Bachelor Thesis

Linear-Time Static Analysis of RISC-V Binary Code

Thomas Wulz
twulz@cs.uni-salzburg.at

October 18, 2021

Supervisor: Professor Christoph Kirsch
ck@cs.uni-salzburg.at

Department of Computer Science
University of Salzburg

Abstract

Selfie[1] is, among other things, a compiler from a small subset of C (C*) to a small subset of RISC-V (RISC-U). We present algorithms for static analysis of such RISC-V binaries along with a C* implementation. By traversing the binary forward and backward, we collect data about known register values and liveness of registers for each instruction. Our approach has some shortcomings when analyzing binaries with recursive function calls, as these calls result in a lot of information being lost, which can propagate through the program. However, it only requires linear time and space in the size of the binary. In future work, this data can be used to optimize the binaries generated by selfie, as they are almost entirely unoptimized.

Contents

1 Introduction

1.1 Problem Description

The goal of this project is to perform static analysis on RISC-V binaries generated by the selfie[1] compiler to enable optimizations, while only having access to the compiled program. Specifically, we aim to collect known concrete register values and liveness information for each instruction. This data enables a variety of binary optimizations.

1.2 Selfie

Selfie is a self-contained 11-KLOC C* implementation of a compiler (`starc`), a RISC-U emulator (`mipster`), a RISC-U hypervisor (`hypster`), and a tiny C* library (`libcstar`) utilized by selfie[1]. C* and RISC-U are small subsets of C and RISC-V defined by selfie. The name comes from its self-referential property: It can be used to compile itself and it can run the resulting binary using `mipster` or `hypster`, and this instance can in turn compile or run selfie again.

1.3 RISC-U

RISC-U is a small subset of RISC-V defined by selfie. Each register is 64 bits wide and each instruction is encoded in 4 bytes. It only consists of 14 instructions, which are listed here in Table 1 along with their effects:

`lui rd,imm`	`rd = imm * 2^12` with `-2^19 <= imm < 2^19`
`addi rd,rs1,imm`	`rd = rs1 + imm` with `-2^11 <= imm < 2^11`
`ld rd,imm(rs1)`	`rd = memory[rs1 + imm]` with `-2^11 <= imm < 2^11`
`sd rs2,imm(rs1)`	`memory[rs1 + imm] = rs2` with `-2^11 <= imm < 2^11`
`add rd,rs1,rs2`	`rd = rs1 + rs2`
`sub rd,rs1,rs2`	`rd = rs1 - rs2`
`mul rd,rs1,rs2`	`rd = rs1 * rs2`
`divu rd,rs1,rs2`	`rd = rs1 / rs2`
`remu rd,rs1,rs2`	`rd = rs1 % rs2`
`sltu rd,rs1,rs2`	`if (rs1 < rs2) { rd = 1 } else { rd = 0 }`
`beq rs1,rs2,imm`	`if (rs1 == rs2) { pc = pc + imm }` `else { pc = pc + 4 }` with `-2^12 <= imm < 2^12`
`jal rd,imm`	`rd = pc + 4; pc = pc + imm` with `-2^20 <= imm < 2^20`
`jalr rd,imm(rs1)`	`tmp = ((rs1 + imm) / 2) * 2; rd = pc + 4; pc = tmp` with `-2^11 <= imm < 2^11`
`ecall`	system call with number in `a7`, parameters in `a0-a3` and return value in `a0`

Table 1: The RISC-U instruction set[1]

Every instruction except for `beq`, `jal`, `jalr` and `ecall` also always has the trivial side effect of `pc = pc + 4`, i.e. moving the program counter forward by the instruction size. Additionally, it should be noted that `divu`, `remu` and `sltu` interpret their arguments as unsigned integers.

1.4 High-level Overview

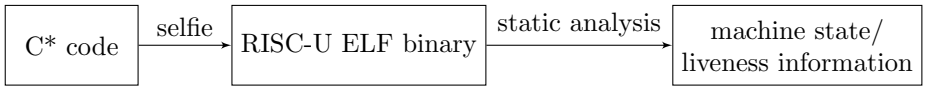

Figure 1: High-level overview

We start with the source code of a program written in C*, such as selfie itself or the analyzer described in this work. The selfie compiler is used to generate a RISC-U binary in ELF format, which is then used as the input to our analyzer. The implemented static analysis passes produce a control-flow graph and information about known register values and register liveness for each instruction. The overall process is visualized in Figure 1, and Figure 2 provides a closer view of how the three analysis passes relate to each other.

