A Secure Parser Generation Framework for IoT Protocols on Microcontrollers

Sameed Ali
sameed.ali.gr@dartmouth.edu
Dartmouth College, Hanover

Sean Smith
sws@cs.dartmouth.edu
Dartmouth College, Hanover

Abstract

In the recent years, vulnerabilities found in the packet parsers of Bluetooth Low Energy (BLE) protocol have called for a need to have secure lightweight protocol packet parsers for microcontrollers. Since these packet protocol grammars consist of packets of limited size it is possible to parse them efficiently via Finite State Machines (FSM). However, parsing via FSMs would require developers to either express the grammars via regular expressions or constructed hand-coded parsers. Unfortunately, hand-coding parsers is error-prone; furthermore, due in part to certain constructs found in such grammars which are not commonly found in text-based regular grammars. In addition, expressing binary grammar constructs in regular expression is not only challenging and error-prone but the resulting expressions are often complex and unreadable. Thus the lack of an alternative language for describing these constructs is a hindrance to the use of finite state machines to generate parsers which are safe, secure and computationally bounded. This paper presents a novel secure parser generation framework which consists of an easy-to-use parser description language called "Microparse" and a toolkit that utilizes finite state machines to generate lightweight parsers for microcontrollers. To demonstrate the viability of this approach, we have applied our framework to generate parsers for the BLE protocol running on an Ubertooth One Microcontroller. We demonstrate that the generated FSMs are lightweight enough to be run on devices with very limited resources, and are easier to use for developers; we offer this method as a potential solution for the various bugs found in the implementation of the BLE firmware in the recent years.

1 Introduction

In the recent years, numerous vulnerabilities have been discovered in the packet parsers of the BLE microcontrollers [13]. These parts of the parsers that were vulnerable were responsible for parsing the Link Layer of the BLE packets. These vulnerabilities are significant not only because of the ubiquity of the BLE protocol [12] but also because of their severity. The revealed vulnerabilities allow a remote attacker in the wireless range of a BLE device to achieve Denial of Service (DoS) and potentially Remote Code Execution (RCE) on the target device.

It is important to note that the discovered vulnerabilities were not due to a flaw in the BLE protocol; rather, it was the *buggy implementation* of the protocol that made them possible.

These discoveries not only show that many Bluetooth Low Energy (BLE) devices have vulnerabilities in the lower layers of their BLE stack but also demonstrate a need for a simple and secure parsing mechanism for packet parsers operating in low-resource environments. These parsers need to be lightweight, requiring minimal memory and computing resources; they need to be easy for developers to construct; and they need to provide secure parsing for packet protocols.

We posit that finite state machines can be used to construct secure computationally-bounded parsers but remain underutilized - despite the availability of lightweight compile time regular expression libraries for microcontrollers - because writing regular expressions to parse binary grammars is not only challenging and error-prone but results in expressions which are complex and unreadable.

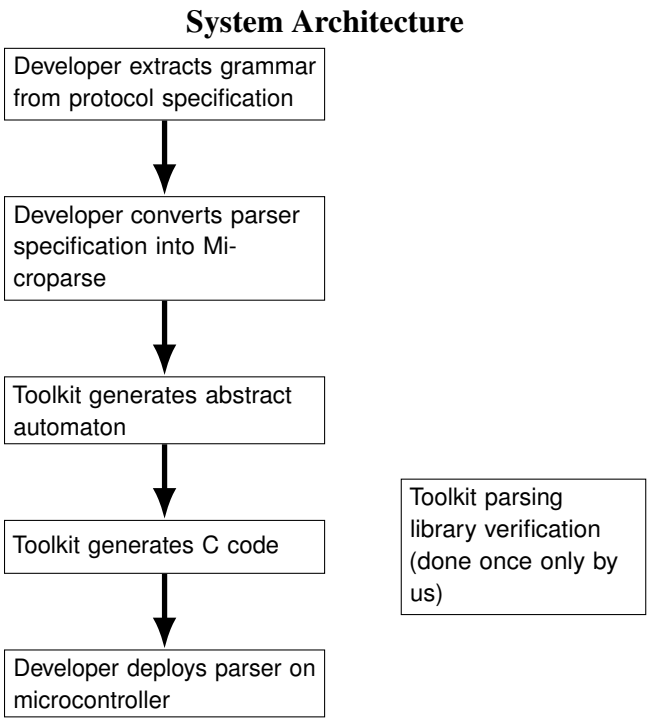
To that end, this paper makes the following **contributions**:

- We provide a method to develop secure parsers for resource-constrained environments and thus prevent *implementation* vulnerabilities in packet parsers found in microcontrollers.
- We develop *Microparse*, an alternative *parser description language (PDL)* for developers to easily represent the finite state machines that can parse grammars commonly found in network protocol packets.
- We provide a compiler from this parser description language to C code for packet validation.
- We verify the memory safety and termination of the functions in the provided parser combinator library used

by the generated C-code.

- We use these tools to construct an implementation of a BLE link layer packet parser for microcontrollers in C without using any external C libraries (thus making it feasible to run inside such resource-constrained systems.

Figure 1: Architecture diagram of the system. The Microparse compiler takes the grammar description then generates C code.



