# Grammatical Inference and Language Frameworks for LANGSEC

Dr. Kerry N. Wood
U.S. Army Research Laboratory
Adelphi, MD
kerry.n.wood.ctr@mail.mil

Dr. Richard E. Harang
U.S. Army Research Laboratory
Adelphi, MD
richard.e.harang.civ@mail.mil

***Abstract*—Formal Language Theory for Security (LANGSEC) has proposed that formal language theory and grammars be used to define and secure protocols and parsers. The assumption is that by restricting languages to lower levels of the Chomsky hierarchy, it is easier to control and verify parser code. In this paper, we investigate an alternative approach to *inferring grammars* via pattern languages and elementary formal system frameworks. We summarize inferability results for subclasses of both frameworks and discuss how they map to the Chomsky hierarchy. Finally, we present initial results of pattern language learning on logged HTTP sessions and suggest future areas of research.**

***Keywords—grammatical inference, LANGSEC, language identification, pattern language, elementary formal system (EFS)***

## I. Introduction/motivation

In [1], Sassaman et al. propose the use of formal language theory for security (LANGSEC). LANGSEC has primarily focused on the use of formal languages (1) to better define data transiting between components and (2) to verify and restrict parsers of those data transactions. As such, it suffers from the same limitation that has plagued other formal systems in that it must be "cooked in" at design. Protocols must implement a language that is context-free or lower on the Chomsky hierarchy, and parsers must be verified to correctly recognize and then parse that grammar.

The merits of this approach are fairly obvious: namely, there are theoretical decidability bounds when languages are context-free or simpler, and correctly formatted messages may be deterministically recognized and (under slightly stronger technical requirements) unambiguously parsed. Unfortunately, it also requires that

- the language be defined a priori and be restricted to specific Chomsky classes,
- the language be complete for the application, or extensible while still remaining within the Chomsky class,
- the parser be properly implemented and verified.

The final bullet above is clearly difficult, bugs creep into even the smallest and simplest codes. However, the first two bullets are just as problematic in practice. From a programmer and application perspective, the language of protocols can never be too malleable. LANGSEC would paraphrase Einstein, "a language should be as complex as is necessary, but no more complex." The protocol language must meet the application's requirements and stay in a specific Chomsky class. Unfortunately, real-world systems are well known to suffer from creep. Even in the cases in which languages or protocols are well defined, esoteric, seldom-used, and likely insecure components frequently find their way into the code [2][3][4].

Even if a LANGSEC approach is taken for a particular project, there is still the problem of providing tools to programmers to facilitate proper development and testing. In truth, it may be impossible to secure systems post facto. On the other hand, analyzing existing systems through the lens of LANGSEC may provide insight into shortcomings and highlight the tools necessary to enable LANGSEC as a practical discipline in software development.

In this paper, we discuss methods for inferring existing unknown grammars from examples. Such an approach may be useful, for example, when an unknown or poorly documented protocol is being examined or a subset of an established grammar is prohibitively complex to designate in advance in a generative fashion, but data containing examples of the target grammar (and perhaps counterexamples of sentences in the original protocol but not desired in the restricted version) can be obtained or filtered from existing data. Grammatical and language inference is a well-established field with decades of research. We do not present this document as a comprehensive survey of grammatical inference as a whole (readers may refer to [5] as a more complete reference). Rather, we focus on techniques and frameworks that we believe are applicable and useful to LANGSEC. In particular, a framework should be inferable, applicable, and map to the Chomsky hierarchy (Table 1).

| | |
|---|---|
| **Inferable** | The framework should have a learning algorithm that takes input strings from examples, produces a grammar or similar structure, and has tractable complexity. |
| **Applicable** | Otherwise known as membership queries. Given a string, can it be efficiently (polynomial time) determined if the string is a member of the target language? |
| **Map to Chomsky** | The framework should have classes or subclasses that map to the Chomsky hierarchy, thus allowing for the computational limiting argument mentioned previously. |

**Table 1 Grammar framework requirements for LANGSEC.**

DOI 10.1109/SPW.2015.17

The restrictions on inferability and applicability are obvious for real-world use. The final requirement is a key tenet of LANGSEC [1], and makes possible the application of the framework to real-world protocols. We may find that although a protocol is defined to be complex with respect to the Chomsky hierarchy, the vast majority of messages passed in the language fall into a simpler, easier-to-verify class.

This document is organized as follows: in Section II, we provide an overview of learning models pertinent to grammatical inference. In Section III, we introduce the two most relevant non-Chomsky generative grammars we are aware of, pattern languages and EFSs, and some important subclasses of each. Section **Error! Reference source not found.** documents initial results of using pattern languages for learning weblog data and for fuzzing test cases. To close the document, we provide an overview of future work in Section V, and our conclusions and brief overall discussion are in Section **Error! Reference source not found.**.

## II. LEARNING

This section provides the groundwork for the remainder of the paper. First, it is necessary to collect preliminary notation and assumptions. Then, we review the two main models of learning for formal languages: (1) learning in the limit (henceforth: "Gold-style") and (2) probably approximately correct (PAC). It is important that both models be differentiated, because inference difficulty is determined not only by language complexity but by learning type. Where possible and appropriate, we call out potentially confusing overlaps of notation or lexicon. We once again remind the reader that this document merely introduces methods that we believe are useful to LANGSEC. Readers interested in more comprehensive resources may review [5], [6], and [7].

### *A. Preliminaries*

Notation among different grammatical inference methods can be frustratingly inconsistent. We take the time to redefine notation for clarity and completeness of this reference. Much of the notation and definitions are borrowed from [8] and [9].

#### *1) Sets, alphabets, languages, and strings*

If not otherwise specified, script notation refers to classes or sets (e.g., $\mathcal{G}$), whereas standard font indicates a singleton (e.g., $G$). Because languages can be referred to as singletons (a single language), results of generative grammars, or sets of strings, we use double-bar notation to indicate the amorphous standard (e.g., $\mathbb{L}$) and to describe the proper context.

Consider a finite alphabet $\Sigma$ and a string $s$ composed of a finite number of symbols taken from $\Sigma$. The empty string is denoted $\epsilon$, and $\Sigma^*, \Sigma^+$ are the set of all strings (including $\epsilon$ ) and all strings of one or more character. A language $\mathbb{L}$ is a subset of $\Sigma^*(\mathbb{L} \subseteq \Sigma^*)$. For notational convenience, a language can also be represented as a function of its generative grammar. That is, $\mathbb{L}(G)$ is the language generated by grammar $G$. (The difference is important, $\mathbb{L}$ is a set of strings, whereas $G$ is a representation in a framework or function.) To avoid confusion, and as a reminder that order matters in strings and not in sets, we denote the length of a string $s$ as $|s|$, the length of a pattern $\pi$ as $|\pi|$, and the cardinality of a set $S$ as $\#S$.

