Avoiding Hardware Aliasing
Verifying RISC Machine and Assembly Code for Encrypted Computing

Peter T. Breuer and Jonathan P. Bowen
School of Computing, Engineering & the Built Environment
Birmingham City University, Birmingham, UK
Email: Peter.T.Breuer@gmail.com, jonathan.bowen@bcu.ac.uk

Abstract

‘Hardware aliasing’ classically arises when a processor through failure or design has fewer bits than required to address each memory location uniquely, and also – nowadays – in the context of homomorphically encrypted computing. In such contexts, different physical locations may sporadically be accessed by the same address due to factors not directly under the control of the programmer but still deterministic in nature. We check RISC machine and assembly code to ensure that each memory address is calculated in exactly the same way at one write to and the next read from it, which allows programs to work correctly even in a hardware aliasing context.

1. Introduction

We are interested in an unfamiliar situation, called hardware aliasing. It used to be very familiar indeed to mainstream programmers and users alike, being one cause of the ‘DLL hell’ that Windows ‘98 was prone to: use of dynamic linked libraries (DLLs) under Windows was problematic for a variety of reasons including poorly defined versioning control and installation engineering, but, in particular, different versions of the same library loaded at the same memory address. So if one application loaded one version of the library and a second application loaded another, then the second application would sporadically and unpredictably – as far as the programmer and user were concerned – find itself executing in the first library’s code.

Users of DOS were even more familiar with the concept of hardware aliasing, as expanded memory managers (EMMs) such as QuarterDeck’s QEMM [1] remapped memory so that the VGA screen and BIOS read-only memory opportunistically shared the same addressing as random access memory (RAM) for applications. What a program accessed at runtime depended on the memory manager, and that decided according to a number of cues, including the kind of machine code instruction doing the access. Programs had to be written in a special way in order to make use of expanded memory reliably.

Despite the venerable history, modern-day computer scientists seem largely unfamiliar with the triage required in order to deal with hardware aliasing. In modern contexts, however, hardware aliasing does arise. Barr [2] says that the term “hardware aliasing . . . [is] used to describe the situation where, due to either a hardware design choice or a hardware failure, one or more of the available address bits is not used in the memory selection process.” In other words, the memory address bits accessible to the processor are insufficient to distinguish two different locations. For the sake of example, suppose processor arithmetic is based on 16 bits and there are 17 bits of memory addressing available, then (‘logical’) address 0x1234 might access either (‘physical’) address 0x01234 or 0x11234, depending on what the extra bit is set to in the addressing lines on each occasion. Which it is is not under the control of the one machine code instruction, but it is reproducible: running exactly the same sequence of machine code will produce exactly the same memory access at the same point again.

In this paper, a means of preserving the intended behaviour of RISC [3] machine code and assembly language in some hardware aliasing contexts is set out. We are motivated particularly by the case of a KPU (Krypto-Processor Unit). That is a generic processor design in which encrypted data circulates through memory and registers without ever being decrypted [4] at the same kind of speeds as we are used to, employing principles of ‘fully homomorphic’ encryption [5], [6]. Nearly any current CPU design is capable of modification to a KPU format, since it is in every way
values 0 to 2³₂, etc. In a physically 64-bit architecture, the logical tern '10' represents the logical value 2, '11' represents 3, etc. In a physically 64-bit architecture, the logical values 0 to 2³₂ – 1 may be represented by up to 2³₂ different bit-patterns each, none of them conventional, and the scheme will vary with the encryption.

The trick may be understood by appreciating that the bit-patterns representing data inside the machine are not the conventional ones in which the physical pattern ‘10’ represents the logical value 2, ‘11’ represents 3, etc. In a physically 64-bit architecture, the logical values 0 to 2³₂ – 1 may be represented by up to 2³₂ different bit-patterns each, none of them conventional, and the scheme will vary with the encryption.

The connection with hardware aliasing is that many different bit-patterns represent the same logical value. More cipher codes assigned to each plain-text number means better security, and a more catastrophically intrusive aliasing effect: to a 32-bit code running in a KPU, all the 2³₂ different 64-bit bit-patterns representing the same 32-bit logical address 0xf000f000 look the same. If the running code compares any two for equality, the processor will report ‘true’, yet those different 64-bit patterns access different memory locations because the computer memory and other devices not integral to the processor are not privy to the encryption. The running program sees the same address sporadically alias different memory locations.

