Compressing Terms in ProVerif with Hash Consing and Variable Renaming

Matthew Zahra

Word count: 4993 - via texcount

MCompSci Computer Science - Part B

Trinity 2025

Acknowledgements

I would like to extend my utmost thanks to my supervisor, Dr Vincent Cheval, for bringing me onto such an exciting project, for answering my numerous questions and, for listening to my far-fetched ideas. I would also like to thank my family and my girlfriend for their unwavering support and for listening to my ongoing battle with OCaml.

Abstract

ProVerif is a symbolic cryptographic protocol verifier used to prove properties about protocols. Once a protocol and security property have been formulated in ProVerif's own extension of π-calculus it represents them as a set of Horn clauses and uses saturation to try to reach a decision. These clauses consist of a universally quantified disjunction of terms, which are combinations of function, constant, and variable symbols. Terms are currently represented as memory inefficient trees, meaning ProVerif can be difficult to run on consumer hardware. To combat this, we exploit hash consing techniques, allowing the sharing of structurally equal subterms by representing terms as directed acyclic graphs (DAGs). However, for complex protocols (e.g. TLS) this level of compression is still insufficient. We then look to rename variables to increase subterm sharing between clauses, ensuring this renaming is injective with respect to each clause. We quantify the difficulty of finding an optimal solution to this problem by proving it to be NP-hard and explore solving the problem via an Integer Linear Program. We then present and benchmark six tractable approximation algorithms, with the best (Level 5) producing excellent levels of memory compression on our benchmark.

Contents

1 Introduction and Background

1.1 ProVerif

Security protocols appear in a wide range of applications and being able to guarantee their integrity is a necessity. There are several tools that try and verify security protocols, such as DeepSec [1] and Tamarin [2], but this report focuses on the tool ProVerif [3]. ProVerif is an open source automatic cryptographic symbolic protocol verification tool that has been developed for over 20 years (it is written in OCaml and an overview of its research can be found at [4]). It allows us to verify several different properties such as correspondence properties (events which are temporally interdependent) and equivalence properties (ensuring adversaries cannot differentiate two different events). ProVerif has verified many established protocols such as TLS [5] and was used by Google to verify EKEP [6].

ProVerif accepts a description of the protocol and the property to verify in its own extension of applied π-calculus [7]. It outputs one of: true; false (with an attack showing this); unable to prove property. The last case is due to the undecidability of the verification problem - due to some approximations ProVerif may generate a false attack or sometimes loop forever. However, when it outputs true or false then these are known to be correct (proofs found at [3]).

Below is an example of the π-calculus translation of a protocol which ProVerif successfully found a vulnerability in.

```
let processS =
    in(c, (h2: host, m: bitstring));
    get keys(=h2, k2) in (* get the key of h2 from the key table *)
    let (h1: host, n1: nonce, n2: nonce) = decrypt(m, k2) in
    get keys(=h1, k1) in (* get the key of h1 from the key table *)
    new k: key;
    out(c, (encrypt((h2, k, n1, n2), k1), encrypt((h1, k), k2))).
```

Figure 1: Extract Yahalom protocol description.

ProVerif takes a protocol description and translates it into a set of first order logic Horn clauses. It saturates this with some user-provided lemmas to derive a simpler set of Horn clauses ($\mathbb{C}_{sat}$). If this terminates, it translates the security properties into a set of request clauses $\{R_i\}$ and saturates $\mathbb{C}_{sat} \cup R_i$, for each request clause R_i. If this terminates, it tests if the corresponding security property is satisfied. If this is not true for some R_i, ProVerif derives an attack to show the violation. If the attack is then shown to be genuine, the protocol does not satisfy the property in question; if the attack is a false attack then it says it cannot prove this.

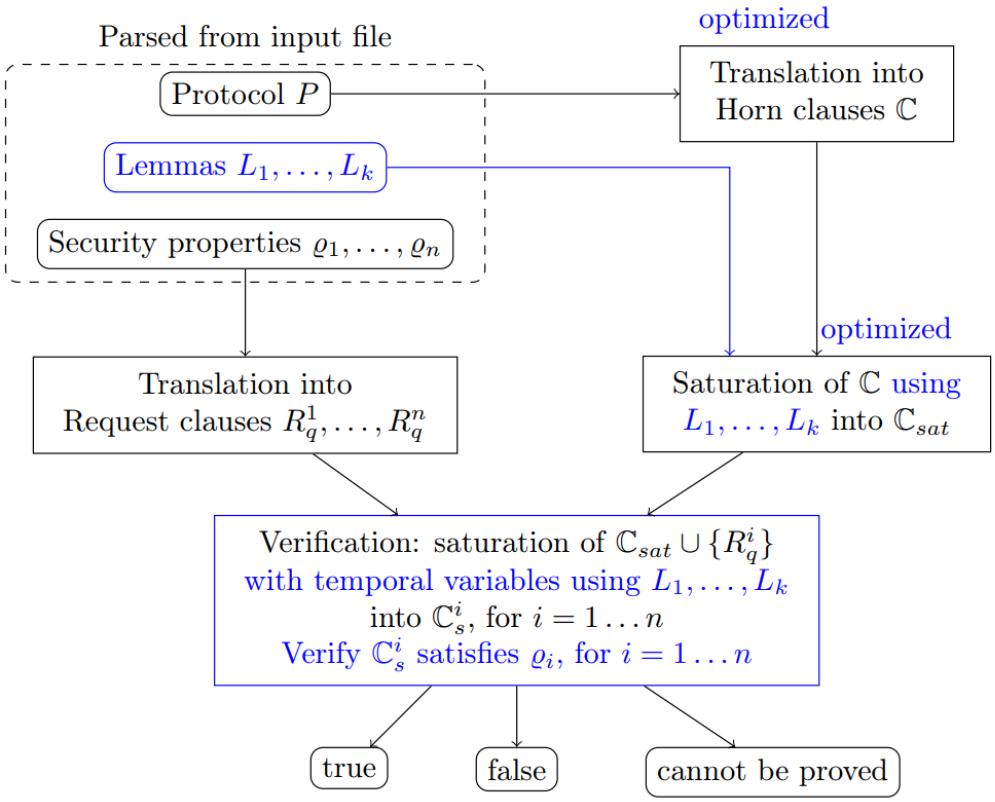


Figure 2: ProVerif's process from [8].

