5-21-2020

On Session Languages

Prashant Anantharaman
Dartmouth College

Sean W. Smith
Dartmouth College

Dartmouth Computer Science Technical Report TR2020-881

On Session Languages

Prashant Anantharaman
Department of Computer Science
Dartmouth College
Hanover, USA
pa@cs.dartmouth.edu

Sean W. Smith
Department of Computer Science
Dartmouth College
Hanover, USA
sws@cs.dartmouth.edu

May 21, 2020

Abstract

The LangSec approach defends against crafted input attacks by defining a formal language specifying correct inputs and building a parser that decides that language. However, each successive input is not necessarily in the same basic language—e.g., most communication protocols use formats that depend on values previously received, or on some other additional context. When we try to use LangSec in these real-world scenarios, most parsers we write need additional mechanisms to change the recognized language as the execution progresses.

This paper discusses approaches researchers have previously taken to build parsers for such protocols and provides formal descriptions of new sets of languages that could be considered to be a sequence of languages, instead of a single language describing an entire protocol—thus bringing LangSec theory closer to practice.

1 Introduction

LangSec is motivated by the real-world problem of crafted input attacks. Too often, software which receives input makes implicit assumptions about the structure and content of that input, but acts on the input without fully checking whether the input actually satisfies these assumptions. To address this problem, the LangSec approach uses the tools of formal language theory:

- to precisely define the set ("language") of expected inputs,
- to build a recognizer (a.k.a. parser) for that language,
- and then to insert that recognizer between the source of the input and the software that consumes the input.

This basic LangSec approach itself implicitly assumes that the language accepted by some software module stays the same over time. In communication protocols, the grammar to be accepted at any given time may depend on the state machine of the protocol. Similarly, file formats get altered over time since specifications are rewritten. In reality, the "correct" language may vary, This paper explores some issues that arise when we try to extend LangSec to handle such real-world scenarios.

Section 2 opens by reviewing tools from formal language theory. Section 3 discusses languages that change over system execution. Section 4 discusses some initial approaches from prior work. Section 5 considers some formal language extensions to handle the general case. Section 6 poses some questions raised by this structure. Section 7 concludes with ideas for future directions.

2 Tools

2.1 Basics

We quickly review the basics of formal language theory.

A *language* is a set of strings over some finite alphabet Σ.

One way to specify a language is with a *grammar* G:

- a set of variables V disjoint from Σ,
- a specific *start* variable $S \in V$,
- and, a set of production rules, each showing how something on the left-hand side can be replaced by something on the right-hand side.

A string $s \in \Sigma^*$ is in the language corresponding to this grammar exactly when we can start with S, repeatedly apply production rules, and end up with s.

We group grammars into classes based on the format of the production rules. In a computer scientist's view of the Chomsky hierarchy, the typical grammar classes of interest are *regular expressions* and their proper superset, *context-free grammars* (with occasional nods towards *context-sensitive grammars*, a proper superset of CFGs).

Another way to specify a language is with an abstract machine that says "yes" when given a string in that language. Theory gives us two flavors:

- a *decider*, which says "yes" exactly when a string is in the language and "no" if not.
- a *recognizer*, which says "yes" exactly when a string is in the language, but may give no answer if not.

Finite state machines (FSMs) are a basic class of deciders. An FSM has a finite set Q of states (with a distinguished start state and one or more accepting states), and a transition function $\delta : Q \times \Sigma \longrightarrow Q$. When an FSM can start in the start state, take transitions according to the symbols in an input string, and end in accepting state, then it accepts that input. As is well known, FSMs decide exactly the languages specified by regular expressions. If we add a stack to an FSM, we obtain a *pushdown automaton*; these recognize the CFGs but decide only a proper subset (e.g., the ones recognized by a deterministic pushdown automata). Computer scientists then typically jump to full Turing machines, which recognize the computable languages (but decide only a proper subset of them).

Other Language Classes of Interest. Although regular expressions and CFGs get the most attention in typical computer science curricula, other language classes exist that may be of practical interest to those exploring defense and analysis of software. The area of compiler design and construction identifies various subclasses of CFGs (such as *LR* and *LL(1)*) which can be easier to parse automatically. More

recent work on parsing has identified *parsing expression grammars (PEGs)* (e.g., [8]), which can be easy to parse unambiguously and also interestingly contain languages beyond CFGs (although it is also currently conjectured that CFGs contain languages that are not PEGs). Another class of potential interest are *visibly pushdown languages (VPLs)* (e.g., [2]), which are those accepted by a PDA restricted to pushing on certain input symbols and popping on certain other input symbols. VPLs lie strictly between the regular languages and CFGs, and some interesting predicate testing problems undecidable for CFGs become decidable for VPLs.

