 2 with $M = M'$.

Variations of the above process have been investigated by several research groups [3, 19, 9] with considerable success on *source code* derived from real-life examples. However, there has been considerably less work on applying SMC to verify *machine-level* programs. In this paper, we show that in spite of the absence of high-level information, such as variable names and branch conditions, the effectiveness of SMC extends to even low-level software. More specifically, we present a SMC-based procedure for verifying safety properties of PowerPC™ assembly programs. Our approach, which we call Assembly Iterative Refinement or AIR, consists of two broad phases:

1. **Decompilation:** In this stage, we translate the target assembly program A and safety property ϕ_A into an equivalent C program P and safety property ϕ_P. In essence, we treat each register as a global variable. Each procedure in A leads to a corresponding procedure in P, and each assembly instruction produces one or more C statements. We present further details of the decompilation process in Section 3. Note that C is particularly suited as the target language of decompilation because: (i) C supports bit-level operations that are critical for preserving the semantics of assembly instructions during decompilation; (ii) we are able to build on existing infrastructures for C verification to perform the next stage of AIR.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 36-45

*This research was conducted as part of the Predictable Assembly from Certifiable Components (PACC) initiative at the SEI. The SEI is a Federally Funded Research and Development Center sponsored by the US Dept. of Defense and operated by Carnegie Mellon University.

2. **Verification:** In this stage, we use SMC to check $P \models \phi_P$. Since decompilation is semantics-preserving, the result obtained in this stage for P also applies to the original assembly program A. Furthermore, any counterexample obtained with respect to P is transformed to a corresponding counterexample with respect to A. Further details about verification can be found in Section 4.

AIR extends the applicability of SMC to assembly program verification, and yields several tangible benefits. First, AIR does not require source code, and thus is applicable to software – such as legacy, proprietary, and commercial-off-the-shelf (COTS) software – for which source code is often not available. Second, unlike source code analysis, AIR analyzes exactly what is to be executed and makes no assumptions about any compiler technology being used. Thus, it eliminates the need to ensure compiler correctness, and reduces the size of the trusted computing base. This is especially desirable when analyzing safety-critical systems. Third, AIR is not tied to any specific high-level programming language, and consequently is more versatile than source-code verification. In particular, AIR is able to sidestep features, such as pointers, structures, and object-orientation, which are problematic for source-level analysis tools. Finally, since AIR decompilation is semantics preserving and targets the C language, it enables us to leverage existing and emerging C analysis tools for verification. Thus, even though we experiment with SMC-based tools, other C verifiers (e.g., CBMC [7] and F-Soft [21]) are also applicable.

We have implemented AIR and obtained encouraging results on several non-trivial benchmarks derived from Linux device drivers and an embedded OS. Further details can be found in Section 5. The rest of this paper is organized as follows. In Section 2, we survey related work. The decompilation and verification stages of AIR are described in Sections 3 and 4, respectively. Finally, we present experimental results in Section 5 and conclude in Section 6.

2 Related Work

Following SLAM [3], several other projects – e.g., MAGIC [9] and BLAST [19] – have investigated the use of the SMC paradigm for C source code verification. A number of projects – such as SPIN [20], Java PathFinder [32], BANDERA [17], BOGOR [16], Behave! [4], and ZING [1] – have also looked at software verification, but not necessarily via SMC. Instead, their focus has been on other languages, such as Java, and other program features, such as concurrency. Decompilers have been traditionally developed for binary understanding and reverse engineering, and not for verification *per se*. Nevertheless, the use of decompilation for verification has been suggested by Breuer and Bowen [6], and by Curzon [14] for verifying micro-code.

The verification of low-level software [12, 27] has also received a lot of attention. A number of approaches are based on either theorem proving, type checking, or static analysis. For example, Boyer and Yu have verified object code for the MC68020 processor using the Nqthm theorem prover [5]. Yu [33] has proposed the use of certified assembly programming and type preserving translations for ensuring the safety of low-level code. His techniques are powerful but require considerable manual intervention, e.g., via type annotations and the use of proof assistants. Yu and Shao [34] have also proposed a logic based type system for the static verification of concurrent assembly programs.