It is pertinent to mention that although this paper demonstrates its techniques on the BLE LL packet format, the general technique is applicable to *any* wireless packet format. In the future, our hope is to apply it to other IoT protocols.

The intuition behind having a developer write a description in an easy-to-use parser description language and then use a tool that generates the parser from that description is that bugs are inevitable when coding in low level-languages which do not provide memory safety like C. In addition, expressing binary grammars in regular expression - although theoretically possible - is very hard, unreadable and error-pone.

Moreover, if the parser's input language (the packet format) is overly complex, writing a parser according to the programmer's expectation can be undecidable in the general case - i.e no general algorithm may exist that can verify the input validity correctly and no amount of patching can fix it. This is because if a complex packet format is modeled as a formal language it may fall into an language category that is not Turing decidable. This is an insight we gather from the prior work done by the LangSec community.

Although it seems unlikely that may be the case for simple protocol formats, automata theory reveals that even a finite automaton augmented with two counters is equivalent to a Turing machine. Thus it is important to model these languages as formal grammars to really see how expressive they are and constrain their expressivity to prevent exploits.

Furthermore, modeling parsers using a language-theoretic approach allows us to define a parser description language which provides us the ability to prove parser equivalence.

We posit that an easy-to-use parser description language (PDL) should be used for generating parsers as opposed to hand-writing parsers. The PDL will be a language with limited expressivity by design, and this limited expressivity will reduce the bugs in parser that are often found in low level code.

This idea of using domain specific languages is not new and is already common place in other areas of computer science, such as SQL [11] for databases and the P4 language [9] for defining software defined network's control plane.

This concept has also been applied for parser generation before to create parser description languages such as Kaitai struct [2] and DFDL [1].

Prior work has constructed verified parsers for various grammar formalisms [8] [19] [18]. Some [23] have also constructed verified parsers expressed in a PDL which guarantees that the generated parser does not have memory-corruption vulnerabilities and will eventually terminate.

However, prior research does not aim towards deploying their code on resource constraint devices like microcontrollers. Our work aims to address this limitation and provide secure parsers for ready for deployment on such devices. The parsers generated by our approach are able to be deployed on devices with as little as 16K bytes of RAM. Our parser implementations are verified to ensure parser termination and that the parsing function is memory corruption free. In addition, our parsers do not function only as validators i.e. they extract the data from the payload by storing them in named registers thus eliminating the need for writing hand writing parsers.

Moreover, researchers [20] have also looked at developing new language classes by augmenting regular languages with additional features to allow them to handle binary grammatical constructs such as the length field.

Our approach differs from theirs in that we assume the existence of a maximum finite length value for the length construct. This allows us to use finite state machines to parse these binary formats. We argue that, in practice, this assump-

tion is realistic as most binary formats have a limit on the maximum size a packet can have and are thus expressable via a finite state machine, albeit a very large one.

We argue that to prevent parser vulnerabilities, a PDL with known formal expressivity has to be designed and implemented. To that end, we have identified a formal automaton (Finite State Automata) which is has known decidable properties and have compiled the PDL to it. This limits the computational power of the parsers that can be generated from the PDL to the computational power of the automaton and thus ensures that a only parsers with limited computational complexity can be defined.

Generating parsers based on finite state machines raises the obvious question: why not use regular expressions directly to describe these parsers, rather than an alternative PDL? One major limitation of regular expression is that the design of regular expressions makes it cumbersome and unintuitive to describe common constructs found in binary grammars. In contrast, the PDL we propose ("Microparse") is designed specifically for generating parser for binary formats, and avoids these limitations. Further, along with Microparse, we also provide a compiler which generates parsers in C code from their description in Microparse.

Currently, crafting a finite state machine parser for machine binary languages requires coding one up by hand which gives rise to various bugs and vulnerabilities. By providing a framework to generate an FSM implementation from Microparse we aim to stymie the rise of such vulnerabilities in the wild.

2 Microparse Description Language

Microparse is a domain specific language written in Clojure [15]. It is a high-level description of the parser - hence, it focuses on providing a simple, readable and easy-to-understand description of the grammar at hand, rather than on the implementation details for parsing data.

Real world binary protocols commonly use grammatical patterns that cannot be easily expressed in regular expressions. We have constructed Microparse so that it permits developers to intuitively describe these patterns while also remaining computationally bounded.

2.1 Structures found in IoT binary protocols

Before we can understand the design of Microparse, it is important to know the kinds of grammar structures commonly found in packets of IoT protocols.

The commonly found structures are as follows:

- SEQ: SEQ is the simplest construct. It represents a sequence of fields that come after one another. One example would be finite size bit field, coming one after

Figure 2: An illustration of the TAG construct. It allows only A, B and C as valid values for the "Packet Type" field. An example of what we call the TAG construct is the "PDU" field in the BLE LL packet format shown in Figure 5.