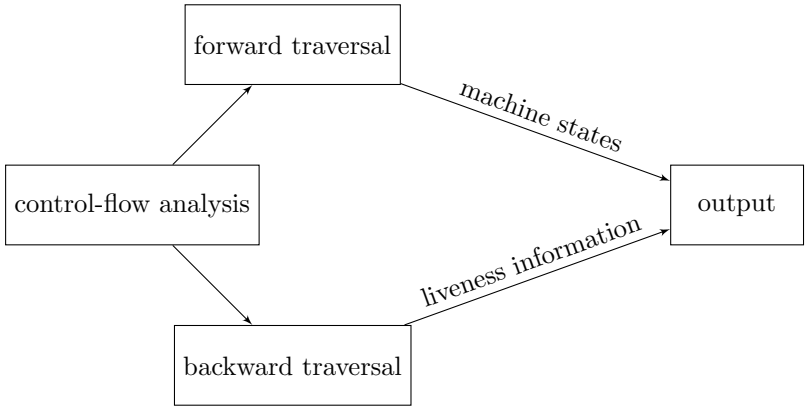

Figure 2: The three analysis passes

1.5 Running example

To demonstrate the output of our algorithm, we use this simple C* program as a running example:

```
uint64_t x;

uint64_t main() {
```

```
    uint64_t a;

    if (x) {
        a = 1;
        a = a * 2;
    }
    else {
        a = 0;
    }

    return a;
}
```

The actual value of `x` does not matter, as we only analyze the code generated for the main function and the concrete value has no impact on our analysis passes. The RISC-U assembly code is given in Figure 3 in control-flow graph form, and the corresponding output of our register value and liveness analysis is given in Section 2.4.7 and Section 2.5.7 respectively.

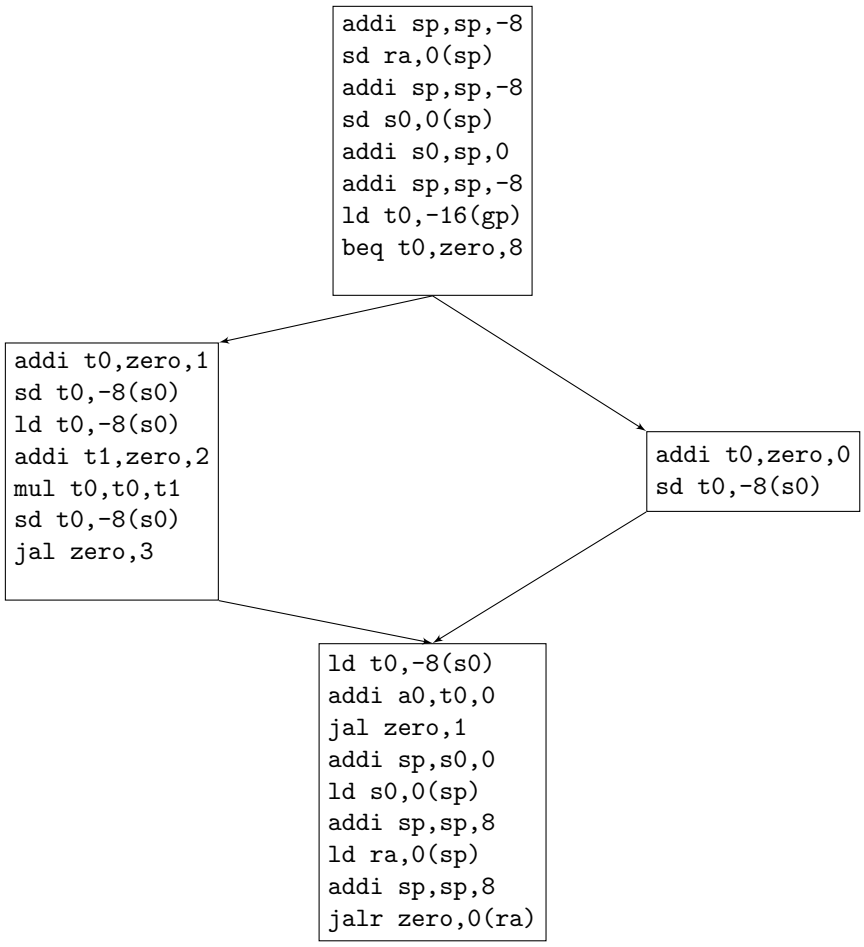

Figure 3: Control-flow graph of our running example

2 Design and Implementation

2.1 Basic Approach

To keep the implementation similarly simple to the selfie compiler, we do not use any intermediate representation, but instead work directly with the disassembled machine code of the binary. The basic idea is to traverse the binary like a CPU would while following all possible branch paths. While traversing the binary, we collect information about the known machine state at each instruction. After the analysis finishes, we have some information about the machine state at each instruction. In a separate pass, we traverse the binary backwards to collect information about the liveness of registers. As shown in Section 3, the time and space complexity of this process is $\mathcal{O}(n)$ with n being the number of instructions

in the binary being analyzed. Like selfie, the analyzer is implemented in C*.