2.2 Other Things in the Automata Stable

Transducers. Chapter 6 in Alagar and Periyasamy [1] provides a deeper treatment of FSMs than what might be found in typical undergraduate textbook (e.g., [12]). This treatment includes formal discussion of *transducers*—FSMs that give "output" while processing an input string—and name two particular flavors: *Mealy machines*, where outputs happen on transitions, and *Moore machines*, where outputs happen in states.

Extended FSMs. Chapter 7 then gives a nice discussion of *extended FSMs*, albeit marred by some bugs (e.g., in the bounded buffer example, p 107). Basically, an EFSM has three new extensions:

- it can have internal variables,
- transitions can have guards (e.g., predicates over these variables) and actions (changing the values of variables),
- and states can be *complex*: meaning the state itself is a call to another FSM.

Register Automata. *Register automata* (defined at `http://automata.cs.ru.nl/Syntax/Register` and many other places) are a special case of Mealy EFSMs, where variables and actions on them are limited to some simple register operations.

Definition 1. A register automata (RA) is represented as a 6 tuple (R, Q, q_0, v_0, F, Δ) [6]. R is a finite set of registers, Q is a finite set of states, $q_0 \subseteq Q$ is the start state, v_0 is the initial assignment of the registers in R. The transitions Δ are written as $p \xrightarrow{\varphi/\ell} q$, where p is the initial state, q is the next state, the transition occurs if the input $x \in [\varphi]$, and the input registers in ℓ are satisfied. The output of the transition is written to the output register in ℓ.

Hower et al. [10] showed how to infer register automata from control flow in an infinite domain and demonstrated that this approach outperforms the L* learning algorithm. Cassel et al. [5] proposed a more succinct, canonical form of representing register automata.

Symbolic Register Automata (SRA) are register automata, where the alphabets are given using Boolean algebra that may have an infinite domain, and the transitions are defined using first-order predicates. SRAs are strictly more expressive than RAs, and are extremely useful when defining automata for alphabets such as the UTF-16 specification. With UTF-16, there would be 2^{16} transitions out of every state, and using first order predicates collapse those transitions to one relation.

D'Antoni et al. [6] compare the performance of SRAs to Symbolic Finita Automata (SFAs) (SFAs are DFAs with the alphabets given using Boolean algebra and transitions in first-order logic) by creating additional states for all the possible values registers could take. For a lot of their experiments, the corresponding SFAs did not fit in memory. They concluded that SRAs are a succinct model to represent state transitions.

Procedural Automata. Frohme and Steffen [9] define systems of *procedural automata* (Definition 1 in that paper, with useful clarification on page 5 of its PDF). We have a set of FSMs, but some transitions can be labeled as calls to another FSM in this set. The paper notes that any language accepted by a system of procedural automata is a CFG. These seem to be EFSMs, without the guards, variables, and actions.

2.3 Session Types

Somewhat orthogonal to the world of formal languages, the field of *session types* has evolved to provide "a theoretical foundation of communication-centered programming" [7]. The emergence of service-oriented programming in the real world gave rise to a need to formally describe the communications between parties carrying out some distributed computation. The literature often uses the example of a user, an ATM, and a bank as a model: each stage in the interaction permits different types and formats of messages. This has been an active research field, developing extensions and features to handle various distribution scenarios, and integrating session typing into programming languages.

To the extent that the "language" of correct inputs for a software module may change based on communication, session types may provide some helpful tools. We plan to consider this further in future work.

3 Languages that Change

As Section 1 noted, in many real-world programs, the "correct" language of expected inputs can change over the course of execution. One basic example is when interaction with the software goes through distinct phases; the language of inputs in the "introductory handshake" phase may differ from the language during the "data exchange" phase. However, one may easily imagine variation that is much deeper; e.g., colleagues in the power grid have suggested that a relay should not just accept any properly formatted command, but rather should "know" that in the current state of the physical power system, only commands with specific parameters are reasonable.

