Automatic Construction of Hardware Traffic Validators

Jason Dahlstrom[1], Brandon Guzman[2], Ellie Baker[2] and Stephen Taylor[2]
[1]Web Sensing LLC, Lebanon NH, USA
[2]Thayer School of Engineering, Dartmouth College, Hanover NH, USA
Jason.Dahlstrom@websensing.com
Ellie.F.Baker.22@dartmouth.edu
Brandon.J.Guzman.22@dartmouth.edu
Stephen.Taylor@dartmouth.edu

Abstract: This paper describes a fully automated process that creates a custom hardware traffic validator directly from a formal grammar and deploys it within a specialized network security appliance. The appliance appears as a hidden, all-hardware "bump-in-the-wire" that can be inserted within any network segment; it stores and validates messages on-the-fly, and either forwards or drops individual packets in real-time. Consequently, it serves to disrupt and mitigate stealthy remote attacks that leverage zero-day exploits and persistent implants. Allowed traffic, files, and mission payload formats are specified formally using a standard Look-Ahead, Left-to-Right (LALR) grammar that operates on ASCII and/or binary data. The grammars can be expressed either in Backus-Naur Form (BNF), used by industry standard tools such as Bison, or through state-of-the-art combinators, such as Hammer, under development within the DARPA SafeDocs program. Bison and Hammer compiler tools are used to generate standard shift/reduce parsing tables. These tables are post-processed to improve their compactness and practical viability. The optimized tables are then combined with a generic push-down automaton to form a complete parser. The parser is then automatically transformed into a hardware circuit using High-Level Synthesis (HLS). The result is a composable block of circuitry that can be directly inserted into a generic communications harness embedded within a Field Programmable Gate Array (FPGA) on the network appliance.

Keywords: parsing, LALR grammar, traffic validation, FPGA, Bison, Hammer

1. Distribution

This research is supported by the Defense Advanced Research Projects Agency (DARPA) under contract W31P4Q-20-C-0033. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views of the Department of Defense or the U.S. Government. The paper is released with the following distribution determination:

Distribution A: (Approved for Public Release, Distribution Unlimited).

2. Introduction

Many organizations handle sensitive data and files relating to military missions, trade secrets, intellectual property, private personal data, and/or classified projects (NBAR, 2021). Traditionally, these organizations have been well advised to implement an *airgap* that physically isolates computers containing sensitive information from the Internet, to protect against theft. Unfortunately, airgaps come with a substantial cost in productivity and assume that profession staff, with the expertise to handle sensitive information, is co-located. Airgaps are increasingly impractical given the need to connect critical systems, such as industrial plant, to cloud-based analytic platforms (e.g., Google Analytics) or support condition-based vehicle maintenance (Adams 2012, Carter 2013, Shanthakumaran 2010). Similarly, individual air, space, and ground vehicles increasingly rely upon standard networking technologies to link embedded control systems with sensors, actuators, and human machine interfaces through industry standard buses (e.g., CAN, J1939, MIL-STD-1553, USB) and communications interfaces (e.g., Gigabit Ethernet (GigE), OpenVPX, and PCIe). Network connected personal devices – phones, tablets, and laptops – are increasingly being used to manage and interact with these systems. Though conceptually air gapped, these systems are intermittently connected with military installations to provide mission parameters, or to effect maintenance and upgrades. Though network boundary protections are used during these activities, there are many threat vectors that circumvent these protections, and airgaps in general; these include unintended network connections, insiders, zero-day exploits, supply chain interdiction, and persistent implants (Kushner 2013).

This paper describes an automated process to validate network traffic employing Field Programmable Gate Array (FPGA) technology. This technology is employed at the core of an all-hardware *network appliance* that continuously validates the integrity of network traffic. This combines the isolation of an airgap, with the

convenience of an Internet connection, at considerably less risk than an open connection: it provides a base-of-trust in hardware that both disrupts, and is impervious to, stealthy remote attacks perpetuated through zero-day exploits and persistent implants. Two such appliances are shown in Figure 1 – an Ethernet appliance and a Smart Network Interface Card (Smart NIC). Both consist, by design, of a ***single*** FPGA chip lying between industry standard buses and network connections. Each appliance thereby forms a "bump-in the-wire" with the FPGA acting as a communication bridge. Consequently, the FPGA can monitor and interact with all systems attached to its interfaces, and may act to store, validate, drop, or forward traffic. Both appliances can use a variety of pin-compatible FPGA sizes from the low-cost Spartan-7 devices pictured on the Ethernet device, to the Artix-7 pictured on the Smart NIC; however, only the Artix devices allow partial reconfiguration.

