Logic For Exploit Detection: Utilizing Proof Search for Exploitability Detection in Compact Software Systems

by

Kaveh Eskandari Miandoab

A Master’s Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

Kaveh Eskandari Miandoab

Thursday 25th April, 2024

APPROVED:

______________________________
Professor Rose Bohrer, Major Thesis Advisor

______________________________
Professor Xiaoyan Sherry Sun, Reader
Abstract

Compact, yet complex systems and software are widely utilized, both on administrative, organizational, and personal levels, for the processing of large amounts of data. These software make imperative decisions and are used as a main tool in everyday workings such as voting software and automatic moderation systems. However, these systems inherently carry the danger of being exploitable either due to their design or owing to the data that they are based on, which can adversely affect the individuals impacted by their usage. In this work, we aim to identify and mitigate this behavior in compact software systems by catching possible exploitability bugs prior to the implementation of the system. Our workflow consists of two main steps, we first model the system as the steps the program takes to reach a conclusion using linear logic programming, and then apply proof search for the early identification of design exploits in the system. We wish to provide a complete and sound framework for the detection of exploits in a software’s design, and we believe that our framework can reduce the required workload for the removal of these design mistakes in later stages of development. Our main focus in this project is the detection of exploits that can be generated by following set a number of steps in accordance with the program logic, thus we refrain from focusing on security exploits, and those with external intervention, such as code injection. More concretely, our framework is best used to catch bugs that arise due to the program logic. Additionally, the proposed framework can be used for education to showcase how a bug propagates through a given software, and how a small change in a program can lead to vastly different results.
## Contents

1 Introduction ................................................. 1
   1.1 Modeling Software as a Linear Logic Program ............... 2
   1.2 Proof Search ........................................ 4

2 Related Work ............................................... 7
   2.1 Linear Logic Programming ................................ 7
   2.2 Proof Search ........................................ 8
   2.3 Applications ......................................... 9

3 Approach .................................................. 10
   3.1 Conversion to Linear Logic Program ....................... 10
   3.2 Proof Search ........................................ 13
      3.2.1 Soundness Proof Outline .......................... 15
      3.2.2 Completeness Proof Outline ........................ 16
   3.3 Proof Search Algorithm ................................ 16

4 Implementation .......................................... 20
   4.1 Implementation Details ................................ 21
   4.2 Time Complexity ...................................... 23

5 Evaluation ................................................. 24

6 Conclusion and Future Work .............................. 28

7 Acknowledgments .......................................... 30

8 References ................................................. 31
1 Introduction

Small, yet complex software [17] such as voting systems and operating system schedulers, are shown to exhibit exploitable behavior in their implementation. Although there is a recent interest in the exploitability of artificial intelligence models due to their popularity and increased usage [34], traditional software is just as likely to contain unwanted behavior due to shortcomings in design and implementation which arise from either personal biases or mistakes in these processes [14]. Given the widespread adoption of software systems, it is imperative for the social good to work towards the identification and mitigation of such flaws in software systems. As a motivating example, consider a voting system that given a number of candidates and voters, counts the number of votes received by each candidate, and returns the winner of the voting process. Suppose that there is a bug in the software such that a candidate’s vote can be counted twice, or a voter can vote several times. Under these circumstances, it is possible that the software miscounts the number of cast votes, returning a candidate that is not the real winner. Considering the importance of such software in several aspects of society, such mistakes can be catastrophic, and as such, require tools that can effectively catch bugs of the same nature prior to deployment.

Currently, the process of detecting bugs prior to software deployment consists of either manual testing or static analysis through software such as FlawFinder and Cppcheck [4]. However, these tools usually focus on specific languages such as C and C++ and have been shown to be unreliable when used in a context requiring a specific domain knowledge [20]. In order to bridge this gap, we propose a novel framework that works by taking into consideration the program logic as opposed to its syntax, enabling the programmer to declare their program alongside any required domain information. Due to its unified notion of logic, the proposed framework is language-agnostic and only requires conversion to a general linear logic form to run, and can specifically check for predefined unwanted behavior in a given software.

More concretely, in this work, we aim to identify shortcomings with respect to exploitability on a design (or system) behavior level rather than a syntactic level. Our
approach consists of two main phases, we first model a given software as a linear logic program divided by its execution logic states such that each state belongs to a stage that needs to be executed fully before the next stage begins. We then apply proof search algorithms with respect to an unwanted system behavior on the linear logic representation of the program in order to identify these behaviors. Given that the system is modeled as a linear logic program, we argue that running a proof search algorithm can soundly identify unwanted behavior (assuming that it directly results from the program’s logic) that can be viewed as a state of the system that the developer wishes to avoid. Considering that a high number of bugs and exploitability issues directly arise from design flaws in the program logic [12], we believe that early identification of these bugs is as important as other security requirements.

For the remainder of this section, we briefly introduce our approach to each phase, offer further details regarding the motivating example, and provide our expected contributions.

1.1 Modeling Software as a Linear Logic Program

Logic programming is a declarative paradigm of programming that utilizes formal logic to both define a domain and execute a user-defined set of rules in that domain. Logic programming assumes that ‘a single formalism suffices for both logic and computation, and that logic subsumes computation’[25]. Due to its declarative properties, logic programming is especially well-equipped to be used as a description-level tool for the analysis of otherwise procedural software as it can be used to define the behavior of any given program while providing a high-level abstraction over implementation details.

A common logic program is written in the form \( H \iff B_1, B_2, ..., B_k \), where a logical statement \( H \) is only true if logical statements \( B_1, B_2, ..., B_k \) are true. It is easy to observe that this formulation directly corresponds with the definition of a given software, as each logical statement can be defined as a correspondent to a program state, which requires the satisfaction of prior states to be true. As an example, take into consideration the motivating example. Suppose that in some given procedural language in which the voting
software is implemented, a function \textit{ReturnWinner()} exists which returns the name of the winning candidate. While it may be possible to call \textit{ReturnWinner()} at any given time, the programmer may want to invoke the function only when another function \textit{CountVotes()} has been called before. Observe that this directly corresponds with our definition of a logical statement such that \textit{ReturnWinner()} corresponds with \( H \) and \textit{CountVotes()} alongside any other prerequisite function might correspond with \( B_1, B_2, ..., B_k \), yielding the logical statement \( \textit{ReturnWinner()} :\neg \textit{CountVotes()}, \textit{Function}_2, ..., \textit{Function}_k \)

Even though logic programming offers a strong mathematical tool set for the declaration of a given domain, it performs poorly when required to define stateful domains \[30\]. A stateful domain is one that takes into consideration the possibility of the mutability of the program world through execution. For instance, the simple propositional logic used in the previous example cannot take into consideration the dynamic nature of a voting process, and hence, cannot correctly model logic such as \textit{CurrentWinner}(\textit{Candidate}) or \textit{NumVotes}(\textit{Candidate}) when the result of these functions are prone to change through the execution of the program.

