WORK IN PROGRESS: A VERIFIED PARSER GENERATOR FOR MICROCONTROLLER APPLICATIONS

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

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

Abstract

This paper presents a parser generation toolkit which generates verified parsers from a user supplied parser description. This parser description is described using a novel parser description language (PDL). The parsers generated by this parser description language are not only verified (to ensure parser termination and to prevent memory corruption) but also strictly limited in expressivity as the PDL is backed with a formal language model (a finite state machine).

The limited expressivity will ensure the generated parsers have mathematical bounds on their complexity, allowing us to reason about properties such as decideability and equivalence.

These parsers are designed to handle large scale finite-state machines and are also space-efficient enough that they can be deployed on microcontrollers with limited computing resources. Further, the PDL is designed to be concise and descriptive enough to describe frequently found grammatical constructs found in packets of network protocols.

We evaluated the parsers generated for the Bluetooth Low Energy (BLE) LL (link layer) packets by deploying them on the firmware of the Ubertooth One device and attacking them by hand-crafted malformed BLE LL packets. Our initial results show the hardened parsers are effective against the malformed packets. Our goal is to test it against a wide variety of known BLE LL exploits to empirically evaluate the security of the hardened parsers and measure the effect these parsers have on the throughput and performance of the BLE protocol.

1 Introduction

The recent disclosures of Sweyntooth [5] vulnerabilities show that many Bluetooth Low Energy (BLE) devices have vulnerabilities in the lower layers of their BLE stack. These vulnerabilities are significant not only because of the ubiquity of the BLE protocol [4] 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.

The severity of these vulnerabilities is augmented by the fact that these vulnerabilities affected many medical devices including medical implants. In some cases, such as the case of medical implants, upgrading the firmware is not trivial and upgrades pose a risk that any error during the upgrade process might make the implant unresponsive.

Furthermore, BLE devices often run real-time operating systems which do not have modern OS exploit prevention mechanisms such as ASLR or DEP. Thus it reduces the bar for malicious attackers to exploit them.

These vulnerabilities were discovered by researchers when they started fuzzing the lower layers of the BLE protocol. 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. This paper addresses these vulnerabilities and hence the focus of this paper will not be to evaluate the security of Bluetooth protocol itself, but rather to develop methods which can prevent *implementation* vulnerabilities in the link layer of the Bluetooth Low Energy (BLE) protocol in the future.

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

- Provides a parser generator which corresponds to a formal language model (finite state machines) whose memory safety and termination has been verified.
- Defines a Parser Description Language (PDL) for IoT binary packet formats with a specific application to Bluetooth Low Energy (BLE) Link Layer (LL) packets.
- Constructs a compiler from the parser description language to C code for packet validation.
- Constructs 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 the target system)

It is pertinent to mention that although the focus of this work is based on the BLE LL packet formats, the general technique is applicable to *any* wireless packet format. Our

hope is to extend this work in the future and apply it to other IoT protocols as well.

The intuition behind writing a parser description language is that bugs are inevitable when coding in low level languages which do not provide memory safety like C.

Further, 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 modelled 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. Moreover, modeling parsers using a language theoretic approach allows us to define a parser description language which allows us prove parser equivalence.

We posit that a 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 for databases and the P4 language for defining software defined network's control plane.

Prior work [8] in verified parsing has 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 shortcoming 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.

Moreover, researchers [7] 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 assumption 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 to describe these parsers? One major limitation of regular expression is that their design makes it cumbersome and unintuitive to describe common constructs found in binary grammars—the PDL we propose is designed specifically for generating parser for binary formats, and avoids these limitations.

We have designed the PDL to describe certain grammatical constructs found in binary grammars that are not possible to be defined by conventional regular expressions. These constructs are frequently found in a wide variety of machine binary languages. To our knowledge crafting a finite state machine parser for them requires coding one up by hand which gives rise to various bugs and vulnerabilities. By providing a framework to generate verified FSM implementation from our new PDL we aim to stymie the rise of such vulnerabilities in the wild.

2 Parser Definition Language (PDL)

The parser definition language is a domain specific language which is a high level description of the parser, and does not have implementation level details in it.

Real world binary protocols and file formats commonly use grammatical patterns not easily expressed in regular expressions. We instead define a parser definition language that makes it easy and intuitive to use these patterns—while still remaining within FSMs.

Our PDL is defined by the following constructs.

- LEN: The length construct consists of a number N, followed by N bytes of data. An example definition is shown in Figure 1. The $gen-len$ function is a PDL function which generates the FSM for the LEN construct.
- Repeat-N: The repeat construct is a function which takes another parser function (such as tag, length) as input and a finite number N and repeats that function N times.
- Repeat-*: This constructs takes as input another parser function (such as tag, or length) and repeats it an infinite number of times. We have not implemented this construct at the time of writing and plan to implement it in the future.