Reps et al. [30] have used static analysis algorithms to recover information about the contents of memory locations and how they are manipulated by executables. They have also created CodeSurfer/x86, a prototype tool for browsing, inspecting and analyzing x86 binaries. Our technique is based on model checking, is completely automated, and targets PowerPC$^{\text{TM}}$ assembly code. Balakrishnan et al. [2] have used model checking to analyze stripped device driver executables. Their approach is not based on decompilation to C, but on a tight combination of their own model checker and control-flow-graph-based internal representation for the target executables.

Another approach for ensuring the correctness of low-level programs is source code verification combined with compiler validation, i.e., proving that the compiler always produces a target code which correctly implements the source code. In practice, proving compiler correctness is extremely tedious. Furthermore, any change in the compiler necessitates its revalidation. Our technique is impervious to the underlying compiler technology. A complementary technique is translation validation [28] where instead of validating the compiler, each individual run of the compiler is followed by a validation phase which verifies that the target code produced on that run correctly implements the submitted source program. Both compiler validation and translation validation assume that the source code is available and has been independently verified. Our approach does not require such an assumption.

3 Decompilation

The first stage of AIR is the decompilation of the target assembly program A into an equivalent C program P. In this section we describe the decompilation procedure using a small assembly program as a running example. For ease of understanding, we start with the source code from which A was compiled. Fig. 1 shows a small C code fragment P_0 on the left and the result of compiling it with `gcc` on the right. P_0 is derived from MICRO-C, a lightweight operating system for real-time embedded applications. For brevity and simplicity we eliminated some irrelevant code from the actual MICRO-C sources. Also, the assembly A on the right of Fig. 1 does not contain some book-keeping information generated by `gcc`.

We use the 32-bit variant of the PowerPC™ instruction set architecture (ISA) and assume little-endian mode. The PowerPC™ architecture defines 32 32-bit general purpose registers (GPRs) – referred to in A as `%r0` through `%r31`. In addition, there are 32 64-bit floating point registers (FPRs) – referred to as `%f0` through `%f31` – and a few special registers (SPRs), e.g., condition register (`%cr`), link register (`%lr`), etc. An assembly program consists of a set of blocks, each beginning with a label. A label is either a procedure name (`OSMemNameSet`) or begins with a dot (`.L1 ... .L4`). The procedures `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL` acquire and release a global lock for achieving mutual exclusion. We wish to verify whether our program satisfies the following property: **(Safety)** `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL` are invoked alternately, beginning with a call to `OS_ENTER_CRITICAL`. Note that **Safety** is representative of a general class of safety specifications with respect to the acquisition and release of resources. Also, our example program does not satisfy **Safety**. Indeed, if the conditions of the first two `if` statements are both satisfied, then `OS_EXIT_CRITICAL` gets called twice in a row without any intervening call to `OS_ENTER_CRITICAL`. One possible fix for this problem is to add a `return` statement as indicated by the comment in the C code.

3.1 From assembly to C

The decompilation process converts the assembly program A to a C program, using the following general strategy:

- The 32 GPRs are declared as global `int` variables `r0` through `r31`. The 32 FPRs are declared as global `double` variables `f0` through `f31`. The SPRs are also declared as `int` variables `cr, lr` and so on. All integer data is assumed to be in signed (two's complement) 32-bit format and all double data is assumed to be in IEEE 64-bit double precision format.
- Each label corresponding to a procedure name yields a procedure declaration. Since an assembly program passes and returns all values via registers (i.e., global variables), our procedures are `void-void`, i.e., they have no parameters or return values. In our example, we obtain a single procedure declaration:

```
void OSMemNameSet(void)
```

```
struct os_mem {
  void *OSMemAddr ;
  void *OSMemFreeList ;
  unsigned int OSMemBlkSize ;
  unsigned int OSMemNBlks ;
  unsigned int OSMemNFree ;
  char OSMemName[32] ;
};

typedef struct os_mem OS_MEM;

void OSMemNameSet(OS_MEM *pmem,
        char *pname,unsigned char *err )
{
  unsigned char len;
  OS_ENTER_CRITICAL();
  if ((unsigned int )pmem == (unsigned int )((OS_MEM *)0)) {
    OS_EXIT_CRITICAL();
    (*err) = 116;
    //bug : there should most likely be a return here
    //return;
  }
  if ((unsigned int )pname == (unsigned int )((char *)0)) {
    OS_EXIT_CRITICAL();
    (*err) = 15;
    return;
  }
  if ((int )len > 31) {
    OS_EXIT_CRITICAL();
    (*err) = 119;
    return;
  }
  OS_EXIT_CRITICAL();
  (*err) = 0;
  return;
}
```

```
OSMemNameSet:
  stwu %r1,-48(%r1)
  mflr %r0
  stw %r31,44(%r1)
  stw %r0,52(%r1)
  mr %r31,%r1
  stw %r3,8(%r31)
  stw %r4,12(%r31)
  stw %r5,16(%r31)
  bl OS_ENTER_CRITICAL
  lwz %r0,8(%r31)
  cmpwi %cr7,%r0,0
  bne %cr7,.L2
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,116
  stb %r0,0(%r9)
.L2:
  lwz %r0,12(%r31)
  cmpwi %cr7,%r0,0
  bne %cr7,.L3
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,15
  stb %r0,0(%r9)
  b .L1
.L3:
  lbz %r0,20(%r31)
  rlwinm %r0,%r0,0,0xff
  cmplwi %cr7,%r0,31
  ble %cr7,.L4
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,119
  stb %r0,0(%r9)
  b .L1
.L4:
  bl OS_EXIT_CRITICAL
  lwz %r9,16(%r31)
  li %r0,0
  stb %r0,0(%r9)
.L1:
  lwz %r11,0(%r1)
  lwz %r0,4(%r11)
  mtlr %r0
  lwz %r31,-4(%r11)
  mr %r1,%r11
  blr
```

Figure 1: A running example.

- Each label beginning with a dot results in a corresponding label in the C program. We strip off the initial dot to conform to valid ANSI-C syntax. Thus, the C program generated in our example contains four labels `L1` through `L4`.
- Each assembly instruction gets translated to an equivalent sequence of C statements. In the rest of this section, we describe the translation process for the instructions that appear in our example. Note that the size of the resulting C program is linear in the size of the input assembly program.

3.2 Translating assembly instructions

PowerPC™ follows the Reduced Instruction Set Computer (RISC) or the load-store paradigm. Thus, there are no arithmetic, logical, or control-flow instructions that operate directly on data stored in mem-

ory. All operations are performed on GPRs, FPRs, and SPRs. In order to operate on memory data, the operands are loaded explicitly into registers, and the result is stored explicitly back to memory.