#### *2) Recursive languages*

A class of languages $\mathcal{L} = \mathbb{L}_1, \mathbb{L}_2, \ldots$ indexed by $J \subseteq \mathbb{Z}^+$ is a family of recursive (a.k.a. "decideable") languages if there is a set of functions $f : \Sigma^* \times I \rightarrow \{0,1\}\ s.t.\ f(s,i) = 1$ iff $s \in \mathbb{L}_i$. The index of a language need not be an integer. One could specify the index via a grammar or pattern for clarity, w.l.o.g. The distinction and differing notation between a class of languages ($\mathcal{L}$) and the individual languages ($\mathbb{L}_i$) is best described as mathematical convenience. At times, we will refer to a class via set notation ($\mathcal{L}$) and others via an indexed set of languages ($\mathbb{L}_1, \mathbb{L}_2 \ldots$). Language classes are most useful when we discuss results for a single language that may extend to further languages collected in a class.

#### *3) Texts and informants*

Now that we have sufficient notation to describe and categorize strings, it is important to discuss how those languages are presented to a learner. A *target* language is the language the learner is trying to learn. *Positive examples* are strings that are known to be in the target language. That is, given a target language $\mathbb{L}$, define $S_+$to be a set of all strings in $\mathbb{L}$ ($S_+ = \{\forall s \in \mathbb{L}\}$)[10]. *Negative examples* are the natural complement, all strings other than the positive examples ($S_- = \Sigma^* - S_+$). A *complete presentation* of language $\mathbb{L}$ is a (potentially infinite) set of positive and negative *labeled* examples. Formally, a complete presentation is an infinite sequence of strings and labels, $(s_1, t_1), (s_2, t_2), \ldots$ where $t \in \{0,1\}$ , $\{s_i |\ i \in I, t_i = 1\} = S_+$, and $\{s_i |\ i \in I, t_i = 0\} = S_-$.

A *positive presentation* of a language $\mathbb{L}$ is precisely $S_+$, the infinite set of strings that define $\mathbb{L}$ [9]. (For this definition, the label ($t$) is dropped, and instead we generate an infinite sequence of $\{s_1, s_2, \ldots\}$ such that $\{s_i | i \in I\} = \mathbb{L}$. ) Note that the definition is exhaustive; the positive presentation of language $\mathbb{L}$ is *all* strings ($S_+$) contained within the language ($\mathbb{L}$).

### *B. Inductive inference from positive examples*

Shapiro [11] defines a model inference problem as follows: "Given the ability to perform experiments in some unknown model $M$, find a finite set of hypotheses, true in $M$, that imply all true observational sentences." That is, if given a set of outputs, can we identify the generating process?

For the purposes of LANGSEC, inductive inference is best described via the definition of an *inference machine* [9][12][13].

**Definition 1 : (Inductive inference in the limit)** Assume a Turing Machine (or equivalent) $M$, called the *inference machine.* $M$ is presented with example strings from grammar $G$ sequentially $(s_1, s_2, \ldots) : (s_i \in S_+)$, and at each iteration, generates hypotheses $(h_1, h_2, \ldots)$ about the actual target grammar $G$. If after some finite number $n$, $h_i = h_n\ \forall i > n$, then we say that $M$ has *converged* to $h_n \equiv h_{final}$.

If, after a finite number of samples, $M$ either terminates or settles on a single, unchanging hypothesis, then we can say that $M$ has converged. Note that the convergence of $M$ does not indicate *correctness* of $h_{final}$. If $h_{final}$ represents the generative grammar for $(s_1, s_2, \ldots)$, which we refer to as $G$, then $M$ is said to have converged *correctly.* Clearly, to be polynomial-time learnable, hypothesis generation must be polynomial time in terms of the input data.

Gold [13] defines inductive inference for formal languages as *learning in the limit,* and shows that it is theoretically possible to infer any indexed family of recursive languages from *complete* data. Unfortunately, he also shows that not all families are inferable from *positive* presentations. Even regular languages fall into this latter class, and cannot be inferred from positive data alone. Angluin [9] shows that languages with specific *thickness* and *elasticity* properties are inferable from positive presentations. The specifics of thickness and elasticity are beyond the scope of this paper (loosely, they relate to the ability to find subsets of strings unique to a particular language within the class under consideration); however, with these notions, Angluin [9] introduced pattern languages, and showed that this structure is indeed inferable from positive data. We discuss pattern languages in detail in Section III.A.

## C. *Probably Approximately Correct*

The reader may note that the requirements for *learning in the limit* (*inductive inference)* are quite stringent and poorly suited to real-world inference. First, a learner must have access to all examples of a language ($S_+$ in its entirety). Second, the learner must converge on a hypothesis, and the final hypothesis must be correct ($s \in \mathbb{L}_{target} : \forall s \in \mathbb{L}(G)$). Finally, the only requirement on the time within which the learner converges is that it be finite. PAC [14] learning relaxes the first two constraints by (1) sampling the positive and negative examples, (2) removing the requirement on exhaustive examples via probabilistic error bounds, and (3) in the most common formulation restricting the computational resources by demanding that the learner attain those bounds in polynomial time with respect to the various parameters of the learning task. PAC algorithms are parameterized by an error probability $\sigma$ and a confidence probability $\delta$. Whereas grammatical inference literature would refer to a subset of $\Sigma^*$ as a string (or word), PAC literature might refer to the subset as a *concept.* A *concept class* ($\mathcal{C}$) is a non-empty set of concepts ($\mathcal{C} \subseteq 2^{\Sigma^+}$). In contrast to Gold-learning, PAC is presented with a subset of a *complete presentation* of strings. This presentation is most often "generated" via an *example oracle,* denoted $EX()$. The learner requests an example from the oracle, who responds in constant time with a random, labeled example $(s_{i_i}, t_i) \subseteq (S_+ \cup S_-)$. In theory, it is possible that the oracle would return an infinite length example. Doing so would require infinite processing time from a polynomial-time learner. Therefore, we allow the learner to request an example no longer than $m$ values by calling $EX(m)$.

The following definitions are primarily taken from [5], with references to [15] and [16] as well.

**Definition 2 : ($\boldsymbol{\epsilon - good\ hypothesis}$)** As in Definition 1, let $G$ be the target grammar, $H$ be the hypothesis grammar, and $\mathcal{D}$ be the distribution over strings $\Sigma^*$. For some $\epsilon > 0$, $H$ is an $\epsilon$-good hypothesis for $G$ if $\Pr_D\big(x \in \mathbb{L}(G) \oplus \mathbb{L}(H)\big) \leq \epsilon$, where $\oplus$ represents symmetric difference.

In plain language, Definition **2** can be summarized as follows: the probability of randomly drawing a string that is in either $\mathbb{L}(G)$ or $\mathbb{L}(H)$, but not in both is limited by $\epsilon$. (If a hypothesis is *perfectly correct,* $\mathbb{L}(G) \oplus \mathbb{L}(H) = \emptyset$.)