- TAG: The tag construct takes a dictionary of key-value pairs and the bit-width of the tag as input. Upon reading $bit-width$ input bits, the construct parses a tag-value depending on the value contained in the bytes read. An example TAG definition is shown in Figure 1. The $gen-tag$ function is a PDL function which generates the FSM for the TAG construct.

- SEQ: It sequences two grammatical constructs and composes them together.

The parsing result of each construct is stored inside a register. These registers are finite in number and are write-once only. The use of registers allows us to reduce the number of states in the resulting finite state machine. It is important to note these registers do not increase the computational power of the automaton. They are used in a restricted manner to only reduce the states of a large FSM and thus do not allow the PDL to parse languages that are not regular.

The description of these PDL constructs requires that each construct has a "width" key whose value defines how many bits define that construct, a "name" key defining the name of the construct, and a "type" key whose value defines what kind (length, tag, etc) of grammatical construct it is.

TAG construct The TAG construct describes a fixed-length field with fixed labels called "tags". An example of this is the PDU field in the BLE LL packet header. This is a fixed length 4-bit field which describes the type of the BLE LL packet.

If the type of the construct is `TAG`, it must give as input a hash-map of the tag names as keys and their valid binary values represented as a string as their values. An example of this is shown in Figure 1.

The *gen-tag* function takes the width of the tag (in this case 4-bits), the hash-map, and the name of the construct as a string as input.

LEN construct The length constructs consists of two parts. The first part is fixed the second is variable sized. The fixed-size part contains a number in it. This number tells us the size in number of bytes of the variable sized part.

In the BLE LL packet, the length construct is the last field in the header. In this case the fixed part of the length construct is the fixed 8 bits labeled "len" in the header, and the variable part is the payload.

An example of the length construct is shown in Figure 1. When defining the length construct in the PDL, the *gen-len* function takes as argument the size of the field (in this case 8), the endianness as a keyword (:LSB or :MSB) and a name as a string. The length construct will ensure that any construct which follows it will only consume the number of bytes defined by the length field's input, and halt afterwards.

For example, in the case of BLE LL packet if the packet payload length is three bytes, then the parser for the payload of the packet will only be allowed to consume three bytes of input.

Repeat-N construct This construct takes another construct and a number *n*as an input and applies it over the input data *n* times.

SEQ construct The SEQ constructs takes a vector of other constructs and sequences them together. It may be thought of as the concatenation operator which concatenates grammatical constructs together. An example of how this SEQ construct is defined in the PDL is shown in Figure 2. This example shows the concatenation of *PDU, RFU, CHSel, TxAdd, RxAdd, len-field* together to define the grammar of the packet header.

OR construct This construct is not yet implemented but we aim to implement it in the future. This constructs is similar to the OR operator in the regular languages. We aim to implement this construct in the future with the addition of a predicate on the transition edge, so that the transition to the subsequent construct is allowed only if the predicate is satisfied. This will allow us to sequence based on predicates so parts of grammar can be combined more efficiently. For example, the common header of BLE LL packets, is the same but the payloads differ based on the type of packet. In such a format the OR operator will be used to combine the header with the different payload grammars that follow.

Parser composition The parser definition language is designed so that simple parsers can be constructed by TAG, LEN, REPEAT-N constructs and then composed by the SEQ construct. This allows the developer to sequence the individual parser definitions together to construct a parser for the complete packet. Figure 7, 8, 9, 10 show the composition and visualization of a BLE LL packet.

2.1 Language complexity of the PDL

The PDL language is as expressive as a finite state machine. The compiler converts the input given to it to a finite state machine which can then be visualized. This finite state machine is then used to generate the C code.

Expressing the same finite state machines with a regular expression results in unreadable regular expressions. Further, it is inconvenient to express certain grammatical constructs like the length construct as a regular expression. This construct requires the depth of the FSMs which are sequenced after it to be limited ensuring that only a certain maximum number of bits are consumed. Defining such a construct in a regular expression is not only challenging but also unreadable. Our PDL allows us to describe such finite state machines with ease while still being readable. Although, a length construct with

Figure 1: This image shows how the TAG and LEN construct is defined in the PDL.

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

;; length field defined as a LEN construct
(def len-field
  (gc/gen-len 8 :LSB "header_len"))
```

Figure 2: This image shows how to define SEQ construct in the PDL. The variables PDU, RFU, ChSel, TxAdd, and RxAdd are TAG constructs and len-field is a LEN construct.