It turns out to be not hard to write code that avoids the problem. There is a programming discipline to be followed. Existing assembly code and machine code may be altered to be safe in a hardware aliasing context – the proviso is that the aliasing is produced deterministically by the processor, in that

1) a copy of an address accesses the same location;
2) recalculating the address from scratch in exactly the same way as the first time accesses the same location again.

The rationale is that an address is just a number produced by the processor from certain inputs. Provide physically the same inputs again and the processor will repeat the same physical transformations to produce physically the same outputs. There is no randomness.

‘Copying’ is just the trivial case, nevertheless it is one point where existing assembly and machine code needs modification. RISC machines in particular move data between registers by adding zero. That may transform the bit-representation non-trivially. We are used to the idea that the result of adding zero to the bit-pattern 0x1 is just the same bit-pattern 0x1 again, but in the context of a KPU, for example, the logical number 1 may be physically represented by the encoding bit-pattern 0x42, and adding zero may produce the physical bit-pattern 0xdeadbeef4a11 although it is just another encoding of the number 1.

Therefore code needs to be checked for compiler substitutions of ‘add zero’ where the ‘move’ instruction was intended. Stack movements and accesses to the elements of a string or an array also need checking.

Another modern case that presents as hardware aliasing is an embedded processor with, say, 40 bits of arithmetic but 64-bit addressing. The extra lines will be connected to 64-bit address registers in the processor, so their values change as the register is filled, but the processor arithmetic cannot test the extra bits. Code ideally should be generated according to the general class of processor without regard to customizations, and one way to do that is to emit code that can cope with hardware aliasing. The variation in what gets addressed by a given code is not from moment to moment, but from processor to processor.

Our verification method applies logic to assembly language/machine instructions, which may have been produced by hand or by compiler. The logical deductions are indexed by what the machine-level instruction is intended to do – as discussed above, exactly the same ‘add’ instruction may have been used both for ‘move’ and to access the bottom element of an array – and fitting together these deductions builds a logically consistent ‘decompilation’ [7] of the whole code.

The logic aims to show that accesses to (32-bit) words on the stack are via addresses calculated as

\[((base + 4) + 4) + \ldots + 4\]

from the stack base, where the offset may not overstep the subroutine’s stack frame. That means it is impossible to access a word in the parent (caller’s) frame – such accesses are often generated by compilers as optimizations, and we must show that this does not occur in the code; such ‘tricks’ would not work in a hardware aliasing context because the two different calculations for the address of the word in the parent frame could give physically different bit-patterns, and memory would access different locations.

That access to a word in the local frame is always via \((base + 4) + \ldots + 4\) for the address implies that there can be no two different ways of calculating the same address, so no opportunity for hardware aliasing.

Our intended target consists of the small fragments of not more than twenty or thirty lines at a time of
hand-coded assembly language code that make up 10% of the Linux kernel. Getting that kernel running in a hardware aliasing context is our eventual aim.

2. Basic notions

Say subroutine foo starts by decrementing the stack pointer by 32 to make space for a local frame of 8 variables of one word (4 bytes). Before return, the pointer is incremented back to its original value sp0. The following is the code emitted by a compiler:

```assembly
foo:
addiu sp sp -32       # stack pointer register ← 32
... move machine code ...
addiu sp sp 32         # stack pointer register += 32
jr ra                  # jump to return address in ra reg.
```

In an aliasing context, the ‘+32’ calculates \((sp0 - 32)+32\), producing a different alias of the stack pointer sp0 initially in the sp register. It is not the same pattern of bits as it was initially, and the caller gets a pointer that does not point to its own data. This works instead:

```assembly
foo:
move gp sp             # copy stack pointer to gp reg.
addiu sp sp -32        # stack pointer register ← 32
... move machine code ...
move sp gp             # copy stack pointer back from gp
jr ra                  # jump to return address in ra reg.
```

This code takes an extra register (gp) and instruction (move), but it recovers exactly the initial bit-pattern.

How may we show the second code is ‘aliasing-safe’? Our analysis abstracts to a stack machine: whereas a RISC machine manipulates a stack pointer register explicitly via generic register and memory instructions, the stack machine uses instructions (push, pop, etc) that work at the level of a stack abstraction. The abstraction disambiguates machine code that looks the same but serves different purposes, such as when ‘addiu sp sp -32’ manipulates an address to increase stack space, and when it is used to calculate data.