**Definition 3 : ($\boldsymbol{Polynomially\ PAC - learnable}$)** A *class* of grammars (or a concept class) $\mathcal{G}$ is said to be PAC-learnable if there exists an algorithm $\mathcal{A}$ that:

- Given any grammar of size $n$, for any (probabilistic) distribution $\mathcal{D}$ over $\Sigma^*$, for every $\delta > 0$ and $\epsilon > 0$, if $\mathcal{A}$ is given access to $EX(m)$, $m$, $n$, $\epsilon$, and $\delta$, then with probability at least $1 - \delta$, $\mathcal{A}$ outputs an $\epsilon$-good hypothesis with respect to $\mathcal{G}$.
- If $\mathcal{A}$ runs in polynomial time with respect to $\frac{1}{\epsilon}, \frac{1}{\delta}, |\Sigma|, m,$ and $n$, then $\mathcal{G}$ is PAC-learnable.

Definition **3** has two important components. First, algorithm $\mathcal{A}$ must have a polynomial-limited computation time in terms of the parameters. Second, the *confidence parameter ($\delta$)* is related to the probability that $\mathcal{A}$ will not output an $\epsilon$-good hypothesis. Colloquially, $\epsilon$ is the accuracy of the hypothesis, and $\delta$ is the probability that the algorithm will not find a hypothesis meeting that requirement.

**Definition 4 : ($\boldsymbol{consistency\ hardness / PAC\ learning}$)** If the consistency problem for a class ($\mathcal{C}$) of concepts $(c_1, \ldots,)$ is $NP$-hard, assuming $NP \neq RP$, the class is not PAC-learnable [17] (Thm. 6.2.1).

## D. *Informant Learning and Consistency*

Section II.B discusses learning from positive examples and is often simply referred to as Gold-learning. If both positive and negative examples are presented to a learner, it is often called *informant learning.* For brevity, we omit a formal definition of informant learning. It is identical to Definition **1**, with the additional constraint that strings are labeled as to their inclusion in the target language $\mathbb{L}$. Informant learning differs from PAC learning in that PAC is allowed to randomly sample input examples from both the positive and negative sets.

A learner is said to be *consistent* if, using its current hypothesis, it can correctly identify string membership in a target language for all examples seen to that point. The Lange and Wiehagen (LW) [18] pattern inference algorithm discussed later in the paper learns *inconsistently.* At any given iteration, the current hypothesis may not correctly identify a previously seen example as a language member. Many learning algorithms cannot provide consistency, and consistency has important implications for PAC learning.

**Definition 5 : ($\boldsymbol{consistency}$)** Borrowing notation from Section II.A.2) a learner $Q$ is said to be *consistent* if $f(s,Q) = 1 : \forall s \in S_+,\ f(s,Q) = 0 : \forall s \in S_-$

| **Language Category** | **Gold-Style[a] Inferable?** | **Inferable in Polynomial Time?** | **Polynomial-PAC Learnable?** | **Membership Query** | **Consistency Problem** |
|---|---|---|---|---|---|
| Pattern language $\mathbb{L}(\pi)$ | Yes[b] [18] | Yes[b] [18] | No | $NP$ - complete [9][19] | $NP$-complete |
| Regular pattern $\mathbb{L}^R(\pi)$ | Yes [20] | Yes | No [24] | PTIME[c] | $NP$-complete |
| Extended (erasing) $\mathbb{L}_\epsilon(\pi)$ | No [25] | □ | □ | □ | □ |
| Extended (erasing) regular $\mathbb{L}_\epsilon^R(\pi)$ | Yes [20][22] | No | No [24] | □ | □ |
| Finite unions of patterns[e] $\mathcal{P} = (\pi_1, \pi_2, \dots)$ $\mathbb{L}(\mathcal{P}) = \cup_{\{\pi \in \mathcal{P}\}} L(\pi)$ | Yes [12][8][26] | Yes | No | No (membership in a single pattern is $NP$ – complete) | $NP$-complete |

**Table 2 Pattern Language Categories**

[a] From positive data.
[b] Lange and Wiehagen present an algorithm that is not consistent, but does infer patterns.
[c] Regular patterns produce regular languages, which are testable with deterministic finite automata.
[d] For any regular pattern $\pi$ and word $w$, checking whether $w \in \mathbb{L}(\pi)$ is decidable in $O(|\pi| + |w|)$ time.
[e] Patterns are not closed under unions. (The class of unions is richer than the class of pattern languages.)
[f] Membership in a single pattern language is $NP$-complete.

## III. Frameworks

### A. *Pattern Languages*

After Gold's negative result [13] regarding the inductive inferability of even the simplest Chomsky classes from positive presentations of languages, interest in inference waned. The introduction of pattern languages by Angluin [19] as a framework that *is* inferable from positive presentations re-ignited interest in the research.

For much of Section III.A.1) we use results and notation from [20] due to clarity and conciseness. Higuera [5] is a gentle introduction to pattern languages, and Ng and Shinohara [21] provide a more recent reference that documents the evolution of learnability of pattern languages from 1980 to present.

#### *1) Overview*

A pattern language uses the same notation for alphabets and strings of Section II.A, with the addition of variables. The set of variables $\mathcal{X} = \{x_1, x_2, \dots\}$ is disjoint from the alphabet ($\mathcal{X} \cap \Sigma = \emptyset$). To prevent confusion, elements of $\Sigma$ are called *constants* and elements of $\mathcal{X}$ are called *variables.*

**Definition 6 : (patterns / regular / k-var)** Given an alphabet of constants $\Sigma$ and a set of variables $\mathcal{X}$, a pattern ($\pi$) is a string comprising constants and variables ($\Sigma^* \cup \mathcal{X}$). We define $\mathcal{P}$ as the set of all patterns over $\Sigma$ and $\mathcal{X}$ ($\mathcal{P} = \pi_1, \pi_2, \dots = \Sigma^* \cup \mathcal{X}$). A pattern $\pi$ is *regular* if each variable in $\pi$ appears exactly once. A pattern $\pi$ is *k-var regular* if it is regular and contains at most $k$ variables.

Example: Let $\mathcal{X} = \{x, y, z\}$ be a set of variables and all other symbols be constant. Then, pattern $xaybza$ is a regular 3-variable pattern, but $xx$ is not regular [22].

**Definition 7 : (substitutions / extended patterns)** [20] For a pattern $\pi$, denote a substitution as $\pi\theta$, in which we replace variable $x_1$ wherever it occurs with (non-null) string $s_1$, variable $x_2$ with string $s_2$, etc. For clarity, substitution $\theta$ can be denoted $\pi\theta = [\frac{s_1}{x_1}, \frac{s_2}{x_2}, \dots, \frac{s_k}{x_k}]$. A substitution is *extended* if null or if "erasing" substitutions are allowed (i.e., a variable may be removed from the pattern entirely without substituting a constant).

**Definition 8 : (pattern language)** [18] For a pattern $\pi$, the language $\mathbb{L}(\pi)$ is the set of all strings that can be obtained by applying Definition **7** to $\pi$.

