Accelerating Legacy String Kernels via Bounded Automata Learning

Kevin Angstadt  
University of Michigan  
Computer Science and Engineering  
Ann Arbor, MI, USA  
angstadt@umich.edu

Jean-Baptiste Jeannin  
University of Michigan  
Aerospace Engineering  
Ann Arbor, MI, USA  
jeannin@umich.edu

Westley Weimer  
University of Michigan  
Computer Science and Engineering  
Ann Arbor, MI, USA  
weimerw@umich.edu

Abstract
The adoption of hardware accelerators, such as FPGAs, into general-purpose computation pipelines continues to rise, but programming models for these devices lag far behind their CPU counterparts. Legacy programs must often be rewritten at very low levels of abstraction, requiring intimate knowledge of the target accelerator architecture. While techniques such as high-level synthesis can help port some legacy software, many programs perform poorly without manual, architecture-specific optimization.

We propose an approach that combines dynamic and static analyses to learn a model of functional behavior for off-the-shelf legacy code and synthesize a hardware description from this model. We develop a framework that transforms Boolean string kernels into hardware descriptions using techniques from both learning theory and software verification. These include Angluin-style state machine learning algorithms, bounded software model checking with incremental loop unrolling, and string decision procedures. Our prototype implementation can correctly learn functionality for kernels that recognize regular languages and provides a near approximation otherwise. We evaluate our prototype tool on a benchmark suite of real-world, legacy string functions mined from GitHub repositories and demonstrate that we are able to learn fully-equivalent hardware designs in 72% of cases and close approximations in another 11%. Finally, we identify and discuss challenges and opportunities for more general adoption of our proposed framework to a wider class of function types.

ACM Reference Format:

1 Introduction
The confluence of several factors, including the significant increase in data collection [34], demands for real-time analyses by business leaders [28], and the lessening impact of both Dennard Scaling and Moore’s Law [73], have led to the increased use of hardware accelerators. Such devices range from Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) to more esoteric accelerators, such as Google’s Tensor Processing Unit or Micron’s D480 Automata Processor. These accelerators trade off general computing capability for increased performance on specific workloads; however, successful adoption requires additional architectural knowledge for effective programming and configuration. Support for acceleration of legacy programs that limits the need for architectural knowledge or manual optimization would only aid the adoption of these devices.

While present in industry for prototyping and application-specific deployments for quite some time, reconfigurable architectures, such as FPGAs, are now becoming commonplace in everyday computing. In fact, FPGAs are in use in Microsoft datacenters and are also widely available through Amazon’s cloud infrastructure [2, 25, 49].

Adopting hardware accelerators into existing application workflows requires porting code to these new programming models. Unfortunately, porting legacy code remains difficult. The primary programming model for FPGAs remains Hardware Description Languages (HDLs) such as Verilog and
VHDL. HDLs have a level of abstraction akin to assembly-level development on traditional CPU architectures. While these hardware solutions provide high throughputs, programming can be challenging—programs written for these accelerators are tedious to develop and difficult to write correctly.

Higher levels of abstraction for programming FPGAs have been achieved with high-level synthesis (HLS) [61], languages such as OpenCL [79], and frameworks such as Xilinx’s SDAccel [95]. While HLS may allow existing code to compile for FPGAs, it still requires low-level knowledge of the underlying architecture to allow for efficient implementation and execution of applications [85, 99]. In fact, optimization techniques for OpenCL programs have been demonstrated to have detrimental impacts on performance when software is ported to a new architecture [10]. Therefore, there is a need for a new programming model that supports porting of legacy code, admits performant execution on hardware accelerators, and does not rely on developer architectural knowledge. We focus our efforts on automata computations, a class of kernels with applications ranging from high-energy physics [93] to network security [71] and bioinformatics [68, 69, 83], for which hardware acceleration is both strongly desired and also capable of providing significant performance gains [8, 33, 35, 36, 39, 80, 88, 91, 96].

In this paper, we present AutomataSynth, a new approach for executing code (including legacy programs and automata computations) on FPGAs and other hardware accelerators. Unlike HLS, which statically analyzes a program to produce a hardware design, AutomataSynth both dynamically observes and statically analyzes program behavior to synthesize a functionally-equivalent hardware design. Our approach is based on several key insights. First, state machines provide an abstraction that has successfully accelerated applications across many domains [68–71, 82, 83, 90, 92, 93, 98] and admit efficient implementations in hardware [33, 88, 96], but typically require rewriting applications. Second, there is an entire body of work on query-based learning of state machines (e.g., see Angluin for a classic survey of computational learning theory [7]), but these algorithms commonly rely on unrealistic oracle assumptions. Third, we observe that the combination of software model checking (e.g., [15, 16]) and recent advances in string decision procedures (e.g., [31, 47, 81, 84]) can be used in place of oracles for certain classes of legacy code kernels, such as those that recognize regular languages.

While AutomataSynth is based on a general approach for synthesizing hardware designs from high-level source code, we focus in this paper specifically on synthesizing Boolean string kernels (functions that return true or false given a string). We propose to accelerate these string kernels using automata processing, which requires representing functions as finite automata [33, 88, 96]. We demonstrate how software model checking, using a novel combination of bounded model checking with incremental loop unrolling augmented with string decision procedures, can answer oracle queries required by Angluin-style learning algorithms, resulting in a framework to iteratively infer automata corresponding to legacy kernels.

We evaluate our prototype implementation of AutomataSynth on a benchmark suite of string kernels mined from public repositories on GitHub. Our evaluation demonstrates that our approach is viable for small functions and exposes new opportunities for improving current-generation tools. We identify four key challenges associated with using state-of-the-art methods to compile legacy kernels to FPGAs and suggest paths forward for addressing current limitations.

In summary, this paper makes the following contributions:

- **AutomataSynth**, a framework for accelerating legacy string kernels by learning equivalent state machines. We extend an Angluin-style learning algorithm to use a combination of iterative bounded software model checking and string decision procedures to answer oracle queries.
- A proof that AutomataSynth terminates and is correct (i.e., relatively complete with respect to the underlying model checker) for kernels that recognize regular languages. The proof leverages the minimality of machines learned by L* and the Pumping Lemma for regular languages.
- An empirical evaluation of AutomataSynth on 18 indicative kernels mined from public GitHub repositories. We learn 13 exactly-equivalent models and 2 near approximations.

