Hardware Sequence Combinators

Stephen Taylor[1] and Gunnar Pope[2]
[1]Thayer School of Engineering, Dartmouth College, Hanover NH, USA
[2]BitStory.AI, LLC, USA

stephen.taylor@dartmouth.edu

Abstract: Recent advances in formal methods for constructing parsers have employed the notion of *combinators*: primitive elemental parsers with well-defined methods for combining them in sequences or through choice. This paper explores the subtleties associated with leveraging *sequence combinators* to produce compact, custom hardware traffic validators. This involves a fully automated process that takes as input a formal grammar specifying message formats and produces a parsing circuit capable of validating traffic headers and payload content. The resulting circuit is deployed through network *guard* appliances that employ Field Programmable Gate Array (FPGA) devices, or alternatively, within the on-chip FPGA associated with System-on-Chip (SoC) devices, such as the Xilinx UltraScale MPSoC. Each guard appliance acts as a hidden "bump-in-the-wire" that either forwards or drops individual packets based on the message parsing outcome, thereby hardening network segments against zero-day attacks and persistent implants. Guards may operate on a wide variety traffic protocols and formats including TCP/IP, CAN/J1939, or MIL-STD-1553. The central step in parser construction is to build a collection of standard shift/reduce parsing tables that can be employed by a push-down automata to check each byte in a message. Typically, these tables are sparse, resulting in excessive use of FPGA circuit resources to represent them. By leveraging sequence combinators, along with other optimizations, we have been able to produce highly compact representations that can reduce table size by up to 95% for non-trivial grammars. Depending on the grammar, this translates directly into FPGA resource reductions. The reductions now make it viable to implement complex parsers on small, inexpensive FPGA's, or alternatively combine parsers with encryption and encapsulation to enhance guard capabilities.

Keywords: Parsing, LALR grammar, Traffic validation, FPGA, Bison, Hammer

1. Background

Air, space, and ground vehicles, like many other embedded systems, 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, 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, the systems are intermittently connected within operational installations to provide mission parameters, or to effect maintenance and upgrades. Though network boundary protections are often in place during these activities, there are many threat vectors that circumvent these protections and airgaps in general; these include unintended network connections, insiders, supply chain interdiction, zero-day exploits, and persistent implants (Kushner 2013). Consequently, we seek embedded devices that may disrupt the attack surface and harden existing systems.

In a previous paper (Dahlstrom et al. 2022), we described a fully automated process that builds a custom hardware traffic validator from a formal language *grammar* describing the message header and payload format. The validators can be deployed in a variety of network appliances, such as the small footprint *guard* pictured in Figure 1, that combines traffic validation with hardware encryption and encapsulation (Dahlstrom and Taylor Aug 2019, Oct 2019, 2020).

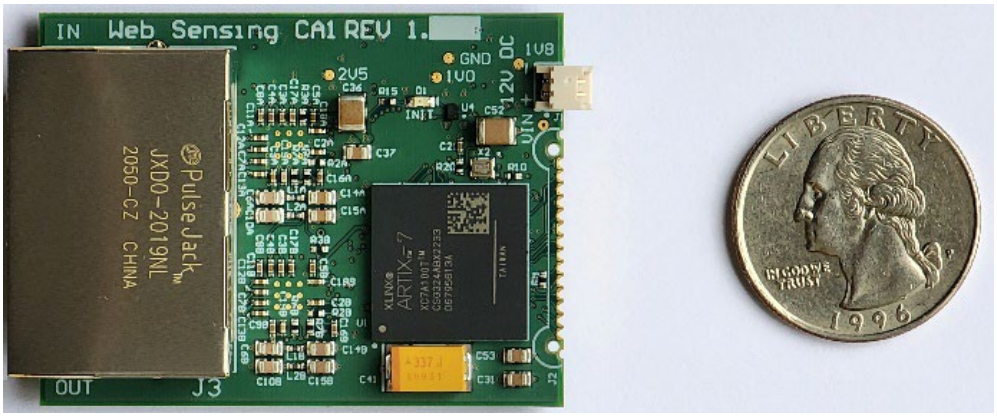

Figure 1: Small Footprint Guard (Size: 2"x1.5"x0.7", Weight: 0.7oz, Power: ~2W)

At their heart each guard consists, by design, of a ***single*** Field Programmable Gate Array (FPGA) lying between industry standard buses and network connections. The guard 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; for example, the guard shown in Figure 1 forms a bridge between two Gigabit Ethernet (GigE) interfaces using the dual connector on the left of the device. Guards 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 thereby mitigating reverse engineering; No software is present in the device, mitigating persistent software implants and zero-day attacks (Taylor 2018).

