1 Background

This chapter provides background information for the work that follows throughout the paper. First, we discuss the existing nonnull type qualifier extension implemented as a language extension to `ABLE`C. Next, we discuss the two new extensions written for this project: an extension allowing for easier use of asynchronous IO and an extension allowing the use of the Wuffs programming language in the parsing of messages.

1.1 Nonnull Qualifiers

One of the most common sources of bugs in the C programming language is errors related to null pointers. The dereferencing of a null pointer is considered undefined behavior in C (cite ISO/IEC 9899 clause 6.5.3.2 paragraph 4, footnote 87), and is extremely undesirable in programming. We introduce an extension to `ABLE`C to deal with null pointers (referred to herein as `ABLE`C-nonnull). This extension allows for compile-time checking for any possible null dereferences of pointers.

The fact that `ABLE`C-nonnull allows for compile-time checking of possible null pointer dereferences is incredibly valuable to a programmer and allows for the avoidance of many headaches typically associated with dealing with pointers in C.

1.2 Asynchronous I/O

Asynchronous I/O is a major part of the difficulty in creating a server implemenation in C, particularly when using threads is not a viable option. We don't want the server to be stuck while we wait for it to perform some I/O operation like reading or writing to an existing connection, when the server has other tasks it could be doing, like establishing new client connections or parsing an already-received message. In C, one of the primary ways of performing asynchronous I/O is using the `epoll` API. However, this API is not very user-friendly, particularly when compared to langauges like Go or Javascript that have very nice facilities for handling asynchronous operations (the `select` statement in Go or the `async` or `await` keywords in Javascript).

More information on the Asynchronous I/O extension will go here. Some ideas on what could be discussed: the lack of easy asynchronous tools in vanilla C. The prevalence of these tools in languages like Go and Javascrip

1.3 Wuffs

A final extension implemented for this project involves the Wuffs language. Wuffs is a language created by Google for memory-safe handling of untrusted file formats. Originally intended for use in security-specific portions of a program, Wuffs allows for compile-time checking of various security vulnerabilities, as well as some semi-rigorous proofs of program safety.

A simple parsing program in C is shown below. The same parsing program is also shown in Wuffs as a comparison of the two languages. Both of these programs take as input a string of numbers, for example, `"123"`, and return the associated numeric value, in this case, 123.

Leaving aside the syntactic peculiarities of Wuffs, note that the Wuffs code is signficantly more verbose than the C code in the above examples. However, we consider the edge cases of parsing some string, particularly with regards to integer overflow. In C, an integer overflow error can occur, but it does so silently. Obviously, in a real C parser implementation, we would implement security checks to cover these overflow cases. The point, however, is less about the fact that this is possible in C and more about the fact that these checks are mandatory in Wuffs. The Wuffs compiler will not allow an addition expression that could possibly overflow the maximum integer value.

This compile-time security checking for basic security vulnerabilites is incredibly useful, particularly in contexts where we are parsing unknown code or strings that could be malicious. We don't need to worry about runtime security checks when we can use Wuffs to guarantee no buffer overflows, out-of-bounds array accessing, integer overflow, or integer underflow at compile time.

Figure 1: Parsing in C

```
uint32_t parse(char * p, size_t n) {
    uint32_t ret = 0;
    for (size_t i = 0; (i < n) && p[i]; i++) {
        ret = 10 * ret + (p[i] - '0';
    }
    return ret;
}
```

Figure 2: Parsing in Wuffs

```
pub func parser.parse?(src: base.io_reader) {
    var c : base.u8
    while true {
        c = args.src.read_u8?()
        if c == 0 {
            return ok
        }
        if (c < 0x30) or (0x39 < c) { // '0' and '9' in ASCII
            return "#not a digit"
        }
        //Rebase from ASCII to numeric value
        c -= 0x30

        if this.val < 429_496729 {
            this.val = (10 * this.val) + (c as base.u32)
            continue
        } else if (this.val > 429_496729) or (c > 5) {
            return "#too large"
        }

        this.val = (10 * this.val) + (c as base.u32)
    } endwhile
}
```