```
(def adv-header
  (reduce os/seq-graphs
          [PDU RFU ChSel TxAdd RxAdd len-field]))
```

an unbounded size of the payload requires automatons more powerful than a finite state machine to parse. In practice, the maximum size of the payload is bounded. This bound allows us to parse it using a finite-state machine by constraining the maximum size of the payload.

2.2 Example: BLE LL packet format description in PDL

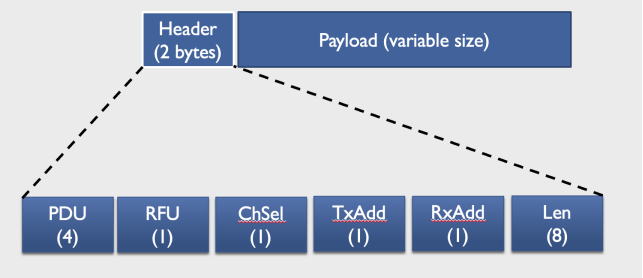
As an example, we show how the design of the parser definition language captures all valid ways in which a BLE LL packet can be constructed.

In this section, we explain the description of a BLE LL packet. The BLE LL packet format is shown in the Figure 3. 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.

Figure 3: Structure of a BLE link layer packet



- 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.

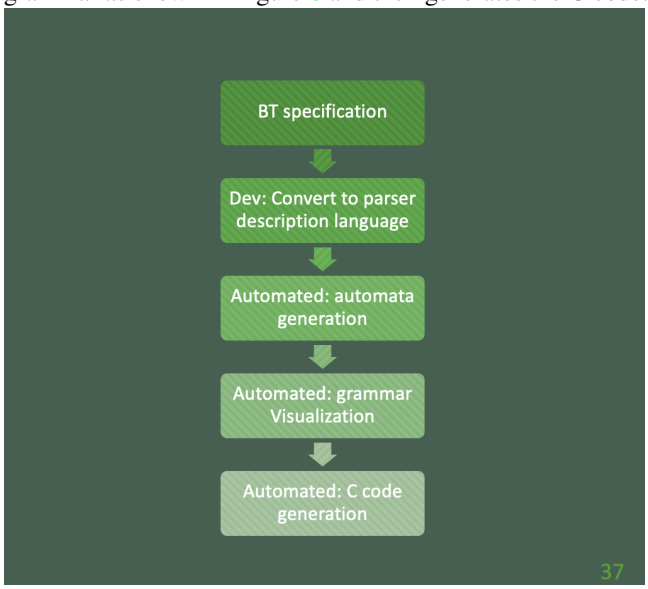
The inner sections of the payload have similar grammatical constructs (i.e. fixed size fields representing a tag or a fixed sized field representing a length of a payload) and their details have been omitted for brevity. The details of the grammatical constructs are explained in section 2. Figure 1 and Figure 2 show the parser description of this packet format.

3 Methodology

3.1 System Architecture

The architecture of the overall system is shown in Figure 4. The developer will read the specification and write a parser in the parser description language. The compiler which will then produce the FSM and visualize it for the developer in case she wants to see the automata and visually verify it. It then generates the C code. The generated C-code is a struct. This struct contains in itself a description of a finite state machine. The struct is then input into a predefined simulator function which simulates the finite state machine on the input. The simulation function is verified by the Frama-C [3] static-analysis framework to ensure that it terminates, only has access to memory that it needs to and is free of memory corruption bugs. Thus for any given input finite-state machine struct, the simulation function is designed to work safely and efficiently. This design also simplifies the C code verification and makes optimizing the simulator function independent of the PDL compiler.

Figure 4: Architecture diagram of the system. The PDL compiler takes the PDL grammar description and visualizes the grammar as shown in Figure 8 and then generates the C code.



3.2 Compiler construction

The PDL to C compiler is written in Clojure [6] 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 then generates C code which simulates a finite state automata. As mentioned earlier, the compiler dynamically generates a C struct and also outputs the pre-defined verified simulation function so that the generated code is able to simulate the automaton.

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

The compiler generates the finite-state machine definition as struct. This struct contains a description of the finite-state machine. The simulation function then takes this struct as an input and tries to parse the input based on the FSM description. This simulation function is optimized to ensure that it can simulate an FSM with a very large number of states.

Furthermore, the simulation function is verified to ensure that it only accesses the memory it is allowed to and eventually terminates.

4 Verification of the C code

The verified C-code FSM simulator function takes as input the array of input bytes of type `uint8_t`, the length of the input buffer (in bytes) and the description of the FSM as a struct.

The C-code is written in Frama-C [3] which is a static analysis engine for C.

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 at the top of the function definition can be seen in Figure 11 and in Figure 12. 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.

5 Optimizing the compiler

The struct is generated by the compiler from the PDL and is defined as a global constant. This struct contains an array which consists of tuples representing the transitions of the FSM graph. An edge tuple is a struct which contains a start node and an end node allowing us to know the transitions from a given node. 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.

This was our initial attempt at encoding the graph and our 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.

To optimize the size of the resulting binary, we rewrote the simulation function to a function which simulates an FSM of depth `d` but does not requires the transition tables. The function is shown in Figure 12.

The difference of this function to the old one is that it simulates a finite-state machine of depth `d` by labeling the first node as 0 and then calculating the label of the left and right child. Since the FSM graph has at most two children one for a 0 bit and one for a 1 bit, it can calculate the label of the child and jump there.

If we have a binary tree with root labeled n with left child labelled $n+1$ and right $n+2$. Then for a any given node m in the tree, the left child's label can be calculated by the formula $2m+1$ and the right child will be $2m+2$ provided the tree root node is labelled 0. We use this formula to simulate a very large finite-state machine without needing a large transition table.

When it terminates it returns the label of the node the FSM halted on. This eliminates the use of transition table required by the first approach, and now we only need to remember the final node label in the packet description generated by the PDL compiler.

A disadvantage of this approach is that it can only parse 32 bits in one function call. This is because it represents the current state as an integer which is 32 bits long.

Figure 5: FSM struct as an output of the compiler. This goes as input to the simulation function.