To remedy this issue while modeling software, we utilize linear logic programming \[16\]. In this form of logic programming, logical statements are used to demonstrate a given state of a program, and the transformations applied to these statements as proofs are further applied to model the behavior of the program. More specifically, we take inspiration from Ceptre \[26\], a rule specification language originally designed for the rapid prototyping of interactive systems such as video games with a focus on narrative generation, to develop a \textit{Python} \[10\] program that accepts a given linear logic program definition and an unwanted program state as input, and searches over the program space via proof search to find possible traces that lead to the unwanted state. Although Ceptre is not initially designed for the detection of possible exploits in a given software, we believe that its strong specification capabilities and its inherent utilization of proof search provide us with a strong inspirational means to successfully develop a \textit{Python} program that models otherwise sequential software as a linear logic program.
We choose Python as our main implementation tool for exploit detection for the following reasons:

• **Popularity and Wide Adaption.** Python is one of the most commonly used programming languages in both academia and industry, allowing us to provide a tool that is both adaptable and easy-to-learn by the current experts of the language.

• **Ease of Use and Prevalence in Education.** Python is often used as a first introduction to computer science and programming and provides an easy-to-use interface for beginners such as undergraduate students. Our tool can be seamlessly integrated into these courses to provide an additional debugging tool while remaining within the boundaries of the target education concepts.

• **Extensibility and Mutability.** As opposed to closed-source or tools written in lower-level languages such as C and C++, Python libraries are fairly easy to understand and highly extensible. The choice to use Python as our main development language allows the users to change and extend upon the provided tool, adding further use cases and capabilities based on their requirements.

1.2 Proof Search

Proof search is a form of Symbolic Artificial Intelligence [13] in which a search tree over the problem domain is explored to discover a sequence of inference rules that constitute proofs for a given goal statement. It is easy to see that given proof rules that constitute a program, one can prove, using a sound and complete proof search algorithm, that a given exploit does not exist in the software. For instance, consider our motivating example. Suppose that an unwanted state is one in which an individual who is not a candidate wins the election. More concretely, assuming that the atomic predicate $\text{Winner}(\text{Character})$ exists, the unwanted state is $\text{Winner}(x)$ where $x \notin \text{Candidates}$. A proof search over the program space with respect to this state would start from the unwanted space, and search backward with the goal of finding a possible sequence of actions that start from the initial
program state, and end in the unwanted state. If no such proof can be given, we deem the system exploitable, and thus in need of revision. Concretely, we assume a closed-world approach, meaning that the truthfulness of a statement implies the falsity of its negation. More strongly, our framework provides proof for the existence of an exploit in the system, and the sequence of steps that need to be taken in order to reach the exploit, which can act as a strong debugging tool in the case of hard-to-find design bugs.

As a stronger motivating example, suppose that our voting software only returns True if no candidate has voted for themselves. However, there is an exploit in the software definition by which a person can first cast their vote, and then sign up for candidacy. In order to prove that no exploit exists, one needs to prove $\forall C (\neg Voter(C) \land Candidate(C) \lor (Voter(C) \land \neg Candidate(C)))$. By utilizing a sound and complete proof search, it is easy to prove that $\exists C (Voter(C) \lor \neg Candidate(C)) \land (\neg Voter(C) \lor Candidate(C))$, and thus a sequence of actions exists that results in a person being both a candidate and a voter. Observe that in such a formulation, unwanted states are either direct negations of all possible program end states or intermediate states that should not be reached at all.

Additionally, owing to the staged nature of the program formulation that is expected, the proposed framework allows one to search the program state locally, meaning that instead of tracing the unwanted state to an initial configuration, one can trace the state to a predefined stage of the program. This allows the user to safely formulate modular software, in which bugs might arise through the execution of certain modules rather than directly being caused by single step in the program logic. Section 4 provides further technical details on how the framework handles cases where we want to search through a localized portion of the program state.

Concerning implementation details, we utilize backward proof search [31] as our method of proof search in this work. Backward search involves starting from a goal statement, an unwanted final system state in our context, and moving backward in the inference tree with the goal of reaching the software’s initial state (config state). It is trivial to see that if such a path exists, a given exploit is satisfiable, and thus the software is exploitable, and
in need of revision. Backward search provides several advantages over the more traditional forward search which is widely utilized in the literature ([26], [35]). While forward and backward proof searches are equivalent in expressive power [24], it is difficult to trace a program logic to an unwanted state directly via forward chaining without exhausting the program space. This is due to the inherent property of backward chaining, in which the program is aware of the state to expand in order to reach the unwanted state. Conversely, forward chaining requires the expansion of all nodes at each given state as the program space leading to the unwanted state is not initially observable. Furthermore, notice that each stage might have multiple entry points, and as such, in order to perform a local search through forward proof search, one has to perform the search multiple times starting from each entry point individually, which is not efficient and requires an assumption over the program state at the time of the search. Backward search, however, does not require multiple runs over a localized area of the program state as it directly searches for states that lead to the current target state, and is at worst of the same time complexity as the forward search. Additionally, backward search does not require an assumption over the program space as the states are generated through backward chaining, and thus the only assumption is the presence of resources required by the state leading to the target state.

We make the following contributions through this project:

- Develop a novel Python-based tool that can soundly detect logic-based bugs and exploits in a given program.

- Introduce a sound and complete language-agnostic framework that can be used to model programs of varying nature and detect bugs in compact software.

- Implement and evaluate 3 real-world examples.

- Evaluate the program over $\sim 5000$ computer generated instances.

- Offer our finalized framework as an open-source tool that can be utilized in educational or industrial settings.
2 Related Work

In this section, we briefly discuss the literature surrounding our approach. We first analyze linear logic programming and the languages that utilize linear logic in their design principles. We then discuss proof search and the methods by which one can apply proof search to a linear logic system.

2.1 Linear Logic Programming

Linear logic, first introduced in 1987 [16], is a subset of logic programming that allows for the utilization of statefulness, and concurrency in a logical specification by emphasizing formulas as resources, a property absent in the classical logic [11]. Therefore, linear logic programming inherently allows for the declaration of computational software that are locally stateful.

Following the introduction of linear logic, a number of linear logic programming languages were introduced that make use of the stateful and concurrent capabilities of linear logic.

LO [1] was first introduced in 1990, and acts as an extension over Horn clauses that allow for OR-Concurrency, which subsequently allows structured processes, and the coordination between these processes to be declared.

Lolli [21], was introduced in 1994. It is a linear logic programming language based on a fragment of linear logic that deals with linearity, rather than intuitionistic logic, in its proof search. Lolli has proven useful in modeling of programs that depend on this linearity, such as theorem proving, natural language processing, and database programming.