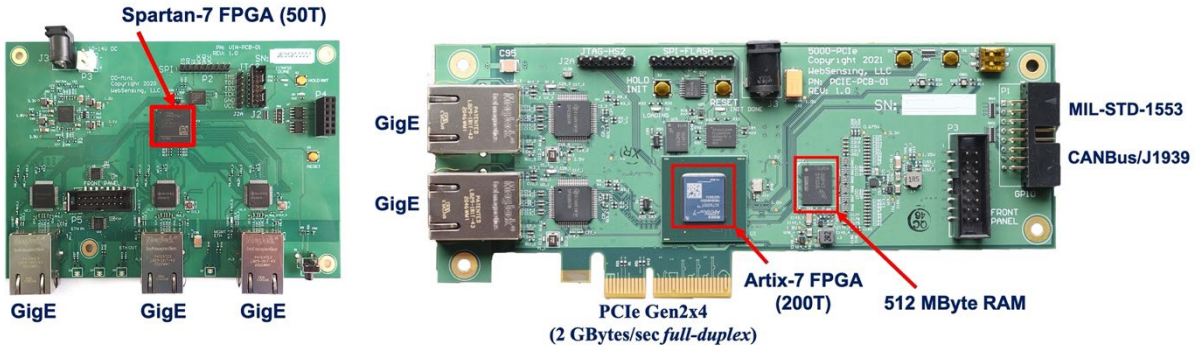

Figure 1: Ethernet (left) and Smart-NIC (right) appliances

The appliances offer several key security advantages: All sensitive data -- encryption keys, buses, and algorithms -- is strictly hidden within the security perimeter afforded by the FPGA chip-boundary (Dahlstrom and Taylor 2018, Aug 2019, Oct 2019, 2020, 2021) thereby mitigating reverse engineering if an appliance is lost or stolen; No software is present in the device, mitigating malicious implants and zero-day attacks (Kushner 2013); any connection can be used as an out-of-band channel to adapt the device to alternate mission profiles, augment its internal functions, or upgrade the device; Extensive anti-tamper and circuit destruction techniques enhance resilience; Large files and data repositories can be held within RAM attached to the FPGA, without violating the strict visibility constraints imposed by the FPGA chip-boundary, by encryption and decryption algorithms embedded inside the FPGA. For versatility, all circuits resident in the appliances are developed using a rapid prototyping technology called High Level Synthesis (HLS). This process allows algorithmic specifications to be designed and tested in C, C++, or System-C. The working code is then automatically transformed into a standalone, reusable, *hardware block*. These reusable circuit plugins can be directly integrated into a static circuit design in the FPGA for production deployments. Alternatively, the block can be treated as an Open Container Initiative (OCI) compliant *container*; Using a technique known as *partial reconfiguration*, the FPGA can then be partitioned into segments and containers can be dynamically loaded into a partition linking it into the overall function of the device on-the-fly. To manage this process, we have developed a thin, hypervisor-like hardware layer termed a *Nanomarshal* (Dahlstrom et al, Nov 2019).

3. Parser plugins

Recall that each network appliance is concerned with monitoring the flow of traffic across its interfaces and *validating* both that messages adhere to an industry standard protocol and that message content is valid in the context of some tactical mission. Generally, its action on detecting a valid message is to allow it to pass; conversely, its action on detecting invalid data is to drop the message -- mitigating potential exploitation -- and/or generate an alert. To achieve message validation, the TSNIC uses custom *parsing engines* inserted across its communication paths. Parsing is the general process of taking an input stream of symbols and understanding their format (syntax) and meaning (semantics). For example, compilers such as GCC use a parser to validate that a computer program, written in some programming language such as C/C++/Java/Fortran is written correctly (i.e., is syntactically valid), and to understand its structure (i.e., its semantics) for the purpose of machine code generation and optimization. Parsers are tools that apply a collection of formal *grammar rules*, defining some input language, to determine if the input adheres to the rules. For example, the following 3-rule grammar G defines a language in which a stream of symbols is valid only if it begins with the character 'a', ends with 'c', and contains one or more intervening 'b' characters:

G : 'a' Bs 'c' ;
Bs: 'b' | Bs 'b' ;

The "or" symbol | designates an alternative definition for the rule defining "Bs". This grammar accepts as valid the input streams *abc*, *abbc* and *abbbc* etc, but rejects any other stream, e.g., *a, ac, aaa, ccc, adx, abbbx*, etc. Individual characters such as 'a' are *terminal symbols* that must be present in the input stream; all other symbols are non-terminals representing intermediate structural elements. For binary grammars, hexadecimal terminal values can also be used, for example, the value '\xFF' represents a single byte value corresponding to 255 in decimal.