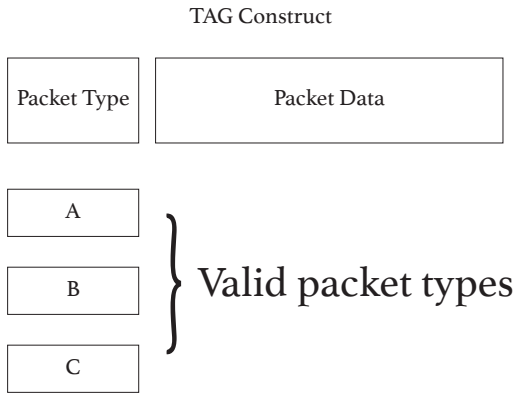


Figure 3: An illustration of what we define as the CASE construct. This illustration shows that there is a different payload grammar for each unique packet type. In the BLE LL packet format the payload grammar is determined by the value of the "PDU" field in the header. The CASE construct captures this relationship.

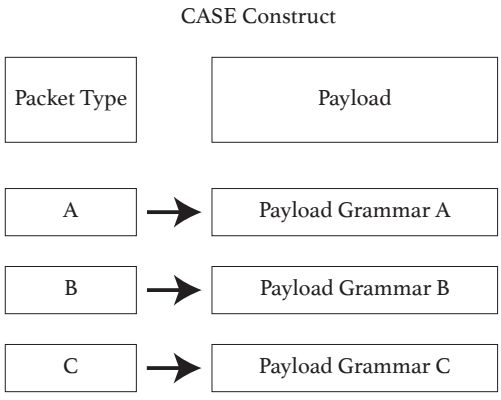
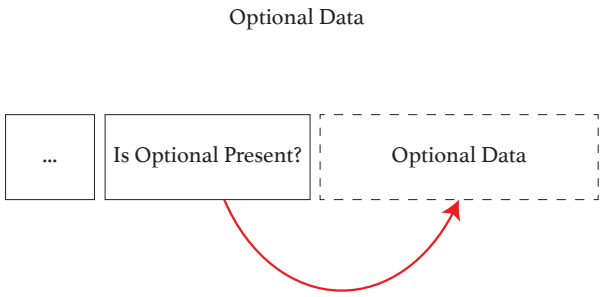


Figure 4: An illustration of an optional field. The red arrow shows the dependency relationship between the two fields. The optional data will be present only if the "is optional present?" field has a non-zero value.



another. For example: A 16 bit integer value followed by another 16 bit integer value.

- TAG: The TAG represents a field (a set of bit/bytes in the input) which is allowed to have a known set of values. (See Figure 2.) For example: a packet type field can only have a known set of values which represent the type of packets that are valid.
- LEN: The LEN construct consists of a number N, followed by N bytes of data. The LEN Field is the bits/bytes representing the number N and the LEN Data is the data which is of the size N.
- REPEAT: The REPEAT construct consists of a repeating set of bits/bytes.
- CASE: The CASE construct captures the *dependencies* within a grammar such as partial grammars which are conditioned on the parsed result of another field. (See Figure 3). Different types of payloads have different payload grammars. To parse a payload one needs to know a payload type which is contained in the header of the packet. In such cases, the grammar of the payload packet is dependent on the type of the payload. CASE construct represents such dependencies present within grammars. Additionally, the CASE construct can also represent optional fields found in some grammars. For example, the Figure 4 shows an optional data field which is present only if another field contains a certain value.

As mentioned earlier, these constructs show the various structures found in IoT protocol grammars. Microparse allows a way to represent grammars as a combination of these constructs allowing the parser representation to be readable and easy-to-understand in sharp contrast to regular expressions.

Microparse consists of functions which represent these constructs. Each of these functions in Microparse thus is responsible for describing the bits and bytes that are represented by the corresponding construct.

Parser composition Microparse is designed so that simple parsers can be constructed by generating combinations of the provided functions of SEQ, TAG, LEN, REPEAT, and CASE constructs. This allows a developer to combine the individual parser functions together to construct a parser for the complete packet. This design choice has been inspired by the parser combinator paradigm [16].

2.2 Language complexity of Microparse

Microparse is no more expressive than a finite state machine. The Microparse-to-C compiler converts the input to a finite state machine which can be visualized by our toolkit (if needed) and then used to generate the C code.

Expressing the same finite state machines with a regular expression results in expressions which are unreadable. Further, it is inconvenient to express certain grammatical constructs like the length construct as a regular expression. Microparse allows developers to describe such finite state machines with ease while still remaining readable. Although, a length construct with an unbounded payload size requires automata more powerful than a finite state machine to parse, in practice, the maximum size of the payload is bounded. This bound allows our framework to parse it using a finite-state machine by constraining the maximum size of the payload.

2.3 Implementation of the Microparse Language

As mentioned earlier, Microparse is modeled after the aforementioned constructs. Microparse implements parser description functions for each of the aforementioned language constructs. This subsection will provide the details of those functions and how they are used to construct parsers.

SEQ construct The sequence construct is defined by the *gen-seq* function in Microparse. This function takes a list of constructs as an input and sequences the constructs together to make a new parser which is a combination of these individual parsers. An example of the use of this construct can be seen in the definition of packet-grammar in Figure 6. After the parsers for the individual constructs inside a packet are defined, they can be sequenced together in order to generate a single parser of the whole packet.

TAG construct The TAG construct in Microparse is defined by *gen-tag* function. This function takes as input the dataunit, a count, the map of valid values and a string description of the construct. The dataunit informs the parser if input is to be read in bits or bytes. The count tells how many of bits/bytes to read. The map is given as input to inform the parser about the possible valid values and their names. The description is a textual description of the parser for debugging purposes.