ProVerif was recently updated (adding the blue sections in Figure 2) which improved its time efficiency [8], but not its space efficiency. While protocol descriptions are often small text files, ProVerif's internal state can blow up to many GBs, which is problematic for users. We aim to tackle this problem.

1.2 Defining Terms

We denote by $\mathcal{V}$ the set of variables, $\mathcal{C}$ the set of constant symbols and $\mathcal{F}$ the set of function symbols. Terms in ProVerif are typed - we assume each type takes some value in $\mathbb{N}$. We denote by $\text{type}(c)$, $\text{type}(x)$, $\text{type}(f)$ and $\text{type}(\bar{f}_i)$ respectively the type of constant c, variable x, output of the function symbol f and the i-th argument of f. Terms in ProVerif are variables, constants, or an application of a function symbol on terms. We denote by $\mathcal{T}$ the set of well-typed terms. We overload the function "type" on terms so that the type of a term is defined as the type of its root symbol. In particular, we have $\text{type}(f(t_1, \ldots, t_n)) = \text{type}(f)$. A term t is *well-typed* when it respects the types and arity of its function symbols - i.e. variables and constants are always well-typed, and $t = f(t_1, \ldots, t_n)$ is well-typed when $f \in \mathcal{F}$ of arity n and for all $i \in \{1, \ldots, n\}$, $\text{type}(t_i) = \text{type}(\bar{f}_i)$ and t_i is well-typed. We will only consider well-typed terms.

We define the set of subterms of $t \in \mathcal{T}$, denoted $\text{st}(t)$, as follows:

$$\text{st}(t) = \begin{cases} \{t\}, \text{if } t \in \mathcal{V} \cup \mathcal{C} \\ \{t\} \cup \bigcup_{i=1}^{n} \text{st}(t_i), \text{ if } t = f(t_1, \ldots, t_n) \text{ for some } f \in \mathcal{F} \end{cases}$$

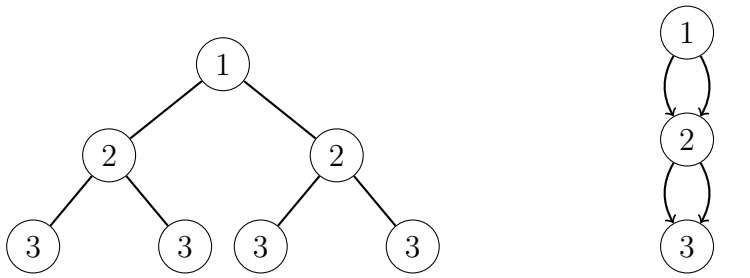
This naturally extends to sets of terms such that $\text{st}(\{t_1, \ldots, t_n\}) = \bigcup_{i=1}^{n} \text{st}(t_i)$.

Given $t \in \mathcal{T}$, we denote by $\text{var}(t)$ the set of variables occurring in t. This definition naturally extends to sets of terms.

1.3 Hash Consing

Currently, ProVerif internally represents terms as trees, with a node for each function, variable and constant symbol appearance, connecting function symbols and their arguments via edges. This means each node is stored separately every time it appears. We propose using *hash consing* techniques [9] to share structurally equal objects. We take inspiration from T. Kutsia [10], where terms are represented as **Di-**

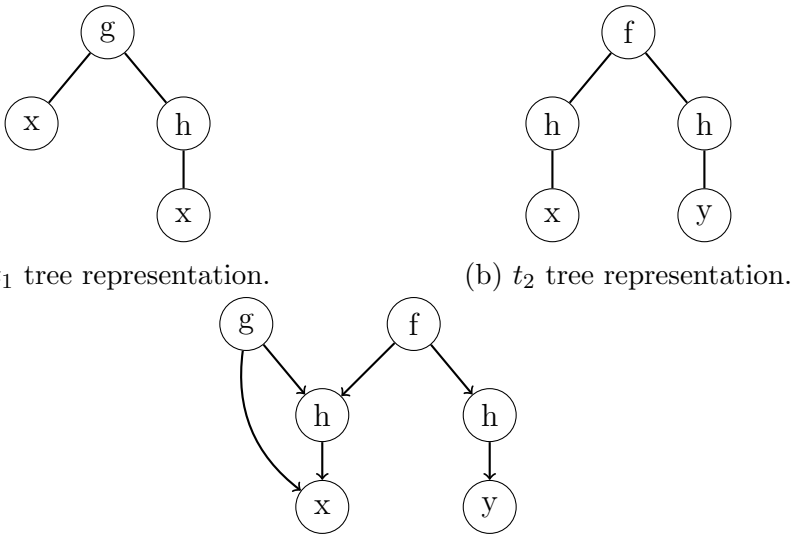
rected Acyclic Graphs (DAGs). This means each subterm node is stored exactly once and different terms can share nodes. Below we see that DAG representation can exponentially reduce memory usage (Definition 1.1).