Our technique is to annotate the machine code with pre- and post-conditions to express what it does on a stack machine model. That follows the classical style of verification under Hoare logic [8], as used in existing verification frameworks such as VCC [9].

2.1. Subroutine foo, annotations bar

Annotation of subroutine foo begins by asserting the precondition that the sp (stack pointer) register is associated with a local subroutine frame of size 24 bytes with words at offset 0, 4 and 8 bytes:

\[
\{ \text{sp}^* = c^{24}0!4!8 \}
\]

The move instruction copies data between registers, li loads a register with a given value, nop is a no-op, lw loads a register from memory, sw stores a register to memory, jal jumps to the given program address and stores the current program address to the ra register (‘subroutine call’) and jr jumps to the program address in the given register (‘return’).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>move r1 r2</td>
<td>li r n</td>
</tr>
<tr>
<td>sw r1 n(r2)</td>
<td>lw r1 n(r2)</td>
</tr>
<tr>
<td>jal a</td>
<td>jr r</td>
</tr>
<tr>
<td>addiu r1 r2 n</td>
<td>ori r1 r2 n</td>
</tr>
<tr>
<td>add r1 r2 r3</td>
<td>subu r1 r2 r3</td>
</tr>
<tr>
<td>beq r1 r2 a</td>
<td>bne r1 r2 a</td>
</tr>
</tbody>
</table>

The n are small integers, r are register names or indices, and a are relative or absolute addresses, assignment is indicated by a left-arrow. The memory cell with address formed by adding n to the address in register r is \(r[n]\).

Table 1. RISC machine code instructions.

Annotating a asserts a binding of registers r or stack slots (n) to an ‘annotation type’ t. One of the register names may be starred to indicate the stack pointer position. The type either has base (‘uncalculated’) u or (‘calculated’) c, and may be decorated with ‘!n’ annotations indicating past writes at that offset from the typed value used as an address. The base type may be superscripted by a natural number n delimiting the offset allowed. Formal type variables x, y, etc may stand-in for types, and formal ‘offset set variables’ X, Y, etc may stand-in for sets of offsets.

- \(a \equiv r^n. . . . (n), . . . = t; \ldots\)
- \(t \equiv c^m!n! . . . u^n!n! . . .\)

The ‘c’ signifies a pointer value in the register. That it is the stack pointer is shown by the asterisk on the register name. The superscript 24 means that the pointer is to an area 24 bytes in size. The ‘!0!4!8’ means the area has already been written to at offsets 0, 4, 8 bytes, respectively.

The first instruction in subroutine foo copies the stack pointer to register gp, so register gp also gets the \(c^{24}0!4!8\) annotation, giving the postcondition below:

\[
\{ \text{sp}^* = c^{24}0!4!8 \} \quad \text{move gp sp} \quad \{ \text{sp}^*, \text{gp} = c^{24}0!4!8 \}
\]

The second instruction in subroutine foo does arithmetic on the stack pointer. After subtracting 32, even though something has been written at offset 8 from the stack base address, now \(32 + 8\) may not give the same bit-pattern as before. There is now nothing that can be said about what has been written at offsets from the new stack base address, so the ‘!0!4!8’ is lost:

\[
\ldots \text{addiu sp sp} -32 \{ \text{gp} = c^{24}0!4!8; \text{sp}^* = c^{32} \}
\]

The new frame is 32 bytes in size, and the stack pointer is still in the sp register, so sp is still asterisked.

The gp register remains undisturbed through the subroutine. The penultimate instruction refreshes the stack pointer register from the value saved in gp, so the stack pointer recovers its initial annotation:

\[
\ldots \text{move sp gp} \quad \{ \text{sp}^*, \text{gp} = c^{24}0!4!8 \}
\]
Table 3. Stack machine instructions

Here, \( n \) is a small integer, \( r \) is a register name or index, \( a \) is a relative or absolute address. Opcodes are colour-coded by kind: stack pointer register movements, stack content, flow control, arithmetic.

\[
\begin{align*}
\text{RISC code} & \quad \text{stack code} \\
\text{move} r_1 r_2 & \quad \text{cspf } r_2 \quad \text{cspt } r_1 \\
\text{jal } a & \quad \text{gosub } a \\
\text{jr } r & \quad \text{return} \\
\text{ja } a & \quad \text{goto } a \\
\text{lw } r_2 n(r_2) & \quad \text{get } r_1 n \\
\text{sw } r_2 n(r_2) & \quad \text{put } r_1 n \\
\text{lb } r_1 n(r_2) & \quad \text{getb } r_1 n \\
\text{sb } r_1 n(r_2) & \quad \text{putb } r_1 n \\
\text{addiu } r r n & \quad \text{addaiu } r r n \\
\text{bnez } r a & \quad \text{ifnz } r a \\
\end{align*}
\]

The model is a register machine with an implicit stack managed by the get, put, push, ... instructions.

