`atoi` Proof of Correctness Report – Team 2

Matthew Sheldon
matthew.sheldon@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Isabella Pereira
isabella.pereira@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Jarrod Rogers
jarrod.rogers@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Naja-Lee Habboush
naja-lee.habboush@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Brandon Wang
brandon.wang@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

I. Introduction

We have been working on proving the formal correctness of the `atoi` function using Picinae on the ARMv8 architecture. The formal verification of `atoi` will allow us to prove exactly how `atoi` works for any given input, which provides the guarantee that it is free of bugs and works as expected at all times, which closes the possibility of a bug or undocumented feature in `atoi` being exploited by attackers. Since `atoi` and functions like it are widely used by modern software, an exhaustive proof of its behavior is also an important step to the formal verification of other software projects that rely on it. Even in other projects which don't use `atoi` directly, but rather a similar function like `atol`, the work we do to prove the correctness of `atoi` may lay the groundwork for proofs on similar functions. Since a proof of `atoi` on the ARMv8 computer architecture using the Picinae system has never been done before, our work on `atoi` will help to advance the state-of-the-art and improve the Picinae system.

II. Overview of atoi

The `atoi` (ASCII-to-integer) function takes, as input, a string, and converts the "initial portion" of said string into a corresponding 32-bit integer. It does this by converting each numeric ASCII character into its associated numeric value one character at a time, and adds each converted value up to produce the final number.

The initial portion of a string starts from the beginning of the string, consists of potentially leading whitespace, an optional sign character, and ends after the first non-numeric sequence in the string. At the start of the numeric sequence, a single "+" or "−" will determine the sign of the number output by `atoi` (providing neither will cause `atoi` to treat it as a positive number). Excepting the sign, the only characters allowed between the start of the string and the start of the numeric sequence in the initial portion is whitespace, which is discarded when `atoi` is processing the number. If this rule is violated, for example by having an alphabetic character occur before the first numeric sequence, we call the input "ill-formed", as `atoi` will return **0**. Figure 1 shows a control flow graph for the disassembly of `atoi`.

III. Challenges

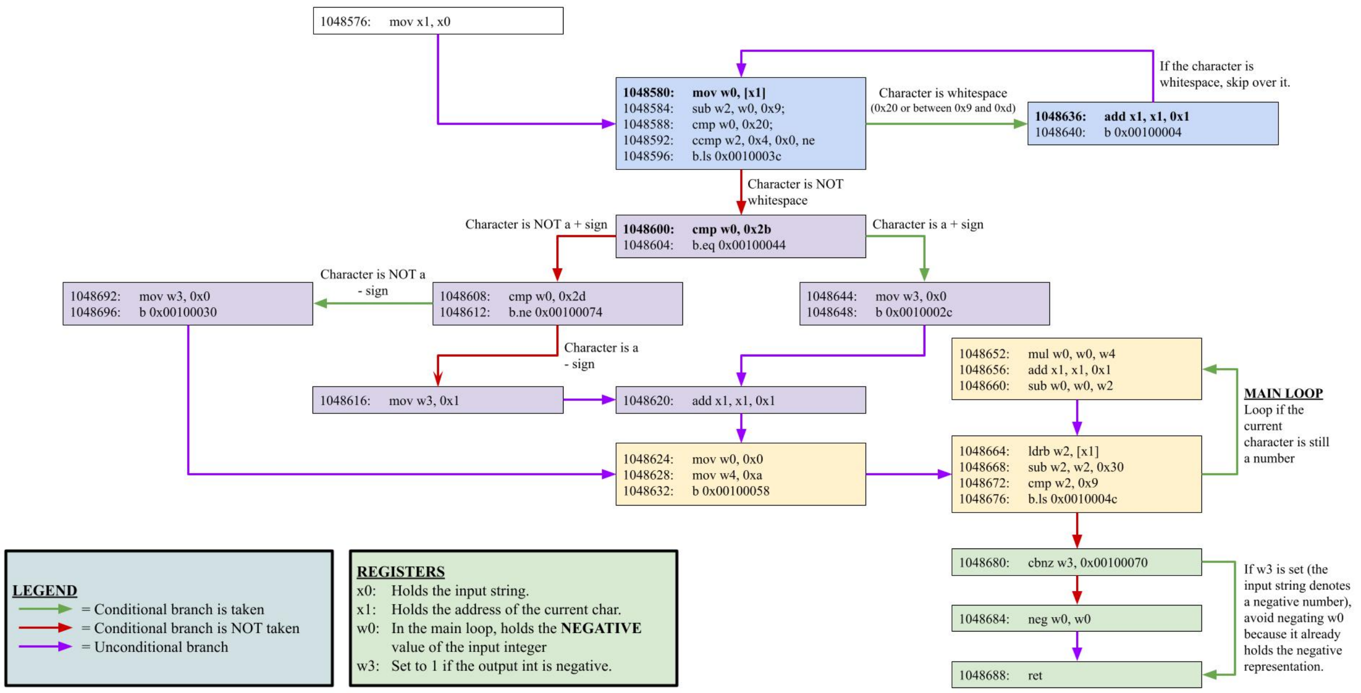
Throughout the course of our work on this project, we encountered several challenges that made proving the correctness of `atoi` more complicated than originally anticipated. Here we detail these challenges.

