Editorial Manager(tm) for Innovations in Systems and Software Engineering
Manuscript Draft

Manuscript Number:

Title: Software Model Checking without Source Code

Article Type: SI: NFM 2009

Keywords: Software verification; Model checking; Assembly programs; Abstraction; Iterative Refinement

Corresponding Author: Sagar Chaki,

Corresponding Author's Institution: Carnegie Mellon University

First Author: Sagar Chaki

Order of Authors: Sagar Chaki; James Ivers

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.

Manuscript
Click here to download Manuscript: asm-ver.tex

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
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

**Innovations in Systems and Software Engineering: a NASA Journal manuscript No.**
(will be inserted by the editor)

# Software Model Checking without Source Code

**Sagar Chaki · James Ivers**

Received: date / Accepted: date

**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.

**Keywords** Software verification · Model checking · Assembly programs · Abstraction · Iterative Refinement

## 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 $\phi$ iteratively, as follows:

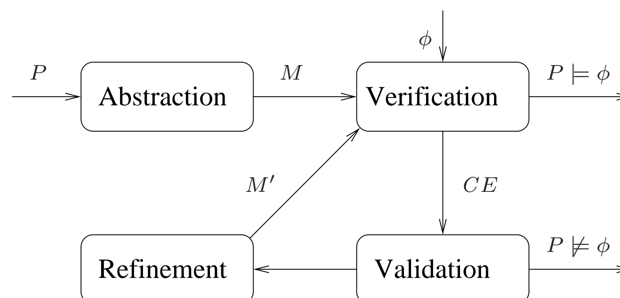



**Fig. 1** Overview of Software Model Checking (SMC).

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 $M = M'$. Typically, $M'$ is constructed via predicate abstraction of $P$ using a new set of predicates derived from the spurious counterexample $CE$.

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.

Sagar Chaki
Software Engineering Institute, Pittsburgh, PA, USA
Tel.: +1-412-268-1436, Fax: +1-412-268-5758
E-mail: chaki@sei.cmu.edu

James Ivers
Software Engineering Institute, Pittsburgh, PA, USA
Tel.: +1-412-268-7793, Fax: +1-412-268-5758
E-mail: jivers@sei.cmu.edu

The overall SMC process is shown in Figure 1. Variations of SMC 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$^{\text{TM}}$ 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 $\phi_A$ into an equivalent C program $P$ and safety property $\phi_P$. Decompilation is semantics-preserving, and based on the following core ideas:
   - Each register is transformed to a global C variable. The type of the variable is derived from the type of the corresponding register so that the bit-width is preserved and appropriate operations are enabled. For example, 32-bit general purpose registers are transformed to 32-bit `int` variables, and 64-bit floating point registers are transformed to 64-bit `double` variables.
   - Each procedure in the assembly program $A$ is transformed to a corresponding procedure in the C program $P$. We will see later that procedure blocks in PowerPC$^{\text{TM}}$ assembly code can be identified in a straightforward manner.
   - Each assembly instruction is transformed to one or more C statements. This transformation is based on the semantics of the assembly instruction.

   We present further details of the decompilation process in Section 3. Note that C is particularly suitable 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.
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.

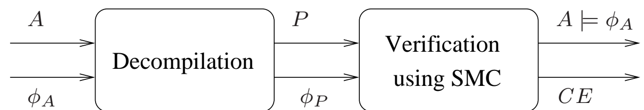



**Fig. 2** Overview of Assembly Iterative Refinement (AIR).

Figure 2 gives an overview of AIR. 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 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 [33], 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. In particular, both Java PathFinder and SPIN have been used effectively to analyze low level concurrent programs (in Java bytecode and Promela, respectively).

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, 28] 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 [34] 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 [35] have also proposed a logic based type system for the static verification of concurrent assembly programs.

Reps et al. [31] 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™ 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 representations 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 [29], where instead of validating the compiler, each individual run of the compiler is followed by a validation phase which checks 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. 3 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. 3 does not contain some book-keeping information generated by `gcc`.

### 3.1 PowerPC™ Assembly Programs

We use the 32-bit variant of the PowerPC™ instruction set architecture (ISA) and assume little-endian mode. The PowerPC™ architecture defines thirty-two 32-bit general purpose registers (GPRs) – referred to in $A$ as `%r0` through `%r31`. In addition, there are thirty-two 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`).

In our example, 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.

