Checking LTL[F,G,X] on Compressed Traces in Polynomial Time

Minjian Zhang
minjian2@illinois.edu
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA

Umang Mathur
umathur3@illinois.edu
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA

Mahesh Viswanathan
vmahesh@illinois.edu
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA

ABSTRACT

The problem of checking if a program execution meets a formal specification arises in many software engineering tasks including runtime verification and designing test oracles. When online analysis is not possible, execution trace logs are stored for offline postmortem analysis, often in a compressed format to reduce disk space and warehousing requirements. A straightforward method for checking if a compressed execution satisfies a property is to first decompress it and then analyze the resulting uncompressed execution.

In this paper, we consider the problem of checking if an execution trace, compressed using a grammar-based lossless compression scheme, satisfies a specification expressed in linear temporal logic, without explicitly decompressing it. In general, this problem is known to be intractable (PSPACE-hard in the size of the compressed trace and the LTL formula). We show that the problem can be solved in *polynomial time* for the fragment LTL[F, G, X], which comprises of all Boolean and modal operators of LTL except the *until* operator. Our algorithm for analyzing SLPs (a grammar-based compression scheme) is effective in practice — for a suite of large execution traces obtained from open source projects, our algorithm shows significant speed ups when compared with the performance of checking LTL properties over corresponding uncompressed traces.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Linear Temporal Logic, Compression, Runtime Verification

ACM Reference Format:
Minjian Zhang, Umang Mathur, and Mahesh Viswanathan. 2021. Checking LTL[F,G,X] on Compressed Traces in Polynomial Time. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468557

1 INTRODUCTION

Consider the *membership problem* that can be abstractly defined as follows: Given a program execution τ, determine if τ is a "good" behavior. This computational problem plays a key role in several approaches whose goal is the engineering of reliable and secure software. The first sub-area where it arises is *runtime verification* [10, 27] where one dynamically monitors the behavior of a system with to determine if the it conforms to system requirements. This approach can be used to augment testing by observing system behavior along paths that were inadequately exercised during testing, thereby resulting in increased code coverage. The key computational task in runtime verification is to solve the *membership problem* — check whether the monitored execution needs satisfies a system requirement. A second sub-area of interest is the design *test oracles* [9]. Given executions of a system exercised by a test suite, a test oracle is a program that distinguishes between correct and incorrect behaviors of the system. Thus, a test oracle can be seen as solving the membership problem for the specific system being tested. The membership problem also plays a key role in intrusion detection [43]. Log files that record the interaction between a network of elements need to be examined to detect patterns of "intrusive behavior" so that corrective measures can be taken to avoid security compromises. Since these log files are large, and there are multiple patterns of intrusive behavior, intrusion is typically detected automatically by a program that solves the membership problem to determine if the log files do not contain an intrusive pattern. Finally, the membership problem also needs to be solved when *statistically model checking* a system. Statistical model checking [5, 44] is an approach to verify quantitative properties of stochastic systems. In this approach, the model checker executes the stochastic system a few times to draw a statistical sample of system behaviors, and then use hypothesis testing to determine the likelihood of a property being true of a system. A crucial step in this process is building an oracle that determines for each execution, whether it satisfies a desired logical property.

One practical challenge in each of these application areas that rely on solving the membership problem is the size of the program execution that needs to be analyzed. Program traces that arise in runtime verification, testing or statistical model checking are often huge, containing millions of events. Long traces are often necessary to exercise large parts of the code base to ensure good code coverage. Log files analyzed for intrusions often record interactions that take place over long windows of time, sometimes over multiple days. The challenge therefore is two-fold: how to store such long traces/logs, and how to effectively analyze them. The common solution to address the warehousing needs for such traces is to compress them and then store them in a compressed format.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468557

Given that program executions need to be compressed to address storage costs, an important question in the context of the membership problem is, to find effective solutions to the problem when the input trace is compressed. This question is not new, and has a rich history, especially in theoretical computer science [7, 8, 12, 15, 16, 31, 38, 46]. The short summary of results in this space is as follows. There is always a naïve algorithm to solve the membership problem on compressed traces — uncompress the trace and check membership. In many situations this is often (provably) the best algorithm possible [7, 16, 38]. However, there are exceptions where the membership problem can be solved in time that is polynomial in the size of the *compressed trace* [30]; one notable example is dynamic race detection on compressed strings [23].

The main question we investigate in this paper is the following: Given a program trace τ in compressed form and a formula φ in linear temporal logic (LTL) [39], determine if τ satisfies φ. LTL is widely used in testing and verification. It's popularity relies on the fact that, on the one hand it is rich enough to express many requirements that typically arise in software engineering, and on the other hand, the absence of explicit quantification, makes it simple enough for a practitioner to easily write properties. Compression schemes we consider are those where a program trace is represented using a *straight line program (SLP)*. SLPs are special context-free grammars where the language of the grammar contains exactly one string, namely, the trace it represents. Several lossless compression schemes. like run-length encoding and Lempel-Ziv encodes [50], can be efficiently converted into SLPs with similar size. There are efficient implementations of compression algorithms that produce an SLP representation of a given execution [3, 21, 22, 25, 37, 48–50].

The problem of determining if a finite trace compressed using an SLP satisfies an LTL property, has been studied before. The problem is known to be intractable — it is PSPACE -hard [33]. Therefore, we ask if there is a rich fragment of LTL for which the problem can be efficiently solved. We consider the fragment LTL[F, G, X] which is the collection of all LTL formulas that are built from propositions using boolean operators, and only the temporal operators X (next), F (eventually or finally), and G (always or globally); in particular, U (until) cannot be used in the formulas of LTL[F, G, X]. The fragment LTL[F, G, X] is expressively very rich. Over infinite traces, LTL[F, G, X] can express properties in each class of the *safety-progress* classification of temporal properties introduced by Manna and Pnueli [32][1]. Our main result is that the problem of checking if a finite trace represented by an SLP satisfies an LTL[F, G, X] formula can be decided in time that is polynomial in the size of the SLP (compressed trace) and the formula.

We now outline the technical challenges and our theoretical contributions in obtaining this result. The principal idea used in verification, runtime verification, and automatic test oracle generation for temporal properties is to exploit the connection between LTL formulas and automata — translate the formula into an automata, and then "run" the automaton with the program or trace to verify or test. For runtime verification or test oracles, the automaton constructed from the formula needs to be *deterministic*. This idea can also be used when checking compressed traces where you effectively "run" the deterministic automaton on the grammar representing the trace.

However, even for LTL [F, G] formulas[2], the smallest nondeterministic automaton is exponential and the smallest deterministic automaton is doubly exponential in the size of the formula [6]. Theoretical lower bounds establish that this cannot be improved. Our first observation is that if the finite (uncompressed) trace is processed *right-to-left* instead of left-to-right, then there is an *exponential sized, deterministic* automaton for each LTL[F, G, X] formula, that can solve the membership question. "Running" an automaton left-to-right or right-to-left on an SLP is very similar and so this change does not fundamentally change the algorithm for compressed traces. However, the fact that the automaton is exponential sized would affect the complexity; for compressed traces, the running time of an algorithm using this automaton would be exponential in the formula. To combat this, we observe that the automaton we design for LTL[F, G, X] has special "monotonicity" properties and has a small "diameter". These two observations can be combined to observe that there are "essentially" $O(m)$ state changes (m here refers to the size of the LTL[F, G, X] formula) when the automaton is run on *any* (uncompressed) trace, no matter what the length of the trace is. Finally, we exploit the special structure of the states of this automaton, to design an algorithm for compressed traces. To prove that this algorithm indeed runs in time that is polynomial in the formula size and the grammar, requires carefully counting the number of substrings that arise in a string represented by an SLP.

We evaluate the performance of our algorithm for checking compressed traces over open source Java projects obtained from GitHub (largely derived from prior study [28]). We also use 10 LTL[F, G, X] properties describing specifications for the use of iterators, collections, file objects, etc., Our evaluation suggests that, large traces from open source projects can be effectively compressed (with an average compression ratio of more than 600×) and that compressed traces can be effectively checked against these specifications, leading to significant speed ups (averaging at 34×).

The rest of the paper is organized as follows. Section 2 discusses background relevant for the exposition. Section 3 presents an overview of our algorithm for checking LTL[F, G, X] formulae on compressed traces, and Sections 4 and 5 discuss the technical details of the algorithm. We present our evaluation in Section 6, related work in Section 7 and concluding remarks in Section 8.

2 PRELIMINARIES

In this section we present preliminary notations about execution traces, LTL monitoring and the SLP compression format.

2.1 Execution Traces

In many approaches whose goal is to either prove the correctness of a software or find errors, a key computational problem that needs to be solved is the *membership problem*, where one needs to determine if a given program execution is correct with respect to a system specification. In this setting, an execution trace (or simply an execution) can be abstractly modeled as a finite sequence of "events" belonging to a set (say) Σ. The set of events Σ is determined by what

[1] Or in every Borel class that has ω-regular properties.

[2] These are LTL formulas that only have F and G (and no X or U) as temporal operators.

```java
class SetTraversal {
    HashSet<Integer> s = new HashSet<Integer> ();
    public void insert(int max) {
        for(int i = 0; i < max; i++) s.add(i);
    }
    public int sumAllExcept(int val) {
        Iterator<Integer> itr = s.iterator();
        int sum = 0;
        if(!itr.hasNext()) return sum;
        while(true){
            int i = itr.next();
            if(i == val) continue;
            sum = sum + i;
            if(!itr.hasNext()) break;
        }
        return sum;
    }
}

class SetTraversalTest {
    @Test
    void testInsertAndSum() {
        SetTraversal st = new SetTraversal();
        st.insert(128);
        int actual = st.sumAllExcept(64);
        int expected = (127 * (127 + 1))/2 - 64;
        assertEquals(expected, actual);
    }
}
```

Figure 1: Java class `SetTraversal` with methods `insert` and `sumAllExcept`. The unit test `testInsertAndSum` tests these two methods at once.

is visible or has been made visible through instrumentation when the program is executed. Thus, an *execution* is $\tau = \mathsf{e}_0\mathsf{e}_1 \cdots \mathsf{e}_{k-1}$ where each $\mathsf{e}_i \in \Sigma$; the *empty trace/sequence* will be denoted by ϵ. Let us fix an execution $\tau = \mathsf{e}_0\mathsf{e}_1 \cdots \mathsf{e}_{k-1}$. The ith event in the execution will be denoted by $\tau[i] = \mathsf{e}_i$. We will denote the substring $\mathsf{e}_i\mathsf{e}_{i+1} \cdots \mathsf{e}_{j-1}$ by $\tau[i : j]$, the suffix $\mathsf{e}_i\mathsf{e}_{i+1} \cdots \mathsf{e}_{k-1}$ by $\tau[i :]$ and the prefix $\mathsf{e}_0 \cdots \mathsf{e}_{i-1}$ by $\tau[: i]$. The *length* of execution τ, denoted $|\tau|$, is the number of events in it which is k. By definition $|\epsilon| = 0$.