## 2 Background and Related Work

AutomataSynth employs a novel combination of tools and techniques from multiple computing disciplines. To the best of our knowledge, this is the first work to combine model learning algorithms, software verification, and architectures for high-performance automata processing. We position our work in the context of related efforts from each area and provide background helpful for understanding our approach.

### 2.1 Finite Automata

In this work, we model the behavior of legacy source code using deterministic finite automata (DFAs) to enable efficient acceleration with FPGAs. Formally, a DFA is a five-tuple, $(Q, \Sigma, q_0, \delta, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet of input symbols, $q_0 \in Q$ is the initial (or starting) state, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function encoding transfer of control between states based on an observed input symbol, and $F \subseteq Q$ is a set of accepting states. A DFA processes input data through the repeated application of the transition function with each subsequent symbol in the input.
input string. After the application of the transition function, a single state within the DFA becomes active. If an accepting state is active after all input characters have been processed, the DFA *accepts* the input (i.e., the input matches the pattern encoded by the DFA).

### 2.2 Accelerators for Finite Automata

There is substantial research on the acceleration of finite automata computation. Automata processing is considered an extremely challenging task to accelerate and is part of the “thirteenth dwarf” in the Berkeley parallel computation taxonomy [11]. Reconfigurable computing has emerged as a suitable platform for accelerating this form of computation [64, 91]. Automata have enabled the acceleration of a wide variety of applications across many domains, including: natural language processing [98], network security [71], graph analytics [70], high-energy physics [93], bioinformatics [68, 69, 83], pseudo-random number generation and simulation [90], data-mining [92], and ML [82].

Xie et al.’s REAPR framework enables high-throughput automata processing using modern FPGAs [96]. Additionally, application-specific accelerators for various types of automata processing have been proposed, such as Micron’s AP [33], the Cache Automaton [80], ASPEN [8], and other ASICS [35, 36, 39, 88]. While accelerators for state machine computation are performant, they require that input problems be phrased in an explicit state machine model, which is uncommon in extant software. Indeed, writing an automaton has been demonstrated to be error-prone and difficult [4, 77], thus leaving an abstraction gap and hindering widespread adoption of automata processing accelerators.

Our development of AUTOMATA SYNTH is complementary to these efforts: we enable the “compilation” of legacy source code for execution on—and acceleration with—automata processing architectures.

### 2.3 State Machine Learning Algorithms

We briefly summarize learning of state machines here, detailing the most relevant instance in Section 3.1. These algorithms are a subset of *model learning* in learning theory and have been the subject of study for several decades [5, 30, 78, 86]. The most common approach is to use *active learning* in which the model is learned by performing experiments (tests) on the software or system to be learned. State machine learning has been applied to the domains of internet banking [1], network protocols [32, 37], legacy systems [54, 72], and describing machine learning classifiers [94].

Most efforts have focused on developing suitable algorithms for learning finite automata [6, 19]. More recent advances simplify the internal data structures of the algorithms, reduce the number of tests necessary to learn a model, or combinations thereof [43, 44, 48, 67]. Learning an equivalent state machine from software remains challenging, and most approaches employ some form of approximation [6, 52].

In this work, we apply this body of model learning research to the problem of adapting legacy source code for efficient execution on hardware accelerators. Our approach attempts to learn a model that is fully equivalent to the original program using software verification techniques, but may also produce approximate results in some situations.

### 2.4 Program Synthesis and Verification

*Program synthesis* is a holistic term for automatically generating software from some input description. Recent efforts have focused on different applications of synthesis, such as sketching [3, 75, 76], programming by example [40], and automated program repair [56, 63]. Many of these approaches employ *counterexample-guided inductive synthesis* (CEGIS) to produce a final solution [76]. CEGIS is an iterative technique that constructs candidate solutions that are tested (typically via formal methods) for equivalence. A *counterexample*, or model of undesirable behavior, is provided if the candidate solution is incorrect, and begins the next iteration of synthesis. We note that CEGIS is largely equivalent to the techniques used in the learning theory community for model learning.

Program verifiers and *software model checkers* prove that a program adheres to a specification or produce counterexamples that violate the specification [15]. These tools typically interleave the control flow graph (CFG) and a specification automaton and explore the resulting graph to determine if any path leads to an error state in the specification.

There has been significant research and engineering effort applied to making these techniques scalable and applicable to real applications [17, 27, 55]. Of particular relevance here are bounded or iterative techniques that address recursive control flow [18, 45], which typically unroll loops a fixed number of times before checking if an error state is reachable in the straight-line portion of the CFG. Most closely related to our work has been the use of bounded model checking to verify string-processing web applications; however, this work often focused on secure information flow rather than constraints over strings [42]. There are also theoretical results on the decidability of straight-line programs on strings, which naturally arise in bounded model checking [53].

A related body of research focuses on extracting program behavior from legacy code for acceleration using domain-specific languages (DSLs), an approach referred to as *verified lifting*. Examples, include extracting stencil computations [46, 57], database queries [26], and sparse and dense linear algebra calculations [38]. By targeting DSLs, verified lifting can leverage known properties of the given problem domain to aid extraction and acceleration. Our development of AUTOMATA SYNTH complements these efforts: we focus on learning hardware descriptions of string kernels for acceleration with FPGAs. For generality, we intentionally limit the domain-specific assumptions leveraged by our approach.

We propose a novel combination of insights from counterexample guided synthesis, theorem proving and automata
We propose AutomataSynth, a framework for learning functional behavior models for off-the-shelf, legacy code implementing regular languages and synthesizing hardware descriptions from those models. Our approach extends Angluin’s L* algorithm [6] by (1) using bounded software model checking with incremental unrolling to implement one of its assumptions, (2) using software testing to implement another of its assumptions, and (3) transforming learned models into homogeneous DFAs for hardware synthesis.

3 Learning State Machines from Legacy Code

We propose AutomataSynth, a framework for learning functional behavior models for off-the-shelf, legacy code implementing regular languages and synthesizing hardware descriptions from those models. Our approach extends Angluin’s L* algorithm [6] by (1) using bounded software model checking with incremental unrolling to implement one of its assumptions, (2) using software testing to implement another of its assumptions, and (3) transforming learned models into homogeneous DFAs for hardware synthesis.