```
const int PDU_VALID_VALS[] = {
    0x00, 0x01, 0x02, 0x03, 0x04,
    0x05, 0x06, 0x07, 0x08
};

const cons_t* SWITCH_ON_PDU[] = {ADV_IND, ADV_DEV_ADDR};

cons_t ADV_DATA     = {"ADV_DATA", 0, 0, 0, SKIP, 8, 0, 0};
cons_t ADV_TYPE     = {"ADV_TYPE", 0, 0, 0, SKIP, 8, 0, &ADV_DATA};
cons_t ADV_LEN      = {"ADV_LEN", 0, 0, 0, LEN, 8, &ADV_TYPE, 0};
cons_t ADV_DEV_ADDR = {"ADV_DEV_ADDR", 0, 0, 0, SKIP, 8, 0, &ADV_LEN}; // 48 -> 8

const_t SWITCH_PKT_TYPE = {"SWITCH_PKT_TYPE", 0, 0, 0, SKIP, 8, 0, &ADV_LEN};
cons_t LEN_FIELD    = {"LEN", 0, 0, SWITCH_ON_PDU, LEN, 8, &ADV_DEV_ADDR, 0};
cons_t CONTROL      = {"CONTROL", 0, 0, 0, SKIP, 4, 0, &LEN_FIELD};
cons_t PDU          = {"PDU", 3, &PDU_VALID_VALS, 0, TAG, 4, 0, &CONTROL};
```

Figure 6: FSM struct definition