#### *2) Inferability/applicability*

It is impossible to summarize the 30 years of research touched off by Angluin's introduction of pattern languages [19] in a brief document. (We were unable to find what we would deem a comprehensive reference beyond [21].) Inference difficulty and language inclusion varies wildly for patterns depending on the number of variables, erasing variables, and pattern regularity. Each of these results is a reference in and of itself. For this work, we summarily ignore special cases of largely academic interest, such as restricting patterns to a handful of variables. Table 2 summarizes the classes of patterns we believe are most useful to LANGSEC. Clearly, the most tractable classes of pattern languages are *regular patterns.* Every regular pattern corresponds to a regular language[1], and finite regular languages can be parsed in polynomial time by a finite state automaton (FSA).

### B. *Elementary Formal Systems*

In an informal sense, regular expressions are analogous to the pattern languages. In a similar vein, Elementary Formal Systems (EFSs) may be compared with the programming language Prolog [27]. EFSs were originally introduced by Smullyan [28] in the early 1960s as a method for building a generative grammar for character strings. It remained fairly obscure until the discovery that it could be used as a unifying framework for grammars [29][22][30]. As stated in [31]:

[1] The converse is not necessarily true [23].

*.. EFS's work as grammars to generate languages, as automata to recognize languages, and as logic programs on character strings.*

Perhaps most importantly, EFS programs with specific structure can be categorized unambiguously into the Chomsky hierarchy, as will be discussed in the following section.

*1) Overview*

Much of the introductory material for EFSs in this section is taken from [10][30][31][32]. As in previous sections, assume a finite alphabet $\Sigma$, a finite set of variables $\mathcal{X}$, and a set of patterns ($\mathcal{P} = \pi_1, \pi_2, ...$). For EFS, consider an additional finite set of symbols ($\Pi = p, q, p_1, p_2, ...$ ), called predicate symbols, and assume that $\Sigma$, $\mathcal{X}$, and $\Pi$ are all disjoint. For a pattern $\pi$, let $v(\pi)$ represent the (unique) set of variables appearing in $\pi$.

**Definition 9 : (ground patterns)** Patterns are defined as usual ($\pi \in (\Sigma \cup \mathcal{X})^+$), and a *ground pattern* contains no variables ($\pi_{ground} \in \Sigma^+$). ( For clarity, $v(\pi_{ground}) = \emptyset$).)

**Definition 10 : (atomic formula / ground atoms)** An atom is an expression of the form $p(\pi_1, ..., \pi_n)$, where $p$ is a predicate symbol with arity $n$ and $\pi_1, ..., \pi_n$ are patterns. If all of the patterns are *ground*, then the atom is also *ground.*

**Definition 11 : (definite clause / substitution)** Given atoms $A, B_1, ..., B_n$, a definite clause is: $A \leftarrow B_1, ..., B_n$. The atom $A$ is the *head* of the clause, and $B_1, ..., B_n$ is the *body.* An atom $A$ can be denoted as a body-less clause ($A \leftarrow$).

Just as in Definition **7**, a substitution is denoted $\pi\theta$. For EFSs, a substitution passes into atoms and clauses. For an atom $A = p(\pi_1, ..., p_n)$:

$A\theta = p(p_1\theta, ..., p_n\theta)$.

For a clause of atoms $C = A \leftarrow B$:

$C\theta = A\theta \leftarrow B_1\theta, ..., B_m\theta$.

As with standard patterns, erasing ($\epsilon$) substitutions are not allowed.

**Definition 12 : (ground clause / Herbrand base)** A clause is called ground if all included atoms are ground. The set of all ground clauses is called the *Herbrand base.*

**Definition 13 : (Elementary Formal System)** An EFS ($S$) is a finite set of constants, variables, and definite clauses $S = (\Sigma, \mathcal{X}, \Gamma)$.

**Definition 14 : (provable clauses / languages)**
A definite clause $C$ is provable from an EFS $S$ if $C$ is obtained by finitely many applications of substitutions and modus ponens. This is denoted by: $\Gamma \vdash \mathrm{C}$.

Given an EFS $S$ and a predicate symbol $p$ of arity 1, the *language* $\mathbb{L}(S, p) = \{ w \in \Sigma^+ | \Gamma \vdash p(w)\}$.

**Example 1** [32]**:** EFS for the language $\{a^n b^n c^n \mid n \geq 1\}$

$$\Gamma = \left\{ \begin{matrix} p(x_1, x_2, x_3) \leftarrow \\ p(ax_1, bx_2, cx_3) \leftarrow p(x_1, x_2, x_3) \\ q(x_1 x_2 x_3) \leftarrow p(x_1, x_2, x_3) \end{matrix} \right\}$$

Where the first clause is the bodyless clause "initiating" the construction of the string, the second allows simultaneous extension of each component of the string, and the final is the "termination" and concatenation to the final single string.

**Example 2** [31] A pattern language $\mathbb{L}(\pi)$ is equivalent to a (single-clause) EFS language $\mathbb{L}(\Gamma, \mathrm{p})$, with:

$$\Gamma = \{p(\pi) \leftarrow\}$$

*2) EFS languages: generating, inferring, and accepting.*

It is important to note that EFSs act as a framework for generating languages and for testing for membership of a string within a language (also known as accepting). If one thinks of an EFS as a Prolog code, if presented with an input string, clauses are applied to the string where possible. If the result is an *empty goal* [32], then the string is a member of the EFS language. A full discussion of the theoretical basis and the methods of EFS use is beyond the scope of this paper, and we refer the reader to [32] and [31].

*3) Classes of EFSs (and Chomsky)*

During our discussion of pattern languages in Section III.A, we introduced subclasses such as regular patterns and erasing patterns in a seemingly ad hoc manner. Pattern languages do not map directly to the Chomsky hierarchy (beyond the simple case of the regular pattern languages), and organizing them is difficult and unnatural. Subclasses of EFS, on the other hand, *do* map to Chomsky. Clearly, this categorization has implications for LANGSEC, and we take time to highlight important definitions and results. Those subclasses and the mappings are discussed in the following subsections.

*a) EFS preliminaries*

We borrow significant notation and definitions from [31]. As before, $v(\pi)$ is the set of variables in pattern $\pi$ and the length of a pattern is denoted by $|\pi|$. The length of an atom is denoted $|p(x_1, ..., x_n)| = |\pi_1| + ... + |\pi_n|$. The number of occurrences of variable $x$ in pattern $\pi$ is denoted $o(x, \pi)$, and it can be extended to occurrences of a variable in an atom $o(x, p(x_1, ..., x_n)) = o(x, \pi_1) + \cdots + o(x, \pi_n)$.

*b) EFS classes*

Definitions **15–19** below show the requirements for specific subcases of EFSs. Classes are defined by the type of clause contained therein, and, therefore, the definitions are combined in a single block.