Example 1. Consider the Java class `SetTraversal` in Figure 1. Every instance of this class has a member variable `s`, which is a set of integer elements. The method `insert` inserts all non-negative integers less than `max` in `s`, while the method `sumAllExcept` returns the sum of those elements of the set `s` which are different from the integer `val`. We remark that the implementation of `sumAllExcept` is functionally correct whenever `val` is not the last value when traversing `s` using the iterator `itr`. If `val` is the last value in `s` when traversing using `itr`, the loop body can execute `next()` (after traversing the node with `itr`), even though there are no remaining elements, which may raise a Java exception (`NoSuchElementException`).

The figure also shows a test class `SetTraversalTest` that implements a unit test `testInsertAndSum` that first calls `insert` on an instance `st` of `SetTraversal` with the argument 128 and then checks if the sum of elements thus inserted (except the element 64) is as expected. The given unit test passes and, in fact, does not expose the bug outlined above. The execution trace generated due to this test, can nevertheless be used to infer the possibility of an exception. If we instrument calls to the methods `hasNext()` and `next()`, then we will observe a trace over the alphabet $\Sigma = \{\mathsf{h}, \mathsf{n}\}$, where h represents a call to `hasNext()` and n represents a call to `next()`. For the unit test `testInsertAndSum`, we will observe the execution trace $\tau = (\mathsf{hn})^{65}\mathsf{n}(\mathsf{hn})^{62}\mathsf{h}$. This is because, in this case, the iterator traverses the set `s` in the order of insertion, and for the first 65 elements (values 0 through 64), the method `insert` correctly calls `hasNext()` before `next()`. However, in the next step, it enters the loop and calls `next` without checking `hasNext()`. All the subsequent loop executions generate the sequence $(\mathsf{hn})^{62}\mathsf{h}$. In subsequent sections, we will discuss how analyzing τ, in fact, can hint at the possibility of an exception. Observe that $|\tau| = 256$, $\tau[: 130] = (\mathsf{hn})^{65}$, $\tau[130 :] = \mathsf{n}(\mathsf{hn})^{62}\mathsf{h}$ and $\tau[2 : 130] = (\mathsf{hn})^{64}$.

2.2 Linear Temporal Logic

Linear temporal logic (LTL) is a popular logic for specifying temporal properties of systems, and is widely used to specify correctness properties. In this section, we introduce the syntax and semantics of LTL and some fragments that are relevant in this paper. Since program executions encountered while testing and runtime verification are assumed to be finite, our semantics for LTL will be defined for finite execution traces. While this is not the classical semantics for LTL, it is standard [13]. We will also be using a "letter semantics" for the logic — models are sequences of letters as opposed to sequences of sets of propositions, and formulas are built using letters as opposed to propositions.

Syntax of LTL. Let us fix a finite alphabet Σ. Then, a formula φ in LTL over Σ is given by the following grammar.

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \mid \varphi\mathsf{U}\varphi$$

Here, a is a symbol in Σ, $\neg$, $\wedge$ and $\vee$ are Boolean connectives and X ('next'), F ('eventually'), G ('always') and U ('until') are temporal modal operators. We will use $\varphi_1 \implies \varphi_2$ as a shorthand for $\neg\varphi_1 \vee \varphi_2$.

Semantics of LTL. The semantics of LTL is given by how an LTL formula φ evaluates over a finite *non-empty* trace $\tau \in \Sigma^+$. We formally describe this evaluation relation $\models_f$ below; the subscript 'f' in $\models_f$ stands for evaluation over *finite* traces.

$$\begin{array}{llll}
\tau \models_f a & \text{iff} & \tau[0] = a \\
\tau \models_f \neg\varphi & \text{iff} & \tau \not\models \varphi \\
\tau \models_f \varphi_1 \wedge \varphi_2 & \text{iff} & \tau \models_f \varphi_1 \text{ and } \tau \models_f \varphi_2 \\
\tau \models_f \varphi_1 \vee \varphi_2 & \text{iff} & \tau \models_f \varphi_1 \text{ or } \tau \models_f \varphi_2 \\
\tau \models_f \mathsf{X}\varphi & \text{iff} & |\tau| > 1 \text{ and } \tau[1 :] \models_f \varphi \\
\tau \models_f \mathsf{F}\varphi & \text{iff} & \text{there is an } i \text{ such that } 0 \leq i < |\tau| \text{ and } \tau[i :] \models_f \varphi \\
\tau \models_f \mathsf{G}\varphi & \text{iff} & \text{for every } i \text{ such that } 0 \leq i < |\tau|, \ \tau[i :] \models_f \varphi \\
\tau \models_f \varphi_1\mathsf{U}\varphi_2 & \text{iff} & \text{there is an } i \text{ such that } 0 \leq i < |\tau| \text{ and } \tau[i :] \models_f \varphi_2 \text{ and for every } j \text{ such that } 0 \leq j < i, \tau[j :] \models_f \varphi_1
\end{array}$$

Remark. LTL, as presented here, only has future time operators. Some presentations include past time operators as well: Y (for 'yesterday'), the dual of X; O (for 'once'), the dual of F; H (for 'historically'), the dual of G; and S (for 'since'), the dual of U. Over finite traces, the following property holds. Let $\widehat{\varphi}$ be the formula obtained by replacing every past time (future time) operator by the corresponding dual future time (past time) operator in φ. Then $\tau \models_f \varphi$ if and only if $\tau^{\mathsf{r}} \models_f \widehat{\varphi}$; here τ^{r} denotes the reverse of τ. This means that over finite traces, LTL with only past time operators is "equivalent" to LTL with only future time operators. Our results, though presented only for LTL with future time operators, also apply to LTL with purely past time operators.

Example 2. Consider the program shown in Figure 1 and the execution τ produced when calls to methods `hasNext()` and `next()` are tracked. As shown in Example 1, $\tau = (\mathsf{hn})^{65}\mathsf{n}(\mathsf{hn})^{62}\mathsf{h}$, where h represents a call to `hasNext()` and n represents a call to `next()`. Intuitively, in a correct implementation, the program should check the existence of a next element (i.e., event h) before accessing the next element (i.e., event n). The program in Figure 1 does not satisfy this intuitive correctness requirement since an event h does not precede the event n when the value 65 is accessed. We can formalize our informal intuition by requiring that 'there are no successive calls to `next()`'. However, this by itself is not enough because the execution nhn does not have successive n events, but the first n event is not preceded by h. So we must also require that the execution does not begin with n. We could write this as $\varphi = (\neg\mathsf{n}) \land \mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}(\mathsf{n}))$. One can see that τ does not satisfy this property (as desired) because $\tau[129 :] = \mathsf{nn}(\mathsf{hn})^{62}\mathsf{h}$ does not satisfy $\mathsf{n} \implies \neg\mathsf{X}\mathsf{n}$.

Fragments of LTL. We will consider a couple of fragments of LTL obtained by restricting the modal operators that appear in formulas. The first fragment is LTL[X] which consists of formulas built from events and Boolean operators using only X operator. The next fragment is LTL[F, G, X] which uses the modal operators X, F, and G, but does not use U. We skip the formal BNF grammar for these fragments.

Formulas in LTL[F, G, X] can be expressed in a normal form that is obtained by pushing X as far in as possible. Since $\mathsf{X}(\varphi_1 \lor \varphi_2) \equiv (\mathsf{X}\varphi_1) \lor (\mathsf{X}\varphi_2)$, $\mathsf{X}(\varphi_1 \land \varphi_2) \equiv (\mathsf{X}\varphi_1) \land (\mathsf{X}\varphi_2)$ and $\mathsf{XF}\varphi \equiv \mathsf{FX}\varphi$, we can push X inside conjunctions, disjunctions and F operators. However, over finite executions X cannot be pushed inside '¬' or 'G' operators. To see this, consider the execution $\sigma = \mathsf{hn}$. Observe that σ satisfies XGn, but not GXn as $|\sigma[1 :]| = 1$ and thus $\sigma[1 :] \not\models_f \mathsf{Xn}$. Similarly, $\tau = \mathsf{n}$ satisfies $\neg\mathsf{Xh}$ but not $\mathsf{X}\neg\mathsf{h}$. The normal form can be described by the following BNF grammar.

$$\begin{array}{lcl} \varphi & ::= & \psi \mid \eta \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \mathsf{F}\varphi \\ \eta & ::= & \mathsf{X}\psi \mid \mathsf{X}\eta \\ \psi & ::= & a \mid \neg\varphi \mid \mathsf{G}\varphi \end{array} \qquad (1)$$

Formulas in LTL[F, G, X] (φ) are one of X-formulas (η), G-formulas (ψ), conjunctions/disjunctions of LTL[F, G, X] formulas, or an F operator applied to an LTL[F, G, X] formula in the normal form. X-formulas are those where the top level operator is X. A X operator can only be applied to either a G-formula or an X-formula. Finally, G-formulas are letters, or negations of LTL[F, G, X] formulas (φ), or have G as the topmost operator. Every LTL[F, G, X] formula can be converted into this normal form, with at most quadratic blowup.

Example 3. Consider the LTL[F, G, X] formula $\varphi = \mathsf{X}(\mathsf{Gn} \lor \mathsf{Fh})$. The normal form for this can be obtained by pushing X as far inside as possible. Thus, $\varphi' = (\mathsf{XGn}) \lor (\mathsf{FXh})$ is the equivalent formula in normal form.

Remark. The fragment LTL[F, G, X] is expressively rich. For example, it can express properties in each class of the safety-progess classification of temporal properties introduced by Manna and Pnueli [32]. Among the pattern-based specifications introduced in [14], most patterns, except precedence, chain precedence, and scopes can be expressed in LTL[F, G, X]. In fact, among the 555 commonly occurring specifications collected and surveyed in [14], approximately 80% of them can be expressed in LTL[F, G, X].

Automata for LTL and its fragments. For LTL properties, algorithms for verification, runtime verification, and test oracle generation, all rely on the translation of logic formulas to automata. For a specification φ, the crucial step therefore, is the construction of an automaton $\mathcal{A}_\varphi$ such that an execution τ is accepted by $\mathcal{A}_\varphi$ if and only if τ satisfies φ. The size of $\mathcal{A}_\varphi$ has big influence on the complexity of the verification/testing algorithm. For runtime verification and test oracle generation, the automaton $\mathcal{A}_\varphi$ needs to be deterministic. Because of the critical role translations from LTL to automata play in algorithms, these have been well studied. Unfortunately, the translation from formulas to deterministic automata can result in at least a double exponential blowup. It is worth emphasizing that the result below holds whether we interpret LTL over finite or infinite executions.

Theorem 2.1 (Alur-LaTorre [6])**.** There is a family $\{\varphi_n\}_{n\in\mathbb{N}}$ of LTL[F, G, X] formulas such that the size of φ_n is n and any deterministic acceptor for φ_n is of size $\Omega(2^{2^n})$.