<table>
<tr><th>Assembly</th><th>C statements</th></tr>
<tr><td colspan="2">Loads and stores</td></tr>
<tr><td><code>lwz %r0,8(%r31)</code></td><td><code>r0 = *((int*)(r31 + 8));</code></td></tr>
<tr><td><code>li %r0,116</code></td><td><code>r0 = 116;</code></td></tr>
<tr><td><code>lbz %r0,20(%r31)</code></td><td><code>r0 = (*((int*)(r31 + 20))) & (0xff);</code></td></tr>
<tr><td><code>stwu %r1,-48(%r1)</code></td><td><code>*((int*)(r1 - 48)) = r1; r1 = r1 - 48;</code></td></tr>
<tr><td><code>stw %r31,44(%r1)</code></td><td><code>*((int*)(r1 + 44)) = r31;</code></td></tr>
<tr><td><code>stb %r0,0(%r9)</code></td><td><code>*((int*)(r9 + 0)) =
((*((int*)(r9 + 0))) & 0xffffff00) | (r0 & 0xff);</code></td></tr>
<tr><td colspan="2">Register operations</td></tr>
<tr><td><code>mr %r31,%r1</code></td><td><code>r31 = r1;</code></td></tr>
<tr><td><code>mflr %r0</code></td><td><code>r0 = lr;</code></td></tr>
<tr><td><code>mtlr %r0</code></td><td><code>lr = r0;</code></td></tr>
<tr><td><code>rlwinm %r0,%r0,0,255</code></td><td><code>r0 = (((r0 >> 32) & 0) | ((r0 << 0) & 0xffffffff))
& (0xff);</code></td></tr>
<tr><td><code>cmpwi %cr7,%r0,0</code></td><td><code>cr = (r0 < 0) ? (cr | 0x8) : (cr & 0xfffffff7);
cr = (r0 > 0) ? (cr | 0x4) : (cr & 0xfffffffb);
cr = (r0 == 0) ? (cr | 0x2) : (cr & 0xfffffffd);</code></td></tr>
<tr><td><code>cmplwi %cr7,%r0,31</code></td><td><code>cr = (r0 >= 0) && (r0 < 31) ?
(cr | 0x8) : (cr & 0xfffffff7);
cr = (r0 < 0) || (r0 > 31) ?
(cr | 0x4) : (cr & 0xfffffffb);
cr = (r0 >= 0) && (r0 == 31) ?
(cr | 0x2) : (cr & 0xfffffffd);</code></td></tr>
<tr><td colspan="2">Conditional and unconditional jumps</td></tr>
<tr><td><code>b .L1</code></td><td><code>goto L1;</code></td></tr>
<tr><td><code>ble %cr7,.L4</code></td><td><code>if(!(cr & 0x4)) goto L4;</code></td></tr>
<tr><td><code>bne %cr7,.L2</code></td><td><code>if(!(cr & 0x2)) goto L2;</code></td></tr>
<tr><td><code>bl OS_ENTER_CRITICAL</code></td><td><code>OS_ENTER_CRITICAL();</code></td></tr>
<tr><td><code>blr</code></td><td><code>return;</code></td></tr>
</table>

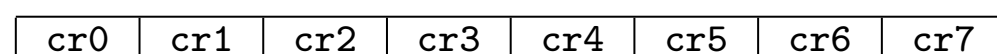
Figure 2: Translation schema from assembly instructions to C statements.

Fig. 2 shows a table with assembly instructions on the left and the corresponding C statements on the right. Among the instructions in Fig. 2, the following have straightforward translations: `li = load immediate`, `mr = move register`, `mflr = move from link register`, `mtlr = move to link register`, `b = branch`, `bl = branch link`, and `blr = branch link return`. The translations for the other instructions can be understood on the basis of their semantics, as described below:

- `lwz = load word and zero`: loads a word from the memory location denoted by the second argument to the register denoted by the first argument.
- `lbz = load byte and zero`: loads a byte from the source memory location S denoted by the second argument to the target register T denoted by the first argument. The higher-order 24 bits of T are set to zero. Due to little-endianness, if we load an integer from S, our desired byte will be laid out at the lower order end. Now we have to zero out the higher-order 24 bits.
- `stwu = store word with update`: stores the word in source register S denoted by the first argument to the target memory location T denoted by the second argument, and then sets the value of S to T.
- `stw = store word`: stores the word in source register S denoted by the first argument to the target memory location T denoted by the second argument.
- `stb = store byte`: stores the lowest byte in source register S denoted by the first argument to the target memory location T denoted by the second argument. Again due to little-endianness, we load the current word stored at T, replace its lowest byte with the lowest byte of S, and store the new value back to T.

- `rlwinm = rotate left word immediate then AND with mask`: rotates left the contents of the source register *S* (denoted by the second argument) by the number of bits *B* (denoted by the third argument), then logically ANDs the rotated data with a 32-bit mask *BM* (denoted by the fourth argument), and stores the result in the target register *T* (denoted by the first argument).