3.1 L* Primer

Dana Angluin’s foundational L* algorithm was popularized in 1987 [6]. Because many of our framework decisions (such as how to implement its required queries and counterexamples in a legacy source code context) and results (such as correctness and termination arguments) depend on the steps and invariants of her algorithm, we sketch it here in some detail. We claim no novelty in this subsection and readers familiar with L* can proceed to Section 3.2.

At its core, the L* algorithm relies on a minimally adequate teacher (MAT) to answer two kinds of queries about a held-out language, L. First, the MAT must answer membership queries, yielding a Boolean value indicating if the queried string is a member of L. Second, the MAT must answer conjecture or termination queries. Given a candidate regular language A, typically expressed as a finite state machine, the MAT responds with true if A = L or responds with a counterexample string for which A and L differ. (Note that automata learning is used in applications where L is not a DFA, and thus this query is typically not resolved by standard DFA equivalence checking.)

These queries are used to construct an observation table that can be transformed directly into a DFA. This table may be defined as a 3-tuple, (S, T, E), where S is a nonempty, finite, prefix-closed set of strings over Σ; E is a nonempty, finite, suffix-closed set of strings over Σ; and T is a function mapping ((S ∪ S · Σ) · E) to {true, false}. (S, T, E) may be visualized as a two-dimensional array where rows are indexed by a value s ∈ S · Σ, columns are indexed by a value e ∈ E, and entries are equal to T(s · e). For ease of notation, Angluin defines row(s) to be a finite function, f, mapping values from E to {true, false} defined as f(e) = T(s · e). Informally, row(s) denotes the values in a particular row of the observation table.

An observation table must be both closed and consistent before a DFA may be correctly constructed. A table is closed if for every t ∈ S · Σ, there exists an s ∈ S such that row(t) = row(s). A table is consistent if, for all s1, s2 ∈ S where row(s1) = row(s2), row(s1 · a) = row(s2 · a) for all a ∈ Σ. These properties ensure that there is a valid transition out of each state in the DFA (closed) and that transitions on any character remain the same regardless of the characters already processed (consistent). Given a closed and consistent observation table, a DFA over the alphabet Σ may be constructed as follows:

\[ Q = \{ \text{row}(s) \mid s \in S \}, \]
\[ q_0 = \text{row}(\varepsilon), \]
\[ F = \{ \text{row}(s) \mid s \in S \land T(s) = \text{true} \}, \]
\[ \delta(\text{row}(s), a) = \text{row}(s \cdot a). \]

Each unique row in the observation table becomes a state in the candidate automaton, and outgoing transitions from a state are defined by the row indexed by the current row’s prefix concatenated with the transition character.

Pseudocode for the L* algorithm is provided in Algorithm 1. A closed, consistent observation table is constructed using membership queries. Then, the table is transformed into a candidate automaton for a termination query. If the MAT responds with a counterexample, the counterexample and its prefixes are added to the observation table. The process repeats until the MAT responds to a termination query in the affirmative. The final automaton is the learned language.

3.2 AutomataSynth Problem Description

In this subsection, we formalize the problem of learning a state machine from a legacy string kernel.

AutomataSynth operates on a function that takes one string argument and returns a Boolean value:

\[ \text{kernel : string} \to \text{bool} \]

These are also called equivalence queries, but we avoid this term to prevent confusion with similar uses of the term in software verification.

\(^{2}\) A set is prefix-closed if for every s ∈ S, every prefix of s is also a member of S. Suffix-closure is defined similarly.
We assume that the source code for this function is provided and that the function halts and returns a value on all inputs (i.e., `kernel` is an algorithm). If `kernel` recognizes a regular language, `AUTOMATA_SYNTH` returns a state machine, `M`, with equivalent behavior to `kernel`. That is, for all `s ∈ Σ^*`, `M(s) = kernel1(s)`. For runs which exceed a resource budget or expose incompleteness in the underlying theorem prover (including functions that are non-regular), our prototype implementation alerts and provides *approximate* equivalence, where `M(s) = kernel1(s)` when the length of `s` is less than an arbitrary fixed length (see Section 4).

In Section 4.4, we present a formal proof that our framework produces an equivalent DFA for input kernels that recognize regular languages. Our empirical evaluation in Section 6 demonstrates that real-world legacy string kernels either recognize regular languages, or our tool can produce an approximation of the original function. We discuss the challenges associated with supporting a broader class of functions in Section 7.1.

### 3.3 Using Source Code as a MAT

We propose extending Angluin’s L* algorithm to learn a DFA representation of a legacy string kernel. To succeed, we must construct a MAT that can answer membership and termination queries about an input string kernel. While the L* algorithm provides a framework for query-based DFA learning, the original work does not define any one mechanism for implementing the teacher. Our proposed MAT implementation leverages the source code of the target function.

**Membership Queries.** We observe that a membership query for a string, `s`, may be implemented by executing the legacy kernel on `s`. The result returned by the function is the answer to the query. For languages akin to C employing integers, we interpret Boolean values in the standard way (i.e., 0 is false and all other values are true). While direct and intuitive in theory, we note that there are several challenges in practice. Following the C standard, many runtime systems assume that strings are null-terminated (i.e., a null character must only occur as the final character in a string). In practice, however, we find that legacy string kernels will sometimes allow null characters in other positions. This often occurs when the length of the input string is known *a priori*. While seemingly innocuous, this deviation from the standard limits the runtime mechanisms by which the legacy kernel may be invoked. We found that compiling the kernel to a shared object and then invoking the function dynamically provided the best stability in our experiments.

**Termination Queries.** At the heart of our problem formulation is the challenge that a legacy string kernel does not admit a direct means for answering termination queries. A recent survey of model learning indicates that testing for equivalence queries [86]; however, our initial efforts found testing alone to be unsuitable for termination queries in this domain. Our insight is that verification strategies from software model checking may be used to test for equivalence between the kernel and a candidate automaton. Traditionally in verification, equivalence would be proven using bisimulation or interleaving of the automaton and the source kernel. However, this formulation presupposes that the “state transitions” are directly encoded in the source code and can be aligned with the state transitions in the candidate automaton. We do not make this assumption in our problem definition in Section 3.2, and we prefer an approach that does not require manual annotation. Indeed, we do not even assume that the states of the equivalent automaton are visited “in order” during the execution of the legacy kernel.