**Definition 15 : (variable bounded)** A clause $A \leftarrow B_1, \ldots, B_m$ is said to be variable bounded if $v(A) \supseteq v(B_1) \cup \ldots \cup v(B_n)$. EFS $S$ is variable bounded if all clauses are variable bounded.

**Definition 16 : (length bounded)** A clause $A \leftarrow B_1, \ldots, B_m$ is said to be length bounded if $|A\theta| \geq |B_1\theta| + \cdots + |B_{n\theta}|$. EFS $S$ is length bounded if all clauses are length bounded.

**Definition 17 : (simple clause)** A simple clause is of the form $p(\pi) \leftarrow q_1(x_1), \ldots, q_m(x_m)$, where $p, q_1, \ldots, q_m$ are unary predicate symbols and $x_1, \ldots, x_m$ are mutually distinct variables appearing in $p$ [31]. EFS $S$ is simple if all clauses are simple.

**Definition 18 : (regular clause)** A simple clause is regular if the pattern in its head is regular. An EFS $S$ is regular if all its clauses are regular.

**Definition 19 : (right/left linear clause)** A regular clause is called right linear if the pattern of the head is $xw$ for some $w \in \Sigma^+$. A regular clause is left linear if the pattern of the head is $wx$ for some $w \in \Sigma^+$. An EFS $S$ is right (left) linear if all its clauses are right (left) linear.

Recall that, in its most concise description, any EFS is a logic program comprising a set of if-then rules that will define or parse a set of words [29]. Definitions **15–19** then define conditions on the structure of the logic program such that the output is necessarily restricted to a class in the Chomsky hierarchy.

*4) Inferability*

As is the case with pattern languages, the inferability of EFSs is an exercise in taxonomy. A small change in EFS structure or in terminology can lead to dramatically different results. Recall that Gold-style learning requires exhaustive (complete) examples of strings in the language, whereas polynomial-PAC learning randomly samples a polynomial number of positive and negative examples. With that in mind, we informally summarize the two most relevant results in learnability of EFSs as follows. First, for Gold-style *learning in the limit*, finite languages that are *at most* context-sensitive complex can be learned by *finite* EFSs. Second, although it is possible to polynomially-PAC learn EFS models for languages, doing so requires additional constraints on the program. These restrictions and the results are presented below. Claims are presented without proof and cited for interested readers.

**Definition 20** [31][29] **: (hereditary clause)** A definite clause $p(\pi_1, \ldots, \pi_n) \leftarrow$
$q_1(\tau_1, \ldots, \tau_{t_1}), q_2(\tau_{t_1}, \ldots, \tau_{t_2}), \ldots, q_l(\tau_{t_{l-1}+1}, \ldots, \tau_{t_l})$ is hereditary if for each $j = 1, \ldots, t_l$, pattern $\tau_j$ is a substring of some pattern $\pi_i$. An EFS is hereditary if all clauses are hereditary.

**Definition 21** [31][29] **: (LB-H-EFS$(\boldsymbol{m}, \boldsymbol{k})$)** Denote LB-H-EFS$(m, k)$ as the class of languages definable by a length-bounded, hereditary EFS with at most $m$ definite clauses such that the number of variable occurrences in the head of each clause is bounded by $k$ $(o(x, p(\ldots) \leq k : \forall x, \forall p_{head})$.

**Claim** 1 [29][33] **:** LB-HEFS$(m, k)$ are polynomial-PAC learnable for any $(m \geq 1, k \geq 1)$.

**Claim** 2 [29] **:** for some $m \geq 1$
(a) Any context-free language is in LB-H-EFS$(m, 2)$.
(b) Any regular language is in LB-H-EFS$(m, 1)$.
(c) LB-H-EFS$(m, k)$ contains a union of $m$ pattern languages that are definable with at most $k$ variable occurrences.

Pertinent results from the preceding section are summarized in Table 3. Once again, we advise care to the reader when interpreting these data because of overlapping or confusing lexicon and the subtle differences between learning methods. For example, *regularity* can refer to pattern languages, a class in the Chomskly hierarchy, and a class of EFS. Taking these three, *all are very different.* Gold's famous result [13] showed that Chomsky-regular languages cannot be learned in the limit. However, regular pattern languages are orthogonal to the Chomsky hierarchy and *can* be learned in the limit. Meanwhile, regular EFSs correspond to context-free languages in the Chomsky hierarchy. We have attempted to provide a terse (and hopefully, useful) summary of research in the area, but even apparently trivial changes in the structure of the language and the learning model can produce significant differences in learnability.

| **Language in Chomsky Hierarchy** | **EFS** | **Gold Inferable** | | **(*Hereditary*)[b] polynomial-PAC Inferable** | |
|---|---|---|---|---|---|
| | *# of clauses* → | $\infty$[a] | $N$[a] | $\infty$ | $(m, k)$ |
| recursively enumerable | Variable bounded | NO | NO | NO | -- |
| context-sensitive | Length bounded | NO | YES | NO | NO[c] |
| context-free | *regular* | NO | YES | NO | YES |
| *regular* | right/left linear | NO | YES | NO | YES |

[a] Upper limit on number of clauses in EFS.
[b] Only hereditary EFSs are PAC inferable. All other EFS are non-PAC inferable.
[c] Length-bounded EFSs are PAC learnable if we ignore computation time.

**Table 3 EFS to Chomsky Mapping**

## IV. Results

Our initial investigation into the use of formal language theory has focused on the use of pattern languages. Pattern languages are known to have good inferability results and are easy to implement, and they are a natural first step to the frameworks discussed in this paper.

### A. Weblog data

As part of its computer network defense service provider operation, Army Research Laboratory has access to log data from production HTTP servers. These data are in the common log-file format [34], as illustrated in Example **3**.

**Example 3 (common logfile format, URL underlined)**

```
127.0.0.1 user-identifier frank [10/Oct/2000:13:55:36 -
0700] "GET /apache_pb.gif.0" 200 2326
```

[a] These IP may represent NAT'ed networks. At the time of writing, we have not investigated.

Table 4 summarizes the descriptive statistics and characteristics of the weblog data corpus.

| | |
|---|---|
| **Total log entries** | ~ $7.2 \times 10^6$ |
| **GET requests** | ~$4.3 \times 10^6$ |
| **POST requests** | ~$2.8 \times 10^6$ |
| **Unique source IP addresses**[a] | 34 |
| **Logging duration / period** | ~ 6 days 4 hours |

[a] These IP may represent NAT'ed networks. At the time of writing, we have not investigated.

Table 4 Weblog data corpus.

We applied `PATTERN_LEARNER` (PL) [5] to the weblog data corpus. (PL and Lange and LW) [18] are largely identical, although the write-ups are slightly different. We selected one arbitrarily to avoid confusion.) All prototypes are in Python, utilizing the PLY lexer [35] for string tokenization. For this section, the symbol alphabet is derived by tokenizing the strings into words (`regex '\w+'`) and then individual symbols (`regex '.'`). As such, the underlined URL of Example **3** would generate the token list: [(/),(apache_pb),(.),(gif)]. Tokenizing on words rather than individual characters reduces the number of symbols the algorithm must consider. Furthermore, it allows a richer and more reasonable comparison of strings; single-character differences no longer throw off string comparisons.