```
enum CONS_TYPE {
    TAG,
    LEN,
    SKIP,
    REPEAT,
};
typedef struct cons_t {
    char* name;
    int valid_vals_len;
    int* valid_vals;
    enum CONS_TYPE type;
    int size; // in bits
    struct cons_t* inner_cons;
    struct cons_t* next_cons;
} cons_t;
```

However, in practice we haven't seen *individual* constructs longer than 8 bits and thus it has not been a problem. It is important to note that this limitation can be bypassed by using a larger data type to increase the number of bits parsed in a single construct.

This new design results in a struct definition as shown in Figure 6. An example of a BLE LL grammar description is shown in Figure 5. The struct contains the size and type of the construct, along with a name and in the case of a TAG construct — a list of valid values that the tag is allowed to take defined as an array of *int* called *valid_vals*.

The struct's contain in themselves a pointer to the next construct which allows the simulation function to locate the next construct after parsing the current one.

The automaton we generate is deliberately limited to reading bits at a time because the set of alphabet of the FSM is the set $\{0, 1\}$. This restriction is enforced because if the alphabet of the FSM was increased to the set of valid bytes (or any set of symbols of a larger size), then parsing a $single-bit$ field would not be possible because the FSM cannot recognize a field which is smaller than the size of a symbol. However for grammars which don't have single-bit fields, such automatons can be designed and used.