2.2 Requirements

This approach relies on being able to infer all possible execution paths from the binary, e.g. being able to recover the full control-flow graph (CFG). Therefore, dynamic function calls (which are not part of C* anyway) are not supported. Furthermore, this analysis algorithm is not ideal for programs which have loops in their CFGs, i.e. those with recursive function calls. While it still produces correct results, some information is lost at the instructions where the loops occur, as described in Section 2.4.6 and Section 2.5.6.

2.3 Building the Control-Flow Graph

Both forward and backward traversal require some information about the control flow, which is collected at this stage. For forward traversal, we only need to know where control-flow paths end due to `exit` being called. Backward traversal requires an inverse CFG where the edge directions are reversed, meaning each instruction's edges now point to all instructions it can be reached from. It also uses a lookup table that maps the address of an instruction to the function it belongs to.

Due to the simplicity of the RISC-U instruction set and selfie's code generation, the control-flow graph is easy to recover without having to resort to advanced analysis techniques. This simple algorithm works similarly to the forward traversal algorithm described in Section 2.4.3, with the key difference being that each (reachable) instruction only gets processed once.

`jalr` would be tricky to handle as it does not have a static jump target, but in the code generated by selfie it only occurs in the form `jalr zero,0(ra)`, which is used to return from functions. Note that selfie emits this instruction as part of the function epilogue, and any `return` statements are implemented as jumps to the epilogue. Therefore, we simply have to add one edge from each `jalr` instruction to every instruction immediately following a call to the corresponding function.

Syscalls can be treated the same as function calls, except for the `exit` syscall, which we can identify by the preceding `addi a7, zero, 93` instruction. In binaries generated by selfie, only the hand-coded implementation of the `exit` function actually contains this syscall. We store the address of this function, as it is later used to identify function calls that are actually calls to `exit`, i.e. ones that are guaranteed not to return.

2.4 Forward Traversal

2.4.1 Overview

This analysis pass traverses the binary from the entry point to all exit points, similar to normal execution. Instead of keeping track of exact register values like

a CPU or emulator would, we only keep track of concrete register values that are always the same at a certain instruction across all possible execution paths. This is achieved by storing a known state for each instruction and updating it whenever the exploration of the current execution path gets there.

Pseudocode for forward traversal is given in Listing 1, and the following sections provide a detailed description of how it works.

```
update_state(instruction, new_state):
  if there is no state at instruction:
    set state at instruction to copy of new_state
  else:
    merge new_state into state at instruction

traverse_recursive(pc, prev_pc, current_ra):
  loop:
    if prev_pc is not -1:
      tmp_state <- copy of state at prev_pc,
                   or initial state if none exists
      apply effects of instruction at prev_pc to tmp_state
      if there is no state at pc:
        set state at pc to copy of tmp_state
      else if merging tmp_state into state at pc resulted in no changes:
        if current_ra is not -1:
          current_func <- peek call stack
          if call stack contains current_ra more than once:
            update_state(current_ra, unknown state)
          else if there is a cached state for current_func:
            update_state(current_ra, cached state for current_func)
          return

    fetch and decode instruction at pc
    if type is beq:
      if rs1 and rs2 are known:
        if rs1 and rs2 are equal:
          traverse_recursive(pc + imm, pc, current_ra)
          return
      else:
        traverse_recursive(pc + imm, pc, current_ra)
    else if type is jal:
      if rd is $ra:
        push pc + imm to call stack
        traverse_recursive(pc + imm, pc, pc + instruction size)
        pop call stack

        if function at pc + imm does not return:
          return

        prev_pc <- -1
      else if rd is $zero
        pc <- pc + imm - instruction size
    else if type is jalr:
      if rd is $zero:
        update_state(current_ra, state at pc)
        current_func <- peek call stack
        if there is no cached state for current_func:
          set cached state for current_func to copy of state at pc
        else:
```

```
          merge state at pc into cached state for current_func
        return

    else if type is ecall:
      if $a7 is known and equal to value for exit syscall:
        return

    pc <- pc + instruction size

traverse_recursive(0, -1, -1)
```

Listing 1: Pseudocode for forward traversal

2.4.2 Machine state

The most important concept being used is that of a known machine state, as it is essential to the algorithm's execution while also being its output once it terminates. While the full machine state would include all memory values, we define our notion of a machine state as the set of CPU register values only. Each state is stored as a data structure containing two pieces of information about every CPU register (excluding the program counter): a flag that indicates whether its concrete value is known and the value itself. If the flag is set to false, the stored register value has no meaning. Without such a flag, there would have to be a "magic" register value that indicates that it is actually unknown, which would cause problems if it was used in the program being analyzed.

When we talk about the machine state "at" a certain instruction, we refer to the state at the moment right before the instruction is executed, with `pc` already pointing to it.