All log entries are assumed to be normal user requests, and therefore the data are considered to be a *positive presentation*. We are not aware of any malicious "attacks" in the corpus.

PL relies on the observation that the pattern can *be no longer than the shortest string in the data-set.* In this corpus, the smallest request was the root directory ("/"). When a pattern learner is applied to a large data-set with a min-length string of one character, the learned pattern is a single variable. (In our output, this would be denoted "`_x0_`".)

To avoid the uninteresting single-variable output, we batched URL data into smaller subsets. The corpus was first broken into POST and GET requests. Within each request type subset, we identified the set of root URLs. "Root" URLs are nothing more than the initial directory structure of a request, stripped of any additional parameters. Log entries were then `grep'd` from the overall corpus into root-specific batches.

Table 5 demonstrates some hand-selected results for each request type. (Full results are available upon request.) URL's have been rudimentarily anonymized; word-strings are mapped to random words of equal length, and HTTP-special characters are unchanged. This mapping preserves notional URL structure but hides URL specifics. (Unfortunately, some context is also lost. In many non-anonymized entries, one can immediately identify the service accessed and data passed in the request.)

We admit without reservation that these results are simple and do not reflect even a partial grammar for HTTP. However, some interesting patterns do emerge. It is important to remember that by definition, PL cannot learn a pattern longer than the shortest string(s) in the input set. In fact, as defined, PL ignores all but the shortest set of input strings. Differences among strings in this set are discovered on a character-by-character comparison, and they are recorded as pattern variables. As documented, neither PL nor LW performs any variable combining as a post-processing step, and it is possible that the output contain a series of neighboring pattern variables.

These facts are imperative for proper interpretation of Table 5. For example, P.1 does not end with a pattern variable. From the form of the pattern, it is fairly obvious that it is a request from a standard endpoint for different files with the same extension. For a set of data that was not a carefully crafted batch, it is possible that this pattern reflects only the shortest examples in the batch, and that longer examples are far different. P.2 shows a string that is most likely an endpoint request. As a result of our full-word lexing, the parameter of the URL is clearly a bang-separated tuple of three values. Finally, P.3 shows the aforementioned artifact of the learning algorithm. PL makes no effort to optimize pattern length, nor does it combine neighboring variables. P.3 has no repeating variables, and one could collect the variables into a single entry w.l.o.g. The bottom block of Table 5 shows that GET requests are similar. G.1 illustrates a long, highly structured request with numerous parameters. G.2, meanwhile, shows an obvious parameter/value pairing for the root URL endpoint. Both patterns are learned over a large number of log entries, and we can be confident that future requests will be similar.

Regular patterns are the simplest of cases, and they do not precisely correspond to a grammar. However, assuming the corpus represents a *positive presentation*, the results do provide insight into the language(s) accepted by specific endpoints of HTTP server(s). If we assume that each root URL corresponds to a code-path, learned patterns may help identify known-good, or flag seemingly bad, requests. We discuss this further in the following subsection and in Section V.

### B. American Fuzzy Lop Test Cases

American Fuzzy Lop (AFL) [36] is a code-fuzzing tool developed by Michal Zalewski and implemented in C. It is fast, powerful, and has elements of both static analysis and standard black-box testing. When one has access to source code, AFL is able to use a modified compiler to inject control trampolines into a runtime. Without source code, AFL can

| Row | Pattern Vars | POST Requests (Anonymized URLs) | # of Samples |
|---|---|---|---|
| P.1 | 1 | /7NWS/XBPD07F/HXZRT/6BR/PR9M/_x0_.Q7OJ | 83199 |
| P.2 | 3 | /0YDO8J33LKSC/DKWPZJ/QCSY1BWNRIG65;R9LZ64B6GI=_x0_!_x1_!_x2_ | 2366 |
| P.3 | 4 | /5HEZP8EKVK3R9RGF9D_x0__x1__x2__x3_ | 204950 |

| Row | Pattern Vars | POST Requests (Anonymized URLs) | # of Samples |
|---|---|---|---|
| G.1 | *17* | /KPGF/7OH0EHB1.HE9?id=_x0_%F95X4L2%_x1__x2__x3__x4_'_x5__x6_%_x7_%_x8__x2__x3__x11_'_x12__x6_%_x14_%_x15_='_x16_' | 6464 |
| G.2 | 4 | /89TM/8J7MNN1/3H6WH/H751AQ57?_t=cd&33XKQ5I6=_x0_&IMK6XIR3=_x1_&AELAN=_x2_&COV=en&89S8A2JUFSVXE=_x3_ | 3096 |

**Table 5 Pattern Learning Weblog Data (pattern variables = "_x<integer>_")**

also be started in pure black-box mode, with the expected loss of performance compared with instrumented code. AFL uses a set of heuristics for test-case generation [37], and it uses the instrumented binary to gauge and guide test-case coverage. During processing, test cases are stored as input files to the binary, in a hierarchical naming format that documents test-case evolution. AFL generates the test cases, runs the binaries, and records any "interesting" inputs that lead to unexpected binary return states (hangs or crashes). As stated in AFL's documentation:

*As a side result of the fuzzing process, the tool creates a small, self-contained corpus of interesting test cases. These are extremely useful for seeding other, labor- or resource-intensive testing regimes…*

### 1) AFL and pattern languages

Test cases are generated iteratively, because AFL permutes previous inputs into new inputs. In general, the file-modification operations are simple bit-level swaps, re-orderings, or value substitutions. For pattern languages, we would expect these operations to appear as pattern variables.

In this subsection, we apply PL to the publicly available corpus of test cases for `libjpeg` on AFL's website. (The corpus is publicly available, can be downloaded, and is static, thus providing a reproducible data-set for this paper.) Files are treated as raw binary and converted to hexadecimal characters using the *NIX "`xxd`" command. At times, we omit short-length test inputs due to the uninterestingly small pattern languages that result.

For these data, and this section, we slightly modified our implementation of PL and denote the new version $PL_\alpha$. In Section A, we highlighted the fact that PL is restricted by the min-length input strings in the input set. We make the following change: assume that the input set has $M$ strings, and one minimum length string of length $\ell$. $PL_\alpha$ selects the first $\ell$ characters of all input strings and treats this as the set of min-length strings. This maintains the pattern length, but avoids single-string outputs. For clarity, $PL_\alpha$ also collects, combines, and renumbers consecutive pattern variables where possible.

Table 6 shows groups of test cases derived from the same source-case (captured in AFL in the command line "`src`" field). With the inferred pattern, it is easy to immediately identify the changes in the byte-strings over the entire generated corpus. In cases C.2 and C.3, note the repeated variables leading to non-regular pattern languages. Substrings have intentionally or unintentionally been mirrored throughout the test set.