Table 4. RISC code decompilation.

<table>
<thead>
<tr>
<th>RISC code</th>
<th>Stack code</th>
</tr>
</thead>
<tbody>
<tr>
<td>move r1 r2</td>
<td>cspt r1</td>
</tr>
<tr>
<td></td>
<td>cspf r1</td>
</tr>
<tr>
<td>jal a</td>
<td>gosub a</td>
</tr>
<tr>
<td>jr r</td>
<td>return</td>
</tr>
<tr>
<td>ja a</td>
<td>goto a</td>
</tr>
</tbody>
</table>

The final return (\( \text{jr} \)) instruction does not change anything, so the calling code has returned as stack pointer a value that is annotated as having had values written at offsets 0, 4, 8 from that base address, and the caller can rely on accessing data stored at those offsets.

That guarantees that subsequent reads will find values that have been written. This system will be coaxed into providing much stronger guarantees in the following sections.

3. Decomposition as proof

The formal logical deductions that we use to reason about annotations are each indexed by an instruction for an abstract stack machine that disambiguates the machine code/assembly language instruction’s use. The machine code instruction may be thought of as decompiled to the stack machine instruction, and the logic reasons about what happens at the higher level (described in Box 1).

Only certain decompilation choices make sense together and that corresponds to the logical deductions being combined validly only in certain ways.

The stack machine instructions are listed in Table 3. Different stack machine instructions compile to the same machine code instruction as shown in Table 4.

At the stack machine level register/memory transfers are effected by the three sets of instructions shown on lines 2, 3 and 4 of Table 3, abstracting the classic load and store (‘lw’ and ‘sw’) RISC instructions.

The machine code processor state is a triple \( (R, M, p) \), where the register state \( R \) is a vector of 32 registers \( r \) each containing a 32-bit value \( R_r \), \( M \) is the memory state, a vector of 2\(^32\) values \( M_a \) indexed by address \( a \), and \( p \) is the address of the current instruction. The stack pointer \( s \) is in the stack pointer register \( \text{sp} \). The abstract stack machine state is a 4-tuple \( (R, K, \mathcal{H}, p) \), where \( R \) consists of the 31 registers excluding the stack pointer register, the stack \( K \) corresponds to the top part of memory \( M \) above the stack pointer value \( s \), the heap \( \mathcal{H} \) corresponds to the bottom part of \( M \) below the stack pointer, and the address \( p \) is that of the current instruction.

\[
\begin{align*}
K_k = M(s + k) & \quad s = R_{sp}, \ k \geq 0 \\
\mathcal{R} r = R_r & \quad r \neq \text{sp}, \ r \in \{0, \ldots, 31\} \\
\mathcal{H} a = M_a & \quad a < s
\end{align*}
\]

Knowledge of the hidden stack pointer value \( s \) and the stack machine state reconstructs the machine code processor state.

Box 1. Formal processor/stack machine relation.

Which to use depends on whether stack (\( \text{put}/\text{get} \)) or heap is accessed, and in the latter case whether an array or a string is accessed. Because of space restrictions, only stack-oriented access will be treated here.

The many possible decompilations of the RISC move \( r_1 r_2 \) (‘copy from register \( r_2 \) to \( r_1 \)’) machine code instruction mainly cover movements of the stack pointer. The \( \text{cspt } r_1 \) instruction stands for ‘copy stack pointer to register \( r_1 \), for example, and it is used when a copy is made in \( r_1 \) of the stack pointer, without changing the register in which the stack pointer is. The copy cannot be used as stack pointer. A copy of a previous value can be restored with \( \text{rspf } r_1 \) (‘restore stack pointer from register \( r_1 \)’), and then used.

The \( \text{mspt } r_1 \) instruction stands for ‘move stack pointer to register \( r_1 \)’ and copies and changes the register the stack pointer is in from \( r_2 \) to \( r_1 \). The \( \text{cspf } r_2 \) instruction stands for ‘copy stack pointer from register \( r_2 \)’ and refreshes the stack pointer register from an exact copy stored earlier in \( r_2 \) (why compilers emit ‘refresh’ instructions will not be explored here).