(a) Tree (b) DAG

Figure 3: Exponential space save by DAG representation.

We maintain a hash consing table of nodes, and every time we create a new node we first check if it already exists, returning it if it does; adding it to our hash consing table if it doesn't. To allow OCaml's garbage collector to reclaim memory that is not live but still present in the table, we use weak pointers in our table - these aren't considered by the garbage collector when it determines if it can erase a value or not.



(a) t_1 tree representation. (b) t_2 tree representation.

(c) t_1 and t_2 DAG representation.

Figure 4: Two terms represented as trees and then as DAGs.

Note: Unless stated assume all terms are represented by a DAG.

Definition 1.1. *We denote by* ***memory*** *the amount of storage needed to represent a set of terms. In both Tree and DAG representation, the total* ***memory*** *is the number of nodes in their representation. For a set S of DAG represented terms, this is equivalent to $|st(S)|$, so minimising its* ***memory*** *is equivalent to minimising the number of subterms.*

1.4 The Renaming Problem

We wish to maximise the memory compression DAG representation gives us by renaming variables.

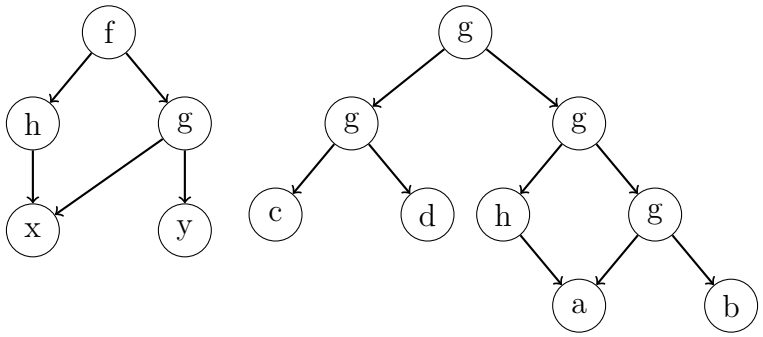
Definition 1.2. *Given $V \subseteq \mathcal{V}$, $\sigma : V \to \mathcal{V}$ is called a variable renaming when $\forall v \in V$, $\text{type}(v) = \text{type}(\sigma(v))$. The application of a variable renaming σ on a term t, denoted $t\sigma$, is the term obtained by replacing every instance of a variable v in t by $\sigma(v)$ (if $v \notin \text{dom}(\sigma)$, let $\sigma(v) = v$). This is naturally extended to sets of terms.*

Different renamings change the total memory. However, not all renamings are allowed. We note that terms $f(x, y)$ and $f(z, z)$ are not equivalent, as in the former the variables are free to take different values. ProVerif produces a universal quantification over a conjunction of Horn clauses, which themselves are a disjunction of terms. Because universal quantification can move over conjunctions but not disjunctions, we allow renamings to modify the variables shared between clauses. However, we enforce that any pre-existing variable sharing between terms within each clause is preserved. We call such a renaming **structure-preserving**. *Note: From now, clauses are treated as sets of terms.*

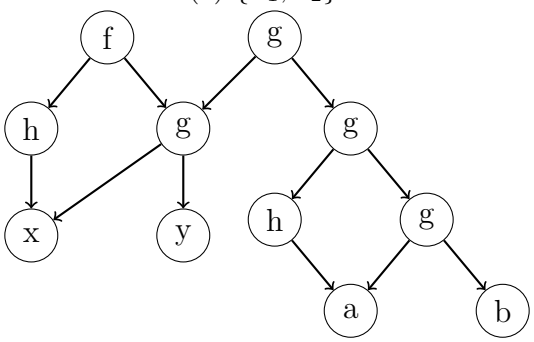
Definition 1.3. *Let a clause $\mathfrak{C} \subseteq \mathcal{T}$ be a set of terms. We say that variable renaming, σ, is structure-preserving with respect to $\mathfrak{C}$ when*

$$\forall u, v \in \text{var}(\mathfrak{C}), \sigma(v) = \sigma(u) \implies u = v$$

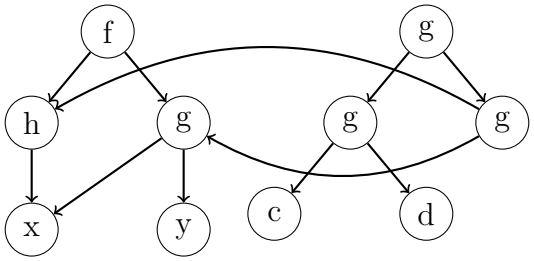
To illustrate different renamings, consider two clauses $\mathfrak{C}_1 = \{f(h(x), g(x, y))\}$ and $\mathfrak{C}_2 = \{g(g(c, d), g(h(a), g(a, b)))\}$ where $a, b, c, d, x, y \in \mathcal{V}$ and $f, g, h \in \mathcal{F}$. Note that $|\operatorname{st}(\mathfrak{C}_1)| = 5$ and $|\operatorname{st}(\mathfrak{C}_2)| = 9$. Since the two clauses do not share variables, $|\operatorname{st}(\{\mathfrak{C}_1, \mathfrak{C}_2\})| = 14$. Applying the renaming $\sigma = [c \to x, d \to y]$ reduces the memory by 3, but applying the renaming $\sigma' = [a \to x, b \to y]$ improves on this by reducing the memory by 4: $|\operatorname{st}(\{\mathfrak{C}_1, \mathfrak{C}_2\}\sigma)| = 11$ and $|\operatorname{st}(\{\mathfrak{C}_1, \mathfrak{C}_2\}\sigma')| = 10$.



(a) $\{\mathfrak{C}_1, \mathfrak{C}_2\}$.