To understand the comparison and jump instructions, we note that the condition register `cr` is logically partitioned into eight sub-registers `cr0 ... cr7`. The sub-registers are numbered from the higher order bits to the lower order bits of `cr` as shown by the following diagram.



cr0	cr1	cr2	cr3	cr4	cr5	cr6	cr7

Thus, `cr7` denotes the lowest four bits of `cr`. Further, suppose that the results of a comparison between X and Y are stored in a condition sub-register R. Then the bits of R must be interpreted as follows. The highest bit is 1 if and only if $X < Y$, the next bit is 1 if and only if $X > Y$, and the next bit is 1 if and only if $X = Y$. The lowest bit is reserved for overflows. We now present the translation scheme for the remaining instructions.

- `cmpwi = compare word immediate`: compares the contents of the register denoted by the second argument with the integer denoted by the third argument treating both values as signed integers, and stores the result in the condition sub-register denoted by the first argument.

- `cmplwi = compare logical word immediate`: compares the contents of the register denoted by the second argument with the integer denoted by the third argument treating both values as unsigned integers, and stores the result in the condition sub-register denoted by the first argument. Since all our C variables are signed, we guard the C statement to be executed on conditions that check for negative values in addition to the actual comparison being performed.

- `ble = branch less equal` : jumps to the label denoted by the second argument if the condition sub-register denoted by the first argument indicates a "less or equal" comparison result.

- `bne = branch not equal` : jumps to the label denoted by the second argument if the condition sub-register denoted by the first argument indicates a "not equal" comparison result.

This completes the description of the translation scheme for the instructions appearing in our example. In total, to perform our experiments, translations were written for eighty nine PowerPC assembly instructions. Complete details of the PowerPC™ ISA are available [29] online.

4 Verification

Once the target assembly program has been decompiled to a C program *P*, the second stage of AIR involves the verification of *P* via SMC. For our experiments, we used the COPPER [13] SMC tool for verification. For the purpose of AIR we only required the ability of COPPER to check for trace containment between sequential C programs and finite state machines. In addition, though our familiarity with COPPER lead to its use in our experiments, any other C verification tool based on the SMC paradigm, such as SLAM [3], MAGIC [9], or BLAST [19], is suitable for use in the verification stage of AIR.

Indeed, the main challenge involved in the use of SMC for AIR verification is tool-independent, and arises from the need for precise handling of bit-level semantics during SMC[1]. In all cases, the handling of bit-level semantics is delegated to the theorem prover used during predicate abstraction. Most often, the

[1]We note that the C bounded model checker CBMC [10] does obey precise bit-level semantics but does not use SMC.

theorem prover (usually Simplify [26] or Vampyre [31]) treats the C bit-wise operators as uninterpreted functions. For source code verification, this is not a major roadblock since many properties verified on source code do not rely on the precise interpretation of bitwise operators. However, in the case of AIR, precise interpretation of bitwise operations is crucial for verifying the C programs generated via decompilation. In our initial experiments, not a single non-trivial property could be verified by leaving the bitwise operators uninterpreted. We attempted several solutions to this problem, as discussed next.

Solution 1: Adding axioms. First, we added extra axioms about C bitwise operators to assist Simplify, the default theorem prover used by COPPER. Unfortunately, this solution is ad hoc, since we had no way of knowing if we had added enough axioms. Also, the performance of Simplify, in terms of both time and memory consumption, degraded dramatically with increasing numbers of axioms. Ultimately, we concluded that this approach would not scale to realistic programs.

Solution 2: Syntactic analysis. Next, we augmented COPPER with a set of syntactic bit-level analyses. Specifically, before invoking Simplify, COPPER performs some simplifications on the formula whose validity has to be checked. The transformations are targeted at specific patterns that arise in formulas due to the structure of assembly programs. For example, a common query to Simplify is the validity of `((E | 0x4) >> 2) & (0x1)`, where `E` is some C expression. Our technique is able to convert such formulas to `0x1`, whose validity can then be easily decided by Simplify. We call this solution *uninterpreted* since all bitwise operators are left completely uninterpreted by the theorem prover.