Numerous other researchers have focused on test-set coverage and the use of expressive grammars for test-set generation, and we make no claims as to the efficacy of our work. However, this approach is a hybrid, low-effort blend of the two techniques that we believe is worthy of future research. We discuss it further in Section V.

### 2) AFL and language discovery

We observe that AFL is performing language discovery in a randomized fashion, and Zalewski alludes to this in [38]. In this reference, AFL is used to test the common image library `libjpeg`. Rather than initializing the analysis with one or more jpeg images, fuzzing is started on file containing the text string "`hello.`" By virtue of its goal of exercising code paths in the binary, as AFL iterates, it begins to generate test cases that (slowly) begin to reflect the JPEG file format. Zalewski [38] refers to this as "pulling jpegs out of thin air." Because AFL has no knowledge of the language "jpeg," it is using `libjpeg` as a recognizer and searching $\Sigma^*$. Learning "jpeg" is a byproduct of AFL's actual goal: finding those sub-languages that lead to unexpected program behavior. Put another way, if `libjpeg` were implemented perfectly, AFL would discover no faults. However, in its efforts to exercise all code paths, it would develop larger and larger test-set coverage of the language "jpeg."

## V. Future Work

Admittedly, our results represent only a cursory investigation into the frameworks described in this paper. The core of this work is designed to act as a literature search for grammatical inference and as an introduction and less-confusing reference. In the following subsections, we provide a brief overview of areas of interest and future work.

### A. Learning with Elementary Formal Systems

EFSs are covered in Section III.B of this paper. At the time of writing, we have not implemented any EFS learning algorithms. Research in the area has waned, and we note that

| I.D. | Vars. | Test set (files) / PL-inferred Pattern Language | # | Case Length (b) ([μ],[σ]) |
|---|---|---|---|---|
| C.1 | 15 | **<corpus>/jpeg/full/images/*src:000619*** | 25 | (504,117) |
| | | ffd8ffe000104a46494600010102001c001c0000ffdb004300281c1e231e19282321232d2b28303c64413c37373c7b585d4964918099968f808c8aa0b4e6c3a0aadaa_x0_8a8cc8ffcbdaeef5ffffff9bc1ffffffaffe6fdfff8ffdb0043012b2d2d3c353c76_x1_4176f8a58ca5f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f_x2_f8f8f8f8f8f8f8f8f8f8ffc1001108002000200301220002110103110 1ffc40018000101000300000000000000000000000000_x3_03000104ffc4002510000_x4_0_x5_0104010305000000000000000_x6_0_x7_3_x8_0_x9_41221312241_x10_71_x11_143361a1ffc4001601010101000000000000000000000000000000102ffc4001a11010002030100000000000000000000000_x12_122241ffda000c03010002110311000f00567_x13_c_x14_ | | |
| C.2 | 6 | **/jpeg/edges-only/images/*src:003554*** | 12 | (260,0) |
| | | ffd8ffdd0004_x0_00b400e8ffdb00438028101efffaffe6fdfff8ffdb0043012b2d2d3c353c64414176f8a58ca5f8f8d500010000f8f8f8f0f8f8f84a64653d79906fc317d3caf8f8f8f8f8f8f8f8f801001c0000ffdb004300281c1e231e19282336232d2bef303c64413c37373c7b585d49649180999600018c8aa0b4ad8a8cc8ffe4daee097affff9bc1ffffffaffd1fdfff8ffdb0043012b2d2d3cf8f8f8f8e3f8f0e9f900f8f84a6a653d79906fdf17d3b7f8f8f8f8f8f8f8f8f8ffca001108_x2_0_x3__x4__x4_2003012200021101031101ffda0006000118012200021101031101ffda00060001030003000100000000000040000000001ffda000801010101010101010_x1_ | | |
| C.3 | 6 | **/jpeg/edges-only/images/*src:000512*** | 12 | (170,80) |
| | | ffd8ffe000104a_x0__x3__x4__x3__x1_0_x2_ | | |

Table 6 Pattern Language for American Fuzzy Lop JPEG Corpus

the applied EFS inference studies of Arikawa [29] and Miyano [24] were performed in the early 1990s and 2000s, respectively. Neither had access to the ubiquitous and high-performance computation available today, and we intend to leverage computational resources to apply their experimental techniques to our own data.

## *B. Hybrid Fuzzing/Analysis/Grammar Inference*

In Section IV.B, we discussed the use of pattern languages with AFL. Previously, we have mentioned our hypothesis that although many contemporary programs are thought to accept a single language ($\mathbb{L}$), in actuality, they accept a union of finitely many sub-languages $\mathbb{L} = (\mathbb{L}_1, \mathbb{L}_2, \ldots, \mathbb{L}_m)$. Moreover, we believe that those sub-languages often correspond to specific code paths in the target binary.

Our plan is to leverage publicly available tools including clang [39] for LLVM [40] and the aforementioned AFL [36] to tie language learning to code paths. Although it is likely impossible to exhaustively enumerate learnable examples, it may be possible to learn *enough* about sub-language grammar to restrict inputs. By tying grammars to code paths, we hope to be able to restrict inputs to those paths that are assumed (or known) safe. Fracturing of a grammar into sub-grammars that have identifiable execution chains will allow monolithic binaries to be leveraged and still avoid components that are apocryphal, old, esoteric, and likely unsafe [41].

## *C. Other Frameworks*

Pattern languages and EFSs are certainly not the only two grammatical inference frameworks available. Notably, some recent works that focused on natural language processing (Clark [42][43][44]) show promise for grammatical inference. We plan to evaluate and compare these methods with those mentioned in this paper.

## VI. Conclusion/Discussion

In this paper, we have provided an overview of pattern languages and EFSs and their inferability. We have attempted to collect and organize results, de-conflict notation, and highlight the points most germane to LANGSEC. EFSs map directly to the Chomsky hierarchy and can be thought of as a Prolog-like programming language. Pattern languages, on the other hand, resemble regular expressions, are orthogonal to the Chomsky hierarchy, and are much easier to implement. Both frameworks have subclasses that are learnable in the limit or under PAC constraints. Our literature search and initial results end with the unavoidable conclusion that grammatical inference is hard and that learning grammar for even the simplest classes is theoretically and computationally difficult. However, LANGSEC suffers from similar difficulties, only the price is paid at design time. To reap the benefits of LANGSEC's approach, programs require Chomsky-limited protocols and parsers. Grammatical inference and the techniques presented here offer the promise of using LANGSEC after the fact. Pattern languages and EFSs have potential for learning, segmenting, and better understanding grammars.

Our initial results with weblog data and fuzzer input are encouraging enough to motivate continued research in this area. We believe that additional work with these frameworks will lead to tools to help secure existing systems and to assist during the design of new ones.

## References

[1] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *Syst. J. IEEE*, vol. 7, no. 3, pp. 489–500, 2013.

[2] "Schneier on Security: Heartbleed." [Online]. Available: https://www.schneier.com/blog/archives/2014/04/heartbleed.html. [Accessed: 12-Jan-2015].