(b) $\{\mathfrak{C}_1, \mathfrak{C}_2\}\sigma$.



(c) $\{\mathfrak{C}_1, \mathfrak{C}_2\}\sigma'$.

Figure 5: Applying σ and σ' to $\{\mathfrak{C}_1, \mathfrak{C}_2\}$.

Definition 1.4. *Let s and s' be 2 distinct terms. We say that s and s' are* ***similar*** *iff:*

(i) s *and* s' *are members of* ***different*** *clauses,* $\mathfrak{C}_1$ *and* $\mathfrak{C}_2$.

(ii) $\exists$ *renaming* σ *that is structure-preserving with respect to* $\mathfrak{C}_1$ *and* $\mathfrak{C}_2$ *such that* $s = s'\sigma$.

We denote this by $s \sim s'$.

We now define the problem corresponding to minimising the total memory. *Note: ProVerif produces clauses whose variables are pairwise disjoint from one another, so we assume this.*

Definition 1.5. *Define MINSUBTERM as the following optimisation problem.*

Given a set of clauses $\{\mathfrak{C}_1, \ldots, \mathfrak{C}_n\}$ *where for* $i \neq j \implies \text{var}(\mathfrak{C}_i) \cap \text{var}(\mathfrak{C}_j) = \emptyset$, *return a variable renaming* σ *such that*

(i) σ *is structure-preserving with respect to* $\mathfrak{C}_i, \forall i \in \{1, \ldots, n\}$

(ii) $\forall \sigma'$ *satisfies (i),* $|\bigcup_{i=1}^{n} \text{st}(\mathfrak{C}_i\sigma')| \geq |\bigcup_{i=1}^{n} \text{st}(\mathfrak{C}_i\sigma)|$

We formally define the decision version of the problem for the purpose of proving the difficulty of MINSUBTERM.

Definition 1.6. *Define k-MINSUBTERM as the following decision problem. Given* $k \in \mathbb{N}$ *and a set of clauses* $\mathfrak{C}_1, \ldots, \mathfrak{C}_n$ *where for* $i \neq j \implies \text{var}(\mathfrak{C}_i) \cap \text{var}(\mathfrak{C}_j) = \emptyset$, *return a YES if there exists a variable renaming* σ *such that*

(i) σ *is structure-preserving with respect to* $\mathfrak{C}_i, \forall i \in \{1, \ldots, n\}$

(ii) $|\bigcup_{i=1}^{n} \text{st}(\mathfrak{C}_i\sigma)| \leq k$

otherwise return NO.

2 Quantifying the Difficulty

To demonstrate there are no optimal polynomial-time solutions, we prove the problem is NP-hard.

2.1 Proof of NP-hardness

To show the difficulty of MINSUBTERM, we reduce k-CLIQUE.

Definition 2.1. *Define k-CLIQUE as the following decision problem. Given as input* $k \in \mathbb{N}$ *and an undirected graph* $G = (V, E)$*, determine whether there exists* $V' \subseteq V$ *such that* $|V'| \geq k$ *and* $\forall u, v \in V'$*;* $u \neq v \implies (u, v) \in E$*.*

Theorem 2.2. *k-CLIQUE is NP-complete.*

Proof. See [11]. □

Theorem 2.3. *k-MINSUBTERM is NP-complete.*

Proof. We begin by polynomial-time reducing k-CLIQUE to an instance of k-MINSUBTERM.

Let $g, h, \mathcal{E} \in \mathcal{F}$ and $\forall i \in \mathbb{N}; v_i, x_i \in \mathcal{V}$. Without loss of generality, let $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ where each $e_i = (v_a, v_b) \in V \times V$. Also assume that $(v_a, v_b) \in E \iff (v_b, v_a) \in E$. The problem is trivial if $k > n$, so assume $k \leq n$. We define the following terms:

(i) $\forall e_i = (v_a, v_b) \in E$; let $s_i = \mathcal{E}(v_a, v_b)$. Let $t_1 = g(s_1, \ldots, s_m, v_1, \ldots, v_n)$. This represents all the vertices and edges of the graph.

(ii) $\forall i, j \in [1, k], i \neq j$; let $p_{i,j} = \mathcal{E}(x_i, x_j)$.
Let $t_2 = h(p_{1,2}, \ldots, p_{1,k}, p_{2,1}, p_{2,3}, \ldots, p_{k,k-1})$. This represents k vertices that are all directly connected to each other via edges.

We now call k-MINSUBTERM with $k = 2 + m + n$ and clauses $\mathfrak{C}_1 = \{t_1\}, \mathfrak{C}_2 = \{t_2\}$. We note that $|\operatorname{st}(t_1)| = (1+m+n)$ subterms* and $|\operatorname{st}(t_2)| = 1+k(k-1)+k = 1+k^2$ subterms. We can trivially see that the best compression a structure-preserving renaming (σ) could achieve is if for each i, $\sigma(x_i) = v_a$ for some a, such that for each

*Assume each vertex is connected to at least one other vertex - if not, problem is trivial if k = 1, otherwise ignore these disconnected vertices.

i, j we have $p_{i,j}\sigma = s_l$ for some l (without loss of generality, we rename variables in t_2 to those in t_1). This is optimal as it causes all strict-subterms of $t_2\sigma$ to appear in t_1. Further, t_2 itself cannot ever be shared as function symbol h doesn't appear anywhere else. This would leave $2 + m + n$ subterms in total, as each argument of h in $t_2\sigma$ would be identical to some argument of g in t_1. We return YES iff this can be acheived.