2. Sequence Combinators

Traffic validation is, in essence, a *parsing* process in which a string of symbols or values – corresponding to an Ethernet Frame – is analyzed for conformity with respect to a formal language grammar (Grune and Jacobs 2007). Classical *parser generators*, such as Bison (Levine 2009), are tools that automatically construct a software parser implementation from a given grammar. In recent years there has been considerable interest in expanding the expressiveness of parsers, grounded in formal methods, to cope with increasingly complex data formats. Some tools achieve this by leveraging a Domain-Specific Language (DSL) to describe the format (e.g. Levillain 2014, Sommer, Amann and Hall 2016); while others employ *combinators*: primitive elemental parsers with well-defined methods for combining them into more expressive parsers (Hutton 1992, Chifflier and Couprie 2015, Couprie 2015, Bogk and Schöpl 2014, Krishnaswami and Yallop 2019, Bangert and Zeldovich 2019). To achieve this, combinators are often provided in a library that allows different parsers to be constructed by invoking combinator functions using some convenient programming language. For example, the Hammer combinator library (Bratus 2016) allows parsers to be constructed via a C-program.

Unlike this body of research, here we are concerned with what mechanisms admit to the efficient realization of parsers in *hardware* using FPGA's; in particular, if the combinator notion can be used directly to affect an improvement in performance or FPGA resource utilization. Consequently, to validate traffic, the guard shown in Figure 1 implements a standard Left-to-Right Look Ahead (LALR) *parser* that operates on Ethernet frames, one-byte at a time, as they pass through the device. The parser checks that each input byte is valid with respect to a grammar expressed in either Bison Backus-Naur Form (BNF) or Hammer. For example, the following grammar, written in BNF describes, a digit used in parsing JavaScript Object Notation (JSON) (Bray 2017) numbers:

digit : '0' | onenine ;

onenine : '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

Reading the grammar: a *digit* is either the terminal symbol '0' in ASCII ***or*** (c.f. '|') is parsed through an alternative rule designated using the non-terminal symbol *onenine*. The *onenine* rule designates nine alternatives, the symbols '1' through '9', explicitly listing all the alternatives. The equivalent Hammer representation is:

H_RULE(zero, h_ch('0'));

H_RULE(digit, h_choice(zero, onenine, NULL));

H_RULE(onenine, h_ch_range(0x31,0x39));

Both these grammars can be transformed into a collection of shift/reduce *parsing tables* that drive a standard push-down automaton that operates with a single stack. Note, however, a subtle conceptual difference between the phrasing: the *h_ch_range() sequence combinator* need simply test that the input is between '1' and '9' rather than include parsing states that test against every potential value in the range. It is this key observation that we exploit here in producing highly compact hardware implementations on FPGA's. To make this concept explicit across both inputs forms, we have extended BNF using a pre-processor – xBNF - that allows the character sequences to be stated explicitly. The equivalent xBNF rules are:

digit : '0' | onenine ;

onenine : ['1' - '9'] ;

3. Automated Parser Construction and Optimization

Figure 2 shows the automation process used to transform a grammar into a hardware parser-plugin that can be deployed on a guard such as that shown in Figure 1. For grammars written in Backus-Naur Form (BNF), the Bison

parser generator tool (Levine 2009) is used to produce a collection of shift/reduce parsing tables, expressed in a machine-readable XML format, describing how the associated push-down automaton should parse the input (leftmost brown path). In Hammer, an equivalent parser can be defined using pre-existing combinators in C linked with the Hammer library (alternate blue 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 parsing tables into a C-header file (pda.h) whose primary content is a large two-dimensional C-array. Rows in the array correspond to *states* in the push-down 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 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.