Parser generators are tools that take a grammar as input and automatically generate a program that implements the associated parser. The most mature and widely used generator is Bison which accepts two primary classes of grammar: Generalized Look-Ahead (GLR) and the more restrictive Left-to-Right Look Ahead (LALR). Both classes of grammar are expressed in Backus-Naur Form (BNF), used above to define the grammar G. Under the DARPA SafeDocs program, new tools are being developed based formal methods. One of the more mature is the Hammer *combinator* library (Bratus, et al. 2016) which provides a collection of well-defined base parsers and methods to combine them to build more complex parsers – all expressed in the C-programming language. The resulting parsers are provably correct by construction. The Hammer library provides a collection of backends that allow different classes of grammar to be implemented, including GLR and LALR.

Though GLR grammars are more general, LALR grammars are sufficient for validating a wide variety of communication protocols and file formats. Their simplicity allows them to be realized by a *push-down automaton* – a finite state machine employing a single stack to store symbols while parsing the input stream. The state machine relies on two core operations *shift* – involving saving a symbol from the input onto the stack -- and *reduce* – involving the application of a grammar rule to detect a structure in the input and reduce the symbols on the stack. The state-machine is generic and common to all grammars, however, the order in which shift and reduce operations are applied to the input is based on a collection of *parsing tables* derived from the grammar, usually referred to as Action/Goto tables (Aho, Sethi, and Ullman, 1988). These tables map the current state of the automaton, to a next state based on the symbol read from the input stream.

The network appliances in Figure 1 can parse any LALR grammar expressed in either Bison or Hammer. Figure 2 shows the automation process used to transform a Grammar into a hardware parser-plug. Using Bison, the input grammar, defined in BNF, is fed directly into the standard Bison parser generator (leftmost path). This produces a set of parsing tables, expressed in a machine-readable XML format. For Hammer, an equivalent parser can be defined using pre-existing combinators in C linked with the Hammer library (alternate path). We have augmented the Hammer library to output the same XML format as Bison. A conversion tool – *xml2h* – is then used to convert the XML tables into a C-header file (pda.h) containing a large two-dimensional C-array. Rows in the array correspond to *states* in the automaton, while columns designate terminal and non-terminal symbols encountered when reading the input stream. Entries in the array designate shift or reduce actions applied in each state. Consequently, the C-array provides a complete definition of how the push-down automaton should operate to validate any particular grammar. The C-array (pda.h) is combined with a generic LALR *automaton* (pda.c) (Aho, Sethi, and Ullman, 1988) and *testbench* code (main.c) to produce a runnable C-program implementing the parser. This software parser is validated using a set of representative test vectors that are successively loaded from files by the testbench (main.c) to ensure that the parser operates correctly.

The parser code is carefully constructed to operate on streaming-data and feed directly into the Xilinx High-Level Synthesis (HLS) tools. These produce a hardware circuit block that implements the software parser derived from the pda.[ch] files. The HLS hardware-software co-simulation tool is then used to validate the hardware parser. It uses the same software testbench and test vectors, used to validate the software version of the parser, to ensure the hardware version operates correctly. The validated hardware is subsequently output as a *parser-plugin* component that can be loaded directly into one of the network appliances shown in Figure 1. The plugin is organized to connect with the communication interfaces on the board through industry standard AXI-streams. These are either statically connected through the Xilinx Vivado tool chain, or dynamically managed through partial reconfiguration by the Xilinx Vitis tool chain.