We propose to verify an alternate reachability property that places additional constraints on the input string. In particular, we observe that a counterexample `t ∈ Σ^+` is in either `L(kernel1)` or `L(M)`, but not in both, and thus will always satisfy the constraint `t ∈ L(kernel1) ⊕ L(M)`, where ⊕ is the symmetric difference operator. Therefore, we ask the software verifier to prove that there is no execution of `kernel1` on `t` where `kernel1` returns `true` and `t ∉ L(M)` or `kernel1` returns `false` and `t ∈ L(M)`. To test this reachability property, we use a novel combination of bounded model...
checking with incremental loop unrolling augmented with a string constraint solver. We discuss the implementation of this verification task in depth in Section 4.

Software verifiers are relatively complete (e.g., [12]), meaning that there are certain programs that cannot be fully verified due to limitations in the underlying SMT solvers. Verifiers often return an answer in three-valued logic: true in our application means that the kernel and candidate automaton were proved equivalent, allowing for termination; false in our application means that the property was not satisfied, and there is a counterexample to provide to the \( L^* \) algorithm; and unknown in our application means that the verifier was unable to prove equivalence, but also does not provide a counterexample. In the case of an unknown answer from the verifier, we halt our algorithm and warn the user that the resulting automaton is approximate; there may be inputs for which the automaton returns an incorrect answer.

### 3.4 Synthesizing Hardware Descriptions from Automata

Once a state machine has been learned using the \( L^* \) algorithm with our custom MAT, the kernel is now amenable to acceleration. There has been a significant effort to accelerate automata using FPGAs [96] and other custom ASICs (e.g., GPUs [91, 97] and Micron’s AP [33]). We convert the learned automaton to a hardware description and synthesize the design for loading onto an FPGA. Other execution platforms are possible [9], but we focus on FPGAs in this work due to their widespread deployment.

### 3.5 System Architecture

Figure 1 depicts the high-level system architecture of our framework. The \( L^* \) learner (shown to the left) queries a MAT (shown to the right) consisting of the legacy source code, software model checker, SMT solver, and string decision procedure. The legacy string kernel is used by the MAT to answer membership queries. Termination queries are transformed by a mapper into a software verification problem that searches for string that distinguish the language of a candidate automaton from the target language implemented in the kernel. The output of the Learner is a DFA that encodes the same computation as the Kernel. We use this DFA to synthesize a hardware design for execution on an FPGA.

### 4 Implementation and Correctness

In this section we lay out formal properties of our implementation, first demonstrating that iterative, bounded software model checking conforms to the required properties for MAT termination queries. We then prove correctness and termination for \textsc{AutomataSynth}. Because \textsc{AutomataSynth} operates on arbitrary source code and employs theorem proving techniques, correctness and termination are relative.

#### 4.1 Bounded Model Checking

There are several SMT-based model checking algorithms that have been employed to verify properties of software. We use bounded model checking, an algorithm best-suited for the queries currently supported by string constraint solvers. In particular, string solvers do not currently support most interpolation queries (e.g., \[29\]), which are used heavily by counterexample-guided [27] algorithms such as SLAM and BLAST [13, 15]. Developing interpolation algorithms that support string constraints is beyond the scope of this work.

**Bounded model checking** enumerates all program paths up to a certain bound that reach a target error state (e.g., an error label in the source code) [18]. For each path, the algorithm generates a set of constraints over the program’s variables, and the disjunction of these constraints is passed to an SMT solver to determine if the constraints for at least one path are satisfiable. If so, the set of variable assignments represents a configuration of the program that would result in an error condition at runtime. In this approach, loops are unrolled a fixed number of times (the “bound”). We follow the standard practice of incremental unrolling (cf. [20]), which iteratively applies bounded model checking for increasing unrolling depths. For programs with unbounded loops this strategy results in a semi-algorithm; however, we demonstrate in Section 4.4 that there exists a finite unrolling that fully verifies a kernel deciding a regular language for our property.
4.2 Reasoning about Strings

As described in Section 3.3, AutomataSynth must verify that there are no strings in the symmetric difference of the legacy kernel and a candidate automaton. We propose encoding the language of the candidate automaton as a regular expression constraint on the input string parameter of the kernel. We then solve the encoded problem using a bounded model checker that reasons about strings. A suitable string decision procedure must support (at minimum):

- Unbounded string length,
- Regular expression-based constraints over strings,
- Access to individual characters of strings,
- Comparison of individual characters and strings,
- Reasoning about the length of strings, and
- The ability generate strings that satisfy a set of constraints.

Additional features supported by string decision procedures can be helpful for representing standard library string functions. Recent decision procedures, such as Z3str3 [14] or S3 [84], support these required properties.

To combine bounded model checking with string decision procedures, we extend the CPAChecker extensible program analysis framework [16] to generate and solve string constraints. We modify CPAChecker’s predicate analysis algorithm to generate “String” sort constraints for string-like types in C programs (e.g., char and char* types [62]). We produce a character extraction constraint for each occurrence of indexing of (and dereferencing) string variables. Additionally, we add support for functions such as strlen.

The C language specification does not directly support regular expressions. To embed these constraints in a program’s source code, we also add an additional function for checking if a string variable conforms to a regular expression.

4.3 Verification for Termination Queries

Listing 1 demonstrates our formulation of termination queries using bounded model checking with incremental loop unrolling and string decision procedures. We construct a wrapper around the source code for the kernel that adds additional assertions to the path constraints used by the software verifier. When the kernel returns true, we add the constraint that the input string cannot be represented by the regular expression representing the candidate automaton. We add a similar constraint for the false case as well. Finally, we ask the verifier to prove that the error label (line 23) is unreachable (note that assume constraints influence reachability).

4.4 Correctness

In this subsection, we conclude our formal development of termination queries based on the combination of bounded model checking with incremental loop unrolling and string decision procedures. We demonstrate that this approach satisfies the Angluin constraints for MAT termination queries.
all strings in $L(K)$ shorter than a given length. The intuition is that the finite unrolling $K'$ corresponds to the use of bounded model checking.