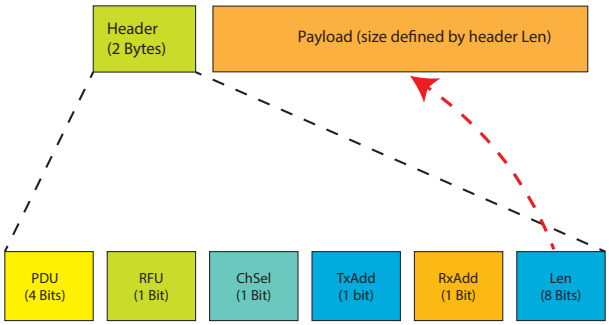
Figure 6 shows the Microparse parser definition of the PDU tag located inside the header of the BLE Linked Layer packet. The PDU defines the packet type and is a 4 bit wide field. The valid values it can take are shown in the dictionary passed as argument to the function. The keys in the dictionary name the various kinds of valid packets types and the values of the dictionary are the binary value the packet type field would take if the packet is of that type. These values are defined by the protocol specification.

LEN construct The LEN construct in Microparse is defined by *gen-len* function. The length function takes as input the dataunit (bit or byte), the endianness (LSB or MSB), the number of bytes and a string consisting of the description of the parser. The function is called *gen-len* and an example of the length construct is shown in Figure 6. In this example length construct is defined by calling the *gen-len* function. This function takes as argument the dataunit (byte), the size of the field (2 bytes), the endianness (LSB) and a description as a string. The length construct will ensure that the parsers which follow it will only be allowed to consume the number of bytes defined by the length field's input, and that the parsing would halt afterwards.

REPEAT construct The REPEAT construct is represented by the *gen-tag* function in Microparse. The repeat construct takes as input a dataunit (bit/byte), a count as a second argument which defines how many bytes/bits to repeat. It can take a special argument called "##Inf" if the repeats are to have no limits on them. It can also take in two optional arguments called min-repeat and max-repeat which define limits on the minimum and maximum times a bit/byte can repeat itself. If these optional arguments are given they will take precedence over the non-optional count argument. An example of the use of this function in Microparse is shown in Figure 6.

CASE construct In Microparse, the CASE construct is represented by the *gen-case* function. The *gen-case* function takes as input a tag construct and a dictionary. The dictionary is a key-value pair of tag values and the subsequent parsing function to run if that tag value is the parse result. An example of this construct is shown in Figure 6 in the definition of the payload-grammar. The PDU is a tag which goes as argument to this function. The possible parsing result of this tag function

Figure 5: Structure of a BLE link layer packet. The red arrow shows the length of the payload is determined by the value contained in the Len field.



go in as the keys of the second dictionary argument to the *gen-case* function and the corresponding values are definitions of other parsers, whose definition is omitted for brevity, which will be run if that tag value is the parse result.

2.4 Example: BLE LL packet format description in Microparse

As an example, we show how the design of Microparse captures all valid ways in which a BLE LL packet can be constructed. To do so, we shall first provide a description of the BLE LL packet. The BLE LL packet format is shown in the Figure 5.

The BLE LL packet consists of a two byte header followed by a variable sized payload.

The header consists of:

- A 4 bit PDU type field which describes the type of packet.
- A 1 bit RFU field which is reserved for future use.
- a 1 bit ChSel field which signals if the device supports BLE 5.0 channel select algorithm 2.
- A 1 bit TxAddr field which informs if Tx addr is randomized.
- A 1 bit RxAddr field which informs if Rx addr is randomized.
- An 8 bit length field which defines in bytes the size of the payload.

The payload grammar differs based on the type of the packet. The inner sections of the payload have similar grammatical constructs and their details have been omitted for brevity. Figure 6 show the parser description of this packet format. The *packet-grammar* definition consists of a sequence of

Figure 6: An example of Microparse code, this figure shows how the gen-tag, gen-len and gen-len functions can be used to describe a parser for the BLE LL packet. (Note that many definitions have been omitted for brevity.)

```
;; PDU defined as a TAG construct
(def PDU
  (gen-tag :bit 4
    {:ADV_IND         "0000"
     :ADV_DIRECT_IND  "0001"
     :ADV_NONCONN_IND "0010"
     :SCAN_REQ        "0011"
     :SCAN_RSP        "0100"
     :CONNECT_IND     "0101"
     :ADV_SCAN_IND    "0110"
     :ADV_EXT_IND     "0111"
     :AUX_CONNECT_RSP "1000"}
    "PDU"))

(def repeat-bytes
 (gen-repeat :byte ##Inf))

;; definitions of ADV_* SCAN_*, AUX_*
;; not shown for brevity.
(def payload-grammar
  (gen-case PDU
   {:ADV_IDV ADV_IND
    :ADV_DIRECT_IND  ADV_DIRECT_IND
    :ADV_NONCONN_IND ADV_NONCONN_IND
    :SCAN_REQ        SCAN_REQ
    :SCAN_RSP        SCAN_RSP
    :CONNECT_IND     CONNECT_IND
    :ADV_SCAN_IND    ADV_SCAN_IND
    :ADV_EXT_IND     ADV_EXT_IND
    :AUX_CONNECT_RSP AUX_CONNECT_RSP}))

(def len-construct
  (gen-len :byte :LSB 2 "header"))

;; some definitions not shown for brevity.
(def packet-grammar
 (gen-seq
  [PDU RFU ChSel TxAdd
   RxAdd len-field payload-grammar]))
```

parser definitions (PDU, RFU, ChSel etc.), and these individual parser definitions correspond to parsers for the fields shown in the BLE LL packet diagram.