2.4.3 Core algorithm

The main part of the algorithm consists of taking the machine state at the previous instruction, applying the effects of the previous instruction to it, and updating the machine state at the current instruction. That is, if there is no state yet, it is simply set to this newly generated one. Else, the state at the current instruction is set to the result of merging the new state with the current one. If this process does not result in any changes to the state at the current instruction or an `exit` syscall is hit, the exploration of the current execution path is terminated. Otherwise, the same procedure is repeated for the next instruction.

Most of the time, the next instruction is the one immediately following the current one in the binary. Unconditional jumps are also trivial to handle, as the next instruction is simply the jump target. However, a conditional branch (`beq`) actually results in the path splitting – this is handled by first exploring the branching path and then continuing to explore the fall-through path. If the values of the registers used as the branch condition are known, we only explore the corresponding path.

Function calls require special treatment, as the control flow continues at the next instruction at the call site as soon as a function returns. Whenever the algorithm reaches the end of a function, it merges the current state with the function's cached state and also the one at the instruction where the execution will continue. After the function has been fully explored, the traversal is then continued starting from that instruction, unless we know that the function does not return (i.e. it is the `exit` implementation). Additionally, whenever a path stops being explored early because merging resulted in no changes, the function's cached state is merged with the one at that instruction. This is necessary to ensure that the effects of a function are still factored into the next machine state at the call site without having to fully explore the function again.

Lastly, if there is no previous instruction, e.g. at the entry point, the unknown state is used instead. This state has every register's value set to unknown, except for the hardwired zero register, which must always be `0` by definition.

2.4.4 Instruction Effects

- `lui`, `addi`, `add`, `sub`, `mul`, `divu`, `remu`, `sltu`

  These are the most straightforward instructions to handle, as they do not involve any memory reads/writes or other side effects. With the exception of `lui` and `addi`, which involve immediate values, they all read values from two source registers and write the result to a destination register. The instruction effects in our machine state model are therefore easily defined: If all input values are known, the output value is also known and trivially computed. If any input value is unknown, the output value is also set to unknown.

- `ld`

  As we do not track memory accesses, the destination register is always set to unknown.

- `sd`

  This instruction has no effect on our machine state as it only writes to memory, which is not tracked.

- `beq`

  Only changes the program counter and therefore does not affect our machine state.

- `jal`, `jalr`

  These instructions store the current program counter in the destination register and modify the program counter. As we do not keep track of the concrete program counter value, the destination register is set to unknown.

- `ecall`

Performing an `ecall` results in the system call being performed and the result being put in the `a0` register, which we have to set to unknown. The only exception is the `exit` syscall, which stops the execution of the program and is therefore out of scope for our tracked machine state, but relevant for the control-flow analysis.

Of course, most of these instructions also have the trivial side effect of incrementing the program counter by the instruction size.

2.4.5 Merging

Merging of states is an operation that takes two states as input and produces a new state. In our implementation, this operation is performed in-place, i.e. the result is stored in one of the input states as opposed to a new third state, as the old state that got "merged into" is no longer needed in our algorithm.

The idea behind this merging process is to combine the information of two machine states by keeping common register values and setting conflicting or unknown values to unknown. This process can be thought of as computing the set intersection of two states, as only the information that is common to them is included in the result. It forms the core of our algorithm, as it is used at every instruction to update the known state to only include information that is shared between all possible execution paths through the binary.

The actual implementation is very simple. For each register, the following rule is applied: If both states have the same known value, then the merged state gets assigned that value. Otherwise, it is set to unknown.

2.4.6 Recursion

While this algorithm works fine for simple programs, it falls apart when recursive functions are involved. Consider this piece of code:

```
uint64_t recursive(uint64_t n) {
    if (n != 1) { // beq
        if (n == 0) { // beq
            // addi t0,zero,0
            return 0;
        }
        recursive(0);
        // wrong assumption here: t0=1, when in reality t0=0
    }
    // addi t0,zero,1
    return 1;
}
```

This function, admittedly with a rather contrived control flow, simply returns `1` when the argument `n` is `1` and `0` otherwise. The source code listing here includes some assembly fragments from the binary generated by selfie, necessary

for understanding how it is explored. Let us take a high-level look at how the algorithm explores this function to figure out what exactly is happening:

1. The algorithm starts exploring the function at its entry point
2. At the first `beq` instruction, it initially explores the branching path.
3. As part of the machine code for the `return 1` statement, the integer literal `1` is loaded into the register `t0`. This information is added to the machine state before the return instruction.
4. At the return instruction, the current path terminates. Since the function's cached state was previously undefined, it is now set to the current state.
5. Next, the fall-through path of the first `beq` is explored, starting with the branching path of the second `beq`.
6. The algorithm gets to the recursive call and follows it, meaning the function is explored again from its entry point.
7. Merging does not result in any changes to the saved state at the start of the function, so the call exploration is terminated and the function's cached state is applied to the next instruction at the call site. This means that the state after the recursive call now includes `t0=1` – which is wrong, as `t0` would actually be `0` at runtime.