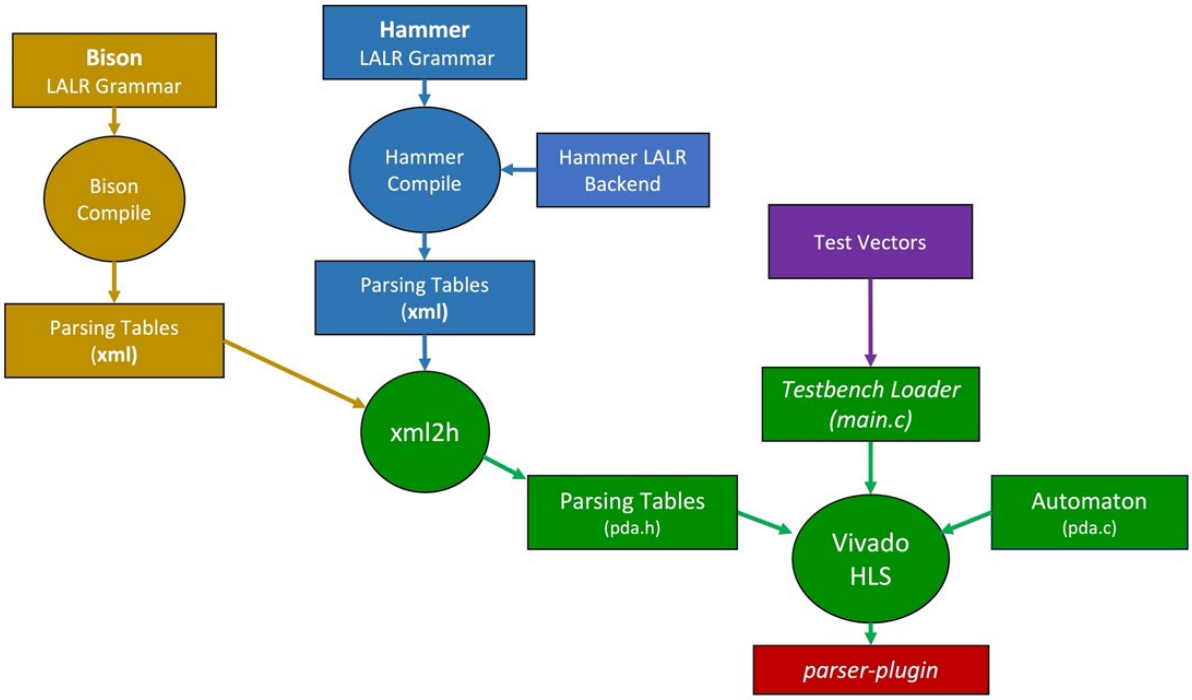

Figure 2: Automated Parser generation process

4. Table Optimization

Unfortunately, though a conventional two-dimensional C-array (pda.h) is a convenient conceptual framework for organizing the automation process, it is impractical since it contains many states that cannot be reached in practice. Consequently, a highly optimized alternative representation is used that removes much of the empty structure from the array. These optimizations follow similar techniques to those employed internally by Bison and are described in detail online (Popuri, 20006). For example, consider the following Hammer Grammar:

```
HParser *init_parser() {
        HParser *G;
        H_RULE(chara, h_ch('a'));
        H_RULE(obrac, h_ch('('));
        H_RULE(cbrac, h_ch(')'));
        HParser *P = h_indirect();
        HParser *M = h_indirect();
        h_bind_indirect(P, h_choice(chara, h_sequence(obrac,M,cbrac,NULL), NULL) );
        h_bind_indirect(M, h_optional(P) );
        G = h_sequence(P,h_end_p(),NULL);
        return G;
}
```

The resulting parsing table for this grammar, involves 10 states:

```
static int16_t table[10][9] = {     /* line 1 */
 {0,5,0,3,2,0,6,4,32767},           /* state 0 */
 {-7,-7,-7,-7,0,0,0,0,0},
 {-6,-6,-6,-6,0,0,0,0,0},
 {-5,-5,-5,-5,0,0,0,0,0},
 {0,5,-3,3,2,8,9,0,0},
 {7,0,0,0,0,0,0,0,0},
 {-4,-4,-4,-4,0,0,0,0,0},
 {0,0,10,0,0,0,0,0,0},
 {-2,-2,-2,-2,0,0,0,0,0},
 {-1,-1,-1,-1,0,0,0,0,0} };
```

Each line in the table represents a state in the push-down automaton. The first 4 entries in each line correspond to the standard ***action*** table associated with detection of a terminal symbol in the grammar; while the last 5 entries correspond to the standard ***goto*** table associated with non-terminal symbols. Positive values in the action table represent shift operations, negative values are reductions, zeros represent parse failures and unreachable states, and 32767 is a reserved value representing the *accept* symbol.

A first optimization (O1) arises from inspection of the data in the tables: notice the entries in lines 3,4,5,8,10 and 11 (bold-faced) of the parsing table. Each designates a *default reduction* (-ve) operation applied for *every* terminal symbol, with unreachable (0) *goto* table entries. Consequently, it is possible to represent each of these lines by a single number designating the default reduction to be applied in that state (7,6,5,4,2,1 respectively). Since the state numbering is arbitrary, it is possible to reorder the states such that all of these transitions fall, in order, to the end of the table, the *goto* segment can then be represented independently, and all of the default lines can be removed and represented by a single array (*deftable*). This array is accessed only when a shift operation exceeds the number of remaining states in the action table. The resulting parsing tables are shown below:

```
 static uint16_t deftable[6] = {
  -7,-6,-5,-4,-2,-1
};

static int16_t actions[4][4] = {
  {2,0,6,0},
  {2,-3,6,0},
  {0,0,0,8},
  {0,10,0,0}
};

static int16_t gotos[4][5] = {
  {5,0,3,7,32767},
  {5,4,9,0,0},
  {0,0,0,0,0},
  {0,0,0,0,0}
};
```

A second optimization (O2) reduces the *gotos* table by adding a default goto table (*defgotos*) then generating a remaining *gotos* table with the defaults removed. Positive values in the *defgotos* table indicate unique defaults; negative values (e.g. -1) indicate where to find a non-default element (i.e. column = (-default)+1) :

```
static int16_t defgotos[5] = {5,4,-1,7,32767};

static int16_t gotos[4][1] = {{3}, {9}, {0}, {0} };
```