6 Evaluation

Deployment on micro controller For evaluation, we deployed the generated code on an Ubertooth One device [2]. This device runs on a ARM cortex M-3 microcontroller with an open source firmware. The max RAM size is 16K bytes. The device comes with an open source BLE scanning application which can be installed on the firmware. It is present in the `bluetooth_lxtx` folder of the Ubertooth repository. This application also has a corresponding host application which can parse and display packets received from the Ubertooth device. 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 FSM description defined in a struct as input. The packet is only 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 device deployed with our hardened parsers. These attacks were launched from a NRF52840 dongle [1] which intentionally sent malformed LL packets. The malformed packets contained invalid values for tags and extraordinarily large length field values. We also tested for packets of different types (such as a SCAN_REQ packet when the grammar only allowed ADV_IND packets) and observed that only the packets which match the grammar description were

Figure 7: An example automata visualization generated by the compiler, this represents an automata that will parse the PDU of the BLE LL packet header.

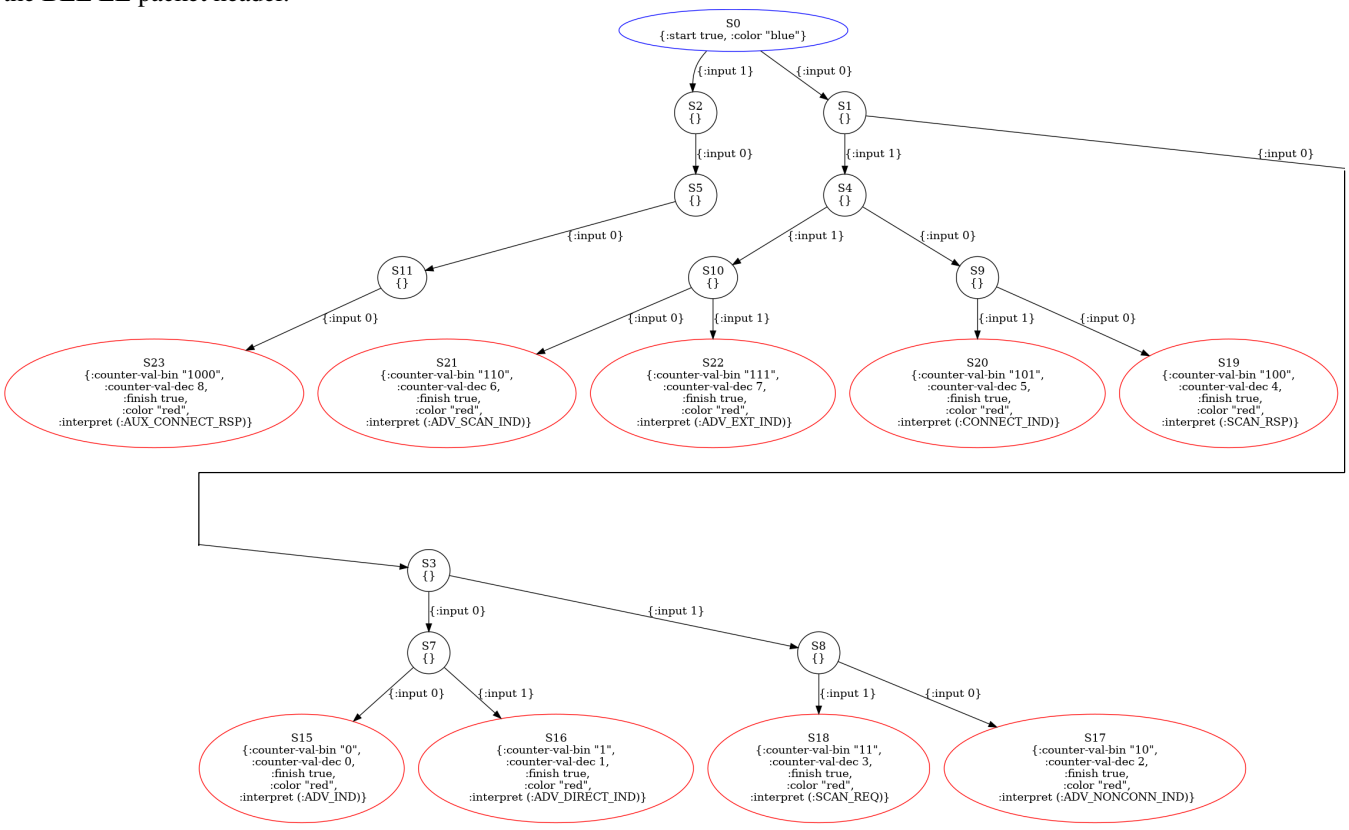


Figure 8: An example FSM visualization generated by the compiler: this shows the resulting FSM when two TAG constructs (PDU and RFU) are concatenated by the SEQ construct.

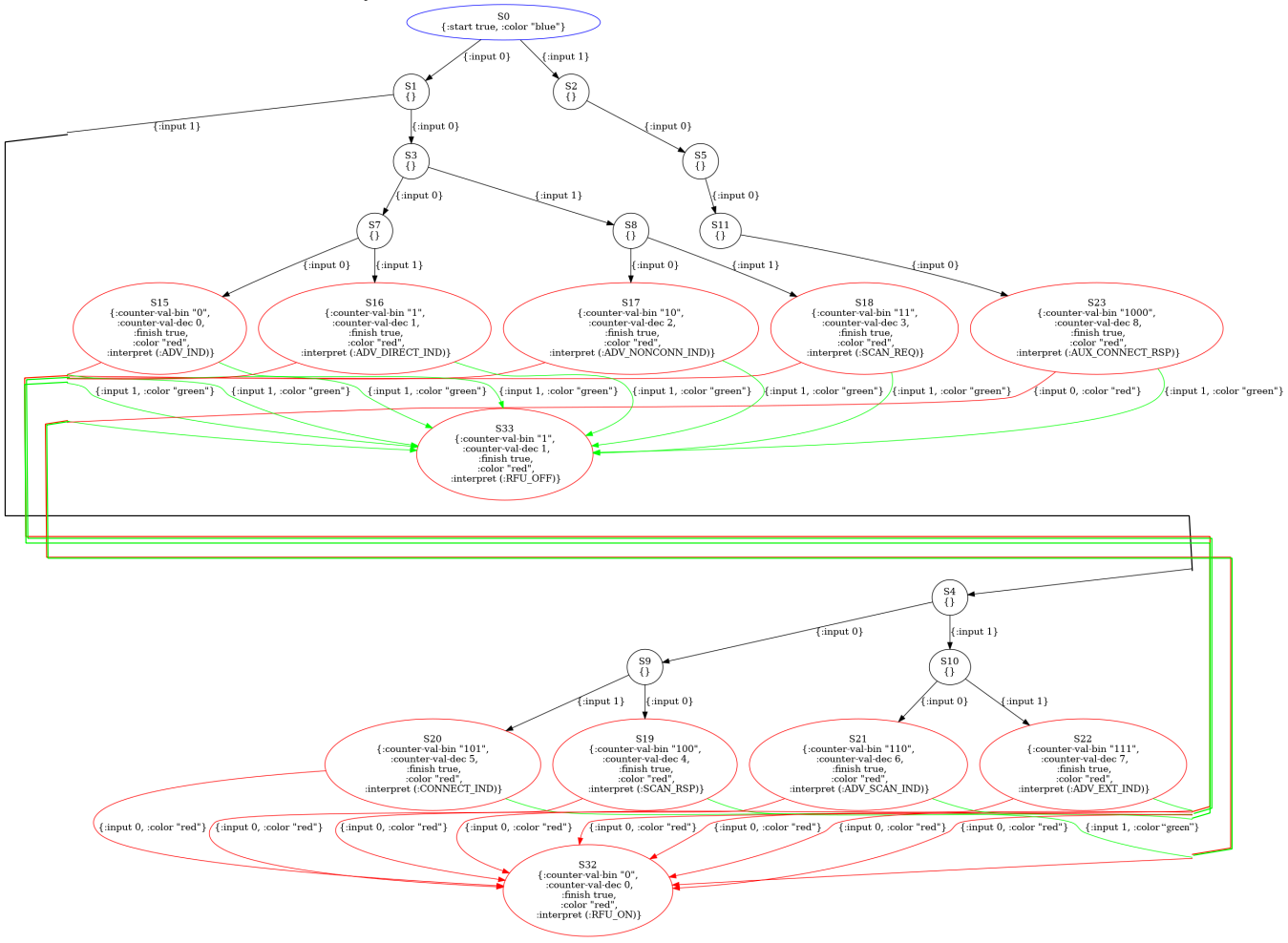