The issue is caused by the fact that there is another possible path through the function where `t0` ends up being `0` right before returning, but due to the interaction between the function's control flow and the algorithm's exploration strategy, it has not been explored yet at the point in time when the recursive call gets followed. A sophisticated solution to this problem could involve a non-greedy approach where the possible states at return sites are explored before following other, potentially recursive function calls. In the interest of sticking to the original idea and simple implementation, we decided to work around this issue by specifically checking for recursion: Whenever the exploration in a function call terminates early and the current call is either directly or indirectly recursive, the state information at the call site is set to the unknown state. This is implemented by maintaining a virtual call stack that contains the functions currently being explored: If the current function appears multiple times on this stack, it means that there are still pending traversals that could affect its cached machine state.

While this approach results in all known register values being lost, it completely mitigates this issue, as register values being unknown can always be considered "correct". Over the entire binary, this mitigation barely affects temporary registers, since they frequently get reassigned. However, `gp` only gets set once at the start of the program and its value is therefore lost forever once we apply the unknown state. For this reason, it makes sense to treat it as a special exception and always preserve its value.

2.4.7 Example

Figure 4 shows the output of the forward traversal algorithm for our running example. For each instruction, the known register values are shown next to it. As we do not track local variables, known register values get lost whenever the value of a variable is loaded from the stack. Therefore, the result of the `mul` instruction cannot be computed, even though it would be constant. Also, it can be seen that merging the paths with `t0=0` and `t1=2` results in neither register being known, as they are unknown in one path each.

Note that `gp` and `a7` have constant values throughout this example, but are omitted for brevity.

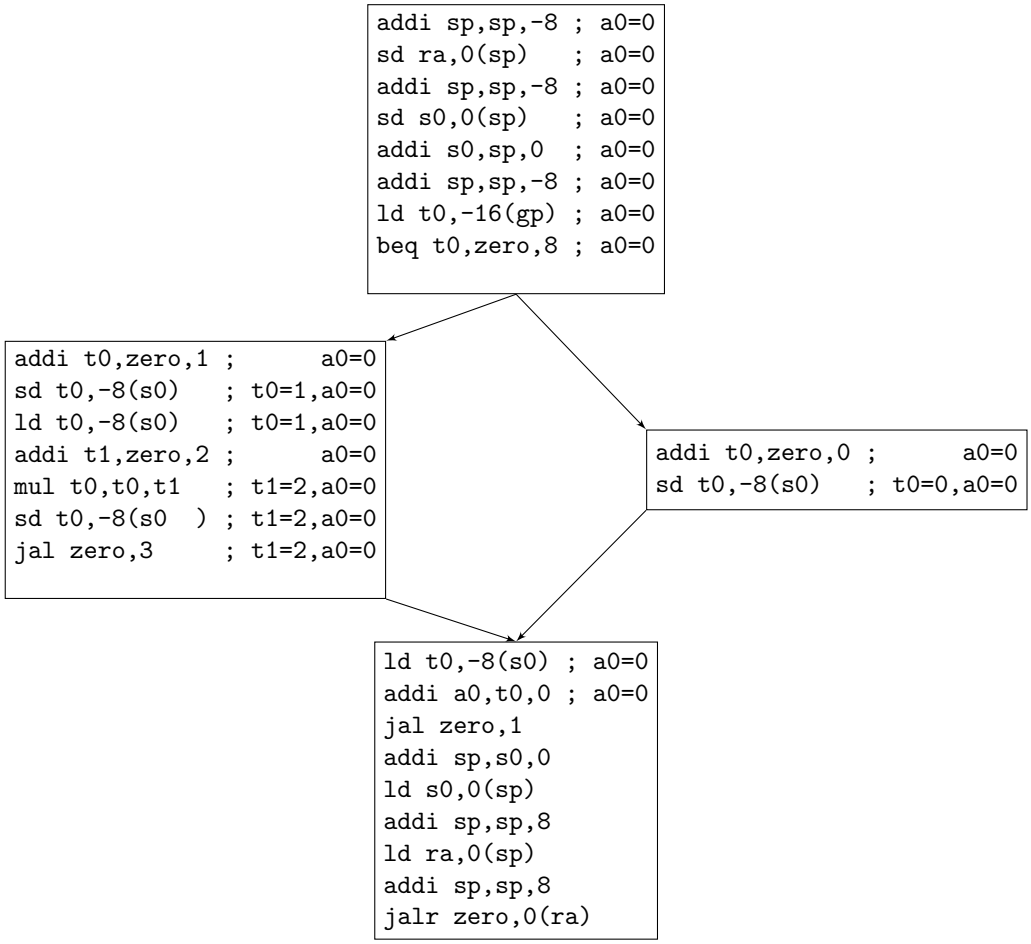

Figure 4: Output of forward traversal for our running example

2.5 Backward Traversal

2.5.1 Overview

This analysis pass has many similarities with forward traversal. As described by Appel, "a variable is live if its current value will be used in the future, so we analyze liveness by working from the future to the past." [2, p. 218] In our case, this corresponds to traversing the binary backwards from the exit point(s) to the entry point. Instead of looking at concrete effects of instructions on register values, we only have to consider which registers a given instruction reads from or writes to. The two analysis passes are complementary and also independent from each other, as they do not rely on each other's results.