This limitation suggests that characterizing the correct inputs of some software module requires considering a *series* of formal languages. As a rough approximation, we might imagine that there exists some partition

$$\Delta_0, \Delta_1, ...$$

of the software's execution into distinct sessions, and a sequence of languages

$$L_0, L_1, ...$$

such that at time $t \in \Delta_i$, the correct language is really L_i.

This thinking suggests two lines of inquiry.

- **Coarseness of Recognition.** If we defend an interface with a validator that recognizes some language L_V but the correct language at time t is really L_t, with $L_V \supsetneq L_t$, then our defense misses the mark by $L_V \setminus L_t$. Can we turn this into a meaningful metric of the quality of any given LangSec defense implementation, and evaluate the tradeoffs between that metric and other engineering costs? This thinking extends to the cases when the correct language L is constant over time, and also to when the validator's language L_V can evolve over time.
- **Languages that evolve.** Formal language theory (e.g., the Chomsky hierarchy and all that) gives us tools to describe a static language via a grammar, and a corresponding machine that recognizes this language. How do these standard tools extend to handle evolving languages? Can these extensions be useful?

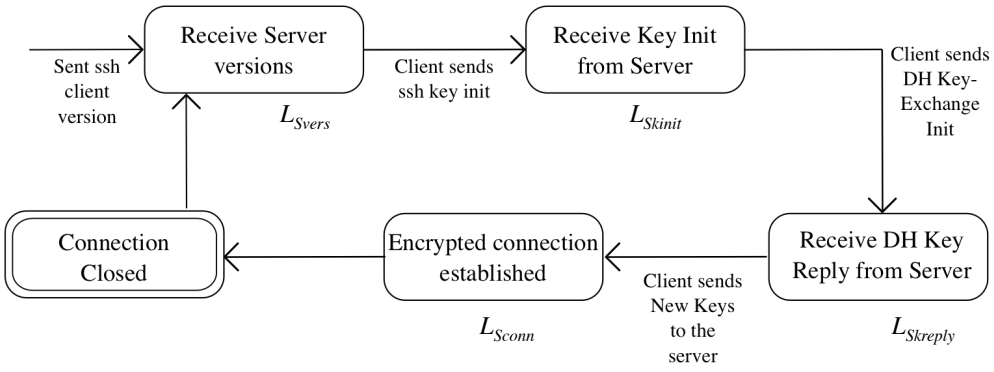

Figure 1: The state-specific parsers for two of the messages in OpenSSH annotated with the languages expected in each state.

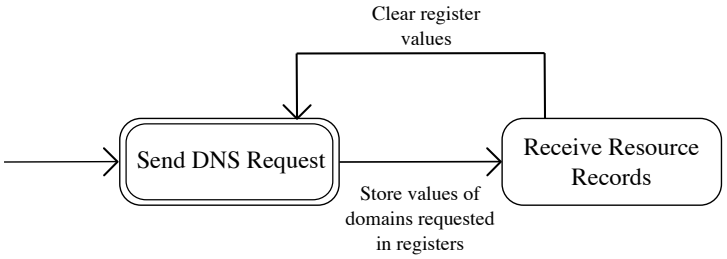

Figure 2: Our DNS parser changes the input filter based on client actions.

4 Initial Approaches

Network protocols are commonly specified via a *finite state machine (FSM)*; each state of the FSM represents a different phase of the protocol, with different communications involved.

Prior work building LangSec defenses for software that spoke such a protocol would build a separate grammar and validator for each phase, and used terms such as *session parsing* or *session languages* to describe this problem.

For example, Tse and Johnson [13] provide a toolset to construct a session parser by first building a DFA for the protocol, and allowing the developer to specify tests that messages must satisfy in order to transition between the states.

Anantharaman et al. [4, 3] described protocol state machines using FSMs, and implemented language parsers specific to the states of the FSMs. They demonstrated these language-based filters for the power protocol IEEE C37.118 as well as IoT protocols such as MQTT. In IEEE C37.118, a configuration frame decides how the consequent data frames are structured, and this dependency makes describing the protocol using FSMs useful. Poll et al. [11] noted that session languages are often defined as prose or as flow diagrams, but rarely as FSMs including all the error conditions.

Figure 1, from some of current work, shows the state-specific parsing stages for two of the messages in OpenSSH. When we, however, looked at the DNS protocol, (Figure 2) we saw an additional wrinkle: the response an input filter accepts must not only be formatted correctly—it must also match the domains the client previously asked.