**Lemma 4.2.** Let $p \in \mathbb{N}$ and $K$ be a program that recognizes a subset of $\Sigma^*$. There exists an $n \in \mathbb{N}$ such that the $n$-finitely-unrolled program $K'$ obtained from $K$ (with all loops replaced with the finite unrolling of the first $n$ iterations and all subsequent iterations removed) satisfies $\forall s \in \Sigma^*, |s| < p \implies K'(s) = K(s)$.

**Proof.** Given $p \in \mathbb{N}$, we construct the set of strings $S = \{ s \mid s \in \Sigma^* \land |s| < p \}$, on which $K'$ must agree with $K$. We let $n$ be the maximum number of iterations performed by $K(s)$ for all $s \in S$. Because $S$ is a finite set, its maximum value is guaranteed to exist and be finite. We construct $K'$ by unrolling $n$ times the program $K$. By construction, the property $\forall s \in \Sigma^*, |s| < p \implies K'(s) = K(s)$ holds. 

We also reason using the standard Pumping Lemma for regular languages. For reference, we recall the Lemma here without proof as defined by Sipser [74, Theorem 1.70].

**Lemma 4.3 (Pumping Lemma for Regular Languages).** If $A$ is a regular language, then there is a number $p$ such that for all $s \in A$, if $|s| > p$ then $s$ may be divided into three pieces, $s = xyz$, satisfying the following conditions: for each $i \geq 0$, $xy^iz \in A$, $|y| > 0$, and $|xy| \leq p$.

The smallest such $p$ is called the *pumping length*. We call out as a Lemma the association between pumping lengths and minimality [74, Proof of 1.70]:

**Lemma 4.4.** The (smallest) pumping length of a regular language $L$ is equal to the number of states in the minimal DFA that recognizes $L$.

Additionally, our proof makes use of two theorems about the output of the $L^*$ algorithm. We paraphrase these results here [6]. See Section 3.1 for $L^*$ definitions, such as $(S, E, T)$.

**Theorem 4.5 ($L^*$ [6], Theorem 1).** If $(S, E, T)$ is a closed, consistent $L^*$ observation table, then the DFA $M$ constructed from $(S, E, T)$ is consistent with the finite function $T$. Any other DFA consistent with $T$ but not equivalent to $M$ must have more states.

We will use the following corollary of this result.

**Corollary 4.5.1.** Let $p$ be the pumping length of the target language, $L$, and $M$ be a DFA constructed from a closed, consistent $L^*$ observation table. The pumping length of $L(M)$ does not exceed $p$.

Finally, we make use of the $L^*$ algorithm termination result. The property we use in our proof has been emphasized.

**Theorem 4.6 ($L^*$ [6], Theorem 6).** Given any MAT presenting a regular language $L$, $L^*$ eventually terminates and outputs a DFA isomorphic to the minimum DFA accepting $L$. Additionally, if $n$ is the number of states in the minimum DFA recognizing $L$ and $m$ is an upper bound on the length of any counterexample provided by the MAT, then the total running time of $L^*$ is bounded by a polynomial in $m$ and $n$.

With these properties in hand, we are now ready to prove our original theorem.

**Proof (Theorem 4.1).** Given a pure program $K$, which decides a regular language, and a candidate DFA $M$ constructed from a closed, consistent $L^*$ observation table $(S, E, T)$, let $p$ be the pumping length of $L(K)$. By Lemma 4.4, the minimal DFA that recognizes $L(K)$ has $p$ states. By Lemma 4.2, there exists a finite unrolling $K'$ of program $K$ such that $\forall s \in \Sigma^*, |s| < p \implies K'(s) = K(s)$. We will show that verifying $K' \equiv M$ is sufficient to verify $K \equiv M$ using bounded model checking.

Verifying the property $\forall t \in \Sigma^* \exists s$ such that $|t| < p \land K'(t) \neq M(t)$.

Case 1: There exists $t \in \Sigma^*$ such that $|t| < p \land K'(t) \neq M(t)$.

Case 2: For all $t \in \Sigma^*$ such that $|t| < p$, $K'(t) = M(t)$ holds.

In the first case, we return $t$ as a counterexample, concluding $K \not\equiv M$. In the second case, we conclude that $K' \equiv M$ and any counterexample must be at least as long as $p$; however, no such counterexample exists. The proof proceeds by contradiction.

Suppose, for the sake of contradiction, that $\exists t' \in \Sigma^*$ such that $|t'| \geq p$ and $K(t') \neq M(t')$. Let $n$ be the number of states in the candidate DFA $M$. We now relate $n$ to the number of states in the minimal DFA recognizing $L(K)$. By Lemma 4.4 and Corollary 4.5.1, $n \leq p$ because the pumping length of $L(M)$ is at most $p$ and the number of states in $M$ is equal to the pumping length of $L(M)$. Additionally, because the finite unrolling $K' \equiv M$, $n \geq p$ by Theorem 4.5. Therefore, the number of states in $M$ is bounded above and below by the pumping length of our target language, implying that $n = p$.

To conclude, we note that $K$ is consistent with $T$ but not equivalent to $M$, and thus by another application of Theorem 4.5 we can conclude that the DFA recognizing $L(K)$ must have more than $n = p$ states. This contradicts the fact, from Lemma 4.4, that the minimal DFA recognizing $L(K)$ has exactly $p$ states. Therefore, no such $t'$ exists.

Because $L^*$ produces a minimal DFA (Theorem 4.6), and $M$ was produced from a closed, consistent observation table, we can conclude that $M$ must be a DFA isomorphic with the minimal DFA accepting the language $L(K)$. Thus, $K \equiv M$.

This means that, using bounded model checking on the program $K'$ (recall that $K'$ is a finite unrolling and thus admits bounded model checking), we either find a counterexample or can conclude equivalence of $K$ and $M$. Therefore, $K' \equiv M \implies K \equiv M$. 

□
From this result, we can establish the following corollary, which allows us to conclude that our approach may be used in a MAT to answer termination queries.

**Corollary 4.6.1.** For a given program $K$, there exists a finite number of iterations of incremental unrolling needed for our approach to respond to a termination query with either a counterexample or a proof of equivalence.