2.3 Compressed Executions

In this paper, we will present algorithms to solve the membership problem when the program execution is compressed. The compressed execution we consider will be encoded by a *straight line program (SLP)*, which is a special context-free grammar whose language has exactly one string, namely, the execution it represents. Several lossless compression schemes, like run-length encoding and Lempel-Ziv encoding [50] can be efficiently converted into SLPs of similar size. Several efficient algorithms that compress strings using SLPs are known [3, 21, 22, 25, 37, 48–50].

Straight Line Programs (SLP). Recall that a context-free grammar is a tuple $G = (T, N, S, R)$, where T is the set of *terminals*, N is the set of *non-terminals*, $S \in N$ is the *starting non-terminal*, and R is the set of *rules* of the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$. A straight line program (SLP) is special context-free grammar, with restrictions that ensure that the language associated with G has exactly one string. In particular, we require that in an SLP, each non-terminal A has exactly one rule where A appears on the left, and that there is a total ordering '<' on non-terminals such that if a non-terminal B appears on the right-hand side of the rule for A then $A < B$. These restrictions ensure that the language associated with each non-terminal has exactly one string, and we denote this string as $[\![A]\!]$.

The size of a rule $(A \to \alpha) \in R$ is defined to be the size $|\alpha|$ and the size of a grammar G, denoted by $|G|$, is defined to the be sum of the sizes of all rules. The compression ratio of a grammar $G = (T, N, S, R)$ is defined to be $\frac{|[\![S]\!]|}{|G|}$, i.e., the ratio between the length of the string that G represents, and the size of G.

Example 4. The grammar shown in Figure 2 is an SLP that encodes the execution $\tau = (\mathsf{hn})^{65}\mathsf{n}(\mathsf{hn})^{62}\mathsf{h}$ of program in Figure 1 from Example 1. The grammar has start symbol S. The rules are designed to ensure that $[\![A_i]\!] = (\mathsf{hn})^i$, $[\![B]\!] = \mathsf{n}(\mathsf{hn})^{62}\mathsf{h}$ and therefore $[\![S]\!] = \tau$.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
S & \rightarrow & A_{65}B & B & \rightarrow & \mathsf{n}C & C & \rightarrow & A_{62}\mathsf{h} \\
A_1 & \rightarrow & \mathsf{hn} & A_2 & \rightarrow & A_1A_1 & A_4 & \rightarrow & A_2A_2 \\
A_6 & \rightarrow & A_2A_4 & A_8 & \rightarrow & A_4A_4 & A_{16} & \rightarrow & A_8A_8 \\
A_{24} & \rightarrow & A_8A_{16} & A_{30} & \rightarrow & A_6A_{24} & A_{32} & \rightarrow & A_{16}A_{16} \\
A_{33} & \rightarrow & A_{32}A_1 & A_{62} & \rightarrow & A_{30}A_{32} & A_{65} & \rightarrow & A_{32}A_{33}
\end{array}$$

Figure 2: A straight line program encoding of the execution $\tau = (\mathsf{hn})^{65}\mathsf{n}(\mathsf{hn})^{62}\mathsf{h}$ from Example 1.

The SLP has 15 rules. All rules have size 2. Thus the size of the SLP is $2 \times 15 = 30$. The compression ratio is therefore $\frac{256}{30} \simeq 8.53$.

Every SLP can be transformed in linear time into Chomsky normal form where the size of each rule is bounded by 2. From now on we will assume that all the SLP grammars are written in Chomsky normal form.

We remark that an SLP can be exponentially more succinct than the uncompressed string it represents. For instance, a string $\sigma = \mathsf{h}^{2^n}$ of size 2^n can be represented by a grammar of size $O(n)$ with rules $S \rightarrow H_nH_n, H_n \rightarrow H_{n-1}H_{n-1}, \ldots, H_2 \rightarrow H_1H_1$ and $H_1 \rightarrow \mathsf{h}$.

The problem of checking if an execution represented by an SLP G satisfies an LTL formula φ has been studied before. A naïve algorithm for this problem involves *decompressing* the SLP G and checking if the uncompressed execution satisfies φ. However, this can be expensive because of the possible exponential succinctness of the SLP, as outlined in the previous paragraph. Is there a better algorithm that doesn't involve uncompressing?

Automata for checking SLPs against LTL. One possible approach, that works without decompressing the SLP, leverages the automata-theoretic connections of LTL. Given an LTL formula φ, we can first construct a deterministic finite automaton $\mathcal{A}_\varphi = (Q, \Sigma, q_0, \delta, F)$ as outlined in Section 2.2. For every non-terminal A in the SLP grammar G and for every state $q \in Q$, we can then inductively compute the state $\delta(q, \llbracket A \rrbracket)$ reached after running the string generated by A starting from the state q. In the base case, the rule corresponding to A is $A \rightarrow a$ ($a \in \Sigma$), and $\delta(q, \llbracket A \rrbracket) = \delta(q, a)$. In the inductive case of $A \rightarrow BC$, we have $\delta(q, \llbracket S \rrbracket) = \delta(\delta(q, \llbracket B \rrbracket), \llbracket C \rrbracket)$. Finally checking if $\llbracket A \rrbracket \models_f \varphi$ amounts to checking $\delta(q_0, \llbracket S \rrbracket) \in F$. This approach would work in time $O(|G| \cdot |\mathcal{A}| \cdot D)$, where D is the time taken to compute the transition function on an input symbol and state. Note that D, in general, is at least $\log |\mathcal{A}|$ because the bits encoding the state have to be read and the encoding of the next state needs to be produced. Based on Theorem 2.1, this is double exponential in $|\varphi|$ and thus intractable. Indeed, the following theoretical lower bound establishes that there is no algorithm that has a tractable asymptotic complexity for the case of full LTL.

Theorem 2.2 (Markey and Schnoebelen [33])**.** Given an SLP $G = (T, N, S, R)$ and an LTL formula φ, the problem of checking if $\llbracket S \rrbracket \models_f \varphi$ is PSPACE-hard.

The main result of this paper shows that this problem can be solved efficiently (in polynomial time) when the formula is from the LTL fragment LTL[F, G, X].

3 TECHNICAL OVERVIEW

Recall (from Section 2.3) that there is a simple automata-theoretic algorithm for checking if a compressed trace (SLP G) satisfies a given LTL formula φ (i.e., $\llbracket G \rrbracket \models_f \varphi$). This simple algorithm constructs a DFA $\mathcal{A}_\varphi$ corresponding to the given LTL formula φ (in time $O(|\mathcal{A}_\varphi|)$), and then inductively computes, for every non-terminal A of G and for every state q of $\mathcal{A}_\varphi$, the next state $q' = \delta(q, \llbracket A \rrbracket)$ obtained after running the trace fragment $\llbracket A \rrbracket$ on $\mathcal{A}_\varphi$. As we previously observed the total running time of this simple algorithm is $O(|\mathcal{A}_\varphi| + |G||\mathcal{A}_\varphi|D)$ (where D is the time to compute the transition function) which, based on the size of the smallest deterministic automaton, is $O(|G|2^{2^{|\varphi|}}2^{|\varphi|})$. The algorithm we propose for formulas in LTL[F, G, X] works on the same automata-theoretic paradigm but with modifications that lead to a polynomial running time. In this section we outline some of the ideas that help us achieve this polynomial time.

Backwards automaton. The first observation that our algorithm relies on is that for the fragment LTL[F, G, X], there is a deterministic automaton that works backwards and only suffers an exponential blow-up (instead of double exponential in the forward automaton). That is, for the input formula $\varphi \in$ LTL[F, G, X], we construct an automaton $\mathcal{A}^{\mathsf{r}}_\varphi$ such that for any trace τ, $\tau \models_f \varphi$ iff τ^{r} is accepted by $\mathcal{A}^{\mathsf{r}}_\varphi$; for an execution $\tau = \mathsf{e}_0\mathsf{e}_1 \cdots \mathsf{e}_{k-1}$ its *reverse* is $\tau^{\mathsf{r}} = \mathsf{e}_{k-1}\mathsf{e}_{k-2} \cdots \mathsf{e}_0$. The algorithm for analyzing the SLP G with this automaton is also straightforward, and proceeds as if the grammar G is reversed (every rule of the form $A \rightarrow BC$ becomes $A \rightarrow CB$). But, this by itself is not enough if we are using the exhaustive paradigm 'compute $\delta(q, \llbracket A \rrbracket)$ for all q and A' because $|\mathcal{A}^{\mathsf{r}}_\varphi| = O(2^{|\varphi|})$. Thankfully, the backwards automaton $\mathcal{A}^{\mathsf{r}}_\varphi$ enjoys a special structure that we exploit, in conjunction with the next observation, to get our efficient algorithm.

Bounding running time with visited states. We next observe that, instead of computing $\delta(q, \llbracket A \rrbracket)$ for all pairs of state q and non-terminal A in the automata-theoretic algorithm, we can afford to only compute $\delta(q, \cdot)$ for states that are actually *visited* (instead of all states). Consider the production rule $S \rightarrow UV$, where S is the starting non-terminal of the input SLP grammar G. Our final goal is to compute the state $q = \delta(q_0, \llbracket S \rrbracket)$. We remark that this can be computed as the composition $q = \delta(q', \llbracket V \rrbracket)$, where $q' = \delta(q_0, \llbracket U \rrbracket)$. If this is the only rule that V occurs in, we only need to compute $\delta(q', \llbracket V \rrbracket)$ (instead of computing $\delta(p, \llbracket V \rrbracket)$ for every state p). Notice that this intermediate state q' would also be visited when running $\mathcal{A}$ on the uncompressed trace $\llbracket S \rrbracket$; this is precisely the state reached after running the prefix $\llbracket U \rrbracket$ of $\llbracket S \rrbracket$. In fact, this observation can be generalized so that we only compute $\delta(q, A)$ for those states q that are ever visited when analyzing the uncompressed trace. We formalize this as follows. For a trace τ and an automaton $\mathcal{A}$ with initial state q_0, let $\mathsf{v}(\mathcal{A}, \tau) = \{\delta(q_0, \tau[: i]) \mid 0 \leq i < |\tau|\}$ be the states of $\mathcal{A}$ that are visited when running τ on $\mathcal{A}$. Then, we can compute $\delta(q_0, S)$ by only computing $\delta(q, A)$ for every non-terminal A and state $q \in \mathsf{v}(\mathcal{A}, \llbracket G \rrbracket)$. This gives us an upper bound of $O(|G| \cdot |\mathsf{v}(\mathcal{A}, \llbracket G \rrbracket)| \cdot D)$, an improvement over $O(|G| \cdot |\mathcal{A}| \cdot D)$ (D denotes the time to evaluate the transition function).