Pseudocode for backward traversal is given in Listing 2, and the following sections provide a detailed description of how it works.

Note: `livedead` is used here to refer to liveness information.

```
update_livedead(instruction, new_livedead):
  if there is no livedead at instruction:
    set state at instruction to copy of new_livedead
  else:
    merge new_livedead into livedead at instruction
  if livedead at instruction changed:
    return true
  return false

recursive_livedead_function(pc, prev_pc, call_pc):
  loop:
    if there is a livedead at prev_pc:
      tmp_state <- copy of livedead at prev_pc
    else:
      tmp_state <- new livedead

    apply livedead effects of instruction at pc to tmp_livedead

    if there is no livedead at pc:
      set livedead at pc to tmp_livedead
    else if merging tmp_livedead into livedead at pc resulted in no changes:
      if call_pc is not -1:
        current_func <- peek call stack
        if call stack contains call_pc more than once:
          update_livedead(call_pc, unknown livedead)
        else if there is a cached livedead for current_func:
          update_livedead(call_pc, cached livedead for current_func)
      if prev_pc is not -1:
        return

    if pc is first instruction of function it belongs to:
      if call_pc is not -1:
        update_livedead(call_pc, livedead at pc)
      return
    else:
      if instruction at pc is a return target:
        for all in-edges to pc in CFG:
          parent_pc <- address of in-edge origin instruction
```

```
          func <- get function that parent_pc belongs to
          push func to call stack
          recursive_livedead_function(parent_pc, pc, pc - instruction size)
          pop call stack
        prev_pc <- -1
        pc <- pc - instruction size
      else:
        for all in-edges to pc in CFG, except last:
          parent_pc <- address of in-edge origin instruction
          recursive_livedead_function(parent_pc, pc, call_pc)

        prev_pc <- pc
        pc <- address of last in-edge origin instruction

recursive_livedead_helper(pc):
  func <- get function that instruction at pc belongs to
  push func to call stack
  recursive_livedead_function(pc, -1, -1)
  pop call stack

  for all in-edges to func in CFG:
    parent_pc <- address of in-edge origin instruction
    if update_livedead(parent_pc, livedead at func):
      recursive_livedead_helper(parent_pc)

for all exit syscalls in binary:
  recursive_livedead_helper(address of exit syscall)
```

Listing 2: Pseudocode for backward traversal

2.5.2 Liveness Information

For each instruction, we simply store a flag that indicates whether a given register is live at this point in the execution of the program. As per the definition of liveness, this indicates whether a register's value may be used at any point in the future.

We define the notion of the liveness information "at" a certain instruction to refer to the point in time right before it gets executed. For example, a register is always set to live at an instruction that reads from it.

2.5.3 Core Algorithm

The algorithm is built on the same principle as the forward traversal, but the fact that we now traverse the binary backwards changes a few things: Notably, we have to utilize the inverse control-flow graph as we otherwise could not always determine the previous instruction from the current one. Additionally, call/jump/branch targets can have multiple "previous" instructions, e.g. when a function is called from multiple places.

We start traversing the binary from all exit points, which binaries generated by selfie only have one of. Now, for each instruction, we follow all in-edges, which are the out-edges in our inverse control flow graph, to the "previous" instructions. Similar to the forward traversal, we create a copy of the liveness

information at the instruction we just came from, apply the effects of the current instruction to it, and merge it with the information at the current instruction.

Again, "function calls" require special treatment. Here, we are dealing with two different concepts that can be considered as such. The easier one to deal with is a "function call" appearing inside the function we are currently exploring, where we reach a return target, i.e. an instruction immediately following a `jal` function call. We can easily identify such instructions using the return target information we collected during the CFG phase. As in the forward traversal, we simply traverse the called function backwards, then merge the resulting liveness information with the one at the current call site and continue from there.

The second type of "function call" occurs when we reach the first instruction of the function we are currently exploring. Here, the path can split, as we must now continue exploring from each instruction that is a call to this function. These call instructions are easily located using the inverse CFG.

2.5.4 Instruction Effects

Most instructions' effects on liveness are obvious: Every source register is considered live, as the instruction will read from it. The destination register is set to dead, as its previous value will be overwritten. In case a register is used both as source and destination, it must be considered live, as its previous value will be read first.