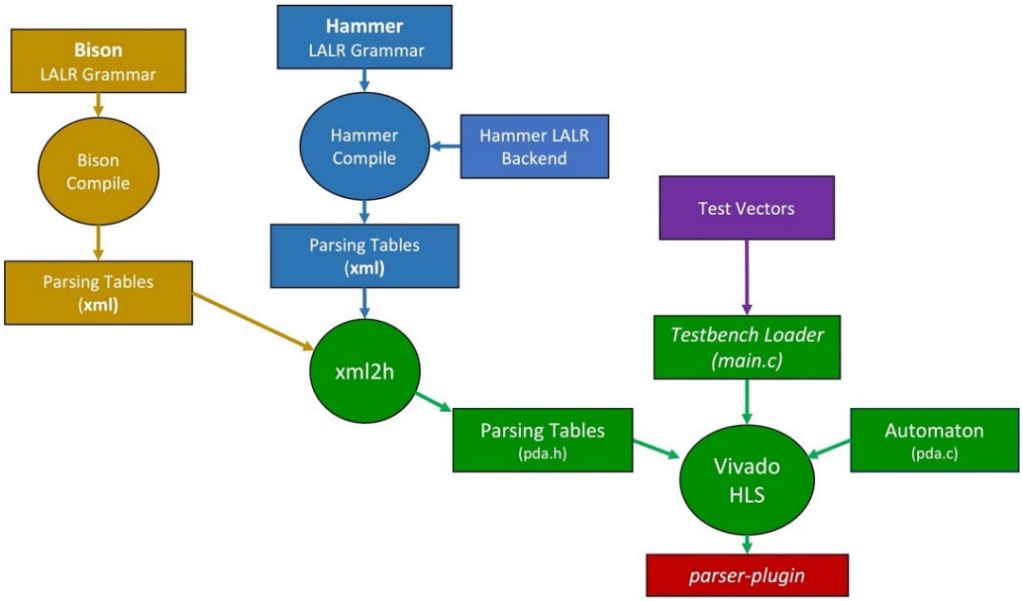

Figure 2: Automated Parser Generation Process

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 a network appliance, such as that shown in Figure 1. The plugin is organized to connect with the communication interfaces on the board through industry standard AXI-streams.

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 *compiler*-style optimizations follow similar techniques to those employed internally by Bison and are described in detail online (Popuri, 20006) and in our previous work (Dahlstrom et al, 2022).

4. Combinator Optimization

We have developed an optimization strategy called *the Sequence Combinator* that leverages the concept described in Section 2, namely that when testing ranges of symbols, direct range calculations are used instead of parsing table entries. The optimization is inserted after parsing tables are generated in Figure 1, as shown in Figure 3 for the Bison path, and thus applies to both BNF and Hammer. The optimization digests parsing tables in XML format (gmr.xml), after compiler-style optimizations have been performed, and produces a compressed set of parsing tables, also in XML format, which are then translated into C format using *xml2h*.

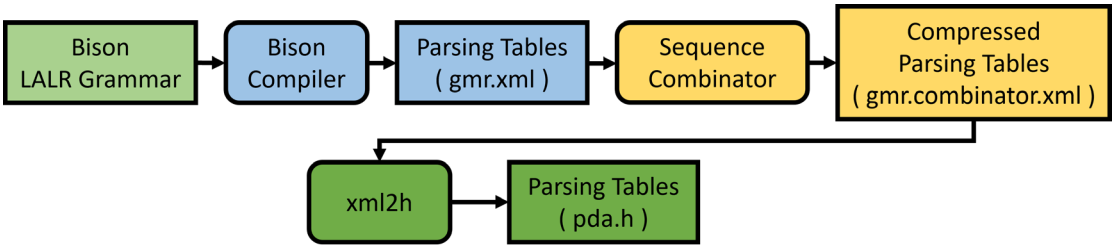

Figure 3: Sequence Combinator Optimization

The *Sequence Combinator* operates by combining sequences of two or more consecutive terminal symbols, $t \in \mathbb{R}^{2:N}$, into a single 'combinator' token, $t_c \in \mathbb{R}^1$, that represents the entire range of token values of t. The subtleties of the transformation are best described using a simple example of a grammar, G, defined in BNF format that accepts any lowercase letter ['a'-'z'] or the string, "foo":

G : letter | string ;

letter : 'a' | 'b' | … | 'z' ; // (26 rules)

string : 'f' 'o' 'o' ;

Notice that the resulting parser will need to distinguish when parsing an 'f' or 'o' which of the letter or string rules are to be reduced by virtue of context.