Solution 3: Using a bit-vector decision procedure. We also compared the above approach to the idea of replacing Simplify with the bit-vector decision procedure CPROVER [22] (we also experimented with CVC Lite [15] but found CPROVER to be faster). We tried two variations of this idea. In the *interpreted* variation, all formulas containing bitwise operators are solved using CPROVER. In the *semi-interpreted* variation, formulas containing bitwise operators are first solved using Simplify. CPROVER is used only if Simplify is unable to decide validity conclusively.

In our example, AIR is able to successfully report the bug in MICRO-C. When the bug is fixed, in accordance with the suggestion in the comment, AIR successfully verifies the safety property. Also our experiments indicate that the *uninterpreted* approach yields the best performance over a set of realistic benchmarks. We now present full details of the empirical validation of our technique.

5 Experimental Validation

We experimented with a set of benchmarks derived from MICRO-C and Linux device drivers. All our experiments were performed on a single core 2.4 GHz Pentium computer running RedHat 9. We imposed a time limit of one hour, and memory limit of one GB. We derived a set of eleven benchmarks – one from MICRO-C, and the rest from Linux 2.6.11.10 kernel drivers – by compiling C source code with `gcc-3.2`. For each example, we checked that a certain "lock" was being acquired and released properly. The nature of the lock varied with the example. For MICRO-C, the lock was an invocation of `OS_ENTER_CRITICAL`, while for the Linux drivers it was a call to `spin_lock`, `spin_lock_irq` or `spin_lock_irqsave`. The "unlock" was derived accordingly.

We initially observed that COPPER is easily able to verify the safety property for all our benchmarks because the locks and unlocks are paired up syntactically. In other words, an analysis of the control flow graph suffices and no further predicate abstraction is necessary. To make our benchmarks more interesting, we added data dependencies between the locks and unlocks. Essentially we guarded the

Name		Interpreted			Semi-Interpreted			Uninterpreted			BLAST			
	KLOC	T	M	I	T	M	I	T	M	I	CE	T	M	I
Micro-C	10	*	164	-	*	164	-	**305**	**70**	31	-	*	67	-
aha152x	33	2319	121	16	*	107	-	**198**	**74**	16	-	*	141	-
DAC960	45	*	193	-	*	142	-	**520**	**107**	16	×	61	128	6278
devices	17	1018	41	17	3523	45	18	**350**	**40**	19	×	202	60	96701
ide	26	510	61	15	557	61	15	**25**	**34**	15	-	*	105	-
ipr	35	*	285	-	*	288	-	**310**	**79**	19	×	30	100	1230
message	21	194	28	26	145	28	25	**29**	**27**	26	×	16	63	831
mxser	22	699	53	19	547	52	19	**129**	**46**	19	×	6	51	1302
synclink	34	123	33	15	106	33	15	**15**	**30**	15	×	40	89	4292
tg3	61	988	77	18	907	77	18	**168**	**70**	21	×	84	152	11496
tlan	31	312	63	7	242	63	7	**73**	**49**	7	×	25	88	2062

Figure 3: Results for non-buggy benchmarks. KLOC = 1000 lines of assembly; T = time in seconds; M = memory in MB; I = # of iterations; * means that the resource was exhausted; - means that no measurement was available; $\times$ means that the counterexample returned by BLAST is spurious. Best figures are highlighted.