Bounding number of states visited. Our third important observation is that any run of automaton $\mathcal{A}^{\mathsf{r}}_\varphi$ (for $\varphi \in$ LTL[F, G, X]) satisfies a "monotonicity" property. This property allows us to bound the number of states visited $|\mathsf{v}(\mathcal{A}, \llbracket G \rrbracket)|$ on any input to $|\varphi||\Sigma|^k$ (independent of $||G||$). Here, k is what we call the *nesting* *depth* of X in φ; a precise definition will be presented in Section 4. This combined with some other observations gives us a running time of $O(|G||\varphi|^2|\Sigma|^k)$.

Further improvements. Our algorithm for analyzing compressed executions runs in strictly polynomial time (and thus does not have the exponential dependence due to the factor $|\Sigma|^k$). We achieve this by performing an involved fine grained analysis of the running time of the algorithm. Further, we also make use of the monotonicity property outlined in the previous paragraph to optimize the space usage of the algorithm.

In Section 4, we describe the automaton construction, and the other observations about monotonicity and the number of visited states in greater detail. We finally present the algorithm for analyzing compressed executions and improvements thereof in Section 5.

4 AUTOMATON FOR LTL[F, G, X]

In this section, we present the construction of a *backwards* deterministic automaton $\mathcal{A}^{\mathsf{r}}_{\varphi}$ such that $\mathcal{A}^{\mathsf{r}}_{\varphi}$ accepts a trace τ^{r} if and only if $\tau \models_f \varphi$; recall that τ^{r} is the reverse of execution τ. The main advantage of this construction is that the size of $\mathcal{A}^{\mathsf{r}}_{\varphi}$ is only exponential in φ (as opposed to doubly exponential). In addition, $\mathcal{A}^{\mathsf{r}}_{\varphi}$ has a special structure that ensures that the number of states visited by $\mathcal{A}^{\mathsf{r}}_{\varphi}$ on any input τ^{r} is polynomial in $|\varphi|$.

Before presenting the construction, we will introduce some conventions and notations that we will use in the rest of this paper. First, as outlined in Section 2, we can assume that LTL[F, G, X] formulas are in normal form given by Equation (1), i.e., X operators have been pushed as far inside as possible. For any formula $\varphi \in$ LTL[F, G, X], we will use $\mathsf{sub}(\varphi)$ to denote the set of sub-formulas of φ. When defining our automata, we will consider a special subset of sub-formulas, called FGX-subformulae, denoted by $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$. These are sub-formulas ψ of φ whose topmost operator is either F, G, or X, and if the top operator of ψ is X, then ψ has a sub-formula with topmost operator G. We formally define this set next.

Definition 1 (FGX-sub-formulas). For a formula $\varphi \in$ LTL[F, G, X], $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$ is the set of sub-formulas defined inductively as follows.

$$\begin{array}{rcll}
\mathsf{sub}_{\mathsf{FGX}}(a) & = & \varnothing & \\
\mathsf{sub}_{\mathsf{FGX}}(\neg\varphi) & = & \mathsf{sub}_{\mathsf{FGX}}(\varphi) & \\
\mathsf{sub}_{\mathsf{FGX}}(\varphi_1 \oplus \varphi_2) & = & \mathsf{sub}_{\mathsf{FGX}}(\varphi_1) \cup \mathsf{sub}_{\mathsf{FGX}}(\varphi_2), & \oplus \in \{\wedge, \vee\} \\
\mathsf{sub}_{\mathsf{FGX}}(\mathcal{M}\varphi) & = & \{\mathcal{M}\varphi\} \cup \mathsf{sub}_{\mathsf{FGX}}(\varphi), & \mathcal{M} \in \{\mathsf{F}, \mathsf{G}\} \\
\mathsf{sub}_{\mathsf{FGX}}(\mathsf{X}\varphi) & = & \mathsf{sub}_{\mathsf{FGX}}(\varphi), & \varphi \in \mathrm{LTL}[\mathsf{X}] \\
\mathsf{sub}_{\mathsf{FGX}}(\mathsf{X}\varphi) & = & \{\mathsf{X}\varphi\} \cup \mathsf{sub}_{\mathsf{FGX}}(\varphi), & \varphi \notin \mathrm{LTL}[\mathsf{X}]
\end{array}$$

Let us look at examples to illustrate these definitions.

Example 5. Consider the formula $\varphi = \neg\mathsf{n} \wedge \mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n})$ from Example 2. The set of its sub-formulas is $\mathsf{sub}(\varphi) = \{\varphi, \neg\mathsf{n}, \mathsf{n}, \mathsf{G}\big((\neg\mathsf{n}) \vee \neg(\mathsf{X}\mathsf{n})\big), (\neg\mathsf{n}) \vee \neg(\mathsf{X}\mathsf{n}), \neg\mathsf{X}\mathsf{n}, \mathsf{X}\mathsf{n}\}$. Similarly, $\mathsf{sub}_{\mathsf{FGX}}(\varphi) = \{\mathsf{G}((\neg\mathsf{n}) \vee \neg(\mathsf{X}\mathsf{n}))\}$. Notice that $\mathsf{X}\mathsf{n} \notin \mathsf{sub}_{\mathsf{FGX}}(\varphi)$ (even though its topmost operator is X) as it does not have a G-sub-formula in its scope.

As is standard in automata constructions for LTL, our automaton $\mathcal{A}^{\mathsf{r}}_{\varphi}$ for φ will track the truth of sub-formulas of φ as it processes the input. Instead of tracking the truth of *all* sub-formulas, our automaton will only track the truth of sub-formulas in $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$. Since $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$ is smaller than $\mathsf{sub}(\varphi)$ (as illustrated by Example 5), this results in smaller automata and better performance in practice. But this is not our only reason for tracking fewer sub-formulas. As we will show towards the end of this section, tracking the truth of fewer sub-formulas reveals that every run of the automaton is "monotonic", which can then be exploited to argue that the number of states visited in the run of $\mathcal{A}^{\mathsf{r}}_{\varphi}$ on any string is small.

Let us fix $\varphi \in$ LTL[F, G, X]. The states of our automaton $\mathcal{A}^{\mathsf{r}}_{\varphi}$ will keep track of which sub-formulas in $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$ are true and which ones are not, on the input seen so far. Thus a state is essentially a *valuation* $h : \mathsf{sub}_{\mathsf{FGX}}(\varphi) \to \{\top, \bot\}$ over φ. We use Val_{φ} to denote the set of all valuations over φ.

While keeping track of the truth of sub-formulae is necessary, it is not sufficient. In order to determine truth of formulas in LTL[X] like $\mathsf{XX}a$, the automaton will additionally also keep track of the last few events seen, in its control state. How many events need to be tracked depends on the number of X operators that appear in LTL[X] sub-formulas of φ. Recall that we are assuming that Xs have been pushed as far in as possible in φ.

For $\psi \in$ LTL[X], define $\mathsf{Xdepth}(\psi)$ to be the nesting depth of X operators in ψ. And more generally, for $\varphi \in$ LTL[F, G, X], we define $\mathsf{Xdepth}(\varphi) = \max\{\mathsf{Xdepth}(\psi) \mid \psi \in \mathsf{sub}(\varphi) \cap \mathrm{LTL}[\mathsf{X}]\}$. For example, LTL[X] sub-formulas of $\mathsf{XG}(\mathsf{h} \implies (\mathsf{XXXn}))$ are h and the sub-formulas of XXXn. Thus, $\mathsf{Xdepth}(\mathsf{XG}(\mathsf{h} \implies \mathsf{XXXn})) = 3$. On the other hand, since the only LTL[X] sub-formulas of $\mathsf{h} \wedge (\mathsf{XGn})$ are h and n, $\mathsf{Xdepth}(\mathsf{h} \wedge (\mathsf{XGn})) = 0$.

To compute the next state h' obtained after reading a symbol e in state h, the automaton needs to update the truth of all sub-formulas in $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$. It turns out that we can, in fact, compute the truth of all sub-formulas in $\mathsf{sub}(\varphi)$ (and thus the valuation h') solely by looking at h, e, the formula φ and the last k events seen in the trace, where $k = \mathsf{Xdepth}(\varphi)$. This definition (of how truth of $\mathsf{sub}(\varphi)$ is updated) is critical not only in defining the automaton but also in stating its correctness. We present this definition before giving the formal definition of $\mathcal{A}^{\mathsf{r}}_{\varphi}$. In this definition, we will use $\Sigma^{\leq k}$ to denote the set of all sequences over Σ of length at most k.

Definition 2. Let $\varphi \in$ LTL[F, G, X], $h \in \mathsf{Val}_{\varphi}$ and $\mathsf{buf} \in \Sigma^{\leq k}$. For any event $\mathsf{e} \in \Sigma$, $\mathsf{post}(h, \mathsf{buf}, \mathsf{e}) : \mathsf{sub}(\varphi) \to \{\top, \bot\}$ is defined inductively as follows.

$$\begin{array}{rcl}
\mathsf{post}(h,\mathsf{buf},\mathsf{e})(a) & = & (a = \mathsf{e}) \\
\mathsf{post}(h,\mathsf{buf},\mathsf{e})(\neg\phi) & = & \neg\Big(\mathsf{post}(h,\mathsf{buf},\mathsf{e})(\phi)\Big) \\
\mathsf{post}(h,\mathsf{buf},\mathsf{e})(\phi_1 \oplus \phi_2) & = & \oplus \begin{array}{l} \mathsf{post}(h,\mathsf{buf},\mathsf{e})(\phi_1) \\ \mathsf{post}(h,\mathsf{buf},\mathsf{e})(\phi_2) \end{array} \text{ if } \oplus \in \{\wedge, \vee\} \\
\mathsf{post}(h,\mathsf{buf},\mathsf{e})(\mathsf{G}\phi) & = & h(\mathsf{G}\phi) \wedge \mathsf{post}(h,\mathsf{buf},\mathsf{e})(\phi) \\
\mathsf{post}(h,\mathsf{buf},\mathsf{e})(\mathsf{F}\phi) & = & h(\mathsf{F}\phi) \vee \mathsf{post}(h,\mathsf{buf},\mathsf{e})(\phi) \\
\mathsf{post}(h,\mathsf{buf},\mathsf{e})(\mathsf{X}\phi) & = & \begin{cases} \big(\mathsf{buf} \models_f \phi\big) & \mathsf{X}\phi \in \mathrm{LTL}[\mathsf{X}] \\ h(\phi)^{3} & \text{otherwise} \end{cases}
\end{array}$$

Having outlined the basic intuition behind the construction of $\mathcal{A}^{\mathsf{r}}_{\varphi}$, we are ready to present its formal definition. In the following, for a function $f : A \to B$ and set $C \subseteq A$, we denote by $f\restriction_C$ the restriction of f to the domain C.

[3] This definition assumes e is at least the second event read. $\mathsf{post}(h, \mathsf{buf}, \mathsf{e})(\mathsf{X}\phi) = \bot$ if e is the first event.