Lygon [18] is a strict extension over Prolog that introduces the notion of consumable resources, an important fragment of linear logic, to Prolog. Lygon forces each clause to be used exactly once, thus removing the need for resource counting. Given that Lygon is a strict extension of Prolog, all classical Prolog programs can also be written in Lygon, while additional features such as concurrency, and inherit statefulness are added for linearity requirements of the language.
Celf [35] is a linear logic based framework for the specification of concurrent, and stateful systems. Celf utilizes CLF type theory [40], an extension over LF type theory [19] that adds the concepts of linear types to allow the usage of linearity in its specifications.

More recently, Martens has introduced Ceptre [26], an interactive rule based linear logic programming language for the modeling of inherently interactive structure, such as video game logic. While originally developed for the modeling of interactive media, Ceptre’s strong linearity as well as its capability to model a set of logic rules as stages that are the direct result of the quiescence of the previous stages, allow for the declaration of other software systems, namely, systems that deal with sequential states and contain strong local statefulness.

We believe that Ceptre offers a strong initial toolset for the modeling of state-based imperative programs, and can be expanded with relative ease to contain backward-proof search capabilities in order to identify possible exploits in a given software. Because Ceptre does not currently offer backward proof search capabilities, its addition to the language can drastically increase its use cases, both in its intended domain, and other domains such as exploit detection and theorem solving. Improving upon Ceptre, we introduce a novel framework that allows for backward proof search in addition to being stage-based. As a future work, we look forward to integrating the capabilities of our framework to Ceptre.

2.2 Proof Search

Proof search, in the context of logic programming, is usually meant to define one of two things. It either points towards the steps that a program takes in order to prove the truthfulness of a given query, or the steps that a program takes in order to find all feasible methods by which a query can be constructed [22]. Our proposed method falls in the second category, as in addition to proving the existence of an exploit, we also wish to produce the steps that an exploiter can take to achieve the said exploit.

To apply proof search to linear logic systems, Di Cosmo et al. [11] provide a general framework. They state that given a sequent $\Gamma : \Delta \rightarrow G$, where $\Delta$ is a multiset of rules and
According to Bruynooghe [2], it is possible to use forward-chaining to apply proof search to a sequent by moving in the direction of the control flow, and checking for the goal properties at each given step. King et al. [23, 24] have analyzed the equivalence of backward-chaining and forward-chaining methods in software verification, where the correctness of a program is measured with respect to a number of assertions. They find that the declarative power of forward-chaining and backward-chaining depends on the conditions over abstract domain operations.

While forward proof search can indeed be utilized to find possible exploits in a given system. We believe that backward proof search is better suited to our context of exploit detection. This is in contrast with forward proof search, which expands the inference tree until the goal statement is observed.

### 2.3 Applications

We observe two major application directions for staged linear logic programs: formal methods for security and privacy [29] as well as social software, as coined by Parikh [32].

Though formal methods broadly construed are used in a wide range of security and privacy tasks [29], we identify focus topics for linear logic programs generally and staged linear logic programs specifically.

Within security, protocols [3] are the best-established application of linear logic, particularly authorization [15, 7, 8] and authentication [37, 38] protocols. This approach has seen application in privacy audits, as well [5].

Social software encompasses the law, civil processes, and communication, among other topics. Within linear logic, traditional social software applications include the law [6] and
voting protocols \[9, 33\]. Outside linear logic, tooling for social software continues to grow, e.g., as demonstrated by the PolicyKit project \[41\]. The examples presented in this paper, though written by the authors, are both on PolicyKit’s examples \[41\] and on prior studies of voting protocols \[9, 33\]. In addition to the above domains, staged linear logic has been applied to narrative generation \[27, 28\], e.g., for procedural content generation in digital media.

3 Approach

Our proposed method of catching possible exploits in a program involves two main steps, we first convert a given program, or its design schema to a linear logic program written in Python, with syntactic inspiration taken from Ceptre \[26\], and we then apply backward proof search to find any possible exploits in the program design. To give a better understanding of our proposed approach, we first define a motivating example as follows.

- Alice, Bob, John, and Sophia are people.
- Sophia and Bob are candidates.
- Each person votes for a candidate.
- If a person is also a candidate, they can’t vote for themselves, otherwise they are disqualified.
- A person cannot vote more than once.
- The person with the highest number of votes wins the voting.

We now explain our proposed methodology, and how it applies to our motivating example.

3.1 Conversion to Linear Logic Program

We first need to convert the given design schema to a linear logic program. For this purpose, we utilize a custom Python program that takes inspiration from Ceptre \[26\]
programming language, and allows the user to define a procedure as a linear logic program using Python data structures and syntax. Further information regarding implementation details is given in section 4. Given that the framework takes inspiration from Ceptre, it is both stage-based and interactive, meaning that the programs are inherently broken into stages, which requires quiescence in stage \( N - 1 \) prior to the execution of stage \( N \), and one can clearly observe and choose the path that the program takes in the proof tree. Observe that the stage-based implementation does not bar the user from implementing a program with no stages, as one can simply place all chunks of their linear logic program in a single stage, during which the software performs as if no stages exist at all.

The stage-based implementation, as well as interactivity, make the provided framework a suitable tool for working with imperative, compact programs that are usually used imperatively. Each linear logic program consists of terms \( \tau \), predicates \( \rho \), and rules \( \kappa \), given an initial state \( \Delta_0 \), which consists of predicates, a rule \( r \in \kappa \) is applied to \( \Delta_0 \), which results in a state change to \( \Delta_1 \). Program execution is achieved through continuous application of \( r_n \in \kappa \) to \( \Delta_i \) until quiescence is reached. More specifically:

\[
\Delta_i \xrightarrow{r_n(S) \rightarrow S'} \Delta_{i+1} | r_n \in \kappa \wedge S \in \Delta_i \wedge S' \in \Delta_{i+1}
\]

Where \( S \) is a set of predicates in \( \Delta_i \), \( S' \) is a set of predicates in \( \Delta_{i+1} \) such that \( S' \) is constructed by applying \( r_n \) to \( S \), and \( \Delta \xrightarrow{r} \Delta' \) indicates a proof step from \( \Delta \) to \( \Delta' \) using \( r \). Note that given the stage-based nature of our framework, each given rule is a member of a stage, meaning that \( r_i \in \kappa_n \), and in order to apply \( r_j \in \kappa_{n+1} \) to any state \( \Delta_k \), it must hold that no \( r_i \in \kappa_n \) applies to \( \Delta_k \), meaning that stage \( n \) is quiescent. More concretely:

\[
(\Delta_i \xrightarrow{r_n}(S) \rightarrow S') \rightarrow \#r_{n-1}(\Delta_i \xrightarrow{r_n-1}(S) \rightarrow S')
\]

Given this definition of program states in the provided framework, it is easy to see that our motivating example can be converted to a linear logic program as follows.

