Contents

1 Introduction

Extensible programming languages were first discussed in the 1960s. As originally discussed in Douglas McIlroy's 1960 paper, the original concept was to use a small number of macros in order to extend a compiler to accept very general extensions to the original language accepted by the compiler [**McIlroy**]. This has been modernized in various ways, including the idea of extensible syntax, compilers, and even extensible runtimes (CITE? wiki has dead link).

One example of an extensible programming language is `ABLEC` [**ableC**], an extensible version of the C programming language, built on top of the `Silver` [**silver**] attribute grammar system. Various extensions exist for `ABLEC` that were created to solve various problems.

This paper examines the uses of extensible programming languages in a real-world application: an Internet Relay Chat (IRC) server implementation. The IRC protocol was originally designed for text-based conferencing [**IRC**], and many server implementations are available. For this paper, we consider the specific implementation of `ngIRCd` (next-generation IRC daemon), written in the C programming language. There are several potential issues with this server implementation. First, any C program must include runtime checks to ensure that all pointers are not null. We introduce an `ABLEC` extension to allow for compile-time checking of possible null pointer dereferences. Another issue with the server is writing easily understandable I/O code in C. While the `epoll` API provided by the Linux kernel allows for some asynchronous I/O, we introduce another `ABLEC` extension to provide improved syntax for asynchronous I/O, akin to what Javascript provides. Finally, we introduce an extension to ensure that parsing code is secure. We do this utilizing the Wuffs programming language [**wuffs**], which allows for compile-time checking for various parsing attacks which must be manually accounted for at run-time in C.

2 Background

This chapter provides background information for the work that follows throughout the paper. We first discuss the Silver attribute grammar system [**silver**], which is used to implement the extensions to the host language. We follow this with a discussion of `AbleC` [**ableC**], the extensible version of C utilized in this work.

2.1 Silver

Silver [**1**], created by Van Wyk et al. is an attribute grammar specification system. Furthermore, Silver is extensible, allowing us to add both general features (pattern matching, for instance) and domain-specific features to Silver. This gives us an attribute grammar specification system with a rich set of language features we can utilize in developing new extensions.

Silver has several nice features useful in generating new language extensions. First and foremost, Silver allows for *forwarding* [**forwarding**] to implement new extensions in cost-effective ways. Forwarding allows language designers to utilize some form of inheritance within their language extensions, saving designers significant time and effort in creating new language features.

2.2 Extensible Programming and Able-C

One of the primary programming languages that is utilized in modern computing when speed or low-level control is vitally important is the C programming language. Unfortunately, C lacks many of the features of more modern programming languages, often making it cumbersome to work with in certain applications.

One way that some have tried to improve upon the C language is through the use of extensions. Writing extensions to the C language, however, can be quite difficult, often involving many complications. For instance, the Cilk extension was introduced to C to allow for easier parallel programming. However, the original implementation of Cilk5 utilized its own type-checker, despite not changing the underlying C type system (CITE Aaron's thesis, specifically the section at the bottom of page 14).

This difficulty in extending the C language was one of the motivations behind creating the `ABLE`C language. Built utilizing Silver, `ABLE`C is an extensible C pre-processor, conforming to the C11 standard [**ableC**]. It takes an "extended" version of C and translates it back into plain C, performing transformations and analyses as it does so.

3 Non-Null Pointer Extension

3.1 Non-null 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, and is undesirable when programming. In this section, we introduce an `ABLE`C extension 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. This static analysis of a program is valuable, saving the programmer both time and effort when compared with the original C code.

3.2 Comparison

Consider the code here from the same function, one rewritten using ABLEC-nonnull. Note the difference in the two examples below.

```
/* return false on failure (realloc failure, invalid src/dest array) */
bool
array_copyb(array * dest, const char *src, size_t len)
{
        assert(dest != NULL);
        assert(src != NULL );

        if (!src || !dest)
                return false;

        array_trunc(dest);
        return array_catb(dest, src, len);
}
```

Figure 1: Function before applying the non-null extension

```
/* return false on failure (realloc failure, invalid src/dest array) */
bool
array_copyb(array * nonnull dest, const char * nonnull src, size_t len)
{

        array_trunc(dest);
        return array_catb(dest, src, len);
}
```

Figure 2: Function after applying the non-null extension

The first example is a simple function to copy a string into an array as it is written in the original project. Note that if this function is passed a null pointer for either of the two pointer arguments, the `assert` statement will fail, causing a call to the `abort()` function, crashing our program.