A. *Overflow/Underflow*

One of the first problems we ran into was determining how to handle integer overflow or underflow scenarios. If the string representation a number greater (or less) than what can be held in a 32-bit signed integer is used as input to `atoi`, it will result in an overflow/underflow and produce a value that doesn't properly correspond to the full input string. `atoi` explicitly does not detect overflow or underflow errors, and will *not* return `0` in the case of such an error. We observed that our specific implementation of `atoi` will simply return the lower 32-bits of what the n-bit representation of the number would have been.

Our initial approach involved using regular expressions on the string to verify that the numeric sequence within would not result in an over/underflow. The principle idea is guided by knowing the bounds of a 32-bit signed integer are static: -2^{31} (-2,147,483,648) on the low end, and $2^{31} - 1$ (2,147,483,647) on the high end. We also figured that since we had been working with regular expressions in the homeworks leading up until this, we would likely be able to come up with something that worked with minimal effort. Because of this, we thought we could craft a regular expression to detect numbers that are outside these bounds in order to anticipate an over/underflowed result. However, as we quickly discovered, this approach was fundamentally flawed.

Since the bounds are static, we knew that any number with between zero and nine digits will not over/underflow. Input numbers with ten or more digits, however, is where this solution starts to get tricky. A number with ten or more digits *could* be a full number that will over/underflow, but *could* also be a small number with many leading zeros, which would not overflow. A solution based on regular expression detection would need to account for these details, and be able to handle cases with a higher precision than just the number of digits (for example, "2147483647" would not result in an overflow,


Character is whitespace
(0x20 or between 0x9 and 0xd)
If the character is whitespace, skip over it.
Character is NOT whitespace
Character is NOT a + sign
Character is a + sign
Character is NOT a - sign
Character is a - sign
MAIN LOOP
Loop if the current character is still a number
If w3 is set (the input string denotes a negative number), avoid negating w0 because it already holds the negative representation.
LEGEND
= Conditional branch is taken
= Conditional branch is NOT taken
= Unconditional branch

```
1048576:    mov x1, x0

1048580:    mov w0, [x1]
1048584:    sub w2, w0, 0x9;
1048588:    cmp w0, 0x20;
1048592:    ccmp w2, 0x4, 0x0, ne
1048596:    b.ls 0x0010003c

1048636:    add x1, x1, 0x1
1048640:    b 0x00100004

1048600:    cmp w0, 0x2b
1048604:    b.eq 0x00100044

1048692:    mov w3, 0x0
1048696:    b 0x00100030

1048608:    cmp w0, 0x2d
1048612:    b.ne 0x00100074

1048644:    mov w3, 0x0
1048648:    b 0x0010002c

1048616:    mov w3, 0x1

1048620:    add x1, x1, 0x1

1048624:    mov w0, 0x0
1048628:    mov w4, 0xa
1048632:    b 0x00100058

1048652:    mul w0, w0, w4
1048656:    add x1, x1, 0x1
1048660:    sub w0, w0, w2

1048664:    ldrb w2, [x1]
1048668:    sub w2, w2, 0x30
1048672:    cmp w2, 0x9
1048676:    b.ls 0x0010004c

1048680:    cbnz w3, 0x00100070

1048684:    neg w0, w0

1048688:    ret
```

REGISTERS
x0: Holds the input string.
x1: Holds the address of the current char.
w0: In the main loop, holds the **NEGATIVE** value of the input integer
w3: Set to 1 if the output int is negative.