1. \( \text{Person} \in \text{Type} \)
2. \((\text{Alice} \in \text{Person}) \otimes (\text{Bob} \in \text{Person}) \otimes (\text{John} \in \text{Person}) \otimes (\text{Sophia} \in \text{Person})\)

3. \(\text{Candidate(Person)} : \text{Predicate}\)

4. \(\text{Voter(Person)} : \text{Predicate}\)

5. \(\text{Votes(Person, Person)} : \text{Predicate}\)

6. \(\text{HasNotVoted(Person)} : \text{Predicate}\)

7. \(\text{Voter}(C_i) \otimes \text{HasNotVoted}(C_i) \otimes \text{Candidate}(C_j) \rightarrow \text{Votes}(C_i, C_j) \otimes \text{HasNotVoted}(C_i) \otimes \text{Candidate}(C_j)\)

8. \(\Delta_0 = \text{Candidate}(\text{Bob}), \text{Candidate}(\text{Sophia}), \text{Voter}(\text{Alice}), \text{Voter}(\text{Bob}), \text{Voter}(\text{John}), \text{Voter}(\text{Sophia}), \text{HasNotVoted}(\text{Alice}), \text{HasNotVoted}(\text{Bob}), \text{HasNotVoted}(\text{John}), \text{HasNotVoted}(\text{Sophia})\)

Note that the above linear logic definition of the program contains an intentional bug, by which a ‘Character’ can vote multiple times. This is due to predicate ‘HasNotVoted’ not being consumed in the voting process, which is a common mistake that can happen in a real-world setting. We will use this bug in the next section to motivate our usage of proof search.

Additionally, due to the stateful nature of linear logic programs, it is possible to convert any imperative program to a linear logic program written in the proposed Python framework. The main challenge of this conversion is capturing the correct logic of the initial program. One can utilize Automata Learning \cite{39} to learn an automata that mimics the behavior of the initial system, and then convert the said automata to a linear logic program. While not the main focus of our project, we plan to experiment with this hypothesis in future work in order to develop a general framework for the conversion of imperative programs to linear logic programs.
3.2 Proof Search

Once a linear logic program is constructed, either from a design schema or by analyzing an already written program, we next apply a backward proof search on the linear logic formulation, with the goal statement being a possible computation product of the program. More concretely, the following steps are taken.

1. Identify the initial configuration of the linear logic system.

2. Identify a linear logic statement that is an undesirable system behavior with respect to its requirements.

3. Apply backward proof search with the undesirable statement as the goal state and the linear logic program as the set of rules.

4. Identify the inference rules such that their application leads to the generation of the goal statement.

5. Temporarily eliminate the bottom stage inference rule from the sequent and replace the current target with one of the states that are able to generate the current target via inference rules.

6. Repeat the process.

The search halts once no trace can be found from the initial configuration to the goal statement.

Suppose that we wish to find if an exploit is accessible in the motivating example by which an exploiter can vote more than once. More concretely, we wish to find a proof such that \( \{Votes(C_i, CA_i), Votes(C_i, CA_i)\} \) holds. Observe that as we consider each predicate as a resource in linear logic programming, \( \{Votes(C_i, CA_i), Votes(C_i, CA_i)\} \neq \{Votes(C_i, CA_i)\} \) as in set theory or propositional logic. We start by applying backward expansion over each required statement. Once a trace is constructed from the initial configuration to the goal statement, we conclude that the statement holds and the system
is exploitable. The following states are reached by applying backward proof search to the current goal statement.

1. \{\textit{Votes}(C_i, CA_i), \textit{Votes}(C_i, CA_i)\} \rightarrow \\
   \{\{\textit{Voter}(C_i), \textit{HasNotVoted}(C_i)\}, \{\textit{Voter}(C_i), \textit{HasNotVoted}(C_i)\}\}

2. \{\{\textit{Voter}(C_i), \textit{HasNotVoted}(C_i)\}, \{\textit{Voter}(C_i), \textit{HasNotVoted}(C_i)\}\} \rightarrow \\
   \{\{\textit{Voter}(Alice), \textit{HasNotVoted}(Alice)\}, \{\textit{Voter}(Alice), \textit{HasNotVoted}(Alice)\}\}

As we reach a state that holds given the initial configuration state, we conclude that an exploit exists by which a person can vote twice.

The backward proof search that we employ is a Breadth-First Search algorithm that, given a goal statement, iteratively expands each term constituting the statement by utilizing its previous stage and continues the expansion until quiescence is reached in the current stage. We particularly refrain from utilizing a Depth-First approach, as it can result in a non-halting algorithm in the case of cyclic resources in the linear logic formulation (\(A \rightarrow B\) and \(B \rightarrow A\)) on top of additional disadvantages discussed in section 1. A Breadth-First proof search algorithm allows us to fully explore the expansion space of a statement \(\Delta\) before transitioning to the subsequent statement \(\Delta'\), thus eliminating the risk of indefinitely expanding a single term. In future work, we plan to provide proof for the termination of our algorithm with respect to any linear logic program. Additionally, it is possible to introduce heuristic methods that only expand terms more likely to prove the goal statement, lowering the overall cost of the proof search algorithm.

Furthermore, it is also possible, utilizing the proposed algorithm, to check for the truthfulness of generative invariants in a given program [36]. A generative invariant for the ruleset \(\Sigma\) is defined as the pair \((\Sigma', \Delta_0)\), describing a set of contexts reachable from the config state \(\Delta_0\) by applying the rules in \(\Sigma'\), such that all rules in \(\Sigma\) conclude in contexts that are members of \((\Sigma', \Delta_0)\). In order to check for a given generative invariant in a program using the proposed algorithm, it suffices to check that:

1. The goal statement satisfies the generative invariant.
2. All subsequent states, including $\Delta_0$, satisfy the generative invariant.

As a simple example, suppose that we have the generative invariant in the form of $(Gen \rightarrow Admin \otimes Gen) \land (Gen \rightarrow Admin)$, indicating states containing at least one Admin instance. In order to check for the satisfaction of this generative invariant utilizing our algorithm, it suffices to check for its satisfaction with respect to the goal statement, and every subsequent statement including the config state $\Delta_0$. Note that in the presence of a generative invariant, its violation at any step should be considered a system exploit, thus compelling the algorithm to return True for that particular search.

This backward approach to proof search in finding exploits is both sound and complete. We next offer two brief proof outlines for both of these cases and hope to provide the complete proof in future work.