### 3.2 From assembly to C

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

- The thirty-two GPRs are declared as global `int` variables `r0` through `r31`. The thirty-two FPRs are declared as global `double` variables `f0` through `f31`. The SPRs are also declared as `int` variables `cr`,

```
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
```

**Fig. 3** A running example. A code fragment (left) and the assembly obtained by compiling it (right).

`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. Thus, in our example, we obtain a single procedure declaration:

  `void OSMemNameSet(void)`

– 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.

### 3.3 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 memory. 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.

Fig. 4 shows a table with assembly instructions on the left and the corresponding C statements on the right. Among the instructions in Fig. 4, the following have straightforward translations:

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

**Fig. 4** Translation schema from assembly instructions to C statements.

- `li = load immediate`: takes a register $R$ as the first argument and an immediate integral value $V$ as the second argument; loads $V$ into $R$; for example, `li %r0,0` loads the integral value 0 into the register `%r0`.
- `mr = move register`: takes two registers as arguments and copies the contents of the second argument into the first argument; for example, `mr %r31,%r1` copies the contents of `%r1` into `%r31`.
- `mflr = move from link register`: takes a single register $R$ as an argument; copies the contents of `%lr` into $R$; for example, `mflr %r0` copies the contents of `%lr` into `%r0`.
- `mtlr = move to link register`: takes a single register $R$ as an argument; copies the contents of $R$ into `%lr`; for example, `mtlr %r0` copies the contents of `%r0` into `%lr`.
- `b = branch`: takes a label $L$ as an argument; jumps to $L$; for example, `b .L1` causes the execution to jump to label `.L1`.
- `bl = branch link`: takes a label $L$ as an argument; jumps to $L$, while storing the return address in `%lr`; for example, `bl OS_ENTER_CRITICAL` causes execution to jump to label `OS_ENTER_CRITICAL`, while storing the next address in `%lr`; typically used to implement a function call.
- `blr = branch link register`: causes execution to jump to the address stored in `%lr`; typically used to implement a return from a function.

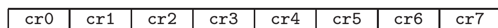
The translations for the other instructions follow their semantics, as described below:

- `lwz = load word and zero`: takes a register $R$ as the first argument and a memory location $L$ as the second argument; loads a word from $L$ to $R$; for example, `lwz %r0,8(%r31)` loads a word from the address at 8 bytes offset from the contents of `%r31` into `%r0`.
- `lbz = load byte and zero`: takes a register $R$ as the first argument and a memory location $L$ as the second argument; loads a byte from $L$ to $R$ and zeroes out the higher-order 24 bits of $R$; for example, `lbz %r0,20(%r31)` loads a byte from the address at 20 bytes offset from the contents of `%r31` into the least significant byte of `%r0`, and then zeroes out the most significant three bytes of `%r0`. Due to little-endianness, in the result of decompiling a `lbz` instruction, after we load an `int` from $L$ into $R$, our desired byte will be laid out at the lower order end

of $R$, and we have to zero out the higher-order 24 bits of $R$.

- `stwu = store word with update`: takes a register $R$ as the first argument and a memory location $L$ as the second argument; stores the word in $R$ to $L$, and then sets the value of $R$ to $L$; for example, `stwu %r1,-48(%r1)` stores the word in `%r1` into the address $L$ at -48 bytes offset from the contents of `%r1` and then stores $L$ into `%r1`.
- `stw = store word`: takes a register $R$ as the first argument and a memory location $L$ as the second argument; stores the word in $R$ to $L$; for example, `stw %r31,44(%r1)` stores the contents of `%r31` into the address at 44 bytes offset from the contents of `%r1`.
- `stb = store byte`: takes a register $R$ as the first argument and a memory location $L$ as the second argument; stores the lowest byte in $R$ to $L$; for example `stb %r0,0(%r9)` stores the contents of `%r0` into the address contained in `%r9`. Again due to little-endianness, in the result of decompiling a `stb` instruction, we load the current word stored at $L$, replace its lowest byte with the lowest byte of $R$, and store the new value back to $L$.
- `rlwinm = rotate left word immediate then AND with mask`: takes registers $R_1$ and $R_2$ as the first two arguments, and immediate integral values $V_1$ and $V_2$ as the last two arguments; rotates left the contents of $R_2$ by $V_1$ bits, then logically ANDs the result with $V_2$, and stores the result in $R_1$; for example, `rlwinm %r0,%r0,0,255` logically ANDs the contents of `%r0` with the bitmask `0xff` and stores the result back to `%r0`.