Compare this to the function we get after applying the non-null extension. Here, we know that both pointer arguments are guaranteed to be non-null. Thus, we no longer need either of the assert statements, nor do we need the check in the `if` statement. This is a toy example implemented to illustrate the point that including this extension allows for shorter functions and fewer safety checks on the part of the end programmer.

4 Asynchronous I/O

4.1 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. Using this API, we are able to keep a list of file descriptors we want the current process to monitor, as well as a list of file descriptors that are ready for I/O. However, this process is extraordinarily tedious, requiring many expensive system calls to set up and maintain the `epoll` instance. When compared with other, more modern languages, the `epoll` API is both more verbose and more difficult to use. Consider a more modern language like Go or Javascript. Both of these languages have their own facilities for asynchronous operation. In Go, we use several constructs, including the `go` and `select` keywords, to implement various aspects of I/O. In Javascript, we utilize both the `async` and `await` keywords, as well as the idea of *promises* in order to achieve some measure of asynchronous operation. The goal of our extension is to include these same facilities in `ABLE`C, allowing for a programmer to more easily write and understand the code performing asynchronous I/O operations.

4.2 Asynchronous I/O Extension

In this section, we discuss the mechanics of the Asynchronous I/O extension, including the specifics of the translation from `ABLE`C code (`.xc` files) to plain C code.

4.3 Asynchronous I/O Implementation

5 Wuffs Parsing

5.1 Wuffs Overview

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.

5.2 Wuffs Extension

In this section, we discuss the mechanics of the Wuffs extension. For more details, examine the concrete syntax file for the `ABLEC`-wuffs extension. The extension is surprisingly simple, given the benefits it adds to a project.

The extension simply looks for a block of code delineated by the markers `WUFFS` and `WUFFS_END`. The code within these two markers is assumed to be valid Wuffs code. This code is loaded into its own `.wuffs` file, which is then compiled using the Wuffs compiler into a C file. In the original `ABLEC` file, we replace the original block of Wuffs code with boilerplate code that includes linking the new C file with the Wuffs functions, as well as calling those functions to parse the input, store the input, and validate the results. Note in the code snippet of the parse function using the extension that all of the Wuffs code is within the larger `parse()` function. This means that we need to do minimal editing of the rest of the file. We can retain the already written functions provided out-of-the-box by the server. In fact, in this case, the only function from the `parse.c` file that needs to be modified is the primary `parse()` function. The end programmer does not need to concern themselves with writing the C code to call any Wuffs functions – the extension is able to take care of inserting those function calls for us, allowing the programmer to simply write the parsing details in Wuffs.

```
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 3: Parsing in C

```
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
}
```

Figure 4: Parsing in Wuffs

5.3 Wuffs Implementation

Below, we have two code snippets of a parsing function. The first snippet is from the original server code. It contains a C implementation of the parser. The second code snippet is an edited Wuffs implementation of the parser, with some code removed for brevity.

We consider the differences between these two implementations. Obviously, there is a difference in syntax between the two languages. Beyond that, however, we consider the differences between the two implementations. First, we must primarily note that the C implementation has no guarantees of safety. If the program compiles, all we can be sure of is that there are no syntax errors in the file. We have no guarantees that, given some malicious input string, we won't be vulnerable to any attacks. In our Wuffs implementation, this code is sent off to the Wuffs compiler during compilation. The Wuffs compiler is able to tell us whether we are vulnerable to buffer overflows or other attacks. This is the largest benefit of using this extension. We get the speed of C code, but the safety and guarantees provided by the Wuffs compiler.

6 Discussion

Discussion of each of the extensions, pros and cons of usage

6.1 Non-null Pointers

Non-null discussion

6.2 Asynchronous I/O

Async IO discussion

6.3 Wuffs

Wuffs discussion

7 Future Work

Discussion of future work

```
GLOBAL bool Parse_Request( CONN_ID Idx, char *Request ){
        REQUEST req;
        char *start, *ptr;
        bool closed;