### 3.2.1 Soundness Proof Outline

Soundness, as used in proof search literature, indicates that if a trace is provided by the algorithm, the trace must lead to an exploit. It is possible to prove the soundness of backward proof search in our case using induction. As a base case, we first show that if a program has no stages, then the algorithm returns an empty trace, as it only chooses a trace based on the set of available stages. Furthermore, it is easy to show that if there is a trace with only a single stage, the algorithm will choose a corresponding state in that stage by definition. Now, we can hypothesize that the algorithm is sound while a trace with stages of length $N$ exists, and prove that the algorithm is also sound while a trace with stages of length $N + 1$ exists. Given that, by definition, the algorithm must choose a logic transformation that directly results in the target statement, the addition of an extra stage can lead to two possible cases as follows:

- **Stage $N + 1$ does not contain a rule that while applied to a state, results in the target statement.** In this case, the algorithm, by definition, does not find a corresponding rule by which backward search can be continued, and considers the stage $N + 1$ as quiescent, thus transitioning to stage $N$ to continue its search. As we know the search...
is sound for $N$ stages, thus the search will soundly complete from stage $N$, making stage $N + 1$ redundant. As such, the algorithm will also be sound with $N + 1$ stages.

- A rule in stage $N + 1$ exists such that when applied to a state, results in the target statement. In this case, the algorithm, by definition, finds at least one corresponding rule by which the backward search can be continued as by base case we know that the algorithm is sound with $|Trace| = 1$. In this state, the algorithm needs only to explore the remaining $N$ stages. We already know that the algorithm is sound with respect to $N$ stages, and as such, it must hold that the algorithm is still sound in this case with respect to $N + 1$ stages.

3.2.2 Completeness Proof Outline

Completeness indicates that if a proof exists for a goal statement, the algorithm is able to find it. It is possible to prove the completeness by construction. Suppose that $N$ traces in a given linear logic program lead to an exploit, by definition, our proof search algorithm potentially checks every trace in a linear logic program in order to find a path from the initial configuration to the goal statement. As such, it is guaranteed to find an answer trace if one exists. We can then eliminate the returned path from the linear logic program, and run the algorithm for the remaining $N - 1$ steps that lead to an exploit, which will return a new answer trace. Given that the algorithm is sound, the returned answers are guaranteed to be correct, and the algorithm can be repeated until no more traces are found, and as such, all exploit traces are covered. Given that the algorithm is able to find every exploit trace, we have now constructed a search method that is complete.

3.3 Proof Search Algorithm

We now provide the complete algorithm by which we perform the proof search. We additionally step through the algorithm and explain in detail the rationale behind each choice in the proposed algorithm.

We define a query $q$ as the conjunctive formula of form $\otimes_i \{p_i(x_i)\}$ and use it as our
target *atom*. More concretely, \( q \) is the *atom* that needs to be proven under \( \vec{s} \). Starting from an arbitrary stage \( n \), in order to prove \( q \) for that particular stage, we are required to find \( \Gamma_r \subset \Gamma_n \cup \Gamma_{n-1} \) where \( r \) represents an atom that is producible from \( q \) via backward chaining or an atom that leads to \( q \) from the stage \( n - 1 \) via forward-chaining and results in quiescence in stage \( n - 1 \), and such that \( (\forall i \in \Gamma_r)i \) is of the form \( A \rightarrow B \otimes q \otimes C \) where \( A \) is of the form \( \otimes \{p_i(\vec{x})\}^+ \) and \( B, C \) are of the form \( \otimes \{p_i(\vec{x})\}^* \). Once \( \Gamma_r \) is produced from \( q \), we iteratively apply the same process to all elements \( i \in \Gamma_r \). Note that in order to reach a quiescent \( \Gamma_r \), an intermediate state containing the rules that are the current product of applying a backward step must be kept. As soon as a stage \( n = 0 \) is reached, meaning that all stages are quiescent, we linearly search through the current \( \Gamma_r \), determining if, for any \( i \in \Gamma_r \), \( p(\vec{pat}) \) matches a state. More concretely, for a run of the algorithm \( R(F) \), we can say that \( \forall i(i \in \Gamma_r \wedge i \vdash p(\vec{pat})) \implies \text{Terminates}(R(F)) \), where \( i \vdash p(\vec{pat}) \) implies a match between \( p(\vec{pat}) \) and \( (i, \Gamma_r) \), where each pattern \( pat \) is either a named element of a type or the special symbol \( \_ \) meaning ‘any value of the given type’.

Algorithm 1 showcases the backward proof search algorithm by which the trace from a given target state to an initial state is constructed. The procedure **ProofSearch** accepts arguments \( \vec{s}, n, p, q \) as input, where \( \vec{s} \) in the linear logic program containing rules, where each rule has a left-hand-side (LHS), and a right-hand-side (RHS), which indicate resources that can be consumed, and generated by the application of that rule, respectively. More concretely, both LHS and RHS are of type \( \Gamma_n \). Parameter \( n \) is an integer referring to the starting stage of the search. Parameter \( q \) is the query that needs to be proven and is of the form \( \Gamma_q \), however, as we are required to know the stage from which the search begins, we add the corresponding \( n \) to \( q \) in line 16. Finally, parameter \( p \) is the algebraic signature that needs to hold once the search is complete, and can either be a single state such as \( q \), or a function characterizing a set of states. Having provided sufficient background, we now explain the proposed algorithm in more depth.

The procedure **ProofSearch** contains two helper functions, the function **For-
Algorithm 1 Pseudocode for proof search algorithm

1: procedure ProofSearch($\vec{s}, n, p, q$)
2: \hspace{1em} function ForwardLinks($\vec{s}, q, n$)
3: \hspace{2em} Let $E := \emptyset$
4: \hspace{3em} for all $r \in \vec{s}$ do
5: \hspace{4em} if $\text{LHS}(r) \subseteq q[1] \land \text{Stage}(r) = n$ then
6: \hspace{5em} $E := E \cup (\text{Stage}(r), (q[1] \setminus \text{LHS}(r)) \cup \text{RHS}(r))$
7: \hspace{3em} return $E$
8: \hspace{1em} function BackwardLinks($\vec{s}, q, n$)
9: \hspace{2em} Let $D := \emptyset$
10: \hspace{3em} for all $r \in \vec{s}$ do
11: \hspace{4em} if $\text{RHS}(r) \subseteq q[1] \land \text{Stage}(r) = n$ then
12: \hspace{5em} $D := D \cup (\text{Stage}(r), (q[1] \setminus \text{RHS}(r)) \cup \text{LHS}(r))$
13: \hspace{3em} return $D$
14: \hspace{2em} Let $F := \emptyset$ be a Queue
15: \hspace{2em} Let $V := \emptyset$ be the Visited Set
16: \hspace{2em} $q' := (n, q)$
17: \hspace{2em} $V := V \cup q'$
18: \hspace{2em} $F$.enqueue($q'$)
19: \hspace{2em} while $F \neq \emptyset$ do
20: \hspace{3em} $C := F$.dequeue()
21: \hspace{4em} if $C[0] = 0 \land C[1] \vdash p$ then
22: \hspace{5em} return $C$
23: \hspace{4em} for all $C' \in \text{BackwardLinks}(\vec{s}, C, C[0])$ do
24: \hspace{5em} if $C' \notin V$ then
25: \hspace{6em} $V := V \cup C'$
26: \hspace{6em} $F$.enqueue($C'$)
27: \hspace{4em} if $C[0] \neq 0$ then
28: \hspace{5em} if $\text{ForwardLinks}(\vec{s}, C, C[0]-1) = \emptyset$ then
29: \hspace{6em} $C[0] := C[0] - 1$
30: \hspace{6em} $F$.enqueue($C$)
31: \hspace{5em} $V := V \cup C$
WARDLINKS (lines 2-7), is designed to find all the forward links from a given state $q$ at a given stage $n$, and works by iterating through all the rules in the given linear logic program $\vec{s}$, and returning states that are the direct result of applying the current rule to $q$ in stage $n$. Conversely, the function BACKWARDLINKS (lines 8-13) is designed to find all the backward links from a given state $q$ at a given stage $n$ and works by iterating through all the rules in a given linear logic program $\vec{s}$ and returning states that are the direct result of applying the current rule to the current state such that it results in $q$. Observe that $\text{FORWARDLINKS} \neq \text{BACKWARDLINKS}$ in our case, as a forward link from $q$ implies the consumption of a subset $t \subset q$ from $q$, and the generation of a new set of resources, while a backward link from $q$ implies the deletion (we refrain from using the term ‘consumption’ in this case, as the deleted resources are assumed to be not present prior to the application of the rule to LHS) of resources that were generated as a product of applying the current rule to LHS, and replacing them with the resources from LHS that were consumed in the rule application process.