Name		Interpreted			Semi-Interpreted			Uninterpreted			BLAST			
	KLOC	T	M	I	T	M	I	T	M	I	CE	T	M	I
Micro-C	10	*	164	-	*	164	-	*	70	-	-	*	67	-
aha152x	33	357	53	10	321	53	10	**84**	**49**	18	×	40	98	8681
DAC960	45	1208	138	13	1017	138	13	**388**	**107**	13	×	70	130	6272
devices	17	1260	*	14	*	46	-	*	43	-	×	281	59	96697
ide	26	*	75	-	*	67	-	*	39	-	-	*	105	-
ipr	35	2009	280	6	1949	280	6	**205**	**76**	6	×	37	99	1230
message	21	62	26	11	76	26	11	**6**	**24**	11	×	16	63	831
mxser	22	115	50	6	108	50	6	**76**	**45**	6	×	8	50	1302
synclink	34	120	35	10	212	36	16	**27**	**32**	16	×	34	89	4292
tg3	61	2115	77	17	849	77	11	**219**	**68**	12	×	88	152	11496
tlan	31	362	63	7	354	63	7	**98**	**49**	7	×	34	89	2062

Figure 4: Results for buggy benchmarks. KLOC = KLOC of assembly; T = time in seconds; M = memory in MB; I = # of iterations; * means that the resource was exhausted; - means that no measurement was available; $\times$ means that the counterexample returned by BLAST is spurious. Best figures are highlighted.

locks and unlocks with a non-deterministic value. Since the same value guards both lock and unlock the examples are still correct.

We experimented using the *interpreted*, *semi-interpreted* and *uninterpreted* approaches presented in Section 4. In the first two cases, the syntactic simplifications were also applied. As a control, we also used BLAST version 1.0. The results of our experiments with these benchmarks are summarized in Fig. 3. Next, for each benchmark, we created a buggy version by artificially inserting errors and repeated our experiments. The results for our experiments with the the buggy examples are summarized in Fig. 4.

We observe that the *uninterpreted* approach exhibits the best overall performance. The *interpreted* approach beats the *semi-interpreted* approach by successfully proving more examples. This indicates that almost all the formulas involving bitwise operators could not be proved by Simplify and hence had to be further delegated to CPROVER. This is also consistent with our initial failure with only Simplify (without the syntactic simplifications). BLAST returns counterexamples for both the correct and buggy examples. Upon closer inspection, all counterexamples returned by BLAST are found to be spurious. We note that this is essentially due to BLAST's dependency on Simplify.

6 Discussion and Conclusion

We have presented AIR, a framework for verifying safety properties of assembly language programs via SMC. We have proposed a number of approaches for more precise handling of bit-level semantics during SMC and empirically validated their relative effectiveness. Overall, our experiments indicate that AIR is effective on real-life benchmarks derived from an embedded OS and Linux device drivers.

It is worthwhile to consider a few issues concerning the AIR approach. Decompiling an assembly program, though much less difficult than verifying the resulting C program, requires careful attention to detail. Whether the target platform is big or little endian and whether a 0 or a 1 is shifted in on >> operations on signed integers are two such intricacies. Correctly modeling elements of the program's environment such as the contents of the PowerPC™ machine state register are more complicated. Other decisions must be guided by the capabilities of the model checker; for example, choosing whether to denote a comparison that treats two operands as unsigned quantities by using type casting and a simple comparison or by using more predicates and checking different conditions depending on the signs of the operands. Moreover, the correct handling of pointer aliasing during verification is crucial for maintaining the overall soundness of AIR. Finally, though AIR is applicable to any assembly program, it is not necessarily a good choice in many cases. The broad applicability of AIR comes at a cost in usability. Encoding properties in terms of elements of the assembly program may be more difficult then encoding the same property against, for example, a C program. Similarly, interpreting a counterexample expressed in terms of an assembly program is likely to be more difficult. However, these concerns are much less important when it is sufficient to know whether a property holds (or not), or when source is unavailable.

In summary, we believe that the AIR approach has important ramifications for the development of effective low-level software verification techniques. Specifically, AIR is applicable to verifying other low-level languages such as Java bytecode and MSIL. Programs in these languages generally contain additional information (such as variable names) that should further increase AIR's effectiveness. AIR is also adoptable for the purpose of using certifying model checking [23] for proof carrying code (PCC) [25]. Certifying model checking in combination with abstraction has been used [24, 8] to construct invariants and ranking functions for the purpose of certifying source code. By generating source code from binaries, AIR enables us to leverage the above technology for the PCC-style certification of binaries. Finally, there is a growing trend of implementing hardware functionality using software, such as microcode, in the domain of hardware-software co-design. We believe that AIR would also be applicable for the verification of such low-level programs.