3 Methodology

3.1 System Architecture

The architecture of the overall system is shown in Figure 1. The developer will read the specification and write a parser in Microparse. The compiler will then produce the abstract automaton and use this to generate C code. The generated C-code calls certain predefined functions which simulate parts of of the FSM. The C code consists of these functions along with addition code which invokes these functions to simulate the FSM.

The predefined functions are part of the provided toolkit. We used the Frama-C [10] static-analysis framework to verify these functions to ensure that they terminate, that they only access memory that they require, and that they are free of memory corruption bugs.

The generated code then invokes these functions to simulate the automaton and complete the parsing process.

These predefined functions are modeled after the functions in Microparse. The *gen-tag* corresponds to the *tag_cons* function in the generated C code which is responsible for parsing a single TAG construct. The return value of the *tag_cons* function is an integer which contains the tag value read from the input.

The *gen-len* corresponds to the *len_cons* function in the generated C code which is responsible for parsing a LEN construct. It does so by ensuring the number of bytes in the len field is exactly the same as the remaining bytes in the input. In case, there is a mismatch in the remaining bytes and the value read it halts the parsing. This ensures that there are a finite number of transitions that the parsing FSM can take after running this function and that this number is equal to the value read from the input.

The *len_cons* function returns the value of the LEN field it read from the input as in integer.

The *repeat_cons* is similarly modeled after *gen-repeat* function in Microparse. It returns a tuple of the type *parsed_result* which contains information about the start and end position of the repeating bytes on the input.

The *gen-case* of Microparse does not correspond to any function in the C code. Instead this Microparse function results in a switch statement in the C code. This switch statement is conditioned upon the result of the tag parser on which the gen case depends and inside the individual case statements it

outputs the C code that is to be run if that particular tag value was the result of the parse the CASE is predicated upon.

An example of the C code generated by Microparse is shown in the Figure 8.

3.2 Compiler construction

The Microparse to C compiler is written in Clojure [15] which is a dialect of lisp. The compiler also has the ability to visualize the finite state machine of the automaton as well so the developer may use this for debugging. The compiler generates C code which simulates a finite state automaton. As mentioned earlier, the compiler dynamically generates the C code and also outputs the pre-defined verified automaton simulation functions so that the generated code is able to simulate the automaton.

The compiler creates the automaton which reads the input from left-to-right and parses the input. It is designed so that it consumes a bit or byte of input for every transition of the automaton and thus it finishes parsing in $O(n)$ time where n is the size of the input.

The compiler generates the finite-state machine as C code with invocations to predefined functions. This code is optimized to ensure that it can simulate an FSM with a very large number of states.

4 Verification of the C code

In order to ensure that the predefined functions written in C do not have vulnerabilities in them, we have verified them using Frama-C [10]. Frama-C is a static analysis engine for C code.

The verified C-code functions takes as input the array of input bytes of type `uint8_t`, the length of the input buffer (in bytes) and the arguments of the grammar construct to be parsed.

Frama-C allows the user to define pre and post conditions on a function and also annotate the function to ensure the functions are memory safe and they can only access the parts of memory they are allowed to access. Annotations also ensure that the function eventually terminates. The annotations are added at the top of the function definition for each of the function verified. The annotations apart from checking for termination, check that the size of the input buffer is not zero, and each element in the array is a valid memory location. Further, it ensures the function is not allowed to write to any non-local memory.

We added these annotations to the functions in C code to ensure that it is safe and free from memory corruption vulnerabilities and that it terminates.

5 Optimizing the compiler

Initial Attempt In our initial attempt, we generated an encoding of the FSM from Microparse and asked the C code to simulate a very large FSM. The FSM was encoded as a struct and was defined as a global constant. This struct consisted of an an array of tuples (consisting of a start node and end node) representing the transitions of the FSM graph.

Depending on the size of the graph, the length of the transition array can be very long. This can have a significant effect on the size of the binary.

These results showed that this approach is not feasible for small microcontrollers as the binary produced was too large for the Ubertooth One which has a microcontroller with 16K of RAM.

Improvement To optimize the size of the resulting binary, we rewrote the simulation function to multiple smaller functions each of which simulates a single construct found inside the grammar. And then instead of generating an encoding of the entire FSM and passing it as an argument to a simulation function, we redesigned the compiler to output C code which is designed to call individual parsing functions one after another to carry out the parse. It is importance to note that the resulting functionality of these smaller functions is still equivalent to a large FSM but these smaller functions are much more efficient to execute for a resource constrained device like a microcontroller. Furthermore, we also utilized the use of write-once-only registers to reduce the number of states of an FSM drastically.