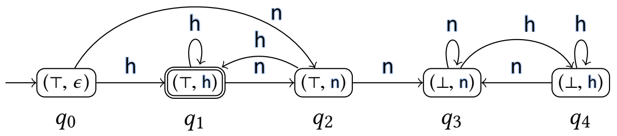


Figure 3: Automaton for the formula $\varphi = \neg\mathsf{n} \wedge \mathbf{G}(\mathsf{n} \implies \neg\mathbf{X}\mathsf{n})$.

Definition 3 (Automaton for LTL[F, G, X]). For $\varphi \in \text{LTL}[\mathsf{F}, \mathsf{G}, \mathsf{X}]$ with $\mathsf{Xdepth}(\varphi) = k$ and event set Σ, the DFA $\mathcal{A}^{\mathsf{r}}_{\varphi} = (Q, \Sigma, q_0, \delta, F)$ is defined as follows.

- The states in Q are triples of the form (h, b, buf) where $h \in \mathsf{Val}_{\varphi}$, $b \in \{\top, \bot\}$ and $\mathsf{buf} \in \Sigma^{\leq k}$. Intuitively, h tracks the truth of FGX-sub-formulas of φ while b tracks whether φ is true on the input read so far. Additionally, buf stores the last k symbols read by the automaton thus far.
- The initial state q_0 is $(h_0, \bot, \epsilon)$ where for every $\mathsf{G}\psi, \mathsf{F}\psi \in \mathsf{sub}_{\mathsf{FGX}}(\varphi)$, $h_0(\mathsf{F}\psi) = \bot$ and $h_0(\mathsf{G}\psi) = \top$.
- The transition function δ is given as follows: $\delta((h, b, \mathsf{buf}), \mathsf{e}) = (\mathsf{post}(h, \mathsf{buf}, \mathsf{e})\restriction_{\mathsf{sub}_{\mathsf{FGX}}(\varphi)}, \mathsf{post}(h, \mathsf{buf}, \mathsf{e})(\varphi), \mathsf{buf}')$, where buf' is the prefix of length k of the concatenated sequence $\mathsf{e} \cdot \mathsf{buf}$.
- The final states $F = \{(h, \top, \mathsf{buf}) \mid h \in \mathsf{Val}_{\varphi}$ and $\mathsf{buf} \in \Sigma^{\leq k}\}$.

Let us illustrate the automaton construction with an example.

Example 6. Consider the formula $\varphi = \neg\mathsf{n} \wedge \mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n})$ from Example 2. The backwards automaton $\mathcal{A}^{\mathsf{r}}_{\varphi}$ for φ is shown in Figure 3; the alphabet is assumed to be $\Sigma = \{\mathsf{h}, \mathsf{n}\}$. The set of sub-formulas $\mathsf{sub}_{\mathsf{FGX}}(\varphi) = \{\mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n})\}$ is singleton, and thus there are 2 valuations in Val_{φ}. Further, $\mathsf{Xdepth}(\varphi) = 1$ and thus the buffer size is at most 1. The states of $\mathcal{A}^{\mathsf{r}}$ are triples (h, b, buf), where $h \in \mathsf{Val}_{\varphi}$, $b \in \{\top, \bot\}$ and $\mathsf{buf} \in \{\mathsf{h}, \mathsf{n}, \epsilon\}$. In the figure, we only show the first component (valuation h) and the third component (the buffer) of the state. Since there is only one formula in $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$, we write the valuation h as the truth value it maps the sub-formula to. The component b can be inferred from the figure — $b = \top$ in a state iff the state is an accepting state (state q_1).

Now consider the traces $\tau_1 = \mathsf{hhnn}$ and $\tau_2 = \mathsf{hhn}$. Observe that the automaton rejects the trace $\tau_1^{\mathsf{r}} = \mathsf{nnhh}$ but accepts $\tau_2^{\mathsf{r}} = \mathsf{nhh}$ as $\tau_1 \not\models_f \varphi$ but $\tau_2 \models_f \varphi$.

The correctness proof of the automaton construction in Definition 3 relies on the following technical lemma which says that the automaton correctly computes the truth of every sub-formula. It can be proved using an easy induction on $|\tau|$ and structural induction on the formula.

Lemma 4.1. *For $\varphi \in \text{LTL}[\mathsf{F}, \mathsf{G}, \mathsf{X}]$, let $\mathcal{A}^{\mathsf{r}}_{\varphi} = (Q, \Sigma, q_0, \delta, F)$ be the DFA as given in Definition 3. For any execution $\tau = \mathsf{e}\sigma$ where $\mathsf{e} \in \Sigma$, for any $\psi \in \mathsf{sub}(\varphi)$, $\tau \models_f \psi$ if and only if $\mathsf{post}(\delta(q_0, \sigma^{\mathsf{r}}), \mathsf{e})(\psi) = \top$.*

We can now state the correctness of our automaton construction.

Theorem 4.1. *Let $\varphi \in \text{LTL}[\mathsf{F}, \mathsf{G}, \mathsf{X}]$ and $\mathcal{A}^{\mathsf{r}}_{\varphi}$ be the DFA given in Definition 3. For any execution τ, $\tau \models_f \varphi$ if and only if $\tau^{\mathsf{r}} \in L(\mathcal{A}^{\mathsf{r}}_{\varphi})$.*

Proof. Let $\tau = \mathsf{e}\sigma$. Observe that $\tau^{\mathsf{r}} \in L(\mathcal{A}^{\mathsf{r}}_{\varphi})$ iff $\delta(q_0, \tau^{\mathsf{r}}) = (h, \top)$ for some h. From the definition of the transition function δ, this is equivalent to $\mathsf{post}(\delta(q_0, \sigma^{\mathsf{r}}), \mathsf{e})(\varphi) = \top$. From Lemma 4.1, this is the same as $\tau \models_f \varphi$ and thereby establishing the theorem. □

Size of $\mathcal{A}^{\mathsf{r}}_{\varphi}$. Observe that the number of states of the automaton $\mathcal{A}^{\mathsf{r}}_{\varphi}$ is $2^{|\mathsf{sub}_{\mathsf{FGX}}(\varphi)|+1}|\Sigma|^k$, where $k = \mathsf{Xdepth}(\varphi)$. Since $\mathsf{sub}_{\mathsf{FGX}}(\varphi) \subseteq \mathsf{sub}(\varphi)$ and $|\mathsf{sub}(\varphi)| \leq |\varphi|$, the number of states is $O(2^{|\varphi|}|\Sigma|^k)$.

We next argue that though $\mathcal{A}^{\mathsf{r}}_{\varphi}$ has $O(2^{|\varphi|}|\Sigma|^k)$ states, in any run, it goes through at most $O(|\varphi||\Sigma|^k)$ states. This is based on the observation that state changes in $\mathcal{A}^{\mathsf{r}}_{\varphi}$ are *monotonic*.

Consider two valuations $g, h : \mathsf{sub}_{\mathsf{FGX}}(\varphi) \to \{\top, \bot\}$. We will say that $g \preceq h$ if for every $\mathsf{F}\psi \in \mathsf{sub}_{\mathsf{FGX}}(\varphi)$, if $g(\mathsf{F}\psi) = \top$ then $h(\mathsf{F}\psi) = \top$, and for every $\mathcal{M}\psi \in \mathsf{sub}_{\mathsf{FGX}}(\varphi)$ where $\mathcal{M} \in \{\mathsf{X}, \mathsf{G}\}$, if $g(\mathcal{M}\psi) = \bot$ then $h(\mathcal{M}\psi) = \bot$.

Lemma 4.2. *For $\varphi \in \text{LTL}[\mathsf{F}, \mathsf{G}, \mathsf{X}]$, let $\mathcal{A}^{\mathsf{r}}_{\varphi} = (Q, \Sigma, q_0, \delta, F)$ be the DFA defined in Definition 3. Let $u \in \Sigma^*$ and let states $(h_1, b_1, \mathsf{buf}_1)$ and $(h_2, b_2, \mathsf{buf}_2)$ be such that $\delta((h_1, b_1, \mathsf{buf}_1), u) = (h_2, b_2, \mathsf{buf}_2)$. Then, $h_1 \preceq h_2$.*

The proof of Lemma 4.2 follows from the definition of the transition function δ and induction on the length of u.

Lemma 4.2 establishes that once the valuation component of the state changes, you never revisit the same valuation. Since the assignment to any $\psi \in \mathsf{sub}_{\mathsf{FGX}}(\varphi)$ can change at most once, the number of valuations visited in any run is bounded by $|\mathsf{sub}_{\mathsf{FGX}}(\varphi)|$, thereby giving a bound on the number of states visited in any run:

Corollary 4.1. *Let $\varphi \in \text{LTL}[\mathsf{F}, \mathsf{G}, \mathsf{X}]$ be a formula over Σ. The DFA $\mathcal{A}^{\mathsf{r}}_{\varphi}$ visits $O(|\varphi||\Sigma|^k)$ distinct states on any input trace $\tau \in \Sigma^*$*

5 MONITORING COMPRESSED TRACES AGAINST LTL[F, G, X]

We will now present our main result — an efficient algorithm to check, given an SLP $G = (T, N, S, R)$ and $\varphi \in \text{LTL}[\mathsf{F}, \mathsf{G}, \mathsf{X}]$, if $[\![S]\!] \models_f \varphi$. Our algorithm follows the template algorithm for SLPs outlined in Section 2.3. That is, we will "run" the automaton $\mathcal{A}^{\mathsf{r}}_{\varphi}$ (Definition 3) on the uncompressed trace $[\![S]\!]$, *without explicitly uncompressing the SLP*. This can be accomplished by computing, for every state q and non-terminal $A \in N$, the state $\delta(q, [\![A]\!]^{\mathsf{r}})$ [4], and then finally checking $\delta(q_0, [\![S]\!]^{\mathsf{r}}) \in F$, where q_0 is the initial state and F is the set of final states of $\mathcal{A}^{\mathsf{r}}_{\varphi}$. As pointed out in Section 2.3, this runs in $O(|G||\mathcal{A}^{\mathsf{r}}_{\varphi}|D)$ time (where D is the time to compute the transition function), which given the description of $\mathcal{A}^{\mathsf{r}}_{\varphi}$, is $O(|G||\varphi|2^{|\varphi|}|\Sigma|^k)$, where $k = \mathsf{Xdepth}(\varphi)$. Now, we can observe that it is not necessary to compute $\delta(q, [\![A]\!]^{\mathsf{r}})$ for every state q, but only for the states visited during a run of $\mathcal{A}^{\mathsf{r}}_{\varphi}$ on $[\![S]\!]^{\mathsf{r}}$. This can be accomplished if we used a "on-the-fly" algorithm for the $\delta(q, [\![A]\!]^{\mathsf{r}})$ computations. For such an algorithm, given the monotonicity properties of $\mathcal{A}^{\mathsf{r}}_{\varphi}$ (Lemma 4.2) and the resulting bound on the number of visited states (Corollary 4.1), we can improve the running time to $O(|G||\varphi|^2|\Sigma|^k)$.