Decreases in the stack pointer are carried out by the \( \text{push } m \) instruction \((m > 0)\). This creates a new stack frame of \( m \) bytes in size, extending the stack downwards. It is implemented by the RISC \( \text{addiu } r r n \) machine code instruction, with \( n = -m \) and \( r \) the stack pointer register.

The formal logic for reasoning about annotations can be seen in the layout below of the annotation in subroutine \( \text{foo} \) from the previous section. Each deduction is denoted by a horizontal line, labelled by the RISC machine code to which it is applied/its decompilation as stack machine code:
The push 32 abstract stack machine instruction makes a new local stack frame of 8 words or 32 bytes and does not increase the size of the current frame.

4. Formal logic

The logic introduced above is formalised in Table 5 with a list of ‘small-step’ deduction schemas expressing what each stack machine instruction does. Bold italic capitals X, Y, etc are ‘offsets variables’. They stand for sets of offset annotations ‘\(x\)’.

For example, the schema for the put gp 4 instruction is derived as follows. It might get precondition \(\{ \text{sp}^{*} = c^{8!}X \}\) when the stack pointer is in register sp initially. Stack frame size 8 might be asserted to accommodate the 4-byte word written at offset 4 by the instruction. The postcondition is \(\{ \text{sp}^{*} = c^{8!}4!X \}\), claiming that 4 is definitely an offset at which something has been written. The ‘X’ stands for the set of offsets at which something has been written by the start of the instruction. It may be that 4 is a member of X, or it may not be. The postcondition says that offset ‘4’ is forced by this instruction. So the deduction for the instruction takes the following form in this instance:

\[
\{ \ldots : \text{sp}^{*} = c^{8!}X ; \ldots \} \quad \text{put gp} \quad 4 \quad \{ \ldots : \text{sp}^{*} = c^{8!}4!X ; \ldots \}
\]

Now, bold lower case letters x, y, etc in Table 5 are ‘type variables’. They stand in for entire annotation types such as ‘\(c^{8!}4!\)’. And cell locations in the stack relative to the base are given in parentheses: ‘(4)’.

Considering the effect (of the put gp 4 instruction) on the gp register, which may be supposed to have the type denoted by the formal variable x initially, and on the stack slot ‘(4)’, the deduction may be filled out to:

\[
\{ \text{gp} = x; \text{sp}^{*} = c^{8!}X \} \quad \text{put gp} \quad 4 \quad \{ \text{sp}^{*} = c^{8!}4!X ; \text{gp},(4) = x \}
\]

Generalising the stack frame size from 8 to \(f\) and the stack offset 4 to \(n\), and generalising registers gp and sp to \(r_1\) and \(r_2\) respectively, the deduction scheme listed for put \(r_2\ n\) in Table 5 is obtained:

\[
\{r_1 = x; r_2 = c^{f!}X\} \quad \text{put} \quad r_1 \quad n \quad \{r_1, (n) = x; r_2 = c^{f!}n!X\}
\]

Registers not mentioned in the schema are unaffected.

Table 5. ‘Small-step’ annotations on stack machine instructions.

The X, Y, etc stand for a set of offsets \([n_1,n_2,...]\) for literal natural numbers \(n\). The stack frame size \(f\) is a literal natural number. The x, y, etc stand for any type (something that can appear on the right of an equals sign).

\[
\begin{align*}
\{r_1 = x; r_2 = c^{f!}X\} & \quad \text{put} \quad r_1 \quad n \quad \{r_1, (n) = x; r_2 = c^{f!}n!X\} \\
\{r_2 = c^{f!}n!X; (n) = x\} & \quad \text{get} \quad r_1 \quad n \quad \{r_1, (n) = x; r_2 = c^{f!}n!X\} \\
\{r^* = c^{f!}X\} & \quad \text{push} \quad n \quad \{r^* = c^{f!}\} \\
\{r_1 = c^{f!}X; r_2 = c^{f!}X\} & \quad \text{rsf} \quad r_2 \quad n \quad \{r_1, r_2 = c^{f!}X\} \\
\{r_2 = c^{f!}X\} & \quad \text{cp} \quad r_1 \quad n \quad \{r_1, r_2 = c^{f!}X\} \\
\{r_2 = c^{f!}X\} & \quad \text{mpt} \quad r_1 \quad n \quad \{r_1, r_2 = c^{f!}X\} \\
\{r_1 = c^{f!}X; r_2 = c^{f!}X\} & \quad \text{rsf} \quad r_2 \quad n \quad \{r_1, r_2 = c^{f!}X\} \\
\{r_1 = c^{f!}X\} & \quad \text{mpt} \quad r_1 \quad n \quad \{r_1, r_2 = c^{f!}X\} \\
\{r_1 = c^{f!}X; r_2 = c^{f!}X\} & \quad \text{add} \quad r_1 \quad r_2 \quad n \quad \{r_1 = c^{f!}; r_2 = c^{f!}X\} \\
\{r_2 = c^{f!}X\} & \quad \text{ori} \quad r_1 \quad r_2 \quad n \quad \{r_1 = c^{f!}; r_2 = c^{f!}X\} \\
\end{align*}
\]