Obviously, to utilize these smaller tables requires changes to the operation of the automaton, since it must now detect and use indirection rather than directly index into a two-dimensional array. Table 1 shows the resulting table size, in bytes, for a variety of parsers taken from the Thayer Parser Experimentation repository online at *https://github.com/lvln/thayer_parsers*: O0 represents unoptimized tables, O1 uses the first optimization, and O2 the second. The last two columns show the percentage reductions adding O1 only (O0-O1) and subsequently adding O2 (O1-O2). Only a single ASCII parser – *json* (without Unicode) – is reported in Table 1 for both Bison and Hammer to provide a point of comparison. The others all combine both binary and ASCII within a single parser and are more representative of expected use cases – on these grammars we see a dramatic cost saving: more than 80% across the board. The second round of optimization produces only a small improvement. Though a third level of optimization is possible (removing the zeros in the final gotos table), the 01-02 results indicate a diminishing return for added complexity in the automaton.

Table 1: Parser optimization results

Parser	Version	O0	O1	O2	O0-O1 %	O1-O2 %
json	Bison	51712	14080	12278	72.8	12.8

Parser	Version	O0	O1	O2	O0-O1 %	O1-O2 %
json	Hammer	74098	17534	13254	76.3	24.4
command	Hammer	480564	74128	69498	84.6	6.2
response	Hammer	151140	6426	6288	95.7	2.1
json.unicode	Hammer	406944	36006	30320	91.2	15.8

5. Notational conveniences

Our experiments with parser compactness showed that Hammer provides a variety of notational conveniences not available in Bison BNF. These primarily revolve around the ability to express *ranges*, *enumerations*, and *sequences* of terminal symbols. For example, in BNF, to specify a hexadecimal digit we might use 25 rules:

```
HexDigit: Digit | LowerAF | UpperAF ;
Digit: ‘0’ | ‘1’ | ‘2’ | ‘3’| ‘4’| ‘5’| ‘6’| ‘7’| ‘8’| ‘9’ ;
LowerAF: ‘a’ | ‘b’ | ‘c’| ‘d’| ‘e’| ‘f’ ;
UpperAF: ‘A’ | ‘B’ | ‘C’| ‘D’| ‘E’| ‘F’ ;
```

Similarly, to recognize any bracketing symbol we might use 8 rules:

```
Bracket: ‘(‘ | ‘)’ | ‘[‘ | ‘]’ | ‘{‘ | ‘}’ | ‘<’ | ‘>’ ;
```

Finally, to represent the values true and false we would use 2, rather ugly, rules:

```
True: ‘t’ ‘r’ ‘u’ ‘e’ ;
False: ‘f’ ‘a’ ‘l’ ‘s’ ‘e’ ;
```

Bison generally assumes that such rules would be handled by a separate lexical analysis program that produces *tokens* to be consumed by the parser. Hammer integrates such operations as elementary combinators, implemented efficiently through comparisons, and consequently builds them formally into a single cohesive and general system. Integration of Lex, which generates lexical analysers, into the automaton framework shown in Figure 2 would of necessity require the added complexity of calling Lex functions fed into the HLS process. Instead, we choose to dispense with Lex completely, and simply pass all input bytes directly into Bison, with each grammar incorporating its own rules for lexical analysis, consistent with Hammers treatment. To simplify these lexical steps, we have developed a pre-processor (implemented in Bison) that unambiguously accepts an extended version of BNF, termed xBNF. The pre-processor accepts the following notations:

[t1-t2] -- Accepts any terminal value in the range **t1** to **t2**, where **t2>t1**.
[t1,t2,...,tn] -- Accepts an enumerated range of terminal values **t1** to **tn**; **n>1**.
“string” -- Accepts any string of terminal characters

Consequently, in xBNF the above examples would be rendered:

```
HexDigit : [‘0’-‘9’] | [‘a’-‘f’] | [‘A’-‘F’] ;          /* ranges */
Bracket:   [‘(‘, ‘)’, ‘[‘, ‘]’, ‘{‘, ‘}’, ‘<’, ‘>’] ;  /* enumeration */
True: “true” ;                                         /* sequences */
False: “false” ;
```

Pre-processing would result in an equivalent set of rules to the original, however, with less non-terminals for intermediate rules (c.f. HexDigit), since these exist purely for human readability. Bison’s BNF input language allows the parsing of binary formats by virtue of the ability to express terminal values in an equivalent hexadecimal format. For example, the rule for LowerAF could have been expressed:

```
LowerAF: ‘\x61’ | ‘\x62’ | ‘\x63’ | ‘\x64’ | ‘\x65’ | ‘\x66’ ;
```

Unfortunately, Bison expects Lex to return the value zero (0) to signify the end of input. Consequently, it is not possible to parse the full range of binary values without special treatment of zero. To resolve this issue, we

replace lex with a simple function that distinguishes the special case of zero in the input stream and returns a special token in its place. Using this convention, it is then possible to add two new notations to xBNF that simplify the parsing of binary formats:

'\x00' -- Accepts the terminal symbol for zero in hexadecimal.
***** -- Accepts any byte (i.e. in the range **'\x00'** to **'\xFF'**)

Table 2 compares the expressiveness of a variety of parsers written in xBNF, Bison BNF and Hammer using the parsers in Table 1. As expected, xBNF consistently improves upon BNF. Hammer is consistently the most complex, this is due to its use of the comparatively complex syntactic structure of the C programming language and the need to handle forward references, as exemplified by lines 6-9 of the example Hammer grammar shown previously in Section 4.