Lines 14-31 showcase the main algorithm, which makes use of the prior helper functions to perform the search procedure. We first define two sets $F$ and $V$, where $F$ is a queue containing the states that are to be expanded and initially contain the target state, and $V$ is a set containing the expanded states, and initially also contains the target state. Next, we find all the states with backward links to the current state in the stage corresponding to the stage of the current state and queue them for further expansion. Each visited state is potentially an element of a trace that leads to the initial configuration. Additionally, the program may reach quiescence in stage $n$, meaning that no unvisited state can be found such that there is a backward link from the current state, or the sequence of rules that are required to produce the target state $q$ does not make use of any rule in stage $n$. To remedy this situation, we additionally search for states in stage $n - 1$ such that there is no forward link from those states to the current state, and the rule corresponding to the forward link results in quiescence at stage $n - 1$. Observe that this is equivalent to enqueueing the current state with the target stage of $n - 1$ if no forward link exists from
the current state at stage \( n - 1 \), meaning that the current state is quiescent at stage \( n - 1 \). Note that checking for quiescence is imperative in this case, as by definition, the staged linear logic program only transitions between stages \( i \) and \( i + 1 \) when stage \( i \) is quiescent, and given that the current state is in stage \( i + 1 \), we can only transition to stage \( i \) if make sure that stage \( i \) is quiescent with respect to the current state.

Once the current target state is at stage \( n = 0 \), showcasing that a trace has been found to the initial program layout, the program checks the satisfiability of the predicate \( p \) with respect to the current state. If \( p \) can be satisfied, we deem the current trace as one that proves the initial target state \( q \). If \( p \) is not satisfied through the current stage, we deem the target state \( q \) unsatisfiable through the ongoing state and continue the search as detailed previously. As outlined in Section 3.2.1 and Section 3.2.2, the provided algorithm is sound, meaning that if a proof is found, it is guaranteed to be correct, and complete, meaning that if a proof exists, it is guaranteed to find the proof in an unbound time limit.

Next, we go through the implementation details of the program, including our choice of data structures, evaluation, and time complexity.

## 4 Implementation

We implement and test our framework in Python. Python\(^1\) is a high-level programming language widely used in both education of computer science, and software engineering. We chose Python due to its simplicity, and prevalence in the computer science community, which allows for the seamless availability, accessibility, and extensibility of our framework. Additionally, we open-source our Python code, and thus it is easy to customize the provided framework by adding novel search strategies and specific use cases without changing the overall program logic, making the framework suitable for both education and professional use.

For the remainder of this section, we focus on implementation and data structure details, as well as analyzing the time complexity of the search algorithm as written in

\(^1\)https://www.python.org/
4.1 Implementation Details

The program, as provided, contains two main components that are implemented as Python classes. Firstly, in order to effectively demonstrate a staged linear logic program, we implement the class SLLP. Remember that from Section 3, each staged linear logic program $\bar{s}$ contains a set of rules $R$ where each $r \in R$ contains a Left-Hand-Side (LHS), a Right-Hand-Side (RHS), and a stage $n$. To apply a rule that lies in stage $n$ for $n \geq 1$, it must be the case that no rule from $n - 1$ can be applied, meaning that rules in stage $n - 1$ are quiescent.

In order to model the aforementioned definition of staged linear logic programs, each instance of the class SLLP refers to a single rule in a given staged linear logic program, and contains the following properties:

- **LHS.** Refers to the left-hand side of the rule, and is structured as a multiset of resources. Using a staged linear logic process, the multiset of rules on the left-hand side is consumed.

- **RHS.** Refers to the right-hand side of the rule, and is structured as a multiset of resources. Using a staged linear logic process, the multiset of rules on the right-hand side is generated via the consumption of the left-hand side.

- **Stage.** Refers to an integer that demonstrates the current stage that the rule is referring to. Stage is used to determine if a specific instance of SLLP can be applied, meaning that Stage-1 is quiescent.

Furthermore, note that although SLLP can be used to model an instance of a rule in $\bar{s}$, one is required to construct a set of instances to develop a complete staged linear logic program if the said program contains more than a single instance. In order to achieve this, we model a complete staged linear logic program as a Python list of SLLP instances. Observe that in our case, $List = Set$ with respect to the Python program as each rule
rule_1 = SLLP(Multiset(['a']), Multiset(['b']), 0)
rule_2 = SLLP(Multiset(['e']), Multiset(['c']), 0)
rule_3 = SLLP(Multiset(['b']), Multiset(['c']), 0)
rule_4 = SLLP(Multiset(['b']), Multiset(['d', 'f']), 1)
rule_5 = SLLP(Multiset(['e']), Multiset(['r', 'n']), 1)
rule_6 = SLLP(Multiset(['c']), Multiset(['k', 'z']), 1)
rule_7 = SLLP(Multiset(['f']), Multiset(['g', 'l']), 2)
rule_8 = SLLP(Multiset(['r']), Multiset(['k', 'z']), 2)

Figure 1: Python Code for the Definition of a Staged Linear Logic with 8 Rules

is unique by itself as (SLLP1.RHS = SLLP2.RHS) \& (SLLP1.LHS = SLLP2.RHS)
\& (SLLP1.Stage = SLLP2.Stage) \Rightarrow SLLP1 = SLLP2 and SLLP2 is redundant.

Figure 1 showcases a staged linear logic program with eight instances of the SLLP class.