`ecall` must be handled differently, as it has implicit source and destination registers: The syscall number is read from `a7` (live), parameters are read from `a0-a3` (live) and the result is written to `a0` (not dead here, due to its potential use as a parameter). We can treat the first `ecall` we process, which is part of the `exit` implementation, as a special exception and only set `a7` to live.

2.5.5 Merging

Merging of liveness information works similarly to that of machine states. It can be interpreted as computing the union of two sets of live registers. In other words, any register that is set to live in at least one of the inputs is set to live in the output. Registers that are dead in both inputs are also dead in the output.

2.5.6 Recursion

We face the exact same issue as with forward traversal again – any recursive calls must be handled specially; otherwise the algorithm will produce wrong results. The same workaround can be used: By keeping track of all functions currently being traversed, we can detect recursive "calls" and set the liveness information to unknown, i.e. set every register to live, if we return early from such a recursive exploration. This mitigates the issue because registers erroneously being considered live does not lead to "dangerous" output, as without this analysis pass, registers would be considered live by default – the interesting information is specifically which registers are guaranteed to be dead at which

point. While this workaround does not have a significant effect on most temporary registers, as they are frequently set, it has the unfortunate side effect of also marking registers that are never read from as live. In theory, this could be avoided e.g. by scanning the binary to find out which registers are never read from, and setting the "unknown" liveness information accordingly, but we did not implement this.

2.5.7 Example

Figure 5 shows the output of the backward traversal algorithm for our running example. For each instruction, the currently live registers are shown next to it. We can see that the temporary registers are only live for a few instructions at a time at most, as they are frequently overwritten. At the end, most registers are dead because they are never read from again before the program exits. The remaining registers are live because they are used by the `exit` implementation executed after the main function returns.

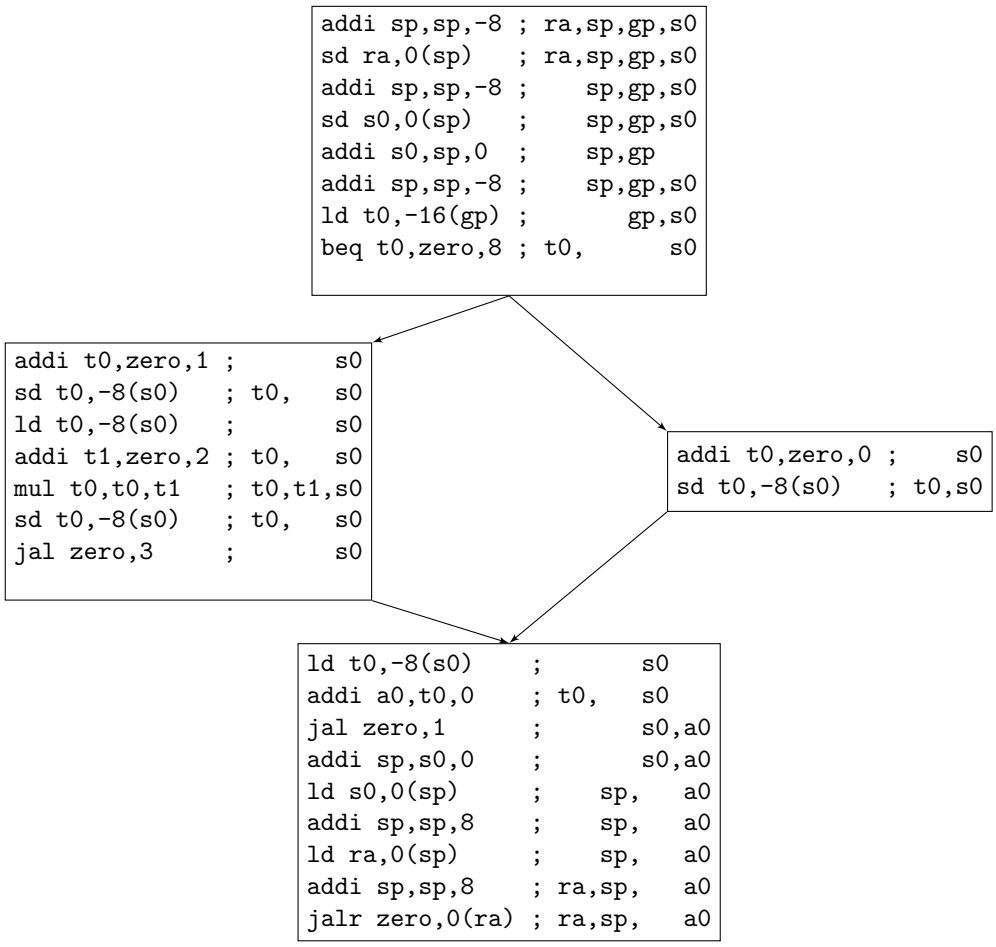

Figure 5: Output of backward traversal for our running example

3 Algorithmic Complexity

Based on the properties of the merging process and the definition of the algorithm, it is easy to prove that its complexity is linear in the number of instruction, both in terms of time and space.