An un-optimized parser will compile this grammar into a set of parser tables requiring: 2 rules for the non-terminal symbol "G", one rule for each character in the set ['a'-'z'] (26 total), and one rule for the sequence of bytes in "foo". The *Sequence Combinator* optimizes this code by combining consecutive sequences of terminal tokens having only one dependent rule (unlike the terminals 'f' or 'o', which exist on the right-hand-side of both non-terminal symbols "letter" and "string"), such that the new logic becomes:

G : letter | string ;

letter : $t_{a,e}$ | 'f' | $t_{g,n}$ | 'o' | $t_{p,z}$; // (5 rules)

string : 'f' 'o' 'o' ;

where $t_{a,e}$, $t_{g,n}$, $t_{p,z} \in \mathbb{R}^1$and represent the tokens for ranges ['a'-'e'], ['g'-'n'], and ['p'-'z'], respectively. This optimization reduces the number of rules needed to represent the 'letter' symbol from 26 rules down to 5 rules and the total number of states in the PDA is reduced from 34 states down to 13. The optimization algorithm is described in pseudocode in Figure 4.

INPUT: gmr.xml file (e.g. PDA parsing tables)
OUTPUT: gmr.combinator.xml file
INITIALIZATION:
0. read file gmr.xml
BEGIN
1. discover symbol ranges
2. find dependent rules
3. find dependent states
4. add new symbols
5. remove unused terminals
6. renumber token-value of symbols
7. find old rules
8. remove old rules
9. create new range rules
10. renumber rules
11. renumber the states
12. write file 'gmr.combinator.xml'
END

Figure 4: Sequence Combinator Optimization Algorithm

The optimization algorithm, shown in Figure 4, is initialized (Step 0) by reading in the Parsing Tables defined in XML format. This file contains the rules, symbols, and states required to implement the grammar parser using a PDA. Step 1 iterates over all rules in the grammar and discovers sets of continuous symbols that map to a single terminal symbol (e.g., 'a'→ "letter", 'b'→ "letter", etc.). Steps 2 and 3 create lists of rules and states that depend on symbols within a given range which will be compressed. For example, this stage of the algorithm discovers that the non-terminal symbols "letter" and "string" are both dependent upon the terminal symbols 'f' and 'o'. Therefore, these symbols must be excluded from the sequence compression algorithm so that each of the terminal symbols can be represented explicitly in the PDA state machine. Step 4 creates new terminal symbols for the compressed ranges (e.g.['a'-'e'] → $t_{a,e}$). Step 5 removes the unused terminal symbols that have been removed by the compression. Step 6 renumbers the token-values for the reduced set of terminal symbols. Step 7-10 are used to find the old rules, remove them, and then renumber remaining ruleset. Step 11 iterates over all states within the PDA and renumbers the old states and references to old rules – creating the finalized grammar object. Lastly, Step 12 writes the new grammar object to file, 'gmr.combinator.xml' where it can be passed to the xml2h.c script to translate the grammar definitions into a PDA.

An additional modification is required within the PDA implementation (pda.c) to map bytes read from the input to their compressed range token (e.g. 'b' → $t_{a,e}$). This logic could be trivially implemented at runtime by mapping the input byte, B, to a compressed token value if it lies within the min/max bounds (for example: 'a'≤ B ≤ 'e' → $t_{a,e}$) but this would incur additional processing during runtime and limit the bandwidth of the PDA. In our research, we discovered that the mapping of symbols to their respective compressed tokens can be achieved *without any additional computational overhead at runtime* by encoding the symbol mapping into the lookup table at the front end of the PDA. For example, let B represent a byte read from the input and $0 \leq B \leq 255$. The original grammar, G, has 28 terminal characters ('a'-'z', 0, and $end) in the Parser Tables. A lookup table, named "tokens", is used to map B ∈ $\mathbb{R}^{256}$ to a non-terminal symbol, S ∈ $\mathbb{R}^{28}$, which is implemented using an array in C, simply as:

S = tokens[B];

0 = min(tokens)

27 = max(tokens)

The Sequence Combinator compresses grammar, G, into only 7 tokens ({ $t_{a,e}$, $t_{g,n}$, $t_{p,z}$, 'f', 'o', 0, $end}) and the new xml2h script modifies the tokens array to encode the compressed mapping in the following manner:

$S_{a,e}$ = tokens[B];

for B={'a', 'b', 'c', 'd', 'e'}

0 = min(tokens)

6 = max(tokens)

The result of this encoding is that the data read from the input is automatically mapped to the compressed symbol representation, during runtime, without requiring any additional logic or processing overhead.