The difference of this approach to the old one is that this approach eliminates the use of transition table required by the first approach. The code generated in C is a series of function calls to a hardened library and thus very readable and easier to debug. The code is simple and easy to understand as well as it only has function calls to the parsing functions and simple if and case statements only.

6 Evaluation

6.1 Deployment on microcontroller

For evaluation, we deployed the generated code on the Ubertooth One device [4]. This device runs on a ARM cortex M-3 microcontroller with an open source firmware. The maximum RAM size is 16K bytes. The device comes with an open source BLE scanning application which can be installed on the microcontroller. It is present in the `bluetooth_lxtx` folder of the Ubertooth One's repository [14]. This application also has a corresponding host application which can parse and display packets received from the Ubertooth One device.

Figure 7: An example FSM visualization generated by the compiler: this shows the resulting FSM when three TAG constructs (PDU and RFU and ChSel) are concatenated by the SEQ construct. The green arrows signify that a '1' input bit is read, and red arrows signify that a '0' input bit is read. States can be reduced if the value of constructs is stored in write-once-only registers. In this example, since the PDU value is stored in a write-once-only register, multiple edges from the leaf nodes in the PDU FSM subgraph (S17-S23) can go to the same subsequent nodes (S32/S33). All green edges from PDU leaf go to RFU-ON (S33) and all red ones go to RFU-OFF(S32). Without the use of fixed registers, for each leaf node, there will be a separate FSM sub-graph which would parse the remaining fields.