5 Towards A General Model

5.1 New Languages

LangSec follows the basic dictum of using the simplest tool possible to describe an input language, to increase the chances of getting it right. However, one can make arguments for any of the tools in Section 2.1: not just FSMs, but also PDAs, PEGS, and perhaps even VPDAs.

Section 2.2 surveyed augmentations to FSMs from the literature. We posit that some of these augmentations may be useful for these other tools, and suggest exploring *extended X*, *register X*, *symbolic X*, and *systems of procedural X*, where *X* might be any of the items in Section 2.1.

5.2 Sequences of Languages

The general idea of the Chomsky hierarchy is that classes of grammars correspond to classes of recognizer machines. In some sense, a grammar represents the computation necessary to decide on what *sequence of input symbols* is correct. However, with session languages, we also have computation to decide what *sequence of grammars* is correct. In the approaches of Section 4, the sequence of grammars was determined by an FSM. However, many classes of grammar/machine pairs exist to describe the complexity of a sequence of elements over some set; both the one for the grammar-sequence and also the input-symbol-sequence could be from any of these classes.

Hence, we propose a way to characterize session languages based on this two-level process.

First, let us consider sequence of languages describing input received on the *external* surface of a software module.

Definition 2. Let $\mathcal{L}_{ext}$ be a finite set of languages over the alphabet Σ. Let $\mathcal{L}_{seq}$ be a language over the *alphabet* $\mathcal{L}_{ext}$. Define $\mathcal{L}_{seq}/\mathcal{L}_{ext}$ to be the set of strings s over Σ such that:

- There exists a sequence $L_0, ...L_k$ of elements, not necessarily distinct, from $\mathcal{L}_{ext}$
- where $L_0, ...L_k$ is a string in $\mathcal{L}_{seq}$
- and s can be written as the concatenation $s_0, .., s_k$, where each $s_i \in L_i$

(In the terminology above, $\mathcal{L}_{seq}$ denotes "sequence" and *Ltwo* denotes "external.")

The two-level structure might be seen as a generalization of EFSMs: where the overall construct as well as the "complex state" constructs may be FSMs, or other things from Section 2.1 (such as PDAs or PEGs), Section 2.2 (such as register automata), or even Section 5.1 (such as "register PDAs").

Having defined two-level languages, we can now put them into classes based on the complexity of their components.

Definition 3. Let C_1 and C_2 be classes of languages. Define C_1/C_2 to be the class of session languages

$$\{\mathcal{L}_{seq}/\mathcal{L}_{ext} \quad : \quad \mathcal{L}_{seq} \in C_1 \text{ and } \mathcal{L}_{ext} \subseteq C_2\}$$

The grammars/recognizers in $\mathcal{L}_{ext}$ play their standard role as specifying an input language. The new twist here is $\mathcal{L}_{seq}$: the structure/computation that describes the sequence of languages.

The examples from Section 4 (except DNS—see below) used FSMs as $\mathcal{L}_{seq}$. Our approach for OpenSSH (Figure 1) lived in FSM/FSM. We could describe $\mathcal{L}_{seq}$ as the regular expression $L_{Sver}L_{Skinit}L_{Skreply}(L_{Sconn})^*$.

However, in general, the top-level $\mathcal{L}_{seq}$ could be any construct from Section 2.1. For example, for an interaction that consists of n requests followed by n responses, we would need a CFG.

5.3 Sequences of Input Languages with Internal Action

In the standard LangSec model, a hardened parser defends software by filtering the inputs coming into it from outside. Hence, we focus on defining the valid inputs; the two-level definitions of Section 5.2 above focus on the complexity of describing sequences of those inputs.

However, the DNS case of Figure 2 introduces a explicit wrinkle (which was probably implicit all along): specifying a valid sequence of inputs in an execution of some software may also requires specifying *internal* actions of the software. We may not need to filter these internal actions to make sure they are properly formatted, but we do need to know what they are, in order to filter the external inputs. The sequence of valid external inputs for Figure 2 can be recognized by a register FSM—but it needs to set its registers from internal actions of the software. For example, we could build the register automata in Listing 1.

In the example, we model the state machine in Figure 2 as a register automata.[1] The two states are the `send_dns_request` and `receive_resource_records` states. The specify the input symbols and output symbols and the assignments to the output symbols in the transitions.