Figure 9: An eagle's-eye view of an example TAG construct, which represents the first four bits of the BLE LL packet. These four bits represents the PDU type of the BLE LL packet. The S0 node is the start node of the finite-state machine and S15 - S23 nodes are the finish nodes of the TAG construct. The whole TAG construct is enclosed in a box which shows the nodes of the finite-state machine which are part of the TAG construct in the larger finite-state machine which parses the BLE LL packet. Figure 8 shows the same finite-state-machine.

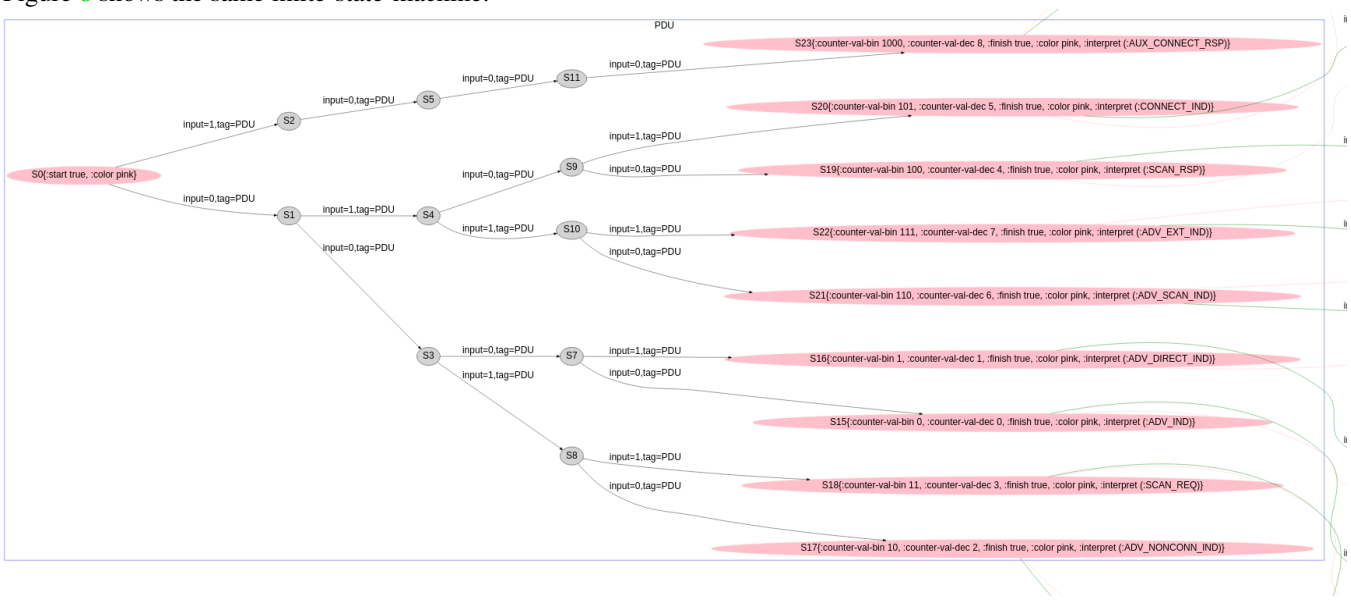


Figure 10: This image shows an eagle's-eye view of the first two TAG constructs of the finite-state machine generated for parsing BLE LL packets. The left most node is the start node. The finite-state machine's inside the square blue boxes signify TAG constructs. This image shows two TAG constructs joined together by the sequence operation. The left most square blue box represents the finite-state machine parser for the PDU field of the BLE LL packet. It contains the finite state machine shown in Figure 9. This image shows how the finite-state machine of Figure 9 fits inside the larger finite-state machine of the BLE LL packet parser.

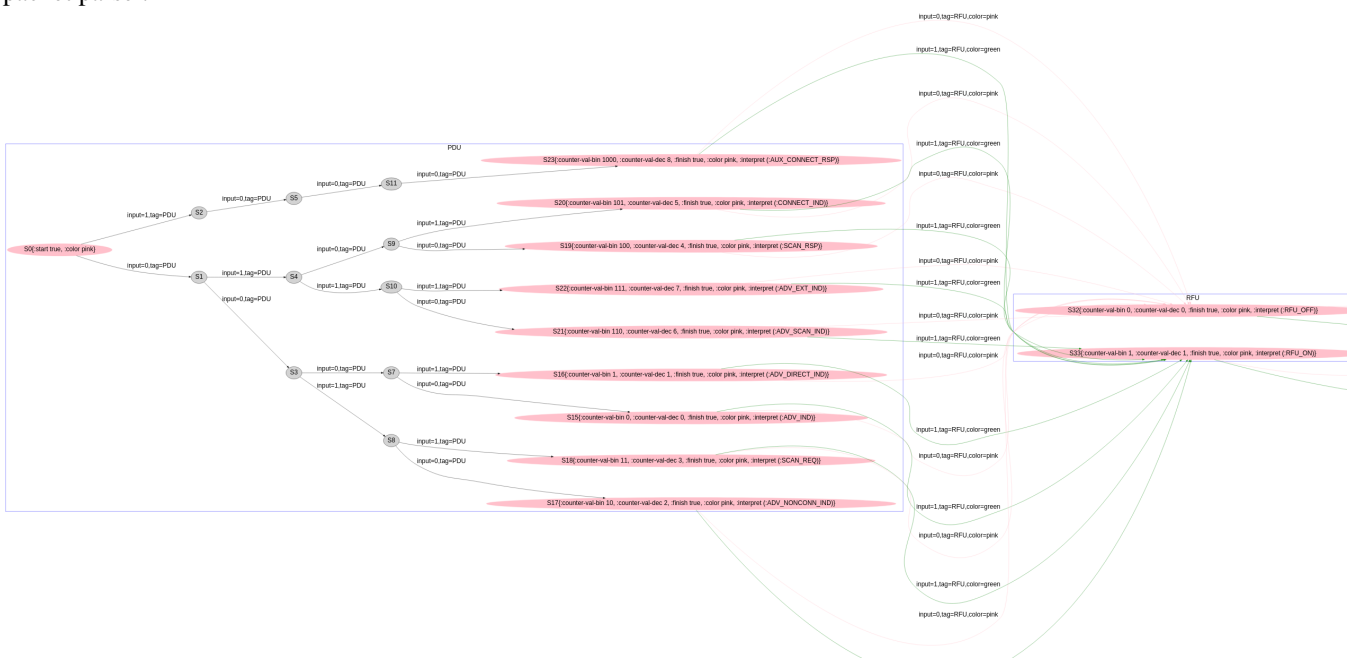


Figure 11: The old simulation function with annotations at the top of the functions. The static-analysis engine verifies the constraints at the top to ensure memory safety and termination. This approach of having a transition table did not work due to memory constraints on the micro controller.