This reduction is sound. If there does exist a clique of size k, without loss of generality say it uses vertices $\{u_1, \ldots, u_k\}$, then by renaming $[x_i \to u_i]$ we achieve the compression above. If there is no such clique, then no renaming that leaves $\leq 2 + m + n$ subterms exists, as this would imply there is a σ such that all strict-subterms of $t_2\sigma$ appear in t_1 (as t_1 always contributes $1 + m + n$ subterms, and $t_2\sigma$ must contribute at least 1 due to its root symbol h). By the interpretation of t_2 this implies that there exist k vertices all connected to one another via an edge, which is a contradiction.

It is clear that this reduction is completed in polynomial-time, since t_1 is of size $\mathcal{O}(|V| + |E|)$, t_2 is of size $\mathcal{O}(k^2)$ and both require a single pass over G. Since k-CLIQUE is NP-complete, this implies that k-MINSUBTERM is NP-hard.

We now show that k-MINSUBTERM $\in$ NP. The certificate would be the resulting renaming. We can verify it is structure-preserving with respect to each clause in $\mathcal{O}(n)$ simply by ensuring it is injective and that it respects types for each clause. To count the number of subterms, we apply the renaming, then explore each term, counting each time we see a new subterm. This is completed in $\mathcal{O}(n)$. This certificate is polynomial in size since it has 1 entry per variable. Hence k-MINSUBTERM $\in$ NP.

Therefore since k-MINSUBTERM is NP-hard and is in NP, k-MINSUBTERM is NP-complete. □

Theorem 2.4. *MINSUBTERM is NP-hard.*

Proof. We reduce k-MINSUBTERM to MINSUBTERM, by running the MINSUB-

TERM algorithm on the input instance. Using the same polynomial-time process, we count the number of subterms this renaming leaves. Since this algorithm produces the optimal renaming, if the resulting count is $\leq k$ return YES, otherwise return NO. This reduction is trivially done in polynomial-time and as k-MINSUBTERM is NP-complete, then MINSUBTERM is NP-hard. □

Therefore, we cannot hope to efficiently achieve optimality.

2.2 Failure of Variable Minimisation

A tempting approach is to look for renamings that minimise the total number of variables. However, consider the following: $t_1 = f(g(x, y), p(x), q(x))$ and $t_2 = f(g(w, z), p(z), q(z))$ are terms that appear in different clauses. Let $\sigma = [x \to w, y \to z]$ and $\sigma' = [x \to z, y \to w]$. We note that $|\operatorname{st}(t_1\sigma) \cup \operatorname{st}(t_2\sigma)| = 9$ while $|\operatorname{st}(t_1\sigma') \cup \operatorname{st}(t_2\sigma')| = 8$, despite both minimising the number of variables.

2.3 Integer Linear Programming Formulation

To find the optimal renaming we phrase the problem as a 0/1 Integer Linear Programming (ILP) problem.

For ease, we rename each variable to a fresh variable. Let $C = \{\mathfrak{C}_1, \ldots, \mathfrak{C}_n\}$ be the set of clauses we want to compress and assume $\operatorname{var}(\mathfrak{C}_i) \cap \operatorname{var}(\mathfrak{C}_j) = \emptyset$ for $i \neq j$. Here $\mathcal{V} = \bigcup_{i=1}^{n} \operatorname{var}(\mathfrak{C}_i)$. Define $\mathcal{V}'$ to be a set of fresh variables, such that $\mathcal{V} \cap \mathcal{V}' = \emptyset$. Further, let $|\mathcal{V}'| = |\mathcal{V}|$ and for each distinct variable in $\mathcal{V}$, let there be a distinct variable of the same type in $\mathcal{V}'$. Below, we only allow variables to take values in $\{0, 1\}$ and we assume a strict total order $(<)$ on all $s \in \bigcup_{i=1}^{n} \operatorname{st}(\mathfrak{C}_i)$.

$$
\begin{aligned}
&\text{minimise } \sum_{s} y_s && \forall s \in \textstyle\bigcup_{i=1}^{n} \mathrm{st}(\mathfrak{C}_i), \mathrm{var}(s) \neq \emptyset && \\
&\text{subject to } \sum_{\substack{j \in \mathcal{V}' \\ \mathrm{type}(i)=\mathrm{type}(j)}} \sigma_{i,j} = 1, && \forall i \in \mathcal{V} && (1)\\
&\qquad \sum_{\substack{i \in \mathrm{var}(\mathfrak{C}) \\ \mathrm{type}(i)=\mathrm{type}(j)}} \sigma_{i,j} \leq 1, && \forall j \in \mathcal{V}', \forall \mathfrak{C} \in C && (2)\\
&\qquad \rho_{i,j} = \sum_{\substack{\alpha \in \mathcal{V}' \\ \mathrm{type}(i)=\mathrm{type}(\alpha)}} \sigma_{i,\alpha}\sigma_{j,\alpha}, && \forall \mathfrak{C}_k \in C, \forall i \in \mathrm{var}(\mathfrak{C}_a), \forall j \in \mathrm{var}(\mathfrak{C}_b), \\
& && a \neq b, \mathrm{type}(i) = \mathrm{type}(j) && (3)\\
&\qquad x_{s,s'} \geq 1 - \frac{\rho_{i_1,j_1} + \cdots + \rho_{i_k,j_k}}{k}, && \forall \mathfrak{C}_k \in C, \forall s \in \mathrm{st}(\mathfrak{C}_a), \forall s' \in \mathrm{st}(\mathfrak{C}_b), \\
& && a \neq b, s \sim s', |\mathrm{var}(s)| = k && (4)\\
&\qquad y_s \geq \prod_{s' < s} x_{s,s'} && \forall \mathfrak{C}_k \in C, \forall s \in \mathrm{st}(\mathfrak{C}_a), s' \in \mathrm{st}(\mathfrak{C}_b), \\
& && a \neq b, s \sim s', \mathrm{var}(s) \neq \emptyset && (5)
\end{aligned}
$$