5. Datatype Optimization

The HLS process shown in Figure 2 is generally performed on standard C code using standard fixed width datatypes. For example, consider the following constant array definition describing the left-hand size of rules in a grammar:

uint16_t lhs[4] = { 1, 11, 2, 13 };

Notice that no value in the array requires more than 4 bits to represent. HLS provides a variable width data specification that makes it possible to use an alternative hardware representation for all data structures used by the push-down automaton. Using this feature, we transform all arrays based on their content and size the data types appropriately:

ap_uint<4> lhs[4] = { 1, 11, 2, 13 };

The result of this transformation is that hardware busses and registers used to represent the *lhs* array in hardware are reduced from 16 bits to 4 bits wide and therefore consume less resources.

6. Resources and Performance

Table 1 reviews the C-structured parsing table data structure sizes prior to HLS, associated with various optimization levels for a full JSON parser and provide insight into the complexity reductions achieved. Notice that these results indicate a dramatic reduction in the size of the structures from a baseline without optimization (-O0), adding compiler-style optimizations that remove sparse areas of the tables (-O1, -O2), and then the subsequent addition of hardware combinators (-O10, -O11, -O12) to each of these baselines. The use of combinators in isolation (-O0 to -O10) reduces tables sizes by 81%; when combined with standard compiler optimizations (-O0 to -12), table sizes are reduced by 95%. Obviously, the Datatype optimization has no impact until synthesis is implemented as it relies on HLS capabilities.

Table 1: JSON Parsing Table Sizes impacted by Combinators

OPT	-O0	-O1	-O2	-O10	-O11	-O12
Terminals	227	227	227	56	56	56
Non-Terminals	48	48	48	48	48	48
Rules	302	302	302	131	131	131
States	346	346	346	175	175	175
Tables (bytes)	192,020	32,552	28,688	37,436	12,716	8,852
% Abs Diff	0.0	0.83	0.85	**0.81**	0.93	**0.95**
% Rel Diff	0.0	0.83	0.12	0.81	0.66	0.3

Tables 2 and 3 shows the actual FPGA resource utilization and average cycles per byte, over a range of representative tests, for two non-trivial parsers used for validating JSON (Bray 2017) and URI (Berners-Lee, Fielding, and Masinter 2005) formats respectively. The first line provides the baseline with no optimization; the second shows with Compiler-style optimizations (-O1 and -O2 in Table 1); the other lines show the cumulative impact of Combinator and Datatype optimizations described in Sections 4 and 5 respectively. Resource counts are for Block RAM (BRAM), Lookup Tables (LUT), Flip Flops (FF), and Digital Signal Processor (DSP) cells. Bandwidth is based on the clock cycles per byte using a 125MHz clock. For comparison, the last line in each table shows the percentage of the Artix 100T resources used with all optimizations applied; This FPGA is used on the small footprint device shown in Figure 1 and has 270 BRAM, 63,400 LUT, 126,800 FF and 240 DSP resources.

Table 2: JSON Parser Resources and Performance

OPT	BRAM	LUT	FF	DSP	Avg Cy/Byte	BW (Mb/Sec)
None	40.5	269	187	1	63	15.9
Compiler	8.5	312	251	1	68	14.7
Combinator	4	305	256	0	57	17.5
Datatype	4	310	218	0	57	17.5
% Artix 100T	***1.4%***	***0.5%***	***0.2%***	-		

Table 3: URI Parser Resources and Performance

OPT	BRAM	LUT	FF	DSP	Avg Cy/Byte	BW (Mb/Sec)
None	7.5	3066	2230	1	58	17.2
Compiler	2	3862	2322	1	63	15.9
Combinator	1.5	3842	2315	0	53	18.9
Datatype	1.5	1284	726	0	53	18.9
% Artix 100T	***0.5%***	***2%***	***0.6%***	-		

With all the optimizations applied, both parsers fit easily within the ARTIX 100T using less than 5% of the resources across the board. This high degree of compactness allows for considerable latitude in the complexity of grammars that can be handled. Consequently, it is possible to produce guards in two flavours: low-cost standalone validators that employ smaller pin-compatible FPGA's, or sophisticated guards that combine the functionality of validation with encryption and encapsulation.

The two parsers show a marked difference in the way HLS allocates resources. The optimizations produce a 90% reduction in BRAM resources on the JSON parser, and an 80% reduction on the URI parser. However, these are lower bounds since block RAM is allocated in slabs and the utilization in each slab cannot be easily determined. In the JSON parser, BRAM allocation leaves little residual logic to be realized with LUT and FF resources, consequently we see only marginal improvements in these areas. However, when substantial levels of these resources are used, as is the case with the URI parser, we see a 67% reduction in LUT and a 69% reduction in FF usage.