Table 2: Source lines of code

Parser	xBNF	Bison BNF	Hammer
json	27	44	72
command	22	41	25
response	7	32	20
json.unicode	32	58	76

Though the source code for xBNF and BNF is more compact, this does not necessarily translate into a smaller parsing table by virtue of the optimization strategies, such as used for combining equivalent states, used internally by Bison and Hammer. To gain an appreciation for this underlying internal compactness, it is useful to consider the number of *terminals*, *non-terminals*, and *states* used by the associated automaton *prior* to any post-processing that optimizes the tables using the *same* algorithms. Recall that, to a first order, the size of the parsing table (*Tsize*) is governed by the number of *states*, *terminals* (action table) and *non-terminals* (goto table) i.e. Tsize = States x (Terminals+Non-terminals). Table 3 shows the results for *json.unicode.*

Table 3: Internal compactness

	xBNF	BNF	HMR
Terminals	228	228	224
Non-terminals	49	45	69
States	346	342	689
Tsize	17,182	15,618	47,765

These results are consistent with a complete study of all the parsers in the repository. The results indicate that the xBNF pre-processing enhancements cause Bison to generate up to 10% larger parsing tables; obviously, the impact of this increase is largely absorbed by optimizations. Hammer tables, in comparison, are substantially larger than the others reflecting its current lack of maturity compared to Bison. A considerable saving was observed for ASCII parsers by virtue of the constrained set of characters (i.e. terminal symbols) employed in a grammar; Binary parsers can employ all 256-bit combinations available in a single byte, plus additional terminals to represent the value of zero (in Bison) and the end-of-input.

6. FPGA resource optimization

The FPGA resources -- Block RAM (BRAM), Flip Flops (FF) and Lookup Tables (LUT) -- needed for a parser is ultimately used to represent its *space-optimized parsing table*s, *pushdown automaton*, and the associated *stack* used during parsing. Table 4 shows the resource utilization associated with an Artix 200T -- for the *json.unicode* parser with a 2Kbyte stack – large enough to parse all of our most complex test vectors. The parser is compared with an *empty* parser that contains no parsing tables. Four versions of the empty parser are shown which vary only by stack size associated with the parser in increments of 2 Kbytes. Values in the table are actual number of resources with the percentage of the overall resources on the chip. Where the overall resource usage is less than 1% it is signified by the notation (~1%).

Table 4: FPGA resource utilization

Parser	BRAM	FF	LUT
json.unicode	16 (2%)	306 (~1%)	942 (~1%)
empty0	0 (0%)	167 (~1%)	533 (~1%)
empty2	2 (~1%)	263 (~1%)	588 (~1%)
empty4	4 (~1%)	263 (~1%)	589 (~1%)
empty8	8 (1%)	263 (~1%)	590 (~1%)

The results demonstrate that the most complex parser tested to date consumes only 2% of the available BRAM resources and an insignificant percentage of the other resources. The automaton alone consumes only 167 Flip Flops and 533 Lookup tables -- less than 1% of each resource. When present, the size of the stack affects only the BRAM resources used and is generally a small percentage of the overall chip resources. This study indicates that for complex parsers, only the smallest and least expensive Spartan-7 FPGA will ultimately be required for production parsers.

7. Latency and Bandwidth

Recall the most complex parser in the repository accepts valid JSON inputs with Unicode (*json.unicode*). The time to store, parse, and forward an Ethernet Frame is proportional to the length of the input that must be traversed by the parser state-machine, to detect either a valid or invalid input. This value could be immediate for short failure cases. Instead, we base analysis on randomly selected *valid* inputs, of various sizes, to force the parser to traverse the entire input and detect success through a variety of alternative paths in the state-machine. This provides an upper bound on latency and lower bound on bandwidth. The performance numbers are obtained directly by counting hardware clock cycles under co-simulation, where the actual circuit, running at 125MHz, is driven from test-vectors provided via a software testbench in High-Level Synthesis – only the cycles used by the actual circuit are registered and have proved to be highly accurate in a variety of prior experiments. Figures 3 and 4 show the measured Latency and Bandwidth curves for a variety of test inputs of differing length. Note that the largest parse is 1600 bytes -- the size of the largest Ethernet frame; messages longer than a single message were shown to be dominated by the byte-to-byte sequential nature of parsing and add linearly to the graphs.

