`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.babboush@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. This will help to advance the state-of-the-art and improve the Picinae system.

II. Overview of atoi

The **atoi** function takes a given input string, and converts the ”initial portion” of it into a corresponding 32-bit integer.It does this by converting each numeric ASCII character into it’s 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 and ends after the first 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”, and **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 a few 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 how to handle integer overflow or underflow scenarios. If a string representing a number which is greater (or less) than what can be held in a 32-bit signed integer is used as the input to **atoi**, it could overflow or underflow and produce a value that doesn’t properly correspond to the input string. **atoi** explicitly does not detect overflow or underflow errors, and will *not* return **0** in the case of such an error.

An intuitive solution would be to convert the string to it’s integer representation and then check to see if the result had over/underflowed during the conversion by **atoi**. This would allow us to anticipate an over/underflow and adjust the expected result so that such behavior can be verified. While neat and intuitive, this would require re-solving this exact problem in the implementation of **atoi** we use to convert the string to an integer in order to check for over/underflow, which would mean verifying the correctness of two different **atoi** implementations.

A different approach that we thought about would involve using a regular expression on the string to verify that the numeric sequence within it would not result in an over/underflow. The principle idea is guided by knowing the bounds of a 32-bit signed integer, -2^{31} (-2,147,483,648) on the low end, and $2^{31}-1$ (2,147,483,647) on the high end. Since we know what these bounds look like, we can craft a regular expression to detect numbers that are higher or lower than these bounds, in order to anticipate an over/underflowed result.

Unfortunately, this solution has it’s own complications. Since we know the bounds, we know that any number with anywhere in between zero and nine digits will not over/underflow. Input numbers with ten or more digits are 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, but ”2147483648” would).

B. ”Proper” vs. ”Improper” Inputs

Similar to the over/underflow issue is that of a ”proper” input, or one 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

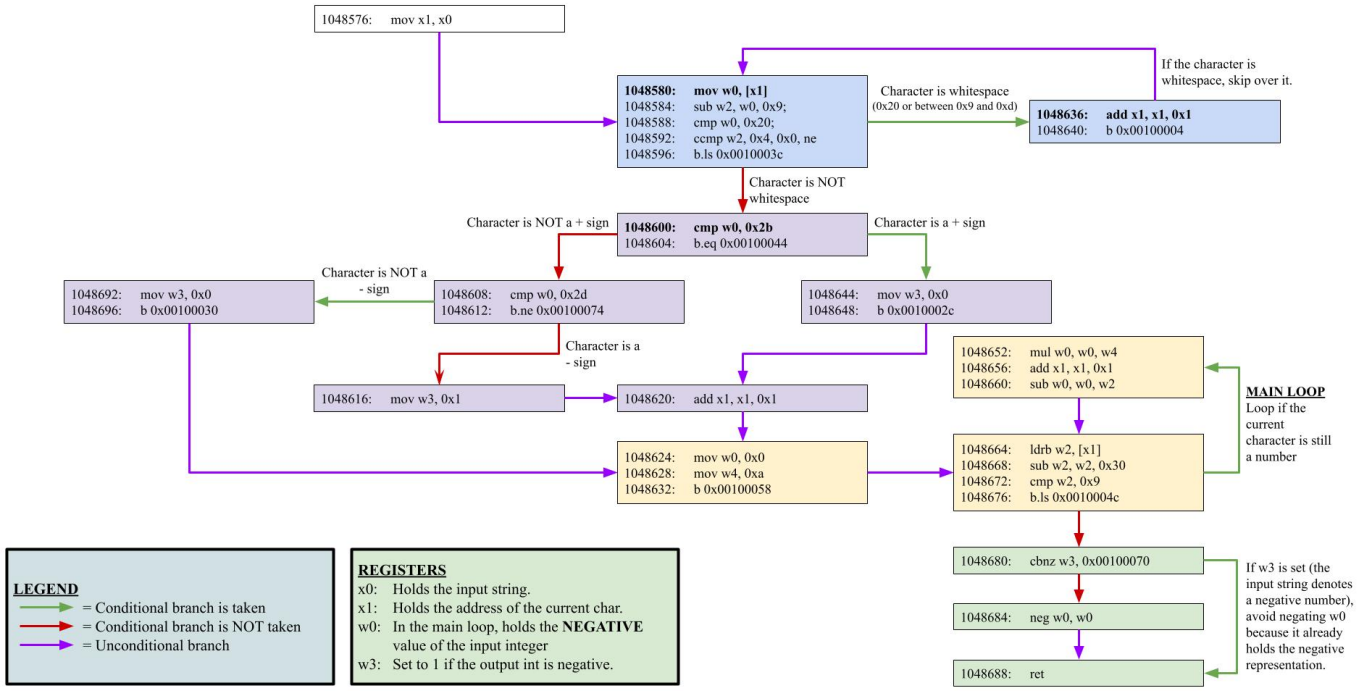

Fig. 1: atoi CFG

equivalent to zero, which means that given an arbitrary string, there is no 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.

IV. Implementation Strategy

A. Major theorems

B. Preconditions

V. Breakdown of Contributions

A. Matthew Sheldon

- TODO

B. Isabella Pereira

- TODO

C. Jarrod Rogers

- TODO

D. Naja-lee Habboush

- TODO

E. Brandon Wang

- TODO

VI. Future Work

- TODO

VII. Evaluation

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, we are proud of what we have accomplished and the work we have contributed towards advancing the state-of-the-art.