References

[1] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV*, pages 484–487, 2004.

[2] G. Balakrishnan and T. W. Reps. "Analyzing Stripped Device-Driver Executables". In *Proceedings of TACAS*, pages 124–140, 2008.

[3] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of SPIN*, pages 103–122, 2001.

[4] BEHAVE! website. `http://research.microsoft.com/behave`.

[5] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM (JACM)*, (1):166–192, 1996.

[6] P. T. Breuer and J. P. Bowen. Generating Decompilers. RUCS Technical Report RUCS/1998/TR/010/A, Department of Computing, The University of Reading, 1998.

[7] CBMC website. `http://www.cprover.org/cbmc`.

[8] S. Chaki. SAT-Based Software Certification. In *Proceedings of TACAS*, pages 151–166, 2006.

[9] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proceedings of ICSE*, pages 385–395, 2003.

[10] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of TACAS*, pages 168–176, 2004.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, (5):752–794, 2003.

[12] D. L. Clutterbuck and B. A. Carre. The verification of low-level code. *Software Engineering Journal (SEJ)*, (3):97–111, 1988.

[13] Copper website. `http://www.sei.cmu.edu/pacc/copper.html`.

[14] P. Curzon. *A Structured Approach to the Verification of Low Level Microcode*. PhD thesis, University of Cambridge, Computer Laboratory, 1991. Tech report no. 215.

[15] CVC Lite website. `http://verify.stanford.edu/CVCL`.

[16] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using The Bogor Extensible Model Checking Framework. In *Proceedings of CAV*, pages 148–152, 2005.

[17] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of ICSE*, pages 177–187, 2001.

[18] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV*, pages 72–83, 1997.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of POPL*, pages 58–70, 2002.

[20] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. 2003.

[21] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software Verification Platform. In *Proceedings of CAV*, pages 301–306, 2005.

[22] D. Kroening. Application Specific Higher Order Logic Theorem Proving. In *Proceedings of the Verification Workshop (VERIFY'02)*, pages 5–15, 2002.

[23] K. S. Namjoshi. Certifying Model Checkers. In *Proceedings of CAV*, pages 2–13, 2001.

[24] K. S. Namjoshi. Lifting Temporal Proofs through Abstractions. In *Proceedings of VMCAI*, pages 174–188, 2003.

[25] G. C. Necula. Proof-Carrying Code. In *Proceedings of POPL*, pages 106–119, 1997.

[26] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.

[27] I. O'Neill, D. Clutterbuck, P. Farrow, P. Summers, and W. Dolman. The formal verification of safety-critical assembly code. In *Proceedings of the International Federation of Automatic Control Safety of Computer Control Systems Conference (SAFECOMP '88)*, pages 115–120, 1988.

[28] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proceedings of TACAS*, pages 151–166, 1998.

[29] PowerPC™ ISA. `http://www.nersc.gov/vendor_docs/ibm/asm/mastertoc.htm`.

[30] T. W. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A Next-Generation Platform for Analyzing Executables. In *Proceedings of the third Asian Symposium on Programming Languages and Systems (APLAS '05)*, pages 212–229, 2005.

[31] Vampyre website. `http://www-cad.eecs.berkeley.edu/~rupak/Vampyre`.

[32] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE*, pages 3–12, 2000.

[33] D. Yu. *Safety Verification of Low-Level Code*. PhD thesis, Graduate School. of. Yale University, 2004.

[34] D. Yu and Z. Shao. Verification of Safety Properties for Concurrent Assembly Code. In *Proceedings of the 2004 International Conference on Functional Programming (ICFP '04)*, pages 175–188, 2004.