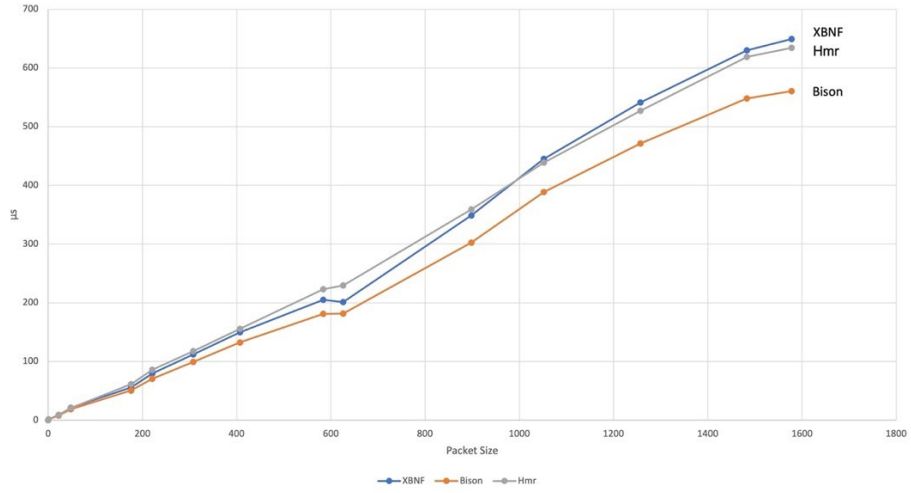

Figure 3: Parser Latency (microseconds)

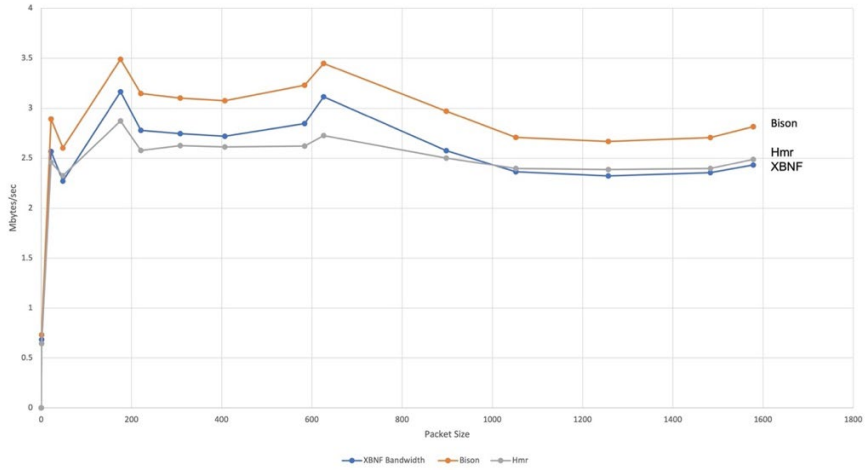

Figure 4: Parser Bandwidth (Mbytes/sec)

Figures 3 and 4 are useful for comparing the relative performance of the automation processes. Note that the random nature of the input drives, what is a relatively complex state machine, through a broad variety of differing state-sequences. In consequence, variability in the results, for example between 200 and 600 bytes is likely dependent on the sequences used in practice in the experiments. It is more important to observe that Bison, by virtue of the maturity of its internal optimization, consistently outperforms Hammer and XBNF in Latency and Bandwidth, but there are no dramatic unexpected deviations in performance inherent in the alternative formulations. The improvements in readability and compactness provided by XBNF, illustrated in Table 2, clearly comes at a cost of up to 15% in latency and bandwidth; its overall performance is comparable to Hammer. It is not yet clear if this overhead can be eliminated through alternative implementation techniques and represents the subject of ongoing research. Using linear approximations, general formulae characterizing the expected overall performance can be constructed as shown in Table 5, where *x* is the packet size.

Table 5: Linear approximations

Grammar	Latency L (microseconds)	Bandwidth BW (Megabytes/sec)
xBNF	L = 0.4225x - 14.022	BW = 0.0005x + 1.9878
Bison	L = 0.3662x - 10.607	BW = 0.0007x + 2.2239
Hammer	L = 0.4131x - 6.4267	BW = 0.0005x + 1.9467

Obviously, these approximations are inaccurate for very small messages, however, they are helpful in gauging practical expectations. The MAVLink network protocol is an unclassified protocol in common use for command and control of drones. It uses control messages that range in size between 12 and 280 bytes in length. Two of the most frequently used messages -- SETMODE and STATUS -- are 18 and 66 octets in length respectively. Table 6 shows the expected latency when, during normal operations, every valid MAVLink message is parsed to the end; additionally, the latency for a maximum sized message of 1600 bytes and the expected number of clock cycles per octet are calculated.