The above is not strictly a linear program due to constraints (3) and (5), but is more intuitive. We can linearise it by doing the following:

Let $x_1, x_2, z \in \{0, 1\}$ and say we have constraint $z = x_1 \times x_2$. This is equivalent to the set of linear constraints (a similar idea lets us linearise inequalities too):

$$z \leq x_1, z \leq x_2, z \geq x_1 + x_2 - 1$$

The variables represent the following:

1. We have one y_s per non-constant subterm s. $y_s = 1$ implies that we count s as it is not identical to some other subterm before it in the ordering after the renaming (see constraint (5)). Since renamings don't affect constant terms, minimising the objective function minimises the total number of subterms.

2. We have one $\sigma_{i,j}$ per old-fresh variable pair. $\sigma_{i,j} = 1$ implies i is mapped to j in the renaming.

3. We have one $\rho_{i,j}$ for each pair of old variables that appear in different clauses.

$\rho_{i,j} = 1$ implies that i and j are mapped to the same variable in the renaming.

4. We have one $x_{s,s'}$ for each pair of similar terms from different clauses. $x_{s,s'} = 1$ implies that after the renaming, s and s' are identical.

The constraints enforce the following:

1. Ensures that each variable is mapped to exactly 1 fresh variable.
2. Ensures that the renaming is structure-preserving with respect to each clause.
3. Indicates if different variables from different clauses get mapped to the same fresh variable.
4. Indicates if after renaming, terms s and s' become identical.
5. Sets $y_s = 1$ if s is a non-constant subterm that has not become identical to some other term before it in the ordering, so should be counted.

The renaming can be extracted by mapping each variable i to j iff $\sigma_{i,j} = 1$. If storing the ILP takes up too much space, we would build it for batches of terms, compressing each individually.

2.4 Solving the Integer Linear Program

To solve the 0/1 ILP described we split the constraints into 5 distinct types:

Positive Implication: $y_1 \times \ldots \times y_k = 1 \implies x = 1$

Negative Implication: $y_1 \times \ldots \times y_k = 0 \implies x = 1$

Exactly One: Given $\{y_1, \ldots, y_k\}$, exactly one is 1; the rest are 0

At Most One: Given $\{y_1, \ldots, y_k\}$, at most one of them is set to 1

One Among: Given pairs $\{(x_1, y_1), \ldots, (x_k, y_k)\}$, $z = x_1y_1 + \ldots + x_ky_k$ (violated if more than 1 pair multiplies to 1 as all values are in $\{0, 1\}$)

To solve the ILP, we try to minimise the number of nondeterministic choices made. We maintain a list of unsatisfied (but not violated) constraints with the current assignment. We pick one and make a nondeterministic decision. We then iterate through the other constraints, making any deterministic assignments - e.g. say we assign $y_i = 0$, and it appears in the product of a Negative Implication, we can then deterministically assign the corresponding $x = 1$. We would terminate this branch if $x = 0$ already, otherwise remove this constraint from the list as future assignments will not change its satisfaction.

We found our solution took an unreasonably long time, even for small inputs. This is particularly problematic due to the undecidable nature of the verification problem - it may become unclear if ProVerif is looping forever or just taking a long time to compress clauses. Two current industry standard solvers are Gurobi [12] and CPLEX [13]. We did not pursue using them as the free tiers were too limited and a paid version is not a suitable extension to ProVerif. We also anticipate that the sheer size of the problem will cause these solvers to take too long. Seen here [14], both Gurobi and CPLEX take $> 100,000$ seconds for a problem with only a few hundred binary variables and constraints. One potential solution would be to relax some constraints to find an approximate solution. We felt that we would still have a very large blow up. We now present an alternative route.

3 Approximation Algorithms

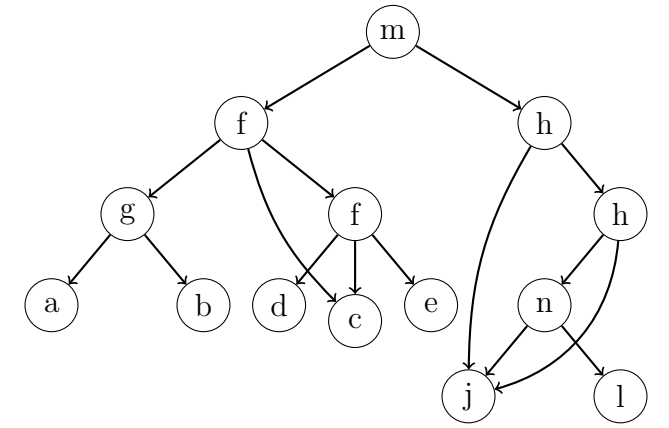
With efficiency in mind, we explore approximation algorithms. We have developed six variants, examined if fresh variables (used in renamings) should be stored in queues or stacks (referred to as containers) and considered introducing randomness to variable selection.