In addition to the construction of staged linear logic programs, the framework requires
the presence of a backward proof search algorithm such that the proof search can be
performed on instances of SLLP, and return True if a trace can be found with respect to
a target query q.

For this purpose, we define a Python function \texttt{stagedBFS}, that accepts four arguments
as input and returns True if a successful search can be performed. The input arguments
to \texttt{stagedBFS} is as follows:

- \textit{Ruleset} is the list of SLLP instances, where each \textit{Ruleset} signifies a staged linear
  logic program \vec{s}.

- \textit{P} is the predicate that needs to hold True in order for the search to be successful.
  Each predicate is an instance of the multiset\footnote{https://pythonhosted.org/multiset/api.html} class.

- \textit{N} is the stage from which the search starts. For a typical staged linear logic program,
  \(N = Max\{r_i.Stage | r_i \in R\}\). However, it is also possible to perform an effective
  localized search by supplying \texttt{stagedBFS} with a custom \textit{N}. This enables further
  flexibility in the search algorithm and allows for the exploration of a limited space if
  the relative spatial position of a bug is known, preventing possible state explosion.
• \( Q \) is the target state that needs to be proven. Practically, \( Q \) is the state from which the backward search is started. Observe that the presence of a backward trace from \( Q \) to an initial state that satisfies predicate \( P \) implies the presence of a forward trace from an initial state that reaches the goal state of \( Q \).

With respect to the implementation of \texttt{Staged\_BFS}, the function has an almost one-to-one correspondence with Algorithm 1. Within \texttt{Staged\_BFS} we define two helper functions \texttt{forward\_links} and \texttt{backward\_links} where \texttt{forward\_links} corresponds with \texttt{ForwardLinks} from Algorithm 1 and \texttt{backward\_links} corresponds with \texttt{BackwardLinks} from Algorithm 1. Additionally, we use a double-ended queue as the data structure of choice for \( F \), and a Python set as the data structure of choice for \( V \) from Algorithm 1. Note that as shown in Section 5, the introduced algorithm merely provides semi-decidability, meaning that although it is guaranteed to halt on inputs that evaluate to \texttt{True} given sufficient time, it is also possible to loop indefinitely if no trace can be found from the goal state \( Q \) to an initial state.

Next, we show that the provided algorithm has the time complexity \( O(n^k) \) where \( n \) is the number of rules, and \( k \) is the length of the shortest possible proof.

### 4.2 Time Complexity

Let \( \vec{s} \) be a staged linear logic program with algebraic signature \( \alpha \), where \( \alpha \) characterizes a set of permitted initial states, and logical signature \( \Sigma \). Let \( n = |\vec{s}| \) be the number of rules. Let \( A \) be a query over \( \vec{s} \). Let \( k \) be the length of the shortest proof of \( A \) under any context \( \Gamma \) matched by \( \alpha \). Then the algorithm terminates in \( O(n^k) \) steps.

Observe that for a proof of length \( k \), it is implied that the proof contains \( k \) rules such that their consecutive application results in the production of the query \( A \). To satisfy this proof, the algorithm iterates through each rule in each given stage and performs the backward substitution if that rule is able to generate the current query. It is easy to see that it is possible to compose a staged linear logic program to have a set of rules such that for each rule in each stage, the algorithm applies the rule to the current query, only
backtracking when reaching stage 0, during which the algorithm detects unsatisfiability with respect to $\alpha$. Given these circumstances, and considering that the shortest proof is of length $k$, the algorithm will traverse the proof graph to the depth $k$ prior to backtracking. As such, based on the assumption that there are $n$ rules present, and all rules are applied with respect to each current query, the algorithm must perform $O(n^k)$ steps prior to termination.

In the next section, we evaluate the correctness and the practical performance of the proposed program by testing the Python implementation of the algorithm on a number of manually and automatically constructed instances.

5 Evaluation

We proceed to evaluate the correctness, and efficiency of the implemented algorithm. This evaluation supports the following research questions:

1. **RQ1**: How does one perform sound and complete backward-chaining search on staged linear logic programs?

2. **RQ2**: How effectively and efficiently does the search perform on staged linear logic programs of different complexity levels?

To support RQ1, we confirm the soundness and completeness of the implementation on the test corpus, i.e., that query solutions are reported if and only if they exist, given sufficient running time. To support RQ2, we measure running time and count how many programs in the corpus complete their queries within a given time budget. We measure the number of rules and number of stages in each program as a proxy for its complexity, so that we can numerically validate the relationship between complexity and running time.

**Corpus Selection** We evaluate the algorithm on a corpus of Staged Linear Logic Programs, which both provide a broad range of complexity levels and representation of realistic use cases. To this end, our corpus is the union of three sources:
A set of minimal examples developed by the authors while testing functional correctness of the implementation,

Randomized examples of varying complexity. These are mainly used to determine the efficiency of the algorithm,

A set of fewer, larger motivating examples developed by the authors based on social software drawn from the PolicyKit project,

In particular: by including the first two sub-corpuses, we can assess performance differences between small and large programs, and by including the remaining sub-corpus, we ensure the results are representative of prior applications identified by the previous work [41].

These three categories respectively contain 6, 3502, and 2 programs which respectively contain a total of 27, 89844, and 10 rules. As a whole, the corpus contains 3510 programs consisting of 89881 rules. For the larger, manually constructed examples in particular, we implement two complex staged linear logic programs, and evaluate them on two contradicting queries, with the first one following the intended behavior of the program, and the other one introducing a bug. The first program is a voting staged linear logic program, in which each instance of type person votes for an instance of type candidate, the goal is to make sure that each person can only vote once. The second program is taken from PolicyKit [41], and follows the procedure by which Wikipedia elects new moderators. Each tenured user can request to be granted a moderator permission, which is then either approved or rejected by another moderator. The goal example for the second example is to make sure that the decision is consistent, meaning that a user can’t be both approved and rejected for their request. For the automatically generated test programs, we ensure that 2502 out of the 3502 generated programs are expected to have a solution, while the remaining 1000 are generated randomly. Additionally, we divide the automatically generated test programs into two subgroups $S_1$ and $S_2$, where $S_1$ contains programs with varying levels of stage complexity (Stage numbers between 0 and 10) while having a maximum of 5 rules per stage. Inversely, $S_2$ contains programs with varying number of
rules at each stage (Maximum of 5 to 25 rules per stage) while having 5 stages in each program.

The initial corpus consisted only of staged linear logic programs, i.e., rulesets, but the evaluation requires querying those rulesets. The author manually (for the examples developed by the author) and automatically (for the randomly generated examples) augmented the corpus by writing queries for each example program, both examples that have a solution and those that do not. We make no claims about how frequently the queries correspond to novel bugs detected in pre-existing programs, as our goal is rather to measure algorithm performance and correctness.