The main observation in this section is that this "on-the-fly" algorithm in fact has a running time that is polynomial in the size of G and φ. This requires us examine this algorithm in some detail, and analyze its running time carefully.

Recall that a state of $\mathcal{A}^{\mathsf{r}}_{\varphi}$ is of the form (h, b, buf) where $h \in \mathsf{Val}_{\varphi}$, $b \in \{\top, \bot\}$ is a Boolean recording the truth of φ, and buf is the buffer tracking the last k symbols read. Now, consider a non-terminal A,

[4] Recall that $\mathcal{A}^{\mathsf{r}}_{\varphi}$ runs the execution in reverse.

and state (h, b, buf). Let $(h', b', \mathsf{buf}') = \delta((h, b, \mathsf{buf}), [\![A]\!]^{\mathsf{r}})$. Based on the Definition 2, we know that the value of the Boolean b does not influence the values of h', b', and buf'. This Boolean b is only needed to determine if the last state (i.e., $\delta(q_0, [\![S]\!]^{\mathsf{r}})$) is a final state, This can alternatively be determined from the valuation h' at the end and buffer buf' using a function analogous to post (Definition 2); we skip giving this definition. Next buf' is nothing but the prefix of length k of the concatenated string $[\![A]\!] \cdot \mathsf{buf}$, which can be computed in an inductive manner based on the rules in the grammar. In the interests of space, we don't give how buf' can be computed but we assume we have a function updateBuffer(A, buf) which returns the prefix of length k of $[\![A]\!] \cdot \mathsf{buf}$.

Algorithm 1: Compute state of automaton $\mathcal{A}^{\mathsf{r}}_{\varphi}$ after reading the string $[\![A]\!]^{\mathsf{r}}$

```
1  function postState($A$, $h$, buf)
2      visited ← visited ∪{($A$, buf)}
3      if $A \to$ e then
4          if (e, buf) ∈ visited then return $h$
5          else /* (e, buf) ∉ visited */
6              visited ← visited ∪ {(e, buf)}
7              $h' \leftarrow \delta((h, \bot, \mathsf{buf}), \mathsf{e})$
8              if $h' \neq h$ then visited ← ∅
9              return $h'$
10     else /* $A \to BC$ */
11         if ($C$, buf) ∈ visited then $h' \leftarrow h$
12         else /* ($C$, buf) ∉ visited */
13             $h'$ ← postState($C$, $h$, buf)
14         buf ← updateBuffer($C$, buf)
15         if ($B$, buf) ∈ visited then return $h'$
16         else /* ($B$, buf) ∉ visited */
17             $h''$ ← postState($B$, $h'$, buf$'$)
18             return $h''$
```

The critical function is really the computation of h' given non-terminal A, valuation h and buffer buf. A pseudocode for this function postState is given in Algorithm 1. We will call postState with arguments A, h, buf only once. After the first call we will memoize this result, and if in subsequent computations, there is a need to compute postState(A, h, buf) we will use the stored result. The data structure storing these previously computed postState results is called visited in Algorithm 1. Observe that monotonicity properties of $\mathcal{A}^{\mathsf{r}}_{\varphi}$ (Lemma 4.2) mean that if the valuation h in the state changes during an execution, the automaton never returns to the same valuation again. Hence, visited just stores the previous calls for the *current* valuation h; as soon as the valuation changes, we reset the data structure visited because the previous calls to postState will never be repeated as h has changed. Moreover, this means that visited only stores pairs (A, buf) when a call to postState (A, h, buf) returns the valuation h.

In line 2, we record the fact that we have made a call postState(A, h, buf) by adding (A, buf) to visited. The computation then proceeds based on the rule for the non-terminal A. If $A \to$ e (e ∈ Σ) is the rule, then we return h if we have computed it before (line 4) or find the new valuation by computing the transition function δ. Note that visited is set to ∅ if the valuation changes (line 8). On the other hand, if the rule is $A \to BC$ (lines 10 through 17), then we compute the result by "running" C and then B.

h_0: [(S, ϵ), (B, ϵ), (C, ϵ), (h, ϵ), (A_{62}, h), (A_{32}, h), (A_{16}, h), (A_8, h), (A_4, h), (A_2, h), (A_1, h), (n, h), (h, n), (A_{30}, h), (A_{24}, h), (A_6, h), (A_{65}, n), (A_{33}, n), (A_1, n), (n, n)]

h_1: [(h, n), (A_{32}, h), (A_{16}, h), (A_8, h), (A_4, h), (A_2, h), (A_1, h) (n, h), (h, n)]

Figure 4: Executing Algorithm 1 on SLP in Figure 2 with property $\varphi = \neg\mathsf{n} \wedge \mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n})$. The valuation h_0 corresponds to $[\mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n}) \mapsto \top]$ and h_1 is $[\mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n}) \mapsto \bot]$. The figure shows the set visited for each valuation. Elements are added to visited from left to right as recursive calls are made. Initially the valuation is h_0, which changes to h_1 when $\delta((h_0, \bot, \mathsf{n}), \mathsf{n})$ is computed.

Example 7. Consider the trace τ from Example 1 of the program in Figure 1. Its compression as an SLP is given in Figure 2. Let us fix the property to be checked to be $\varphi = \neg\mathsf{n} \wedge \mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n})$ from Example 2. Let us see how Algorithm 1 evaluates φ on this SLP.

Recall that $\mathsf{sub}_{\mathsf{FGX}}(\varphi) = \{\mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n})\}$. Thus there are two valuations $h_0 = [\mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n}) \mapsto \top]$ and $h_1 = [\mathsf{G}(\mathsf{n} \implies \neg\mathsf{X}\mathsf{n}) \mapsto \bot]$, with h_0 being the initial valuation. Next, recall that since $\mathsf{Xdepth}(\varphi) = 1$, the buffer is of size at most 1.

The algorithm starts with a call postState (S, h_0, ϵ) (i.e., buffer is empty) which initiates a sequence of recursive calls to postState with different arguments. Initially visited is empty. Line 2 of Algorithm 1 adds (S, ϵ) to visited, and given the rule for S, makes a recursive call postState (B, h_0, ϵ) (lines 11 to 13). The set visited memoizes the result of a call to postState to avoid re-computing an answer — if (A, buf) ∈ visited after returning from a call postState (A, h, buf) then it means that the valuation after running $[\![A]\!]$ from valuation h with buffer buf is h. Figure 4 shows the set visited. The first row shows the set visited while the current valuation is h_0, and the second row shows visited after the valuation changes to h_1. Recursive calls to postState add elements to visited in the order shown from left to right.

The call postState (B, h_0, ϵ) leads to a call postState (C, h_0, ϵ) which then results in a check of $\delta((h_0, \bot, \epsilon), \mathsf{h})$ (lines 6 through 9). Since $\delta((h_0, \bot, \epsilon), \mathsf{h}) = h_0$ (see automaton in Figure 3), visited is not reset. The buffer changes to h (line 14), and a call is made to postState (A_{62}, h_0, h) (line 17). This results in a sequence of recursive calls with non-terminals A_i for values $i < 62$. To see how storing results in visited helps, let us see what happens during the call postState (A_2, h_0, h). A recursive call postState (A_1, h_0, h) is made (line 13) which returns h_0; running the automaton in Figure 3 on $[\![A_1]\!]$ from h_0 leaves the valuation unchanged. The updated buffer (line 14) remains h, and since (A_1, h) ∈ visited (because of the previous call), we do not make additional recursive calls to postState (line 17). Such savings in calls to postState happen in many of the calls involving the non-terminals A_i.

After the recursive call postState (B, h_0, ϵ) returns h_0 (no change to valuation), the algorithm will make a call to postState (A_{65}, h_0, n), since the buffer after reading $[\![B]\!]$ is n. This leads to calls postState

$(A_{33}, h_0, \mathsf{n})$ and postState (A_1, h_0, n) as well as a computation of $\delta((h_0, \perp, \mathsf{n}), \mathsf{n})$. The valuation returned by δ is now h_1, which leads to visited being reset to $\emptyset$ (line 8). The evolution of the set visited after the change of valuation is shown in the second row of Figure 4.

After the call postState $(A_{65}, h_0, \mathsf{n})$ returns with valuation h_1, we need to evaluate whether φ holds. It turns out that given a valuation of each formula in $\mathsf{sub}_{\mathsf{FGX}}(\varphi)$ (namely h_1) and the buffer after A_{65} (namely h), we can compute the truth valuation for *all* subformulas of φ, including φ itself. Details of this process are omitted in the interests of space. Carrying this computation out, we discover that φ is not true and hence the execution encoded by the SLP in Figure 2 does not satisfy φ.

Running time. The running time for each call to postState is dominated by either the time taken for line 7 or for line 14. This is because if we make recursive calls to postState (lines 13 and 17) that time can be ascribed to those recursive calls. Line 7 takes at most time $O(|\varphi|)$ while line 14 takes $O(k)$ time (recall $k = \mathsf{Xdepth}(\varphi)$). Thus, each call to postState takes $O(|\varphi|)$ time. The number of possible calls to postState is at most the number of triples (A, h, buf) which is $|G||\varphi||\Sigma|^k$. Thus, the total running time can be bounded by $O(|G||\varphi|^2|\Sigma|^k)$. This bound has an exponential dependence on $k = O(|\varphi|)$ and we will show that this can be improved.

The key to improving the bound is to do a more careful count of the number of postState calls. Monotonicity (Lemma 4.2) ensures that there are at most $|\varphi|$ different valuations h. Therefore, for any fixed valuation h, we will try to bound the number of pairs (A, buf) that can arise as arguments in a call to postState with h as the valuation. Our observation is that this is much less than $|G||\Sigma|^k$. This is because if (A, h, buf) is an argument to postState, then $[\![A]\!]\text{buf}$ must be a substring of $[\![S]\!]$. Let us fix the uncompressed string, i.e., $[\![S]\!]$, to be τ. As a first step towards counting such pairs (A, buf), we define the notion of when a non-terminal C is *responsible* for generating the pair (A, buf).

Definition 4. A non-terminal C is said to be *responsible* for a substring $\tau[i : j]$ of τ if C is the label of the lowest internal node of the parse tree for τ that has $\tau[i : j]$ as a substring.

Similarly, C is responsible for pair (A, buf) if C is responsible for some occurrence of the string $[\![A]\!] \cdot \text{buf}$ (which is a substring of τ).

Observe that all nodes labeled C are responsible for the same set of pairs (A, buf). This is because such pairs are completely determined by the parse tree with root labeled C. Moreover, there is some non-terminal that is responsible for each pair (A, buf). Thus, we can upper bound the number of pairs (A, buf) by counting the number of pairs each non-terminal C is responsible for. Lemma 5.1 presents one such bound, and its proof is presented in the Appendix.

Lemma 5.1. A non-terminal C is responsible for at most $O(\mathrm{H}(C) + k)$ pairs; here $\mathrm{H}(C)$ is the height of the parse tree whose root is labeled C.