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.




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 $R$ are 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`: takes a condition sub-register $SR$ as the first argument, a register $R$ as the second argument, and an immediate value $V$ as the third argument; compares the contents of $R$ with $V$, treating both values as signed integers, and stores the result in $SR$; for example, `cmpwi %cr7,%r0,0` stores the result of the signed comparison between the contents of `%r0` and 0 into `%cr7`.
- `cmplwi = compare logical word immediate`: takes a condition sub-register $SR$ as the first argument, a register $R$ as the second argument, and an immediate value $V$ as the third argument; compares the contents of $R$ with $V$, treating both values as unsigned integers, and stores the result in $SR$; for example, `cmplwi %cr7,%r0,31` stores the result of the unsigned comparison between the contents of `%r0` and 31 into `%cr7`. Since all our C variables are signed, the result of decompilation guards 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`: takes a condition sub-register $SR$ as the first argument and a label $L$ as the second argument; jumps to $L$ if $SR$ indicates a "less or equal" comparison result; for example, `ble %cr7,.L4` jumps to label `.L4` if either the highest or the third highest bit of `%cr7` is set to 1.
- `bne = branch not equal`: takes a condition sub-register $SR$ as the first argument and a label $L$ as the second argument; jumps to $L$ if $SR$ indicates a "not equal" comparison result; for example, `ble %cr7,.L2` jumps to label `.L2` if the third highest bit of `%cr7` is set to 0.

## 3.4 Semantics Preservation

Suppose we obtain the C program $P$ by decompiling the assembly program $A$. For each instruction $i$ of $A$, let $D(i)$ be the sequence of statements in $P$ obtained by decompiling $i$. A memory configuration $\mu$ is a function from integers to bytes. A register valuation $\rho$ is a function from registers to bit-vectors of appropriate size. An execution state of $A$ is a pair $(\mu, \rho)$ where $\mu$ is a memory configuration and $\rho$ is a register valuation. A variable valuation $\sigma$ is a function from the variables of $P$ to values of appropriate type. An execution state of $P$ is a pair $(\mu, \sigma)$ where $\mu$ is a memory configuration and $\sigma$ is a variable valuation.

Let $\Psi$ be a function from register valuations to variable valuations defined as follows: $\Psi(\rho)$ maps each variable $v$ to the value whose bit-vector representation is $\rho(r)$ where $r$ is the register corresponding to the variable $v$ in accordance with our decompilation scheme. Then, decompilation is semantics-preserving, as captured by the following claim.

*Claim* Suppose we start executing $A$ from any function label `Func` and state $(\mu_0, \rho_0)$, and $P$ from the corresponding function `Func` and state $(\mu_0, \Psi(\rho_0))$. Then, an instruction $i$ is executed subsequently in $A$ from a state $(\mu, \rho)$ iff $D(i)$ is executed subsequently in $P$ from a state $(\mu, \Psi(\rho))$.

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$^{TM}$ assembly instructions. Complete details of the PowerPC$^{TM}$ ISA are available [30] 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. COPPER was developed to verify safety and liveness properties of multi-threaded C programs communicating via shared variables and message-passing. However, 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 theorem prover (usually Simplify [27] or Vampyre [32]) treats the C bitwise 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. For example, one axiom asserted that for any variable $v$, the bitwise-OR of $v$ with 0 is equal to $v$. Unfortunately, this solution is ad hoc, since we had no way of knowing if enough axioms had been added. Moreover, the process of adding new axioms was completely manual. 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 simplifications.** 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 is then 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 CVCLITE [15] but found CPROVER to be faster). To prove the validity of a formula $F$, CPROVER first constructs a Boolean propositional formula $F'$ (via a process known as bit-blasting [22]) such that $F'$ is unsatisfiable iff $F$ is valid. Next, CPROVER uses a satisfiability solver, such as ZCHAFF [23], to check for the satisfiability of $F'$. To construct $F'$, CPROVER interprets $F$ as a C expression, using precise ANSI C semantics.

We tried two approaches of using CPROVER. 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. When calling Simplify, we do not use any external axioms or syntactic simplifications. Thus, this approach is different from both solutions 1 and 2 above. The key goal behind the *semi-interpreted* approach is to minimize the performance penalty, compared to Simplify, incurred when using CPROVER.