The algorithms we designed are fed clauses one at a time and compress them with some knowledge of previous renamings in an *online* fashion. This means we

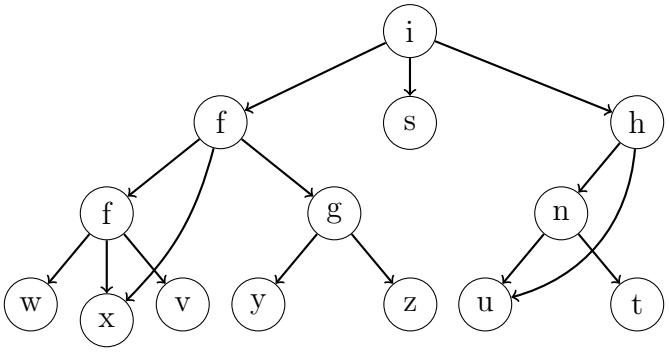
don't assume we have enough memory to represent all the clauses at once. We allow the algorithms to maintain some small intermediate data structure that gives information about previous renamings (for previous clauses in the same protocol) to aid decisions. An algorithm can find details about function symbols or variable types via a single pass over the protocol description.

Since we are using hash consing techniques and DAG representation, we allow algorithms to generate fresh variables whenever they try to draw from an empty container and then store them in future iterations of the container, as well as record new terms in the hash consing table. The containers contain the fresh variables used in renamings, and are stored in the intermediate data structure. We treat all variables as having the same type in the following explanations, but extending them to multiple types is trivial.

We consider the following two terms (from different clauses) to help illustrate some of the algorithms. We colour-code variables to show which are identical post-renaming, and colour-code function symbols iff they are the root of a subterm that is shared across the two terms post-renaming.



(a) t_1



(b) t_2

Figure 6: Terms t_1 and t_2 from different clauses.

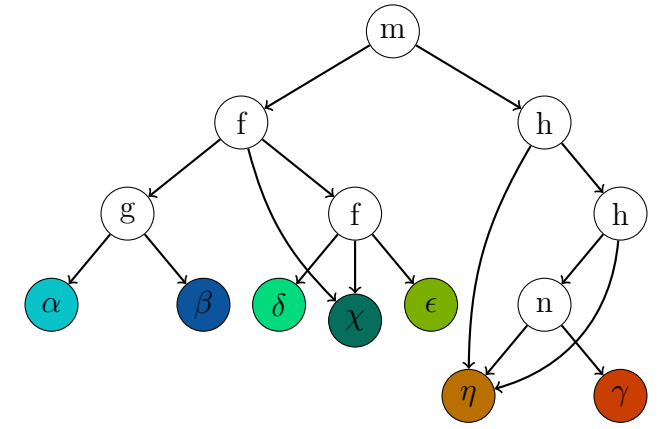
We note that $|\operatorname{st}(t_1) \cup \operatorname{st}(t_2)| = 28$. We explore terms in a Depth-First Search (DFS) approach, exploring a function symbol's children from left to right. We will also utilise queues as our containers in the examples.

The algorithms' correctness is by construction - we only reset intermediate data structures between clauses and ensure renamings are injective with respect to each clause, as when we rename $[a \to b]$ (with b fresh or drawn from a container), we temporarily discard b until the next clause. Hence, all renamings produced are structure-preserving.

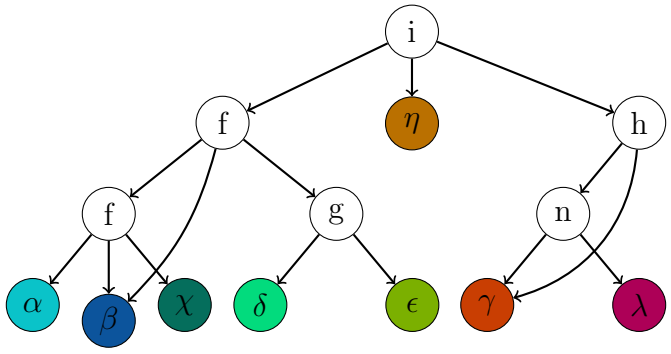
3.1 Level 0 - Naive Greedy

Level 0 explores terms in a DFS approach, renaming variables as soon as it sees them. Its intermediate data structure is just a single container of variables.

For t_1 and t_2, it produces the mapping: $\sigma_0 = [a \to \alpha, w \to \alpha, b \to \beta, x \to \beta, c \to \chi, v \to \chi, d \to \delta, y \to \delta, e \to \epsilon, z \to \epsilon, j \to \eta, s \to \eta, l \to \gamma, u \to \gamma, t \to \lambda]$.



(a) $t_1\sigma_0$



(b) $t_2\sigma_0$

Figure 7: Subterm sharing between $t_1\sigma_0$ and $t_2\sigma_0$.

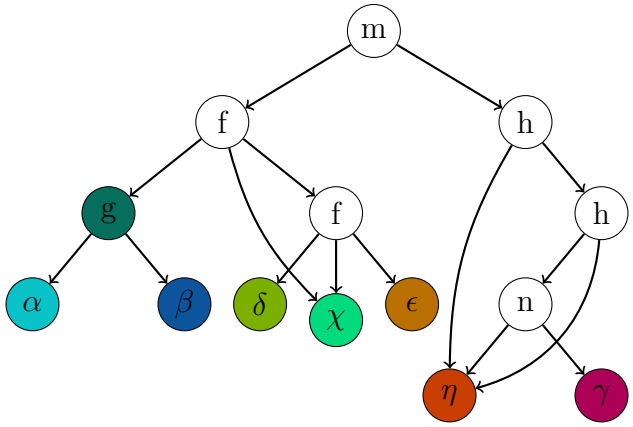
$|\operatorname{st}(t_1\sigma_0) \cup \operatorname{st}(t_2\sigma_0)| = 21$. We share variables between the two terms but we don't share any non-trivial subterms.