Taking $\mathrm{H}(G)$ to denote the height of the grammar (or $\mathrm{H}(S)$), we can use Lemma 5.1 to get the following bound on the running time.

Theorem 5.1. Given an SLP G with start symbol S and formula $\varphi \in \mathrm{LTL}[\mathsf{F}, \mathsf{G}, \mathsf{X}]$, the problem of determining if $[\![S]\!] \models_f \varphi$ can be solved in time $O(|G|(\mathrm{H}(G) + k)|\varphi|^2)$.

Theorem 5.1 follows from observing that Lemma 5.1 shows that the number of calls to postState is bounded by $O(|G|(\mathrm{H}(G)+k)|\varphi|)$ and the running time of each call to postState is at most $O(|\varphi|)$.

6 EXPERIMENTAL EVALUATION

We gauge the feasibility of our proposed approach of monitoring compressed execution traces by comparing the performance of our algorithm against that of the standard approach of monitoring traces without compressing them. The goals of our evaluation are:

(1) **Compression ratios.** The asymptotic runtime of our algorithm varies *quadratically* with the size of the compressed trace (Theorem 5.1). As a result, any speed up (over analysis of uncompressed traces) will evidently only be because of good compression ratios. We, therefore, want to evaluate whether execution traces from real world software projects can be compressed efficiently.
(2) **Performance of algorithm.** Our next goal is to understand how the running time varies with the size of the compressed trace (SLP) in practice. Further, in order to evaluate the practical feasibility of our approach, we want to evaluate whether our algorithm for analyzing compressed traces performs better than the standard approach of analyzing (uncompressed) traces directly, by a good margin. Finally, we want to understand how the speed up varies with factors such as compression ratio.

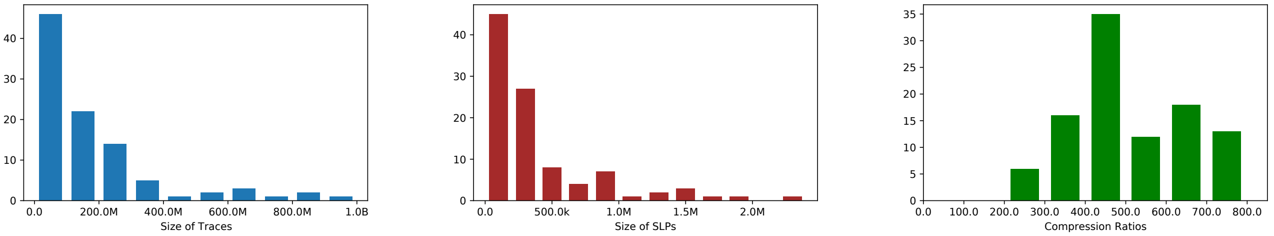
We next describe our implementation and experimental setup (Section 6.1) and then discuss our evaluation results (Section 6.2).

6.1 Implementation and Setup

The broad outline of our experimental setup is as follows. For our set of benchmark programs, we extract execution traces using an off-the-shelf logging tool. We then compress these traces as straight line programs (SLPs) and analyze the SLPs thus generated using our algorithm detailed in Section 5. We also compare the running time of our algorithm with the time it takes to analyze the original uncompressed traces using the standard approach (i.e., running against finite state automata corresponding to our LTL specs).

Implementation. We implemented our algorithm in our tool ZipMOP [4], primarily written in Java (in about 500 LoC). We use JavaMOP [1, 11, 34] for extracting execution traces. JavaMOP instruments a Java program under test and adds monitoring code at each event of interest for checking if the program’s executions meet some formal specification. For our experimental evaluation, we obtained execution traces by modifying JavaMOP so that it logs events to a file. To analyze uncompressed traces against LTL[F, G, X] properties, we use Rabinizer-4.0 [24] publicly available at [2]. Rabinizer-4.0 is a state-of-the art tool for translating LTL formulae into automata. For each of the LTL properties we consider, we obtain deterministic finite automata using Rabinizer-4.0 and check if this automata accepts the trace in consideration.

Benchmarks and Traces. Our subjects are open source GitHub repositories derived from a prior empirical study [28] on GitHub projects, as well as independently obtained from GitHub based on their popularity score (measured by GitHub stars). We use JavaMOP to instrument these repositories so that all events of interest (those



(a) Distribution of trace sizes

(b) Distribution of sizes of compressed traces

(c) Distribution of compression ratios

Figure 5: Sizes of (uncompressed and compressed) traces and compression ratios.

that occur in any of the LTL specs) are logged. We then generated traces by running all test classes of these repositories. We chose the top 100 traces based on the trace lengths. The minimum, maximum and average trace lengths in this set are 52.6M, 1.03B and 209M. The overall distribution is given in Figure 5a.

LTL Specifications. Our LTL properties are also obtained from [28]. Most of these properties specify the expected usage of different data structures and APIs used in these software projects, and are expressed in many different formalisms (regular expressions, ERE, LTL, FSM, etc.,). An example property is $\neg\mathtt{n} \land \mathsf{G}(\mathtt{n} \implies \neg\mathsf{X}\mathtt{n})$ from Example 2 (in Section 2.2), that specifies how an iterator of the `Set` collection must be used — every call to `next()` (denoted by '`n`') must be immediately preceded by a call to `hasNext()` (denoted by '`h`'). Another such property is $\varphi = \mathsf{G}(\mathtt{cr} \implies \mathsf{F}(\mathtt{cl}))$ which states that a resource (such as a buffered stream) must eventually be closed ('`cl`') every time it is created ('`cr`'). We identified that 10 properties were expressible in the fragment LTL[F, G, X], and selected all of them for our study.

Setup. We compare the running times of our algorithm over compressed traces to the time for analyzing the corresponding uncompressed traces against our LTL[F, G, X] specifications. After obtaining traces from our benchmark projects (using JavaMOP), we compress these traces using the Sequitur algorithm [37], available publicly [3], which runs in linear time in the size of the uncompressed trace. For the uncompressed traces, we use Rabinizer-4.0 to generate a deterministic finite state automaton for each property. For every property, Rabinizer-4.0 generates a Rabin automata, which is essentially a finite state machine, together with an acceptance condition for deciding membership of *infinite* words. We manually transformed these automata so that they are suitable for analyzing finite traces. Our experiments were conducted over a 2.6GHz 64-bit Linux machine.

6.2 Evaluation Results

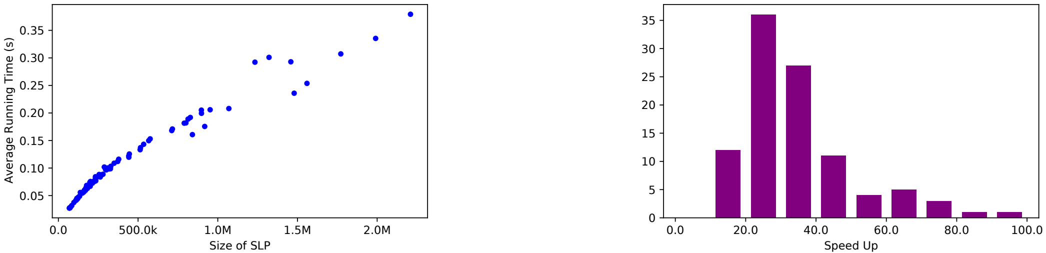
Size of Compressed Traces and Compression Ratios. While the uncompressed traces have lengths varying from 50M to 1.03B, the sizes of the compressed traces (SLPs) all lie between 54k and 1.8M. The average size of the SLPs is approximately $329k$ and the overall distribution is presented in Figure 5b. The compression ratios of each trace was observed to be at least 277. The maximum and average compression ratios are 1016 and 641, and the distribution is shown in Figure 5c. The significant compression ratios hint that most open source projects generate execution traces that have a lot of repetition and thus can be effectively compressed. A plausible explanation of large amount of repetitions is that many unit tests in our subject repositories repeatedly manipulate collection objects (such as lists or sets) in a loop.

Running times. In Figure 6a, we plot the running time (in seconds) for every compressed trace. These times are averaged over the running time of ZipMOP across all the 10 LTL[F, G, X] properties we consider. Further, in order to ensure fair comparison with the analysis over uncompressed traces, we exclude the time to read (uncompressed or compressed) trace files in memory — including I/O times would penalize the uncompressed analysis more heavily as they work over larger files. Observe that all the times are within 0.5 second (excluding I/O time). Also observe that, as expected, the times increase with the size of the compressed trace (SLP). In fact, we can see that the time increases *linearly* with the size of the SLP, despite the worst case dependence of $|G|^2$ as in Theorem 5.1 ($H(G)$ can be $O(G)$ in worst case).

Speed-up over analysis of uncompressed traces. We now compare how the running time over compressed traces compare with the running time of analyzing uncompressed trace logs. Figure 6b shows the speed up $\dfrac{\text{Time to analyze uncompressed trace}}{\text{Time to analyze SLP}}$, where, as before, both the numerator and denominator are average times over all LTL specs. Further, both the times exclude I/O time. The maximum, minimum and average speed ups are 90×, 15× and 34×. The high speed up shows the power of compression in analyzing trace logs as compared to uncompressed versions.

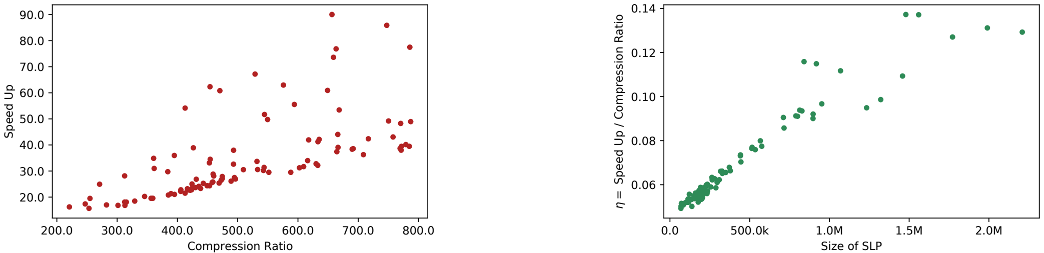
In Figure 7a we show how the speed up varies with the compression ratio. As expected, our algorithm performs better (as against the uncompressed analysis) when the compression ratio is high. This is because the time to analyze an uncompressed trace τ is $O(|\tau|)$ (time to check membership in a finite automaton) and the time to analyze a compressed trace G using our algorithm (Section 5) is proportional to $O(|G|)$ and the speed-up thus increases with the quantity $O(|\tau|/|G|)$, which is the compression ratio.

In Figure 7b, we analyze the efficiency of the algorithm, defined as $\eta = \dfrac{\text{Speed up}}{\text{Compression ratio}}$. The efficiency factor intuitively captures how well can the speed up over uncompressed traces be explained using the compression ratio. We observe that the efficiency values are in the range 0.04 to 0.11, and this is likely because of the constant multiplicative factors involved in the running time of our



(a) Running time v/s size of compressed trace **(b) Distribution of speed up over uncompressed analysis**

Figure 6: Running time and speed over uncompressed analysis



(a) Compression ratio v/s speed up **(b) Efficiency factor v/s size of SLP**

Figure 7: Speed up, compression ratio and efficiency

algorithm for checking compressed traces. Further, the efficiency factor increases (almost) monotonically with the size of the compressed format, implying that higher compression ratios are more effective when the compressed traces are themselves large.