```
/*@ requires TAPE_LEN >= 0;
   requires \valid(TAPE+(0..(TAPE_LEN-1)));
   requires fsm.TAPE_LEN_BITS >= 0;
   requires \valid(fsm.TRANSITION_TABLE+(0..fsm.TAPE_LEN_BITS));
   terminates TAPE_LEN <= INT_MAX;
   assigns \nothing;
 */
int simulate_fsm(const char* TAPE, const int TAPE_LEN, const fsm_t fsm) {
    int cur_state = fsm.START_STATE;
    bool cur_symbol;
    printf("Started at state: %d\n", fsm.START_STATE);
    int i;

    for (i = 0; i < fsm.TAPE_LEN_BITS; ++i) {
        cur_symbol = get_next_bit(TAPE, TAPE_LEN);
        switch (cur_state) {
            case 0:
                printf("transitioned to fail state\n");
                return -1;
                break;
            default:
                // do transition
                // update the cur_state
                // based on read symbol
                if (cur_symbol) {
                    cur_state = fsm.TRANSITION_TABLE[cur_state].t1;
                } else {
                    cur_state = fsm.TRANSITION_TABLE[cur_state].t0;
                }
                printf("Transition state: %d\n", cur_state);
                break;
        }
    }

    // fail state
    if (cur_state == 0 || cur_state > fsm.MAX_STATES) {
```

allowed through by the hardened parser as valid. Further, the Ubertooth device was able to discarded all the malformed packets.

In the future we aim to test out known exploits such as the Sweyntooth exploits against the hardened parsers to ensure that they can successfully prevent such exploits.

7 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 the malformed packets with Link Layer Length Overflow sent by the NRF52840 device.

We also aim to further evaluate the effectiveness of the parsers by attacking the Ubertooth firmware with additional Sweyntooth exploits.

8 Future work

In the future we plan to attempt to generate verified parsers which can recognize a higher language complexity class than regular languages.

We also plan to model more IoT protocols and make the PDL robust enough to model the most popular binary IoT protocols or even more layers of the BLE protocols.

We plan to evaluate the implementation using another BLE device, and establish a BLE connection and measure the throughput of the BLE connection with the verified parsers in place.

Lastly, we plan to formally prove the conversion from the PDL to the FSM and work further on the verification - which only covers memory safety and termination at the moment - to ensure the compiler is free from any logical bugs as well.

Figure 12: Optimized version of the simulation function which represent a state as an integer. The rest of the uses this function to simulate a state machine of depth d, and then compares the final value of i with the accepted list of values. This allows us to eliminate the transition tables and reduce the size of the binary.

```
/*@ requires input_len >= 0;
   requires \valid(input_buffer+(0..(input_len-1)));
   requires d > 0;
   terminates d < 32;
   assigns \nothing;
 */
int simulate_machine(int d, uint8_t* input_buffer, int input_len) {
    // imagine a complete binary tree with node labeled '0' as root.
    // left child has egde which means reading an input "0" right
    // child is reading "1". so a tree of depth d will have a total of
    // 2^(d+1) - 1 nodes where the left most leaf node is labeled
    // 2^d - 1.
    int i = 0;
    int total_number_nodes = (1 << (d + 1)) - 1;
    int starting_leaf_node_val = (1 << d) - 1;

    logger("starting machine...\n");
    logger("value of d: %d\n", d);
    logger("starting leaf node: %d\n", starting_leaf_node_val);
    logger("total_nodes: %d \n", total_number_nodes);
    while (!(i < total_number_nodes && i >= starting_leaf_node_val)) {
        int read_bit = get_next_bit(input_buffer, input_len);
        switch (read_bit) {
            case 0: {
                // left child
                i = 2*i + 1;
                break;
            }
            case 1: {
                // right child
                i = 2*i + 2;
                break;
            }
            default:
                logger("Error reading bit\n");
                return -1;
        }
        logger("next state: %d\n", i);
    }

    // i is leaf now
    int read_binary_value = i - starting_leaf_node_val;
    logger("read bin val: %d\n", read_binary_value);
    return read_binary_value;
}
```

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/sameedali/c-fsm-gen.git for the Clojure compiler code. The Ubertooth One micro controller code is available at https://github.com/sameedali/BLE_parser.git.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.