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

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

verifies the safety property. Also our experiments indicate that the *uninterpreted* approach yields the best performance over a set of realistic benchmarks, indicating that our syntactic simplifications provide the necessary bit-level reasoning. 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 of 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 locks and unlocks with a nondeterministic 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. 5. 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. 6.

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.

We also evaluated the degree of code blowup due to decompilation by comparing: (i) the sizes of the original C source file and the C file obtained via decompilation, and (ii) the sizes of the assembly files obtained by compiling the original C source file and the C file obtained via decompilation. In either case, we observed a blowup of about 2-3 times across our benchmark suite. File sizes were not a bottleneck in any of our experiments. Given the prototypical nature of our implementation, we believe that this is an encouraging sign.

## 6 Discussion and Conclusion

We present 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.

In the context of pointers and aliasing, the decompilation scheme for assembly instructions that access memory is particularly critical. The RISC approach used by PowerPC™ leads to restricted forms of memory accesses, which in turn leads to limited types of pointer operations in the decompiled C programs. Such C programs may be easier to verify via SMC. In contrast, a naive decompilation of `x86` CISC instructions

| 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 |

**Fig. 5** 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; × 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 |

**Fig. 6** 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; × means that the counterexample returned by BLAST is spurious. Best figures are highlighted.

with complex memory access features leads to C programs with complicated pointer operations. Such programs are likely to be much harder to analyze by SMC. Instead, treating each CISC instruction as a sequence of RISC instructions during decompilation may yield C programs that are more amenable to SMC techniques.

AIR decompilation results in a C program with only integer and double variables. We believe that handles the vast majority of instruction sets, which only admit general-purpose and floating-point registers. Handling instruction sets with other, more esoteric registers (like `bool` and `char`), would necessitate the inclusion of corresponding variable types. An important requirement for AIR is the ability to identify basic blocks in the target assembly program, and their mapping to procedures. In our examples, this step was not difficult. However, in general, this requires additional tooling infrastructure.

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. In particular, properties like buffer overflows and illegal memory accesses, incorrect library API usage and synchronization, etc., are likely to be much harder to express and detect at the assembly level. Similarly, interpreting a counterexample expressed in terms of an assembly program is likely to be more difficult. Moreover, the steps of AIR are specific to instruction set architectures. 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 adaptable for the purpose of using certifying model checking [24] for proof carrying code (PCC) [26]. Certifying model checking