7 RELATED WORK

From a theoretical standpoint, the work that is closest to ours is that of Markey and Schnoebelen [33] which established the PSPACE-hardness for the general problem of checking if a string, represented as an SLP, satisfies a formula written in full LTL; our result shows polynomial time tractability for the LTL[F, G, X] sub-fragment. Notably, the hardness in [33] arises from the use of arbitrarily nested until operators in LTL formulae. Lohrey [30] comprehensively surveys algorithmic and complexity-theoretic aspects of language theoretic questions involving SLPs. Galperin and Wigderson [16], and subsequently others[7, 8, 12, 15, 16, 31, 38, 46] showed that graph problems that are tractable on the uncompressed input become intractable when posed over compressed (*succinct*) representations.

Analysis of execution traces or event sequences has been central to the engineering of reliable and efficient software. While in our work, we propose the use of compression in runtime verification [18–20, 27, 35], prior works have focused on the use of compression in race detection [23], profiling [17, 26, 29, 42], or program comprehension using dynamic slicing [45, 47]. More recent works on large scale debugging [41], bug localization [40] and triaging [36] using trace data obtained from stack traces obtained at the time of crashes, while implicitly rely on compression provided by databases that store large columes of trace data, they do not leverage compression in the actual analysis tasks (such as pattern mining or clustering). An interesting avenue for future work would be to develop techniques to speed up such techniques by leveraging compression.

8 CONCLUSIONS

We propose the use of compression as an algorithmic paradigm to improve the efficiency of checking if execution traces conform to specifications written in LTL (linear temporal logic). While this problem is intractable (PSPACE-hard) in general, we establish a polynomial time algorithm for the rich fragment LTL[F, G, X] whose formulae do not include the U operator of full LTL. Our polynomial time algorithm leverages a monotonicity property in the automata theoretic representation of formulae in LTL[F, G, X]. On a comprehensive benchmark suite of open source Java projects, our evaluation confirms that execution traces can be effectively compressed and that the membership problem of traces can be efficiently decided over compressed formats (straight line programs), without decompressing them, resulting into significant speed ups when compared to analysis over uncompressed traces.

ACKNOWLEDGMENTS

Minjian Zhang and Mahesh Viswanathan are respectively supported by NSF CCF 2007428 and NSF CCF 1901069. Umang Mathur was partially funded by a Google PhD Fellowship and by the Simons Institute for the Theory of Computing.

REFERENCES

[1] 2021. JavaMOP. https://github.com/runtimeverification/javamop. Accessed: 2021-02-24.

[2] 2021. Rabinizer 4 - From LTL to Your Favourite Deterministic Automaton. https://www7.in.tum.de/~kretinsk/rabinizer4.html. Accessed: 2021-02-24.

[3] 2021. Sequitur: Inferring Hierarchies From Sequences. http://www.sequitur.info/. Accessed: 2021-02-24.

[4] 2021. ZipMOP. https://github.com/minjian233/ZIPMOP. Accessed: 2021-06-20.

[5] Gul Agha and Karl Palmskog. 2018. A Survey of Statistical Model Checking. *ACM Trans. Model. Comput. Simul.* 28, 1, Article 6 (Jan. 2018), 39 pages. https://doi.org/10.1145/3158668

[6] Rajeev Alur and Salvatorre La Torre. 2004. Deterministic generators and games for LTL fragments. *ACM Transactions on Computational Logic* 5, 1 (2004), 1–25.

[7] José L. Balcázar, Antoni Lozano, and Jacobo Torán. 1992. *The Complexity of Algorithmic Problems on Succinct Instances*. Springer US, Boston, MA, 351–377. https://doi.org/10.1007/978-1-4615-3422-8_30

[8] JoséL. Balcázar. 1996. The complexity of searching implicit graphs. *Artificial Intelligence* 86, 1 (1996), 171–188. https://doi.org/10.1016/0004-3702(96)00014-8

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transaction on Software Engineering* 41 (2015), 507–525.

[10] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. *Introduction to Runtime Verification*. Springer International Publishing, Cham, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[11] Feng Chen and Grigore Roşu. 2007. Mop: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 569–588. https://doi.org/10.1145/1297027.1297069

[12] Bireswar Das, Patrick Scharpfenecker, and Jacobo Torán. 2014. Succinct Encodings of Graph Isomorphism. In *Language and Automata Theory and Applications*, Adrian-Horia Dediu, Carlos Martín-Vide, José-Luis Sierra-Rodríguez, and Bianca Truthe (Eds.). Springer International Publishing, Cham, 285–296.

[13] Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (Beijing, China) *(IJCAI '13)*. AAAI Press, 854–860.

[14] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 411–420. https://doi.org/10.1145/302405.302672

[15] J. Feigenbaum, S. Kannan, M. Y. Vardi, and M. Viswanathan. 1998. Complexity of problems on graphs represented as OBDDs. In *STACS 98*, Michel Morvan, Christoph Meinel, and Daniel Krob (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 216–226.

[16] Hana Galperin and Avi Wigderson. 1983. Succinct representations of graphs. *Information and Control* 56, 3 (1983), 183–198. https://doi.org/10.1016/S0019-9958(83)80004-7

[17] Ankit Goel, Abhik Roychoudhury, and Tulika Mitra. 2003. Compactly Representing Parallel Program Executions. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/781498.781530

[18] Klaus Havelund. 2000. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, Klaus Havelund, John Penix, and Willem Visser (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 245–264.

[19] Klaus Havelund and Grigore Rosu. 2001. Monitoring Programs Using Rewriting. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*. IEEE Computer Society, USA, 135.

[20] Sampath Kannan, Moonzoo Kim, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. 2019. A Retrospective Look at the Monitoring and Checking (MaC) Framework. In *Runtime Verification*, Bernd Finkbeiner and Leonardo Mariani (Eds.). Springer International Publishing, Cham, 1–14.

[21] J.C. Kieffer and E.-H. Yang. 2000. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory* 46, 3 (2000), 737–754.

[22] J.C. Kieffer, E.-H. Yang, G.J. Nelson, and P. Cosman. 2000. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory* 46, 4 (2000), 1227–1245.

[23] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2018. Data Race Detection on Compressed Traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/3236024.3236025

[24] Jan Křetínský, Tobias Meggendorfer, Salomon Sickert, and Christopher Ziegler. 2018. Rabinizer 4: From LTL to Your Favourite Deterministic Automaton. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 567–577.

[25] N.J. Larsson and A. Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

[26] James R. Larus. 1999. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/301618.301678

[27] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. 1999. Runtime Assurance Based On Formal Specifications. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications.*

[28] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 602–613.

[29] Xianfeng Li, Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. 2004. Design Space Exploration of Caches Using Compressed Traces. In *Proceedings of the 18th Annual International Conference on Supercomputing* (Malo, France) *(ICS '04)*. Association for Computing Machinery, New York, NY, USA, 116–125. https://doi.org/10.1145/1006209.1006227

[30] M. Lohrey. 2012. Algorithmics on SLP-compressed strings: A Survey. *Groups Complexity Cryptology* 4, 2 (2012), 241–299.

[31] Antonio Lozano and José L. Balcázar. 1990. The complexity of graph problems for succinctly represented graphs. In *Graph-Theoretic Concepts in Computer Science*, Manfred Nagl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 277–286.

[32] Z. Manna and A. Pnueli. 1987. A Hierarchy of Temporal Properties. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) *(PODC '87)*. Association for Computing Machinery, New York, NY, USA, 205. https://doi.org/10.1145/41840.41857

[33] N. Markey and P. Schnoebelen. 2003. Model Checking a Path. In *CONCUR 2003 - Concurrency Theory*, Roberto Amadio and Denis Lugiez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 251–265.

[34] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. 2011. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Techniques for Technology Transfer* 14, 3 (June 2011), 249–289. https://doi.org/10.1007/s10009-011-0198-6

[35] Moonjoo Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. 1999. Formally specified monitoring of temporal properties. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*. 114–122. https://doi.org/10.1109/EMRTS.1999.777457

[36] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. 2021. Scalable Statistical Root Cause Analysis on App Telemetry. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 288–297. https://doi.org/10.1109/ICSE-SEIP52600.2021.00038

[37] C.G. Nevill-Manning. 1996. *Inferring Sequential Structure*. Ph.D. Dissertation. University of Waikato.

[38] Christos H. Papadimitriou and Mihalis Yannakakis. 1986. A note on succinct representations of graphs. *Information and Control* 71, 3 (1986), 181–185. https://doi.org/10.1016/S0019-9958(86)80009-2

[39] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, USA, 46–57. https://doi.org/10.1109/SFCS.1977.32

[40] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: Bug Localization on Millions of Files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 225–236. https://doi.org/10.1145/3395363.3397356

[41] Rebecca Qian, Yang Yu, Wonhee Park, Vijayaraghavan Murali, Stephen Fink, and Satish Chandra. 2020. Debugging Crashes Using Continuous Contrast Set Mining. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 61–70. https://doi.org/10.1145/3377813.3381369

[42] Manos Renieris, Shashank Ramaprasad, and Steven P. Reiss. 2005. Arithmetic Program Paths. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 90–98. https://doi.org/10.1145/1081706.1081721

[43] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle, Washington) *(LISA '99)*. USENIX Association, USA, 229–238.

[44] Koushik Sen, Mahesh Viswanathan, and Gul Agha. 2005. VESTA: A Statistical Model-Checker and Analyzer for Probabilistic Systems. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems (QEST '05)*. IEEE Computer Society, USA, 251. https://doi.org/10.1109/QEST.2005.42

[45] Tao Wang and A. Roychoudhury. 2004. Using compressed bytecode traces for slicing Java programs. In *Proceedings. 26th International Conference on Software Engineering*. 512–521. https://doi.org/10.1109/ICSE.2004.1317473

[46] Helmut Veith. 1996. Succinct representation, leaf languages, and projection reductions. In *Proceedings of Computational Complexity (Formerly Structure in Complexity Theory)*. 118–126. https://doi.org/10.1109/CCC.1996.507675

[47] Tao Wang and Abhik Roychoudhury. 2008. Dynamic Slicing on Java Bytecode Traces. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 10 (March 2008), 49 pages. https://doi.org/10.1145/1330017.1330021

[48] T.A. Welch. 1984. A Technique for High-Performance Data Compression. *Computer* 17, 6 (1984), 8–19.

[49] En-Hui Yang and J. C. Kieffer. 2000. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. I. Without context models. *IEEE Transactions on Information Theory* 46, 3 (2000), 755–777.

[50] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.