In both parsers, bandwidth improves by approximately 10% but remains at less than 20Mb/sec – 50 times slower than Gigabit line rate – however, this is sufficient for real-time analysis of MIL-STD-1553 and CAN bus traffic. Recall that parsing is an inherently sequential activity that considers one byte at a time. Consequently, for normal, good traffic every byte in a payload must be parsed from beginning to end to determine that it is valid. Invalid packets are an outlier in terms of performance measurement and were consequently not considered in the test corpus. The JSON and URI grammars are relatively complex compared to tactical traffic protocols and buffering is capable of handling bursty traffic at higher rates, however, performance rather than resource related optimizations must be devised to improve bandwidth further.

The performance figures presented here represent a second level of space optimization over the initial prototype described in a previous paper and produced unexpectedly compact implementations. In part this is appears to be because all the core work in parsing input is conducted at the leaves of the parsing tree where terminal symbols are resolved; consequently, the sequence combinator is frequently applied with considerable impact.

7. In Conclusion

The guard technology described here occupies a middle ground between the fully air-gapped and fully connected embedded systems: It trades a marginal increase in risk for the opportunity to observe, optimize, and interact with networked systems remotely through a guard. This paper has discussed improved automated processes that yield highly compact, custom hardware parsers directly from formal language 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 bandwidth is limited by the length of input and the complexity of the grammar. Many additional sources of bandwidth enhancement exist that can be expected to improve the technology further as it matures: running parsers at higher clock rates; using extra resources to parse multiple frames or components of frames concurrently; widening input and output data streams; using data-flow optimizations within the parsing pipeline 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 independently.

Distribution

This research is supported by the Defense Advanced Research Projects Agency (DARPA). 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).**

References

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

Bangert, J. and Zeldovich, N. (2014), "Nail: A Practical Interface Generator for Data Formats", Proceedings of the 2014 IEEE Security and Privacy Workshops, May 2014, pp 158-166.

Berners-Lee, T. Fielding, R., and Masinter, L. (2005), "Uniform Resource Identifier (URI): Generic Syntax", IETF RFC 3986, January 2005.

Bogk, A. and Schöpl M. (2014), "The Pitfalls of Protocol Design: Attempting to Write a Formally Verified PDF Parser," *2014 IEEE Security and Privacy Workshops*, San Jose, CA, USA, 2014, pp. 198-203.

Bray, T. (Ed.), (2017), "The Javascript Object Notation (JSON) Data Interchange Format", IETF RFC 8259 December 2017.

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).

Chifflier, P. and Couprie, G. (2017), "Writing parsers like it is 2017", 2017 IEEE Symposium on Security and Privacy Workshops (SPW), San Jose, May 2017, pp 80-92.

Couprie, G. (2015), "Nom, A byte oriented, streaming, zero copy, parser combinators library in rust", In 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015, pp 142–148.

Dahlstrom, J., Guzman, B., Baker, El, and Taylor, S. (2022), "Automatic Construction of Hardware Traffic Validators", European Conference on Cyber Warfare and Security (ECCWS22), June 2022, pp 60-69.

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

Grune, D. and Jacobs (2007), C.J.H., "Parsing Techniques – A Practical Guide", Springer Monographs in Computer Science, 2007.

Hutton, G. (1992), "Higher-Order Functions for Parsing". Journal of Functional Programming 2, 3 (1992), pp 323-343.

Krishnaswami, N.R. and Yallop, J. (2019), "A typed, algebraic approach to parsing", Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2019, pp 379-393.

Kushner, D. (2013), "The Real Story of stuxnet", IEEE Spectrum, vol 50, no. 3, Mar. 2013, pp 48-53.

Levillain, O. (2014), "Parsifal: A Pragmatic Solution to the Binary Parsing Problem." In 35. IEEE Security and Privacy Workshops, SPW (LangSec) 2014, San Jose, CA, USA, pages 191–197, May 2014, pp 191-197.

Levine, J. (2009), "Flex & Bison", O'Reilly Media, Inc. ISBN: 978-0-596-15597-1, 25 July, 2009.

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

Sommer, R., Amann, J., and Hall, S. (2016), "Spicy: a unified deep packet inspection framework for safely dissecting all your data. In Proceedings of the 32nd Annual Conference on Computer Security Applications, ACM 2016, Los Angeles, CA, USA, Dec. 2016, pp 558-569.

Taylor, S. (2018), "Protecting Embedded Systems from Zero-Day Attacks", IEEE NAECON 2018, July 23-26, pp 165-168.