**Implications.** In our formulation, termination queries return an answer if the bounded software model checking with incremental loop unrolling and the string decision procedures terminates. Our result is therefore relative to the completeness of the model checker and underlying SMT theories (see Ball et al. for a discussion of relative completeness in software model checking [12]). For pure kernels that decide a regular language, we proved that there is a finite bound on the incremental unrolling that will determine equivalence of the kernel and a candidate automaton. In practice, we make use of a timeout on the verification process to ensure timely termination at the expense of correctness in some cases. This design decision results in an approximate solution in cases where either the finite unrolling bound has not yet been reached or the legacy kernel recognizes a non-regular program. The approximate solution is correct for strings of length up to a particular bound but may disagree on larger strings. Our empirical evaluation in Section 6 demonstrates that AUTOMATA SYNTH successfully learns an equivalent state machine for thirteen of eighteen real-world string kernels mined from legacy source code.

## 5 Experimental Methodology

In this section, we describe our process for selecting real-world, legacy string kernels benchmarks as well as our experimental setup for the evaluation described in Section 6.

### 5.1 Benchmark Selection

In our evaluation, we focus on measuring the extent to which AUTOMATA SYNTH learns models for real-world string functions using varied library methods. We construct our benchmark suite by mining legacy string kernels from open-source software projects on GitHub using the following protocol. First, we filter all projects for those with C source code and ordered the resulting repositories by number of stars (i.e., popular repositories first). Next, we use the Cil [62] framework to iteratively parse each source file and extract all functions with an appropriate type signature (see Section 3.2). We filter these functions to exclude those that referenced functions or data outside the compilation unit. We allow the use of common library function (e.g., strlen, strcmp, etc.). In total, we considered 26 repositories and mined 973 separate string kernel functions using this protocol.

After filtering for duplicates and a manual analysis to identify functions that return Boolean values (we note that while C has the _Bool data type, many functions still use integers of varying widths), we collected 18 meaningfully-different real-world benchmarks. Table 1 provides an overview of these string kernels. We use the function name to refer to each benchmark and also indicate the source project for each. Lines of code (LOC) provides a count of the total number of non-comment lines in the post-processed version of the benchmark. Finally, we also indicate whether the kernel is supported by our prototype system. Our prototype implementation supports all but three of these legacy string kernels. The unsupported kernels use computation that is difficult to capture with present string decision procedures.

The kernels in our benchmark suite interact with strings in various manners. Some kernels, such as is_numeric_index, skipline, and cf_is_comment, loop over all characters in the string checking various constraints. Several also make heavy use of strcmp to check for the presence of specific strings (e.g., checkerrormsg, is mounted section name, and start line). We also found examples of kernels (e.g., git_offset_lst_component and AMF DecodeBoolean) that perform single character comparisons. While a developer will likely not be interested in accelerating a single character comparison, these kernels remain indicative of real-world code and allow us to demonstrate a proof-of-concept for synthesizing designs for accelerators such as FPGAs. An evaluation of benchmarks more typical of kernels accelerated by FPGAs (e.g., long-running kernels with hundreds or thousands of states) is left for future work.

<table>
<thead>
<tr>
<th>Function</th>
<th>Project</th>
<th>LOC</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>git_offset_lst_component</td>
<td>Git: Revision</td>
<td>6</td>
<td>✓</td>
</tr>
<tr>
<td>is_encoding_utf8</td>
<td>control system</td>
<td>38</td>
<td>✘</td>
</tr>
<tr>
<td>checkerrormsg</td>
<td>jq: Command-line</td>
<td>4</td>
<td>✓</td>
</tr>
<tr>
<td>checkfail</td>
<td>JSON processor</td>
<td>14</td>
<td>✓</td>
</tr>
<tr>
<td>skipline</td>
<td></td>
<td>17</td>
<td>✓</td>
</tr>
<tr>
<td>end_line</td>
<td></td>
<td>11</td>
<td>✓</td>
</tr>
<tr>
<td>start_line</td>
<td>Linux: OS kernel</td>
<td>11</td>
<td>✓</td>
</tr>
<tr>
<td>is mounted section name</td>
<td></td>
<td>54</td>
<td>✘</td>
</tr>
<tr>
<td>is_numeric_index</td>
<td>MASSCAN: IP</td>
<td>17</td>
<td>✓</td>
</tr>
<tr>
<td>is_comment</td>
<td>port scanner</td>
<td>11</td>
<td>✓</td>
</tr>
<tr>
<td>AMF DecodeBoolean</td>
<td></td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>cf_is_comment</td>
<td>OBS Studio: Live</td>
<td>28</td>
<td>✘</td>
</tr>
<tr>
<td>cf_is_splice</td>
<td>streaming and</td>
<td>22</td>
<td>✓</td>
</tr>
<tr>
<td>is_reserved_name</td>
<td>recording software</td>
<td>39</td>
<td>✓</td>
</tr>
<tr>
<td>has_start_code</td>
<td></td>
<td>18</td>
<td>✓</td>
</tr>
<tr>
<td>number_is_valid</td>
<td>openpilot:</td>
<td>72</td>
<td>✘</td>
</tr>
<tr>
<td>utf8_validate</td>
<td>Open-source</td>
<td>72</td>
<td>✘</td>
</tr>
<tr>
<td>stbtt_isfont</td>
<td>driving agent</td>
<td>24</td>
<td>✓</td>
</tr>
</tbody>
</table>

*Requires strcasecmp support †Requires strtod support ‡Performs math on characters

| Table 1. Benchmark Suite of Real-World, Legacy String Kernels |
5.2 Experimental Setup

Our AUTOMATA SYNTH implementation produces MNRL, an open-source state machine representation language intended for large-scale automata processing applications [9]. We transform the learned DFA to be homogeneous, a property that admits a simplified transition rule while maintaining expressive power and that is amenable to hardware acceleration [10, 23, 33, 91]. We use Brzozowski’s algorithm [21] for converting candidate DFAs to regular expressions as part of the software verification step (see Section 4).

For termination queries, we add string constraint handling to CPAChecker 1.8 [16]. We also extend the JavaSMT framework [47] to support the draft SMT-LIB strings theory interface [81]. We use Microsoft’s Z3 version 4.8.6 SMT solver [31] with the Seq string solver [84] for all queries. All evaluations use an Ubuntu 16.04 Linux server with a 3.0 GHz Intel Xeon E5-2623-v3 with four physical cores and 16 GB of RAM and a maximum time budget of 24 hours.