//Code omitted for brevity

        if (Request[0] == ':') {
                req.prefix = Request + 1;
                ptr = strchr( Request, ' ' );
                if( ! ptr ) {
                        return Conn_WriteStr(Idx, "ERROR :Prefix without command");
                }
                *ptr = '\0';
                start = ptr + 1;
        } else start = Request;
        ptr = strchr( start, ' ' );
        if( ptr ) {
                *ptr = '\0';
        }
        req.command = start;
        if( ptr ) {
                start = ptr + 1;
                while( start ) {
                        if( start[0] == ':' ) {
                                req.argv[req.argc] = start + 1;
                                ptr = NULL;
                        } else {
                                req.argv[req.argc] = start;
                                ptr = strchr( start, ' ' );
                                if( ptr ) {
                                        *ptr = '\0';
                                }
                        }
                        req.argc++;
                        if( start[0] == ':' ) break;
                        if( req.argc > 14 ) break;
                        if( ptr ) start = ptr + 1;
                        else start = NULL;
                }
        }
        if(!Validate_Prefix(Idx, &req, &closed))
                return !closed;
        if(!Validate_Command(Idx, &req, &closed))
                return !closed;
        if(!Validate_Args(Idx, &req, &closed))
                return !closed;

        return Handle_Request(Idx, &req);
} /* Parse_Request */
```

Figure 5: The parse function before adding the Wuffs extension

```
GLOBAL bool Parse_Request( CONN_ID Idx, char *Request ) {
  WUFFS
    pub struct parser?(
      prefix: slice base.u8, cmd : slice base.u8,
      args : array[15] slice base.u8, argc : base.u32[..=15]
    )
    pub func parser.parse?(src: base.io_reader) {
      var c : base.u8 = args.src.read_u8?() //Code removing whitespace omitted
      var i : base.u32[..15] = 0
      var extra : base.u32
      if c == ':' {
        extra = 0
        while true {
          c = args.src.read_u8?()
          if c == ' ' {
            args.src.limited_copy_u32_to_slice!(up_to: extra, s: this.prefix)
            break
          }
          else { extra += 1 }
        }endwhile
        c = args.src.read_u8?() //Prep for reading command
      } else { //Handle command
        extra = 1
        while true {
          c = args.src.read_u8?()
          if c == ' ' {
            args.src.limited_copy_u32_to_slice!(up_to: extra, s: this.cmd)
            break
          } else { extra += 1 }
        }endwhile
      }
      c = args.src.read_u8?()
      if c == 0{ return } //No more args, we are finished
      if c == ' '{ /* Code moving past whitespace omitted */ }
      else { //At least one arg
        extra = 1
        while.loop1 true {
          while.loop2 true {
            c = args.src.read_u8?()
            if c == 0 { break.loop1 } // c is null, we are finished parsing
            if c == ' ' {
              args.src.limited_copy_u32_to_slice!(up_to: extra, s: this.args[i])
              extra = 0
              i += 1
              if i >= 14{ break.loop1 }
              else{ break.loop2 }
            } else { extra += 1 }
          }endwhile.loop2
        }endwhile.loop1
        this.argc = i+1
      }
    } WUFFS_END
} // Parse_Request
```

Figure 6: The parse function after adding the Wuffs extension

References

[1] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, “Silver: An extensible attribute grammar system,” Electronic Notes in Theoretical Computer Science, vol. 203, no. 2, pp. 103–116, Jan. 2010.

[2] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski “Forwarding in attribute grammars for modular language design,” Lecture Notes in Computer Science, pp. 128–142, Apr. 2002.

[3] Google, “Google/WUFFS: Wrangling untrusted file formats safely,” GitHub. [Online]. Available: https://github.com/google/wuffs. [Accessed: 18-Nov-2022].

[4] M. D. McIlroy, “Macro instruction extensions of compiler languages,” Communications of the ACM, vol. 3, no. 4, pp. 214–220, 1960.

[5] Oikarinen, J. and D. Reed, ”Internet Relay Chat Protocol”, RFC 1459, DOI 10.17487/RFC1459, May 1993, ¡https://www.rfc-editor.org/info/rfc1459¿.

[6] T. Kaminski, L. Kramer, T. Carlson, and E. Van Wyk, “Reliable and automatic composition of language extensions to C: The ABLEC extensible language framework,” Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, pp. 1–29, 2017.