Fig. 1: Control Flow Graph (CFG) for `atoi`

but ”214748364**8**” would). For these reasons, we decided to abandon the regular expression approach and instead turn our attention elsewhere.

A more intuitive solution is to, separate from the *atoi* implementation being validated, first convert the string to its integer representation and then check to see if the result over/underflowed during the conversion. This would allow us to detect an over/underflow and adjust the expected result so that such behavior can be verified. While neat and intuitive, we initially dismissed this apprach as it would require re-solving the exact problem of implementing `atoi` and would mean verifying the correctness of two different `atoi` implementations. However, after some thought and discussion with the CS 6335 staff, we realized that this approach would be the most straightforward and efficient way to solve the problem. Namely, since the Gallina implementation of `atoi` would not be burdened with the bit-width restrictions of the ARMv8 architecture – using the built-in and structurally defined Integer definition, it would be able to handle the full range of input values that we would need to test. Then, all that would be left is to determine how many bits of memory would be required to store the integer representation of the number. This would allow us to verify that the input to `atoi` did not result in an over/underflow during the conversion.

B. “Proper” vs. “Improper” Inputs

Similar to the over/underflow issue is defining what a “proper” input to `atoi` looks like (i.e., an input that will not result in an error). While over/underflowing inputs result in undesired behavior, they are not detected by `atoi` as errors. Handling “ill-formed” strings that are detected as errors by `atoi` has also proven to be a challenge itself. The `atoi` function returns `0` when it detects such an erroneous string. Unfortunately, `0` can also be returned by valid conversions from strings with a numeric sequence equivalent to zero, which means that given an arbitrary string, there is no immediate way to tell if it is an “improper” input based on the output from `atoi` alone, which makes proving any property about the correctness of the input very difficult. We worked on trying to solve this issue for a while, but due to time constraints we eventually decided to compromise. The solution we ultimately settled on for this problem is to narrow the scope of our proof to only prove properties about the behavior of `atoi` on proper inputs, leaving validation of the input as a responsibility of the caller.

C. The State Explosion Problem

While working on the project, we observed some odd behavior while executing certain instructions. Anytime a `step` instruction was executed, especially after a `ccmp` operation, execution would take a very long time or bring execution somewhere outside of the expected execution path. In one instance, a `step` instruction executed for over ***two hours*** and then crashed. After code optimization we were able to get the execution time down to a more reasonable (yet still extremely slow) twenty or so seconds per `step` instruction executed. Considering that we have twenty-two step instructions in our main proof, this was still an unacceptably slow speed. Even not considering unacceptable performance, we still had trouble with an executing `step` instruction going rogue and landing at a non-invariant, unsteppable goal. After hours of debugging, testing, and code revisions, we decided to reach out to the CS 6335 staff for assistance with the problem. They were able to figure out that the problem was caused by a bug in the Picinae system which was causing `step` instructions to explode the state, leading to the time spent pointlessly executing unneeded instructions. After discovering the bug, the CS 6335 staff generously wrote up a patch for Picinae that would let us continue our work. Unfortunately, accommodating this patch required refactoring most of the code we had written prior, which, when combined with the time lost trying to find the issue, resulted in losing many valuable work hours through the course of overcoming this challenge.

D. Universal Quantification vs. Existential Quantification

Perhaps the biggest challenge we faced throughout working on this project was an incorrect usage of universal qualifiers throughout the project. We went into the implementation phase of this project with the goal of thoroughly proving the behavior of `atoi` in all possible scenarios. This philosophy, noble as it was, was applied throughout the entire project, including nearly every invariant. This became highly problematic because it multiplied our workload despite being unnecessary for many of our invariants to be exhaustively proven. Worse yet, since the state of many of our invariants relies on the state of invariants before them, fixing this mistake necessitated a refactor of nearly every invariant we had written up to that point, which unfortunately consumed many valuable work hours.