[3] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and others, "The matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014, pp. 475–488.

[4] "Bash specially-crafted environment variables code injection attack | Red Hat Security." [Online]. Available: https://securityblog.redhat.com/2014/09/24/bash-specially-crafted-environment-variables-code-injection-attack/. [Accessed: 12-Jan-2015].

[5] C. De la Higuera, *Grammatical Inference*, vol. 96. Cambridge University Press Cambridge, 2010.

[6] D. Angluin, "Computational learning theory: survey and selected bibliography," in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, 1992, pp. 351–369.

[7] D. Angluin and C. H. Smith, "Inductive inference: theory and methods," *ACM Comput. Surv. CSUR*, vol. 15, no. 3, pp. 237–269, 1983.

[8] T. Shinohara and H. Arimura, "Inductive inference of unbounded unions of pattern languages from positive data," *Theor. Comput. Sci.*, vol. 241, no. 1–2, pp. 191–209, Jun. 2000.

[9] D. Angluin, "Inductive inference of formal languages from positive data," *Inf. Control*, vol. 45, no. 2, pp. 117–135, 1980.

[10] S. Lange, G. Grieser, and K. P. Jantke, "Extending elementary formal systems," in *Algorithmic Learning Theory*, 2001, pp. 332–347.

[11] E. Y. Shapiro, *Inductive inference of theories from facts*. Yale University, Department of Computer Science, 1981.

[12] T. Shinohara, "Inferring unions of two pattern languages," *Bull Inf*, 1983.

[13] E. M. Gold, "Language identification in the limit," *Inf. Control*, vol. 10, no. 5, pp. 447–474, 1967.

[14] L. G. Valiant, "A theory of the learnable," *Commun. ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[15] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Learnability and the Vapnik-Chervonenkis dimension," *J. ACM JACM*, vol. 36, no. 4, pp. 929–965, 1989.

[16] B. K. Natarajan, "On learning sets and functions," *Mach. Learn.*, vol. 4, no. 1, pp. 67–97, 1989.

[17] M. Anthony, *Computational learning theory*. Cambridge University Press, 1997.

[18] S. Lange and R. Wiehagen, "Polynomial-time inference of arbitrary pattern languages," *New Gener. Comput.*, vol. 8, no. 4, pp. 361–370, Feb. 1991.

[19] D. Angluin, "Finding Patterns Common to a Set of Strings (Extended Abstract)," in *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 1979, pp. 130–141.

[20] T. Shinohara, "Polynomial time inference of extended regular pattern languages," in *RIMS Symposia on Software Science and Engineering*, E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, Eds. Springer Berlin Heidelberg, 1983, pp. 115–127.

[21] Y. K. Ng and T. Shinohara, "Developments from enquiries into the learnability of the pattern languages from positive data," *Theor. Comput. Sci.*, vol. 397, no. 1, pp. 150–165, 2008.

[22] S. Arikawa, S. Miyano, A. Shinohara, S. Kuhara, Y. Mukouchi, and T. Shinohara, "A machine discovery from amino acid sequences by decision trees over regular patterns," *New Gener. Comput.*, vol. 11, no. 3–4, pp. 361–375, 1993.

[23] S. Jain, Y. S. Ong, and F. Stephan, "Regular patterns, regular languages and context-free languages," *Inf. Process. Lett.*, vol. 110, no. 24, pp. 1114–1119, Nov. 2010.

[24] S. Miyano, A. Shinohara, and T. Shinohara, "Polynomial-time learning of elementary formal systems," *New Gener. Comput.*, vol. 18, no. 3, pp. 217–242, Sep. 2000.

[25] D. Reidenbach, "A non-learnable class of E-pattern languages," *Theor. Comput. Sci.*, vol. 350, no. 1, pp. 91–102, 2006.

[26] K. Wright, "Identification of unions of languages drawn from an identifiable class," in *Proceedings of the Second Annual Workshop on Computational Learning Theory*, 1989, pp. 328–333.

[27] A. Colmerauer and P. Roussel, "The birth of Prolog," in *History of programming languages---II*, 1996, pp. 331–367.

[28] R. M. Smullyan, *Theory of Formal Systems*. Princeton University Press, 1961.

[29] S. Arikawa, S. Kuhara, S. Miyano, A. Shinohara, and T. Shinohara, "A learning algorithm for elementary formal systems and its experiments on identification of transmembrane domains," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992*, 1992, vol. i, pp. 675–684 vol.1.

[30] T. Shinohara, "Rich Classes Inferrable from Positive Data: Length-Bounded Elementary Formal Systems," *Inf. Comput.*, vol. 108, no. 2, pp. 175–186, Feb. 1994.

[31] S. Arikawa, S. Miyano, A. Shinohara, T. Shinohara, and A. Yamamoto, "Algorithmic Learning Theory with Elementary Formal Systems," *IEICE Trans. Inf. Syst.*, vol. E75-D, no. 4, pp. 405–414, Jul. 1992.

[32] S. Arikawa, T. Shinohara, and A. Yamamoto, "Learning elementary formal system," *Theor. Comput. Sci.*, vol. 95, no. 1, pp. 97–113, Mar. 1992.

[33] S. Miyano, A. Shinohara, and T. Shinohara, "Which classes of elementary formal systems are polynomial-time learnable," in *Proc. 2nd Workshop on Algorithmic Learning Theory*, 1991, pp. 139–150.

[34] A. Luotonen, "The common logfile format (1995)," *URL Httpwww W3 OrgDaemonUserConfigLogging Html Commonlogfile-Format*.

[35] "PLY (Python Lex-Yacc)." [Online]. Available: http://www.dabeaz.com/ply/ply.html. [Accessed: 11-Jan-2015].

[36] "american fuzzy lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed: 28-Nov-2014].

[37] "Binary fuzzing strategies: what works, what doesn't," *lcamtuf's blog*, 08-Aug-2014. .

[38] "Pulling JPEGs out of thin air," *lcamtuf's blog*, 07-Nov-2014. .

[39] "'clang' C Language Family Frontend for LLVM." [Online]. Available: http://clang.llvm.org/. [Accessed: 12-Jan-2015].

[40] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004, pp. 75–86.

[41] "Heartbleed," *Wikipedia, the free encyclopedia*. 30-Dec-2014.

[42] A. Clark, R. Eyraud, and A. Habrard, "A Polynomial Algorithm for the Inference of Context Free Languages,"

in *Grammatical Inference: Algorithms and Applications*, A. Clark, F. Coste, and L. Miclet, Eds. Springer Berlin Heidelberg, 2008, pp. 29–42.

[43] A. Clark, “Learning Trees from Strings: A Strong Learning Algorithm for some Context-Free Grammars,” *J. Mach. Learn. Res.*, vol. 14, pp. 3537–3559, 2013.

[44] A. Clark, “Efficient, correct, unsupervised learning of context-sensitive languages,” in *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, 2010, pp. 28–37.