in combination with abstraction has been used [25,8] 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. Andrews, T., Qadeer, S., Rajamani, S., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: R. Alur, D. Peled (eds.) Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04), *Lecture Notes in Computer Science*, vol. 3114, pp. 484–487. Springer-Verlag (2004)
2. Balakrishnan, G., Reps, T.: "Analyzing Stripped Device-Driver Executables". In: C.R. Ramakrishnan, J. Rehof (eds.) Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08), *Lecture Notes in Computer Science*, vol. 4963, pp. 124–140. Springer-Verlag (2008)
3. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: M.B. Dwyer (ed.) Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01), *Lecture Notes in Computer Science*, vol. 2057, pp. 103–122. Springer-Verlag (2001)
4. BEHAVE! website (2009). `http://research.microsoft.com/behave.`
5. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. Journal of the ACM (JACM) **43**(1), 166–192 (1996)
6. Breuer, P.T., Bowen, J.P.: Generating Decompilers. RUCS Technical Report RUCS/1998/TR/010/A, Department of Computing, The University of Reading (1998)
7. CBMC website (2009). `http://www.cprover.org/cbmc.`
8. Chaki, S.: SAT-Based Software Certification. In: H. Hermanns, J. Palsberg (eds.) Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06), *Lecture Notes in Computer Science*, vol. 3920, pp. 151–166. Springer-Verlag (2006)
9. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular Verification of Software Components in C. In: Proceedings of the 25th International Conference on Software Engineering (ICSE '03), pp. 385–395. IEEE Computer Society (2003)
10. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: K. Jensen, A. Podelski (eds.) Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04), *Lecture Notes in Computer Science*, vol. 2988, pp. 168–176. Springer-Verlag (2004)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) **50**(5), 752–794 (2003)
12. Clutterbuck, D.L., Carre, B.A.: The verification of low-level code. Software Engineering Journal (SEJ) **3**(3), 97–111 (1988)
13. Copper website (2009). `http://www.sei.cmu.edu/pacc/copper.html.`
14. Curzon, P.: A Structured Approach to the Verification of Low Level Microcode. Ph.D. thesis, University of Cambridge, Computer Laboratory (1991). Tech report no. 215
15. CVC Lite website (2009). `http://verify.stanford.edu/CVCL.`
16. Dwyer, M.B., Hatcliff, J., Hoosier, M., Robby: Building Your Own Software Model Checker Using The Bogor Extensible Model Checking Framework. In: K. Etessami, S.K. Rajamani (eds.) Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05), *Lecture Notes in Computer Science*, vol. 3576, pp. 148–152. Springer-Verlag (2005)
17. Dwyer, M.B., Hatcliff, J., Joehanes, R., Laubach, S., Păsăreanu, C., Zheng, H., Visser, W.: Tool-supported program abstraction for finite-state verification. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE '01), pp. 177–187. IEEE Computer Society (2001)
18. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: O. Grumberg (ed.) Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97), *Lecture Notes in Computer Science*, vol. 1254, pp. 72–83. Springer-Verlag (1997)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '02), *SIGPLAN Notices*, vol. 37(1), pp. 58–70. Association for Computing Machinery (2002). URL citeseer.nj.nec.com/524901.html
20. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
21. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software Verification Platform. In: K. Etessami, S.K. Rajamani (eds.) Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05), *Lecture Notes in Computer Science*, vol. 3576, pp. 301–306. Springer-Verlag (2005)
22. Kroening, D.: Application Specific Higher Order Logic Theorem Proving. In: S. Autexier, H. Mantel (eds.) Proceedings of the Verification Workshop (VERIFY'02), pp. 5–15 (2002)
23. Moskewicz, M., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th ACM IEEE Design Automation Conference (DAC '01), pp. 530–535. Association for Computing Machinery (2001). URL http://doi.acm.org/10.1145/378239.379017
24. Namjoshi, K.S.: Certifying Model Checkers. In: G. Berry, H. Comon, A. Finkel (eds.) Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01), *Lecture Notes in Computer Science*, vol. 2102, pp. 2–13. Springer-Verlag (2001)
25. Namjoshi, K.S.: Lifting Temporal Proofs through Abstractions. In: L.D. Zuck, P.C. Attie, A. Cortesi, S. Mukhopadhyay (eds.) Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '03), *Lecture Notes in Computer Science*, vol. 2575, pp. 174–188. Springer-Verlag (2003)
26. Necula, G.C.: Proof-Carrying Code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97), pp. 106–119. Association for Computing Machinery (1997)
27. Nelson, G.: Techniques for Program Verification. Ph.D. thesis, Stanford University (1980)
28. O'Neill, I.M., Clutterbuck, D.L., Farrow, P.F., Summers, P.G., Dolman, W.C.: The formal verification of safety-critical assembly code. In: Proceedings of the International Federation of Automatic Control Safety of Computer Control Sys-

tems Conference (SAFECOMP '88), *IFAC Proceedings Series*, vol. 16, pp. 115–120 (1988)

29. Pnueli, A., Siegel, M., Singerman, E.: Translation Validation. In: B. Steffen (ed.) Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98), *Lecture Notes in Computer Science*, vol. 1384, pp. 151–166. Springer-Verlag (1998)
30. PowerPC™ ISA (2009). `http://www.nersc.gov/vendor_docs/ibm/asm/mastertoc.htm.`
31. Reps, T., Balakrishnan, G., Lim, J., Teitelbaum, T.: A Next-Generation Platform for Analyzing Executables. In: K. Yi (ed.) Proceedings of the third Asian Symposium on Programming Languages and Systems (APLAS '05), *Lecture Notes in Computer Science*, vol. 3780, pp. 212–229. Springer-Verlag (2005)
32. Vampyre website (2009). `http://www-cad.eecs.berkeley.edu/~rupak/Vampyre.`
33. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model Checking Programs. In: Proceedings of the 15th International Conference on Automated Software Engineering (ASE '00), pp. 3–12. IEEE Computer Society (2000)
34. Yu, D.: Safety Verification of Low-Level Code. Ph.D. thesis, Graduate School. of. Yale University (2004)
35. Yu, D., Shao, Z.: Verification of Safety Properties for Concurrent Assembly Code. In: C. Okasaki, K. Fisher (eds.) Proceedings of the 2004 International Conference on Functional Programming (ICFP '04), pp. 175–188. Association for Computing Machinery (2004)