Table 6: Typical latency

Parser	12-byte SETMODE Latency	66-byte STATUS Latency	280-byte MAVLink Latency	Large Packet Latency (1600bytes)	Estimated cycles/octet (1600 bytes)
xBNF	< 1 µs	14 µs	104 µs	662 µs	51
Bison	< 1 µs	13.6 µs	92 µs	575 µs	45
Hammer	1 µs	21 µs	109 µs	655 µs	51

In summary, as one would expect, deep packet inspection requires significantly more time than simple checks on a packet header, such as those required to validate the presence or absence of particular values (e.g., protocol flags, IP addresses, etc.). Latency currently accrues at approximately *50 clock cycles per octet*. At 125Mhz, typical latencies for frequently used, valid MAVLink messages are between 1 and 20µs, with a maximum of 110µs; obviously, invalid messages would be detected more quickly. Though the link may operate at Gigabit rates, parsing causes bandwidth to drop -- by virtue of the need to store, sequentially consider every byte in turn, and subsequently forward individual Ethernet frames -- generally varying between 2.25 and 3.5 Mb/sec.

8. In conclusion

The technology described here occupies a middle ground between the fully air-gapped and fully connected systems and trades a marginal increase in risk for the opportunity to observe, optimize, and interact with networked systems remotely. This paper has discussed an automated process for developing custom hardware parsers using formal grammars. In the extreme, this allows every byte of every message – including payload data -- to be inspected and validated against a formal specification. Obviously, parsing is inherently a byte-by-byte sequential process that drives detection through a potentially large state-machine – consequently, it is not surprising that latency varies linearly with message length and that bandwidth is limited by the store-and-forward nature of Ethernet frames.

The performance figures presented here represent the baseline for an initial prototype and utilize only the resource optimizations described. Many additional sources of performance enhancement exist that can be expected to improve the technology as it matures: running parsers at higher clock rates; using extra resources to parse multiple frames concurrently; widening input and output data streams; using data-flow optimizations within the parsing pipeline rather than packet-by-packet store and forward etc. Moreover, the technology need

not be used in isolation, or, in an all-or-nothing effort to validate every byte: only a subset of message traffic considered "unsafe" might be considered in conjunction with other, higher-performance tests. For example, by quickly checking an IP address or port, the decision might be made to perform whole packet analysis on only a subset of traffic from specific locations. The low resource requirements achieved here offer not only the ability to produce further optimizations, but also to include other algorithms concurrently with parsing. For example, they have already been combined with AES encryption and decryption blocks, as well as IPSec packet encapsulation techniques, both of which are able to operate at GigE line rates.

References

Adams,C. (2012) "HUMS Technology", Aviation Today (aviationtoday.com/2012/05/01/hums-technology).

Aho, A.V., Sethi, R., Ullman,. J.D. (1988) *Compilers*, Addison Wesley.

Barham P, et al (2003) "Xen and the Art of Virtualization", In: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp 164–177.

Bratus,S. et al., (2016) "Implementing a vertically hardened DNP3 control stack for power applications", ICSS'16 ACM Proceedings of the 2nd Annual Industrial Control System Security Workshop, pp 1-9. (c.f. https://gitlab.special-circumstanc.es/hammer/hammer).

Carter, M.C. (2003) "Post Implementation CBM Benefit Analysis", Annual Conference of the Prognostics and Health Management Society.

Dahlstrom, J., and Taylor, S. (2018) "System-on-Chip Data Security Appliance and Methods of Operating the Same", U.S. Patent 10,148,761.

Dahlstrom, J., and Taylor, S. (Aug 2019) "System-on-Chip Data Security Appliance and Methods of Operating the Same", U.S. Patent 10,389,817.

Dahlstrom, J., and Taylor, S. (Oct 2019) "Endpoints for Performing Distributed Sensing and Control and Methods of Operating the Same", US Patent 10,440,121.

Dahlstrom, J., et al, (Nov 2019) "Hardening Containers for Cross-Domain Applications", In Proceeding of MILCOM 2019, Norfolk, VA, USA, pp. 1-6.

Dahlstrom, J., and Taylor, S. (2020) "System-On-Chip Data Security Appliance Encryption Device and Methods of Operating the Same" U.S. Patent 10,616,344.

Dahlstrom, J., and Taylor, S. (2021) "Hardware Turnstile", U.S. Patent 10,938,913.

Kushner, D. (2013) "The Real Story of STUXNET", IEEE Spectrum.

NBAR (2021) -- The National Bureau of Asian Research, The IP Commission Report, 2013, updated 2021.

Popuri, S.K., "Understanding C parsers generated by GNU Bison", Sept 2006 (cs.uic.edu/~spopuri/cparser.htm).

Shanthakumaran, P. (2010) "Usage Based Fatigue Damage Calculation for AH-64 Apache Dynamic Components", The American Helicopter Society 66th Annual Forum, Phoenix, Arizona.