IV. Implementation Strategy

In order to streamline work and organize code, we broke the formal proof into the following components:

A. Major theorems

- `atoi_partial_correctness`: Our primary proof of correctness for `atoi`, limited to scenarios featuring valid inputs, which we were successful in proving. A majority of our proof-related logic is concentrated here.
- `bit_count_correctness`: The proof that the bit counter is correct.
- `whitespace_handler_correct`: The proof that the whitespace handler properly handles whitespace.

B. Preconditions

V. Breakdown of Contributions

Many responsibilities were shared by everyone in the team. These include:

- Reviewing pull requests.
- Attending and contributing to team meetings.
- Contributing to the presentation by brainstorming, working on the slideshow, or presenting.

The contributions made by individual members of our team are as follows (in no particular order):

A. Matthew Sheldon

- Set up repository and environment structure.
- Developed early versions of proof invariants.
- Developed helper lemmas to prove non-whitespace loop-related goals.
- Proof of whitespace-related invariants.
- Proofreading and small adjustments for the report.
- Managed internal team proceedings and final deliverables.

B. Isabella Pereira

- Helped the group plan after receiving guidance from Professor Hamlen.
- Primary contributor to the bit counter and subsequent proof.

C. Jarrod Rogers

- Primary contributor to the project report.
- Assisted with managing the project structure.
- Ensured that all work supported a Linux workflow in addition to Windows.

D. Naja-lee Habboush

- Started the invariants section.
- Defined the Gallina program of `atoi` in Coq.
- Primary contributor to the proof for the Gallina implementation of `atoi`.

E. Brandon Wang

- Conducted initial analysis of disassembled code.
- Creation of the `atoi` CFG (Figure 1).
- Refinement of invariants created by other members of the team.
- Primary contributor to the proof of non-whitespace loop-related invariants.

VI. Evaluation

Looking at how far we were able to take our project, we were able to achieve many of the goals we had originally set out to fulfill. Although our work is incomplete, we are proud of what we accomplished given the time constraints, limited experience, and multitude of challenges we faced in both planning and implementing our formal verification of `atoi`. Some of the things we tried ended up not working, which is discussed in greater detail in section III.

Here are some of the things we are proud of achieving:

- A full Gallina specification of `atoi`
- A mostly-complete formal verification of `atoi` on valid inputs

While were unable to complete this project in the allotted time, we got very close. Here is what is still missing, which is further discussed in the next section:

- We did not finish the implementation of our invariants.
- We were unable to complete our self-specified implementation of `atoi`

VII. Future Work

The current state of the project is not where we had intended for it to be at this point. It is the hope of the authors that future work will be able to complete what we were unable to within the time we had to work. Currently, two major tasks still need to be completed.

The first of these tasks is the final invariant task. While we have completed the first two of three invariant-related tasks, we unfortunately ran out of time to finish the last one. In addition to the maintenance of the digit start index and counting the number of digits (as the invariants currently do), the digit loop invariants must also track the contents of register `w0` in relation to the digits that have been parsed. To do this, the `acc` parameter must be introduced to the digit loop invariants that don't already have it to denote the number of digits that have been parsed so far, and register `w0` must be recorded to contain some symbolic value equal to the sum (or more accurately, difference, as `w0` is subtracted from instead of added onto) of the digit values of all digit characters parsed so far, accounting for place value. For example, if the digit start index is `j` and we have parsed `acc`=3 digits, `w0` should contain a value equivalent to $-digit\_value(mem[p+j+0])*10^2 - digit\_value(mem[p+j+1])*10^1 - digit\_value(mem[p+j+2])*10^0$

The second of the remaining tasks is completing our self-specification of `atoi` and comparing its result to `w0`'s symbolic value using the same input. Our specification of `atoi` will take as input a memory `mem`, string address `p`, index of digit start `j`, and total number of digits `k`, and then compute the symbolic atoi result of the number denoted by `mem[p+j]` to `mem[p+k-1]` and compare it with the contents of `w0`, in the postcondition invariant (with an edge case of 0 digits being handled accordingly).

VIII. Conclusion

As a team we have learned a lot from working on the correctness for `atoi`. Even though there is still more work to be done for a complete `atoi` partial proof of correctness, we are proud of what we have accomplished, the challenges that we were not only able to identify, but also begin to overcome, and the work we have contributed towards advancing the state-of-the-art.