Hence, we revise the definitions of Section 5.2 to include internal actions.

Definition 4. Let $\mathcal{L}_{ext}$ and $\mathcal{L}_{int}$ be finite sets of languages over the disjoint alphabets Σ_{ext} and Σ_{int}, respectively. Let $\mathcal{L}_{seq}$ be a language over the *alphabet* $\mathcal{L}_{ext} \cup \mathcal{L}_{int}$. Define $\mathcal{L}_{seq}/\mathcal{L}_{ext}/\mathcal{L}_{int}$ to be the set of strings s over $\Sigma_{ext} \cup \Sigma_{int}$ such that:

- There exists a sequence $L_0, ...L_k$ of elements, not necessarily distinct, from $\mathcal{L}_{ext} \cup \mathcal{L}_{int}$
- where $L_0, ...L_k$ is a string in $\mathcal{L}_{seq}$
- and s can be written as the concatenation $s_0, .., s_k$, where each $s_i \in L_i$

Again, $\mathcal{L}_{seq}$ denotes "sequence" and *Ltwo* denotes "external." The new $\mathcal{L}_{int}$ describes the *internal* actions of the software; each string in an $\mathcal{L}_{int}$ language can then influence what the parser should expect in future external input. Trivially, $\mathcal{L}_{seq}/\mathcal{L}_{ext} = \mathcal{L}_{seq}/\mathcal{L}_{ext}/\emptyset$.

Definition 5. Let C_1, C_2, C_3 be classes of languages. Define $C_1/C_2/C_3$ to be the class of session languages

$$\{\mathcal{L}_{seq}/\mathcal{L}_{ext}/\mathcal{L}_{int} \quad : \quad \begin{array}{l} \mathcal{L}_{seq} \in C_1 \\ \text{and } \mathcal{L}_{ext} \subseteq C_2 \\ \text{and } \mathcal{L}_{int} \subseteq C_3\} \end{array}$$

```
1 <register-automaton>
2     <alphabet>
3         <inputs>
4             <symbol name="dns_req">
5                 <param name="domain_list" type="list">
```

[1] We use the syntax as described in `http://automata.cs.ru.nl/Syntax/Register#Registerautomatamodel` using XML syntax to describe register automata.

```
                </param>
                <param name="src_ip" type="ip_addr">
                </param>
                <param name="dst_ip" type="ip_addr">
                </param>
            </symbol>
            <symbol name="resource_records">
                <param name="domains" type="list">
                </param>
                <param name="src_ip" type="ip_addr">
                </param>
                <param name="dst_ip" type="ip_addr">
                </param>
                <param name="domain_list" type="list">
                </param>
            </symbol>
        </inputs>
        <outputs>
            <symbol name="requested_domain_values">
                <param name="domains" type="list">
                </param>
            </symbol>
        </outputs>
    </alphabet>
    <globals>
        <variable name="dnsserver" type="ip_addr">
            8.8.8.8
        </variable>
        <variable name="client" type="ip_addr">
            0.0.0.0
        </variable>
    </globals>
    <locations>
        <location name="send_dns_request" initial="true">
        </location>
        <location name="receive_resource_records">
        </location>
    </locations>
    <transitions>
        <transition from="send_dns_request" params="src_ip, dst_ip,domain_list"
    symbol="dns_req" to="receive_resource_records">
            <guard>
                dst_ip==dnsserver && src_ip==client
            </guard>
            <assignments>
                <assign to="requested_domain_values">
                    domain_list
                </assign>
            </assignments>
        </transition>
        <transition from="receive_resource_records" params="src_ip,dst_ip,
    domain_list" symbol="resource_records" to="send_dns_request">
            <guard>
                dst_ip==client && src_ip==dnsserver && domain_list ==
    requested_domain_values
            </guard>
            <assignments>
                <assign to="requested_domain_values">
                    null
                </assign>
            </assignments>
        </transition>
    </transitions>
</register-automaton>
```

Listing 1: Defining the protocol state machine of DNSusing Register Automata.

6 Questions

Prior work (e.g., Section 4) shows real-world parsing problems helped when $\mathcal{L}_{seq}$ was an FSM, and a register FSM. Are there real-world problems helped when $\mathcal{L}_{seq}$ is more powerful?