Recall that, due to the nature of the RISC-U instruction set, each instruction in our control-flow graph can only have up to two out-edges, with the exception of `jalr`. In binaries generated by the selfie compiler, it is used to return from functions, so each `jalr` has one out-edge per call to the corresponding function. The number of these out-edges is bound by the total number of function calls and therefore the number of instructions, as selfie only emits one `jalr` per

function. Therefore, we can establish an upper bound of $2 * n + n = \mathcal{O}(n)$ edges, with n being the number of instructions.

3.1 Time Complexity

Recall the algorithm for merging two states: For each register of the states being merged, the resulting state will either assume the same value or have the register marked as unknown. This means that for the state at any given instruction, the number of unknown registers will increase monotonically, potentially up to the point where all registers are unknown. Combined with the property of our algorithm that a branch stops being explored when merging results in no changes to the state, this means that the number of times a path continues to be traversed after updating the state at any given instruction is linear in the number of registers. Since this is a constant, forward traversal can only continue $\mathcal{O}(1)$ times after each instruction.

This also holds for backward traversal, as the traversal and merging strategies are based on the same principle, with the difference that we only store a liveness flag instead of concrete values.

As the operations performed per instruction, like applying instruction effects and merging, only take constant time, the overall time complexity of our algorithm will depend on how many instructions it processes. We can use the properties established above to prove that this number is $\mathcal{O}(n)$: Each of the $n = \mathcal{O}(n)$ instructions can be processed in such a way that merging results in changes and the path continues to be explored only $\mathcal{O}(1)$ times, resulting in $\mathcal{O}(1) * \mathcal{O}(n) = \mathcal{O}(n)$ instructions being processed. When a path continues to be explored, all out/in-edges are followed in forward/backward traversal to find the next instruction(s) to process. Since the total number of edges is $\mathcal{O}(n)$, this can only result in $\mathcal{O}(1) * \mathcal{O}(n) = \mathcal{O}(n)$ additional instructions, where merging results in no changes and the path stops being explored, being processed. This brings the total number to $2 * \mathcal{O}(n) = \mathcal{O}(n)$.

3.2 Space Complexity

Since we have to allocate a constant amount of memory per instruction to store things such as the edges, machine states and liveness information, the overall space complexity is $\Omega(n)$. Due to the recursive nature of the traversal implementations, it is also tied to the maximum recursion depth. From the property that our algorithm will process $\mathcal{O}(n)$ instructions in total, and the fact that the recursion depth can only increase by one per instruction, it follows that this number is also $\mathcal{O}(n)$. Each nested call only requires a constant amount of memory for its local variables, so the total space required is $\mathcal{O}(n)$.

4 Results

To evaluate the performance and confirm the complexity empirically, we ran the analysis passes on builds of selfie and the analyzer itself, generated by the selfie compiler. The execution time was measured using the `time` utility and an analyzer build compiled with GCC without any optimizations, to prevent features such as dead code elimination from falsifying the results. As the output of the algorithm is not saved, the only I/O involved here is reading the input binary and printing the numbers seen in the table. Also, it should be noted that a lot of the instructions in the analyzer binary are unreachable, due to it being linked against the entirety of selfie while only using a few of its functions. The results are shown in Table 2.

binary	instructions	state updates	liveness updates	max depth (forward)	max depth (backward)	execution time
selfie	34838	43431	46113	68	53	∼310 ms
analyzer	41396	17035	19215	46	19	∼150 ms

Table 2: Results

The number of state and liveness information updates when analyzing the selfie binary is consistent with our complexity analysis, with the former being ∼25% and the latter being ∼32% greater than the number of instructions. As expected, the maximum recursion depth of our algorithm is multiple orders of magnitude less than the number of instructions. For this number to be larger, the programs would need to contain a significant amount of branches or function calls arranged in such a way that a large execution depth is reached before merging stops resulting in changes, e.g. function `a` calls function `b`, which calls `c`, which calls `d`, etc. The recursive function calls present in selfie are not sufficient, as merging soon stops resulting in changes due to the properties described in Section 3.1. The measured execution time is negligible but clearly scales with the number of updates.

5 Conclusion

We introduced a C* implementation of algorithms for static analysis of RISC-V binaries in linear time and space. Since it works directly on the input binary, it is extremely fast in practice with sub-second execution times. However, due to the naive implementation, it delivers significantly worse results for binaries containing recursive function calls, as we have to discard a lot of information to ensure the output remains correct. The output of the analyzer can be used in future work to facilitate binary optimizations such as dead code elimination and peephole optimizations.

References

[1] *The Selfie Project*. `http://selfie.cs.uni-salzburg.at/`.

[2] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge: Cambridge University Press, 2004. ISBN: 978-0-521-60765-0.