**Experimental setup** All experiments were performed on the author’s workstation, which has the processor Intel Core i7 12700 with a clock rate of 2.1 GHz and 12 cores. It has 16 GB of RAM and runs Windows 11 Enterprise version 23H2. Tests were performed using a test script which will be made publicly available.

All tests were executed sequentially, with a timeout of 120 seconds per test. This timeout is motivated by typical use scenarios for automated bug-catching tools, e.g., asynchronous batch-mode execution by continuous integration (CI) software.

**Experimental Results** Experimental results are summarized in Figure 1. For each example, we give it a name, categorize the number of queries with respect to their class and the number of stages to measure complexity and report the percentage of the queries that terminate. We additionally record whether sound results are recorded for each example. There is no completeness column, as completeness is already best assessed by measuring which queries terminate.

**Interpretation of Data** We interpret the data presented in Table 1. We observe that, as expected, an increase in both the number of stages and the number of rules negatively corresponds with the termination percentage of instances. However, an increased complexity in the number of rules has a higher possibility of constructing a program that does not
terminate under evaluation time constraints. This observation is in accordance with the findings of Section 3, as having a higher number of rules is more likely to lead to a longer proof. Additionally, we note that a greater number of stages occasionally corresponds to a higher termination rate, as it decreases the number of rules explored in each individual stage. Given that real users of staged linear logic programs often write shorter programs, even this exponential complexity with respect to the number of rules corresponded to a successful search on a substantial fraction of the corpus. However, the observed timeouts on more complex queries motivate the future development of more sophisticated, optimized search algorithms.

Table 1: Summary of Experimental Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Rules#</th>
<th>Stages#</th>
<th>Sound?</th>
<th>Termination%</th>
<th>Max Rules Per Stage</th>
<th>Example#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy Examples</td>
<td>27</td>
<td>2-4</td>
<td>✓</td>
<td>100</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Real Examples</td>
<td>10</td>
<td>2-3</td>
<td>✓</td>
<td>100</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>S1 Batch #1</td>
<td>4905</td>
<td>1-3</td>
<td>N/A</td>
<td>96.93</td>
<td>5</td>
<td>751</td>
</tr>
<tr>
<td>S1 Batch #2</td>
<td>12935</td>
<td>4-6</td>
<td>N/A</td>
<td>93.33</td>
<td>5</td>
<td>750</td>
</tr>
<tr>
<td>S1 Batch #3</td>
<td>21168</td>
<td>7-9</td>
<td>N/A</td>
<td>90.04</td>
<td>5</td>
<td>750</td>
</tr>
<tr>
<td>S1 Batch #4</td>
<td>8867</td>
<td>10</td>
<td>N/A</td>
<td>92.4</td>
<td>5</td>
<td>250</td>
</tr>
<tr>
<td>S2 Batch #1</td>
<td>8442</td>
<td>5</td>
<td>N/A</td>
<td>97.21</td>
<td>25</td>
<td>251</td>
</tr>
<tr>
<td>S2 Batch #2</td>
<td>17202</td>
<td>5</td>
<td>N/A</td>
<td>89.38</td>
<td>50</td>
<td>405</td>
</tr>
<tr>
<td>S2 Batch #3</td>
<td>13336</td>
<td>5</td>
<td>N/A</td>
<td>78.51</td>
<td>75</td>
<td>271</td>
</tr>
<tr>
<td>S2 Batch #4</td>
<td>2989</td>
<td>5</td>
<td>N/A</td>
<td>72.97</td>
<td>100</td>
<td>74</td>
</tr>
</tbody>
</table>

The timeout rates for negative examples provide important context to our completeness results. Completeness in our context does not guarantee a decision procedure, only a semi-decision procedure. Because backward-chaining search is highly non-deterministic, it is common for negative queries to be non-terminating. For bug-detection applications, this is often acceptable, as timeouts can be understood as the absence of easily-detected bugs. At the same time, this motivates future work on decision procedures.

Finally, we conclude our research and provide a number of possible directions for future research by which we can continue to develop sound and complete proof search algorithms with education and bug detection in mind.
6 Conclusion and Future Work

Throughout this work, we proposed the first backward-chaining proof search algorithm for staged linear logic programs. We introduced a novel algorithm by which it is possible to detect possible software logic-level bugs and exploits via backward chaining from a given goal statement, and iteratively searching the program space to find a trace such that the goal statement can be satisfied starting from an initial configuration with respect to a predicate and concluding in the goal statement. Additionally, we implement our proposed approach in Python and evaluate the algorithm performance with respect to a plethora of manually and artificially constructed test cases. We find that the proposed algorithm, as implemented as a Python program, is able to effectively terminate and detect bugs for structurally complex instances, terminating for 90.04% of instances when bound by the
number of stages and terminating for 72.97% of instances when bound by the number of rules under the testing time constraint. As such, we conclude that the proposed algorithm is effective in the detection of bugs that arise from errors in the programmer’s logic. Finally, we have provided open access to the Python implementation of our algorithm.[4]

Additionally, The experimental evaluation has provided motivation for several potential algorithmic improvements in future work. Firstly, though we provided the first backward-chaining algorithm for staged linear logic programs, there is rich literature on linear logic proof search and automated deduction more broadly. We hope to exploit insights from that literature to develop an optimized search algorithm with greater practical performance via the introduction of heuristics and other techniques. Secondly, the underlying theory of linear logic is known to have connections with other formalisms, such as finite automata,

[4]github.com/TheSittingCat/Ceptre_Backward_Research
for which decision procedures are known. Such decision procedures are often developed by computing a bound on search depth upon which completeness can be guaranteed. We hope to exploit these theoretical connections to develop such a bound and provide a decision procedure. Such a decision procedure would elevate search from a bug-catching tool to a verification tool.

Furthermore, we acknowledge that the currently introduced algorithm requires the conversion of already established programs to linear logic programs, or alternatively, symbolic representations of a linear logic program such that they can be parsed by the provided program. This conversion can be both cumbersome and prone to mistakes. In future work, we would like to focus on the process of automatic generation of a staged linear logic program given the behavior of prewritten software. To this end, we hypothesize that it is possible to automata learning for the automatic generation of an automata such that it mimics the behavior of black-box software, and from which a staged linear logic program can be directly derived.

Several application areas have been identified for staged linear logic programs, such as procedural content generation for games and modeling of social software, including the security and privacy properties of social software. We call for future work applying our algorithm to bug-finding for these domains.

7 Acknowledgments

I would like to thank my advisor, Professor Rose Bohrer, for her unwavering support and dedication to the success and completion of this thesis, as well as her valuable feedback throughout my research journey. Additionally, I would like to thank Professor Xiaoyan Sherry Sun for acting as the reader of this work, as well as their valuable feedback and suggestions along the way. Finally, I would like to thank my wife, Shayesteh, without whom I wouldn’t be able to complete this project in a timely manner, and whose support gave me the hope and dedication to continue this exciting work.
8 References


33