6 Evaluation

In this section, we evaluate AUTOMATA SYNTH on fifteen real-world, legacy string kernels mined from open-source projects. We first evaluate the correctness of the state machines generated by AUTOMATA SYNTH and report runtime and query counts. Second, we evaluated the suitability of the generated automata for hardware acceleration.

6.1 State Machine Learning

Table 2 presents results from our empirical evaluation of AUTOMATA SYNTH on a benchmark suite of fifteen legacy string kernels. We do not report results for the three benchmarks that are not supported. We report the number of membership and termination queries executed for each kernel as well as the number of states in the learned automaton and the total runtime in seconds. The final column indicates if AUTOMATA SYNTH correctly learned the kernel’s functionality. A check mark means that our tool learned a fully-equivalent automaton. We also indicate approximate results in which the maximum time limit was exceeded.

On average, it took seven hours to learn an automaton from the legacy string kernel, with more than half of the benchmarks terminating in fewer than five minutes. AUTOMATA SYNTH correctly learned thirteen of the fifteen benchmarks. The remaining two benchmarks yield approximate solutions, with many of these approximations being extremely similar to the target kernel functionality. In our evaluation, this approximation was always the result of timeouts rather than the relative completeness of the SMT solver used for termination queries. There were no instances in our benchmark set for which the SMT solver returned an unknown result due to a limitation in the string decision procedures.

We determined that there were two primary causes for AUTOMATA SYNTH reaching the timeout without learning a fully-equivalent state machine. First, Brzozowski’s algorithm for constructing a regular expressions can produce large expressions that require simplification to remove redundant and superfluous clauses. This was most relevant to kernels that compared string suffixes with a string constant. We believe this performance limitation is an artifact of design choices in our prototype, which could be solved with more careful construction of regular expressions. Second, some SMT queries were significantly less performant than others. We discuss this challenge in more detail in Section 7.

The relative utility of the membership and termination queries varies between the benchmarks. For example, the function git_offset_1st__component checks a string to see if the first character is a forward slash (/). Using membership queries, AUTOMATA SYNTH learned that the string must be a slash and that any number of characters may follow. The termination query provided a single counterexample of a longer string that was initially misclassified. For this kernel, the membership queries provided much of the "learning". This is in contrast to the stbtt__isfont kernel, which ultimately compares an input string against four hard-coded strings. In this case, the membership queries only provided minimal information. Instead, the termination queries discovered the string constants in the kernel’s source code and provided much of the learned information. In general, membership queries tended to provide more information when each character in the input string was considered separately while termination queries helped to discover string constants used for comparison by the kernels.

| AUTOMATA SYNTH successfully learned automata for five of the eighteen legacy kernels mined from open-source projects. Of these, thirteen were exactly equivalent and two were near approximations. |

6.2 Hardware Acceleration

In this work, we claim no novelty for accelerating automata using hardware accelerators, such as FPGAs. Instead, we leverage existing work in the area of high-performance automata processing. On FPGAs, Xie et al.’s REAPR framework supports high-throughput processing of data with finite automata on FPGAs [96]. For spatially-reconfigurable architectures akin to FPGAs, the dominant factor affecting performance is the number of hardware resources used by a design. For ANMLZoo benchmarks, which contain tens of thousands of states [91], REAPR successfully synthesized designs running in the range of 200–700 MHz. Because the automata learned by AUTOMATA SYNTH are significantly smaller, we expect that similar throughputs could be achieved.

| The finite automata learned by AUTOMATA SYNTH fall within the design constraints of FPGA-based automata accelerators, allowing for high-throughput execution. |
Table 2. Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Membership Queries</th>
<th>Termination Queries</th>
<th>Number of States</th>
<th>Total Runtime (s)</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>git_offset_lst_component</td>
<td>4,090</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>✓</td>
</tr>
<tr>
<td>checkerrormsg</td>
<td>32,664</td>
<td>2</td>
<td>15</td>
<td>86,195</td>
<td>✓</td>
</tr>
<tr>
<td>checkfail</td>
<td>189,013</td>
<td>3</td>
<td>35</td>
<td>86,308</td>
<td>✓</td>
</tr>
<tr>
<td>skipline</td>
<td>7,663</td>
<td>3</td>
<td>3</td>
<td>294</td>
<td>✓</td>
</tr>
<tr>
<td>end_line</td>
<td>510,623</td>
<td>4</td>
<td>44</td>
<td>29,531</td>
<td>✓</td>
</tr>
<tr>
<td>start_line</td>
<td>206,613</td>
<td>2</td>
<td>46</td>
<td>4,813</td>
<td>Approx.</td>
</tr>
<tr>
<td>is_mcounted_section_name</td>
<td>672,041</td>
<td>7</td>
<td>57</td>
<td>86,399</td>
<td>Approx.</td>
</tr>
<tr>
<td>is_numeric_index</td>
<td>10,727</td>
<td>3</td>
<td>4</td>
<td>297</td>
<td>✓</td>
</tr>
<tr>
<td>is_comment</td>
<td>4,090</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>✓</td>
</tr>
<tr>
<td>AMF_DecodeBoolean</td>
<td>2,557</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>✓</td>
</tr>
<tr>
<td>cf_is_comment</td>
<td>4,599</td>
<td>2</td>
<td>4</td>
<td>300</td>
<td>✓</td>
</tr>
<tr>
<td>cf_is.splice</td>
<td>1,913</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>is_reserved_name</td>
<td>350,705</td>
<td>8</td>
<td>42</td>
<td>85,469</td>
<td>✓</td>
</tr>
<tr>
<td>has_start_code</td>
<td>10,213</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>stbtt__isfont</td>
<td>79,598</td>
<td>5</td>
<td>19</td>
<td>13</td>
<td>✓</td>
</tr>
</tbody>
</table>

*AutomataSynth warned of a potential approximate solution due to timeout, but manual analysis confirmed correctness.