In Section 5 above, we defined sequences of the form $C_1/C_2/C_3$, where each C_i was some class of languages. Where does $C_1/C_2/C_3$ itself fit in the hierarchy of languages?

Conjecture. Let C_1, C_2, C_3 be classes of languages. Suppose for some $C \in \{C_1, C_2, C_3\}$ we have each $C_i \subseteq C$. Then $C_1/C_2/C_3 \in C$.

Clearly this is true when we choose classes from among TMs, regular expressions, context-free grammars, and register automata. Is true when the classes include any other items of interest?

Even if $\mathcal{L}_{seq}/\mathcal{L}_{ext}/\mathcal{L}_{int}$ specifies the same languages as a language in some some standard class C, there is still an advantage to thinking this way. A validator written as as some $\mathcal{L}_{seq}/\mathcal{L}_{ext}$ or $\mathcal{L}_{seq}/\mathcal{L}_{ext}/\mathcal{L}_{int}$ may be a much better match to a system's actual requirements—and may be much easier to get right than trying to write the whole thing as one grammar in some standard class.

LangSec traditionally has focused on syntactic correctness of input. However, if we have a program that expects inputs to conform to a language L but still can reject some inputs for semantic reasons (possibly depending on internal system state), then we might be able think of this whole thing as some $TM/L/TM$ language. The semantic becomes syntactic. This general case when we have full Turing machines is absurd—but can we get interesting semantic processing when we have weaker but more tractable classes? Can we build LangSec validators for the power relays of Section 3? Where do we draw the line between the computation of the software application under protection, and the computation of its input validator?

7 Next Steps

In this paper, we described some new ideas to describe the formal semantics of session languages. We defined some new constructs that used newer, and lesser-used classes of automata such as Register Automata, EFSMs, and procedural automata. We also provided an example of a register automata representation of the DNS protocol using an XML-based syntax.

In next steps, we hope to address some of the questions we posed in Section 6. As we discussed earlier, validators described formally as a sequence of languages may be a much more apt fit for practical protocols then the traditional approach of describing the grammar for an entire protocol syntax.

We would also like to explore the metrics that we discussed in Section 5.2—coarseness of recognition and languages that evolve. Future work could explore these metrics to evaluate any LangSec defense mechanism, and by extension any parser that is used to safeguard the rest of the program from unintentional computation.

We believe this approach to session languages is a direction that is worth exploring. We are also looking into building additional constructs to parser-combinator toolkits such as Hammer and Parsley to support such sequences of grammars.

As noted in Section 2.3, the tools of session types may be helpful when dealing with scenarios where the currently correct language for a software module is based on communication with other entities. One approach might be to use session types to model the communication, but use grammars rather than simpler numerical fields as the basis of the message types. We plan further exploration in future work.

Acknowledgements

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR001119C0075 and 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).

References

[1] V. Alagar and K. Periyasamy. *Specification of Software Systems (2nd Ed)*. Springer, 2011.

[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC '04: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, 2004.

[3] P. Anantharaman, M. Locasto, G. F. Ciocarlie, and U. Lindqvist. Building hardened internet-of-things clients with language-theoretic security. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 120–126, May 2017.

[4] P. Anantharaman, K. Palani, R. Brantley, G. Brown, S. Bratus, and S. W. Smith. Phasorsec: Protocol security filters for wide area measurement systems. In *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, pages 1–6, Oct 2018.

[5] Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. A succinct canonical register automaton model. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, pages 366–380, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. Symbolic register automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 3–21, Cham, 2019. Springer International Publishing.

[7] M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: An overview. In *Web Services and Formal Methods: WS-FM2009*. Springer LNCS 6194, 2010.

[8] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[9] M. Frohme and B. Steffen. Active Mining of Document Type Definitions. In *FMICS2018: Formal Methods for Industrial Critical Systems*. LNCS 11119, 2018.

[10] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 251–266, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] E. Poll, J. D. Ruiter, and A. Schubert. Protocol state machines and session languages: Specification, implementation, and security flaws. In *2015 IEEE Security and Privacy Workshops*, pages 125–133, May 2015.

[12] M. Sipser. *Introduction to the Theory of Computation (3rd Ed)*. Cengage, 2012.

[13] K. Tse and P. Johnson. A Framework for Evaluating Session Protocols. In *The Fourth IEEE Workshop on Language-Theoretic Security*, 2017.