... How deductions are combined needs specifying too. Note that a whole program is annotated, including an annotation ‘at the end’, which we will call \(\Phi\). Let \(T\) be the list of annotations, one preceding each instruction address \(a\) in the program, and write \(T \triangleright \{\} \ a \ \{\Phi\}\) for ‘\(T\) lists precondition \(\Theta\) for the instruction at address \(a\) with final annotation \(\Phi\) at the end’. Let Table 5 contain deduction \(\{\Theta\} = \{\Psi\}\) for the instruction \(\iota\) at \(a\) that compiles to \(\kappa\). If \(\Psi\) is the precondition listed in \(T\) for the instruction at address \(a + 4\), then \(\Theta\) is the precondition listed in \(T\) at address \(a\):

\[
T \triangleright \{\Psi\} \ a \ {+} 4 \ \{\Phi\} \\
\frac{T \triangleright \{\} \ a \ \{\Phi\}}{T \triangleright \{\Theta\} \ a \ \{\Phi\}} [a \ | \ \iota / \kappa]
\]

\(T\) is a theory. Note that branches must get the same annotation where they converge as there is only one annotation in \(T\) for that address. Similarly for the one annotation \(\Phi\) ‘at the end’ that counts for all the return exits from a subroutine.

The rule above should be read as an inference. The label at right is a justification: instruction \(\iota\) is at address \(a\) and disassembly \(\kappa\) has been chosen for it.

There are, in all, five ‘big-step’ rules like this; they force instruction annotations to logically cohere. The first rule (above) says a theory is built working backwards through the program. The second rule says building starts at the end, with the jr \(r\) machine code instruction that returns from a subroutine, jumping to the return address stored in register \(r\). The rule says there was a program address (an ‘uncalculable value’, \(u^0\)) in register \(r\) before jr ran – it is the subroutine return address, and it is still there after the jr too:

\[
\frac{T \triangleright \{r = u^0\} \ a \ \{r = u^0\}}{T \triangleright \{r = u^0\} \ a \ \{r = u^0\}} [a \ | \ jr \ r / return]
\]

A u type for the address in register \(r\) means arithmetic is not allowed on it. One may not tamper with a return.
address. The ‘0’ superscript has the effect of forbidding the address from use as a base for offset memory accesses; those would access program instructions.

Jumps (unconditional branches) are handled by a rule that refers back to the jump target:

\[
T \triangleright \{\Theta\} b \{\Phi\}
\]

\[
\frac{T \triangleright \{\Theta\} a \{\Phi\} [a \mid j b / \text{goto} b]}{T \triangleright \{\Theta\} \{\Phi\}}
\]

The annotation \(\Theta\) at the target \(b\) of the jump is identified with that at \(a\). It might be thought that \(b < a\) leads to a loop when constructing theory \(T\), but that is not so; the fixpoint is calculated by unification of variables \(X, Y, x, y, \) etc in the annotations at \(a\) and \(b\).

The branch rule identifies annotations \(\Theta\) at the beginnings of the two paths, one continuing from address \(b\) and one from address \(a + 4\), and requires the tested register to contain a value of type \(c\) (‘calculatable’):

\[
T \triangleright \{\Theta\} b \{\Phi\}
\]

\[
\frac{T \triangleright \{\Theta\} a + 4 \{\Phi\} [a \mid \text{bnez } r b / \text{ifnz } r b]}{T \triangleright \{\Theta\} \{\Phi\}}
\]

The case \(b < a\) (backward branch) is not a problem here either. We have implemented the deduction rules in Prolog, and Prolog’s unification mechanism probably finds a jump/branch fixpoint if it exists [10].