7 Discussion

At a high level, AutomataSynth learns the behavior of a Boolean string kernel through a combination of dynamic and static analyses and emits a functionally-equivalent state machine that is amenable to acceleration with FPGAs. We believe that approaches such as AutomataSynth are very promising and could offer solutions to limitations inherent to current HLS techniques. HLS relies heavily on the structure of C-like source code to produce a hardware description, which were designed for performance on—and as an abstraction of—von Neumann architectures. As such, HLS is unlikely to produce performant FPGA designs from legacy code that was heavily optimized for CPUs [85, 99]. This places a heavy burden on developers tasked with porting code and represents a significant barrier to adoption. Our approach decouples the implementation choices of the legacy program from the emitted hardware design. This allows us to produce a design using a model of computation—state machines—that is performant on FPGAs [91, 96].

This paper represents an initial effort to understand the benefits and limitations of using state machine learning algorithms to compile code for FPGAs. A significant research effort remains for approaches akin to AutomataSynth to be mature enough for industry adoption. In the remainder of this section we identify four key research challenges whose solutions would lead to significant advances in learning-based synthesis for FPGAs. Additionally, we describe candidate future directions to tackle each of these.

7.1 Learning More Expressive Models

We present an approach for accelerating regular language Boolean string kernels with FPGAs. Our prototype soundly transforms such kernels to functionally-equivalent hardware descriptions; Boolean functions with inputs that may be transformed into a serial data stream are also applicable. However, legacy code contains many other types of functions, and these remain an open challenge. Supporting a new function type presents a two-fold challenge: (1) identifying suitable computational models for acceleration and (2) designing or adapting an algorithm suitable for learning these models. Finite automata, as formally defined, produce a single bit of output for each string processed and are limited to recognizing Regular Languages. Additional models, such as Mealy and Moore machines, support transforming an input value into an output value, while others, such as pushdown automata, support more expressive classes of languages.

Several efforts are underway to extend learning algorithms to more expressive computational models [19, 24, 59]. It may also be possible to leverage insights from the architecture community and recent efforts to accelerate automata processing, in which designs often support tagging output report signals with additional metadata [33, 89]. Further, existing DFA learning algorithms may admit learning functions that output an enumerated—or even a multi-bit—value.

An additional challenge is that determining program equivalence is, in the limit, undecidable. For example, Angluin notes that termination queries are not generally decidable for context-free languages [6]. However, existing software verifiers suffer from this same challenge and provide relative completeness [12]. Further, this challenge may be addressed in some cases through careful use of approximation.

7.2 Expressive Power and Performance of String Solvers

Our empirical evaluation of AutomataSynth demonstrated some limitations of present string decision procedures. Certain string operations (e.g., case-insensitive lexicographic
comparisons and casting between characters and numbers to perform arithmetic operations) occur in real-world software but are difficult to represent as constraints in String theories. Additionally, SMT queries generated by bounded model checking algorithms can result in long-running computation.

These challenges are not new: the formal methods community has been laboring to improve string decision procedures for over a decade. Early efforts often focused on the problem domain of identifying cross-site scripting and SQL code injection vulnerabilities (e.g., [41]) and introduced new constraint types. These efforts often reasoned about fixed-sized string variables (e.g., [50]). Subsequent efforts, such as Z3str3, also focus on improving the performance of these decision procedures and have extended support to unbounded strings [14].

**AutomataSynth** is one of the first efforts to combine bounded software model checking with string decision procedures. This combination presents a novel and compelling use case for string solvers that requires new constraint types and optimizations. We make our tool and all of the SMT queries automatically generated by our process available\(^4\) to the community to encourage renewed interest in—and efforts to—improve the performance of string solvers.

### 7.3 Scaling Termination Queries

We found, in practice, that termination queries consumed an average of 66% of the total runtime of **AutomataSynth**. As candidate state machines increase in size, we expect the scalability of termination queries to dominate. This challenge presents an opportunity for innovation. We proposed an approach based on the novel combination of bounded software model checking and string decision procedures; however, alternate formulations of termination queries may provide better performance while maintaining correctness.

Many applications of model learning focus on the use of testing to provide answers to termination queries [86]. We have observed that the application of automated testing presents several challenges, such as producing a suitable quantity and diversity of inputs to identify counterexamples. Test input generators, such as Klee [22], may only support bounded length strings (rather than unbounded).

The application of other software verification techniques may also provide performance gains. Counterexample-guided abstraction refinement verifiers can abstract much of a program’s state to gain performance, but require support for interpolation queries. These are not currently supported by string solvers, but present an additional area for research.

### 7.4 Characterizing and Taming Approximation

Because scalability and decidability of termination queries are challenges, approximation may play an important role in improving the performance of learning-based approaches to synthesizing FPGA designs. Indeed, there is already significant interest in the architecture and software communities for producing approximate programs [58, 60, 65, 66].

Approximation has been a key parameter in model learning algorithms from the start [6, 86]. Results from learning theory often analyze approximation using Valiant’s *probably approximately correct* (PAC) framework, which bounds the probability of the error being less than a fixed threshold for an approximately-learned model [87]. Such results can predict the number of queries necessary to bound the error but *do not characterize the locations or significance of the remaining error*. Anomalous results for frequently-used inputs have a very different impact than anomalous results for seldom-used inputs. Given the design of Angluin-style algorithms, it may be possible to determine which inputs result in approximate solutions. For example, pre-populating the observation table with rows pertaining to known inputs (i.e., those taken from the test suite) ensures that the learned state machine produces the correct output for those relevant values.

### 8 Conclusions

We present **AutomataSynth**, a framework for accelerating legacy regular language Boolean string kernel functions using FPGAs. Our approach uses a novel combination of state machine learning algorithms, software verification algorithms, string decision procedures, and high-performance automata processing architectures to learn the behavior of a program and construct a behaviorally-equivalent FPGA hardware description. We demonstrate a proof-of-concept of this approach using a benchmark suite of eighteen string kernels mined from open-source projects on GitHub. **AutomataSynth** successfully constructs equivalent (or near equivalent) FPGA designs for more than 80% of these benchmarks. We believe this approach shows promise for overcoming some of the limitations of current HLS techniques.

### Acknowledgments

This work is funded in part by: NSF grants CCF-1629450, CCF-1763674, CCF-1908633; AFRL Contract No. FA8750-19-1-0501; and the Jefferson Scholars Foundation. We also wish to thank Kevin Skadron and Lu Feng for their suggestions, encouragement, and expertise in the early stages of this project as well as the anonymous reviewers for their helpful comments and feedback.

### References


Accelerating Legacy String Kernels via Bounded Automata Learning