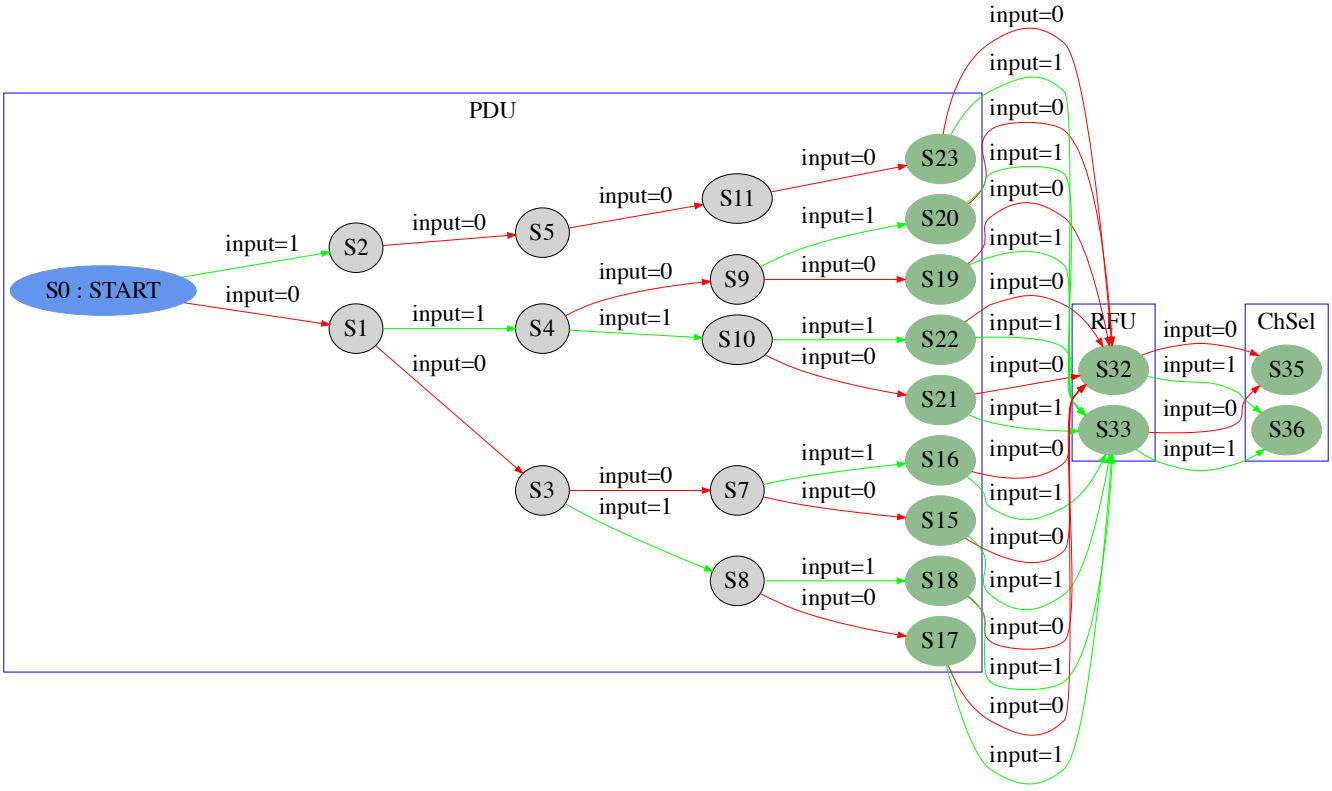


Figure 8: The optimized result of the Microparse compiler. The code shown below is autogenerated from the Microparse description shown in Figure 6. The functions *tag_cons*, *len_cons*, *repeat_cons* are parsing function which parse the TAG, LEN, REPEAT construct respectively. The result of the TAG and LEN function is the value of the tag or the length. In case of repeat, it returns a tuple of type *parsed_result* which has the start and end position of the bytes successfully parsed by the construct. If any of the constructs fails to parse and the parser halts.

```
int run_parser() {
  // T is input tape
  // T_LEN is tape length

  // PDU
  int reg4144=tag_cons(T, T_LEN, BIT, 4);
  // RFU
  int reg4145=tag_cons(T, T_LEN, BIT, 1);
  // ChSel
  int reg4146=tag_cons(T, T_LEN, BIT, 1);
  // TxAdd
  int reg4147=tag_cons(T, T_LEN, BIT, 1);
  // RxAdd
  int reg4148=tag_cons(T, T_LEN, BIT, 1);
  // len cons
  int reg4149=len_cons(T, T_LEN, BYTE, LSB, 2);

  switch (reg4144) {
    case 0b0001: {
        // :ADV_DIRECT_IND
        // AdvA
        // parsed result is a tuple of
        //start and end position
        parsed_result reg4150;
        reg4150=repeat_cons(T, T_LEN, BYTE, 6);
        // Target A
        parsed_result reg4151;
        reg4151=repeat_cons(T, T_LEN, BYTE, 6);
        break;
    }
    case 0b0011: {
        // :SCAN_REQ
        // ScanA
        parsed_result reg4152;
        reg4152=repeat_cons(T, T_LEN, BYTE, 6);
        // AdvA
        parsed_result reg4153;
        reg4153=repeat_cons(T, T_LEN, BYTE, 6);
        break;
    }
    ...
    default:
    debug_printf("ERROR:No case match!");
    return 1;
  }
}
```

We used this host application to verify the accuracy of the packets validated by the injected parser.

We modified the firmware image of the BLE application so that it verifies each packet **before** it gets processed by the rest of the application code. The firmware receives bits from the radio as an array of `uint8_t` bytes. The verified parser takes the array, the integer length of this array, and the generated parser code. The packet is processed if the validator accepts the packet thereby protecting any vulnerable code from attacks.

We tested it out by carrying out malformed packet attacks on the Ubertooth One device deployed with our hardened parsers. These attacks were launched from a NRF52840 dongle [3] which intentionally sent malformed Link Layer(LL) packets to the Ubertooth One device. We chose to send malformed LL packets because this is the layer in which the parsing vulnerabilities were discovered.

The malformed packets contained invalid values for tags and extraordinarily large length field values. It also sent packets which were incomplete packets.

We also tested for packets of different types (such as sending a `SCAN_REQ` packet when the grammar only accepted `ADV_IND` packets) and observed that only the packets which match the grammar description were allowed through by the hardened parser as valid. Further, the Ubertooth One device was able to discard all the malformed packets.

Experimental Results We measured the size of the firmware after injecting the verified parser in the firmware and of the unchanged firmware image file and experimental results show that the injection of the packet validator in the BLE firmware increased the firmware size by 5%.

The hardened parsers were built for ensuring the valid structure of the BLE LL packet. The hardened parsers deployed on the Ubertooth One were able to successfully able to filter out malformed packets sent by the NRF52840 device.

6.2 Comparison with regular expressions

As we have observed, it is theoretically possible to describe BLE LL packets with regular expressions. However, we posit such a description is difficult to understand and complex in comparison to Microparse.

This subsection will evaluate the difference in their approaches by demonstrating an example of the two ways of describing the same FSM.

Consider the case where the developer wants to describe a length construct which has a length field with a finite size. The length construct places "depth constraints" on branches of a FSM where the depth is defined by the input. Since

the "constraints" can only take a finite number of values, it is possible to parse such a grammar via regular grammars. However, describing a readable regular expression for such a grammar is not trivial.

One regular expression could be: `1 E1|2 E2` where `E1` is the regular expression if length constraint is one byte and `E2` is the expression if the length constraint is two bytes and so on. Such a regular expression is clumsy and unreadable. For an n-bit length field, the resulting description requires 2^n repetitions of a very similar expression. In contrast, Microparse provides a simpler way to describe such constructs as shown in Figure 6.

Consider another case where a developer wants to describe a TAG and CASE pattern as is the case in the BLE LL packet where the values of the PDU (the TAG pattern) will later determine the format of the payload (the CASE pattern).

Writing a regular expression of such a construct is unintuitive but one attempt might be to first check the PDU field only allows the predefined values. This can be done by using the choice operator among the valid values of the PDU field. Such an expression would look as follows: `(0000| 0001| ... 1000)` where the bit-strings are the valid values for the PDU field. In contrast, Microparse allows the developer to define the same grammar as shown in Figure 6. The description has human-readable description of the valid bit-strings which can later be used when defining the case construct. To condition the payload format on the value read in PDU, can also be challenging when constructing regular expressions. One way to do so might utilize the ^ operator which will see if the packet starts with the relevant bit-string and depending on the result run the regular expression defining the payload format. For example, the expression might look like: `(^0000) E1 | (^0001) E2 | ...` where `E1` and `E2` are regular expressions for the payload format when the PDU value is `0000` or `0001` correspondingly.

In contrast the Microparse code is more readable and easier to construct. As shown in 6 the language allows the developer to use the human readable name of the bit-strings to define the case pattern.