The final rule is for a subroutine call, the RISC jal \(b\) machine code instruction. The jal instruction saves the next address \(a + 4\) in the ra register (‘return address’) and jumps to code at address \(b\). The callee will later return by jumping back to the address saved in ra. On entry into the subroutine at \(b\), ra gets a \(u^0\) annotation because there is a return address stored in it, meaning an unmodifiable value that cannot even be used as a base for offset memory access. The difficulty is that the same subroutine can be called from many different contexts, eliciting different annotations to match the initial annotation each time. So we use a new list \(T^a\) to collect the annotations (for \(b\) and its called code) which derive from the call from site \(a\). The rule is:

\[
T^a \triangleright \{\text{ra}=u^0; \Psi\} b \{\Theta\}
\]

\[
\frac{T \triangleright \{\Theta\} a + 4 \{\Phi\} [a \mid \text{jal } b / \text{gosub} b]}{T \triangleright \{\Psi\} a \{\Phi\}}
\]

Moreover, although the above rule is correct, it is useful to apply extra constraints expressing the call convention in use. We may require (i) that each subroutine return the stack to the same state it acquired it in, and (ii) that a subroutine make and unmake all of its own local stack frame:

\[
T^a \triangleright \{\text{ra}=u^0; r^* = c^0!X; \Psi\} b \{r^* = c^0!Y; \Theta\}
\]

\[
\frac{T \triangleright \{r^* = c^0!Y; \Theta\} a + 4 \{\Phi\}}{T \triangleright \{r^* = c^0!X; \Psi\} a \{\Phi\}}
\]

The requirement (i) is implemented by returning the stack pointer in the same register \(r\) on entry and return, and with no stack cells visible in the local stack frame handed to the subroutine and handed back by the subroutine (the two \(s\)). The requirement (ii) is implemented by setting the local stack frame on entry to contain no stack, just the general purpose registers.

A technical argument in [10] shows this system enforces the unique addressing scheme for each stack location envisaged in Section 1.

5. Example annotation

The annotation of the main routine of an I/O program that calls ‘printstr’ with the output string address as argument, then calls ‘halt’, is shown in Table 6. The code was emitted by a standard compiler (gcc) and modified by hand to be safe against aliasing, so some gcc compiler quirks are still visible. The compiler likes to preserve the fp register content across subroutine calls, for example, even though it is not used here.

This example requires one piece of the jigsaw that has been left out here to save space. A string address on the heap is introduced on line 7 of ‘main’ by the li a0 (load immediate) instruction, which sets the a0 (‘0th argument’) register for the call to ‘printstr’ on line 8. The li instruction is decomposed as newx (‘new string’), and the string type c\(i\) enters the annotation.

The annotator assumes that the stack pointer starts in the sp register and that ‘main’ is called (from a program loader) with a return address in the ra register. Changes from line to line are marked in grey:

Note that ‘printstr’ gets the following annotation:

\[
\{\text{sp}=c^0; \text{ra}=u^0; v0=c^1; v1=u^1\}
\]

\[
\text{printstr}
\]

\[
\{\text{sp}; \text{gp}, a0=c^0; \text{ra}=u^0; v0=c^1; v1=u^1\}
\]

The annotation on register v0 (‘0th [return] value’) on entry is an artifact. It arises from the annotation rule for branches: the v0 register is copied from register a0 in one branch, which binds v0 to the same type as a0 has on entry, but the other branch does not set register v0. Since each register must end up with the same annotation down both branches and it is not set in one of them, v0 must have its final type before ever getting to the branch, just to satisfy the rule. The same artifact is responsible for the requirement on the v1 register on entry.

The ‘halt’ subroutine also has a peculiar annotation: it does not return and it writes a single byte to a hard-coded I/O-mapped address that stops the processor:

\[
\{\text{ra}=u^0\} \text{halt}\{\text{ra}=u^0; v1=u^1\}
\]
The faux annotation for register \( v_1 \) on output is the taint left by the presence of that write instruction.

### 6. Conclusion

It is sometimes necessary for the programmer to produce code that survives hardware aliasing; e.g., in the context of a homomorphically encrypted processor.

A formal method of annotation has been set out that verifies safety against hardware aliasing for machine code and assembly language, when it suffices for safety that a memory address be calculated in the same way between one write to it and following reads from it.

### References