As we can see from these examples the regular expressions are unreadable, unintuitive, cumbersome to write and may end up extremely long. Coming up with a regular expression to describe the LEN construct was not intuitive and we contend that alternative regular expressions for expressing the same grammar will share similar issues because regular expressions are catered towards expressing patterns found in human-readable text and not binary grammars.

Consequently, it is difficult to capture the structures present in FSM of binary grammars in regular expressions in a readable and concise manner. For example, the conditional relationship between the PDU field and the payload format found in the specification of the protocol cannot be easily expressed in regular expressions. In contrast, the Microparse description of the same packet format is easy to read and understand.

7 Related Work

Prior to our work, researchers have proposed various hardened parser construction toolkits. The Hammar project built a hardened parser combinator toolkit for binary files and binary protocols. It provided a way for developers to describe grammars via a parser combinator like syntax [22].

Later, Nail [5] was proposed which improved upon Hammar by adding the ability to handle dependent fields and stream transformations. In addition, it improved on Hammer's limited ability to parse the length constructs.

Various verified implementations of parsers of known formal languages classes have also been proposed in the past. For instance, Barthwal et al. [6] built a parser generator for SLR grammars and verified the generated parser's soundness, completeness, and non-ambiguity using HOL4 proof assistant [25]. Koprowski et al [17] constructed a verified parser interpreter in the Coq [7] proof assistant for Parsing Expression Grammars (PEGs). And Blaudeau [8] et al proposed a packrat parser interpreter for PEGs verified via the PVS verification system [21].

Lasser et al [18] built a verified LL(1) parser generator. They also used the Coq [7] proof assistant to verify that the generator and produced parsers are sound, complete and terminate on all possible inputs.

More recently, Everparse [24] proposed verified parsers for authenticated message formats. Their parsers were verified in the F* programming language [26] to be safe correct and non-malleable.

However, none of the aforementioned parser construction toolkits are designed for resource constrained devices. Our work aims to address this shortcoming in the literature. Microparse is designed specifically for resource constrained devices and aims to provide a way to generate secure parsers for them.

8 Conclusion

We have presented Microparse, a parser description language, for describing IoT binary protocol grammars. In addition, we have presented a compiler which can generate parsers described in Microparse to parsers in C code which are safe, readable and capable of running on microcontrollers with limited resources. Our approach avoids the pitfalls of handwriting parsers and the problems associated with constructing parsers using regular expressions, while at the same time

ensuring the ease of developing parsers in an expression language and preserving the efficiency of parsers written in a low level language like C. Our experimental results show that this approach can be successfully used on microcontrollers in practice to secure them against attacks.

In the future we plan to model more IoT protocols and model more layers of the BLE protocol in Microparse. We also plan to look at the applicability of our parsers in Bluetooth and WiFi microcontrollers found in otherwise resourceful devices such as smartphones and laptops.

Acknowledgments

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0075 and Contract No. HR001119C0121. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA)

Availability

The source code of this project is available at https://github.com/microparse/microparse.git for the Microparse to C compiler code. The Ubertooth One micro controller code is available at https://github.com/microparse/ubertooth.git.

References

[1] Apache Daffodil | Home. https://daffodil.apache.org/.

[2] Kaitai Struct: declarative binary format parsing language. https://kaitai.io/.

[3] Nordic semiconductor nrf52840 dongle. https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840.

[4] Ubertooth one. https://www.greatscottgadgets.com/ubertoothone/.

[5] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 615–628, 2014.

[6] Aditi Barthwal and Michael Norrish. Verified, executable parsing. In *European Symposium on Programming*, pages 160–174. Springer, 2009.

[7] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[8] Clement Blaudeau and Natarajan Shankar. A verified packrat parser interpreter for parsing expression grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 3–17, 2020.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[10] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, page 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Chris J Date. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[12] Tony Gaitatzis, Andrew Ward, and Linda Manning. *Bluetooth Low Energy: A Technical Primer Your Guide to the Magic Behind the Internet of Things*. ISBN Canada, Ottawa, Ontario, CAN, 2017.

[13] Matheus E Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. SweynTooth: Unleashing Mayhem over Bluetooth Low Energy. page 16.

[14] Greatscottgadgets. Greatscottgadgets/ubertooth: Software, firmware and hardware designs for ubertooth. https://github.com/greatscottgadgets/ubertooth.

[15] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, New York, NY, USA, 2008. Association for Computing Machinery.

[16] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[17] Adam Koprowski and Henri Binsztok. Trx: A formally verified parser interpreter. In *European Symposium on Programming*, pages 345–365. Springer, 2010.

[18] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. A verified ll (1) parser generator. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[19] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. *CoStar: A Verified ALL(*) Parser*, page 420–434. Association for Computing Machinery, New York, NY, USA, 2021.

[20] S. Lucks, N. M. Grosch, and J. König. Taming the length field in binary data: Calc-regular languages. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 66–79, 2017.

[21] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[22] M. L. Patterson. Hammer: Parser combinators in c. https://github.com/UpstandingHackers/hammer.

[23] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. page 19.

[24] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. {EverParse}: Verified secure {Zero-Copy} parsers for authenticated message formats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, 2019.

[25] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[26] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.