3.2 Level 1 - Match Parent Function Symbol

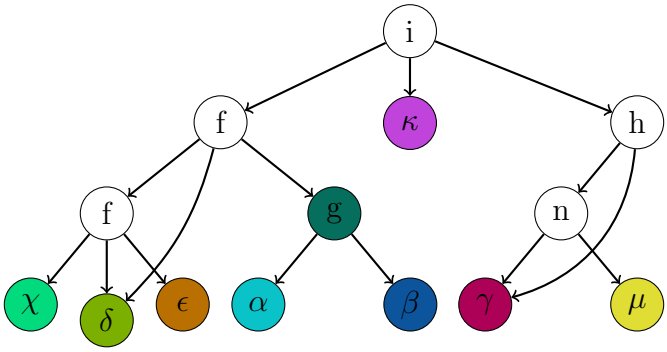
Level 1 explores terms in a DFS approach, distinguishing variables by their parent function symbol. Its intermediate data structure is an array of size $|\mathcal{F}| + 1$ of containers: one per function symbol (and one for variables with no parent symbol). When it encounters an un-renamed variable, it renames it to a variable drawn from the container corresponding to its parent symbol - this depends on where it first sees

the variable. This looks to improve on Level 0 by promoting sharing of subterms that are not just variables.

It produces the renaming $\sigma_1 = [a \to \alpha, b \to \beta, c \to \chi, d \to \delta, e \to \epsilon, j \to \eta, l \to \gamma, w \to \chi, x \to \delta, v \to \epsilon, y \to \alpha, z \to \beta, s \to \kappa, u \to \gamma, t \to \mu]$.



(a) $t_1\sigma_1$



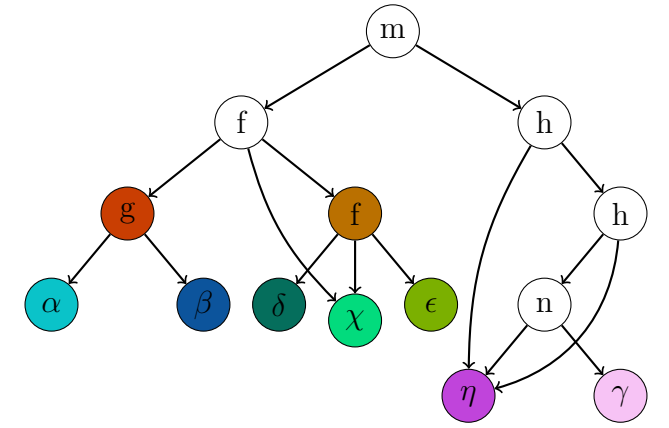
(b) $t_2\sigma_1$

Figure 8: Terms t_1 and t_2 from different clauses.

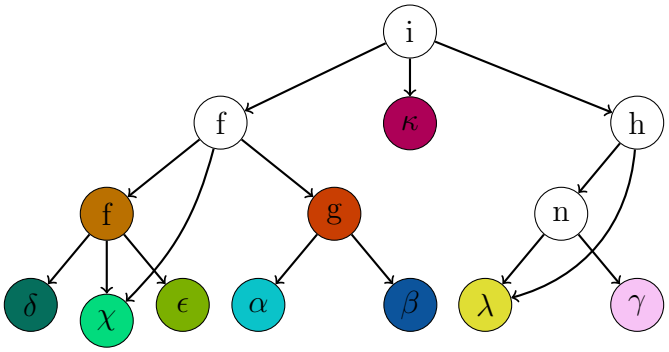
This gives us $|\,\mathrm{st}(t_1\sigma_1) \cup \mathrm{st}(t_2\sigma_1)| = 21$. While this is equal to Level 0, we note the subterm $g(\alpha, \beta)$ is now shared. The complexity of the terms hurts their compression and removing the right-most child of m and i would cause it to outperform Level 0. We note the deepest subterm with parent symbol f is still not shared between the two terms.

3.3 Level 2 - Match Parent Function Symbol and Argument Place

Level 2 explores terms in a DFS approach, distinguishing variables by their parent function symbol and their argument place. Its intermediate data structure is an array of arrays of containers - i.e. index $[i][j]$ is the container for the j-th argument of the i-th function symbol. It renames variables to those drawn from the containers that correspond to their parent symbol and argument place, where they are seen initially.



(a) $t_1\sigma_2$



(b) $t_2\sigma_2$

Figure 9: Terms t_1 and t_2 from different clauses.

It produces the renaming $\sigma_2 = [a \to \alpha, b \to \beta, c \to \chi, d \to \delta, e \to \epsilon, j \to \eta, l \to \gamma, w \to \delta, x \to \chi, v \to \epsilon, y \to \alpha, z \to \beta, s \to \kappa, u \to \lambda, t \to \gamma]$. This gives us $|\operatorname{st}(t_1\sigma_2) \cup \operatorname{st}(t_2\sigma_2)| = 20$. While this is good, it fails to utilise the potential matching

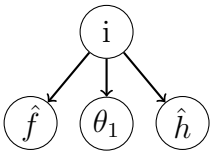
of the far right subterm - $h(n(\theta_1, \theta_2), \theta_1)$.

3.4 Level 3 - Parent Symbol Frequency (Approximation)

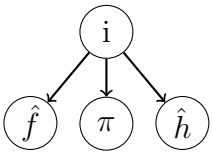
Level 3 looks to improve Level 1 by prioritising renaming variables that have parent symbols that appear frequently. Its intermediate data structure is like Level 1's, but also tracks the frequency of each function symbol. It makes two DFS passes: the first renames variables whose parent function symbol is among the n most common; the second renames anything left over. It updates the function symbol counts as it goes. With only two terms this behaves very similarly to Level 1, but we witness some differences when the number of terms gets large.

3.5 Level 4 - Bottom Up Skeleton Matching

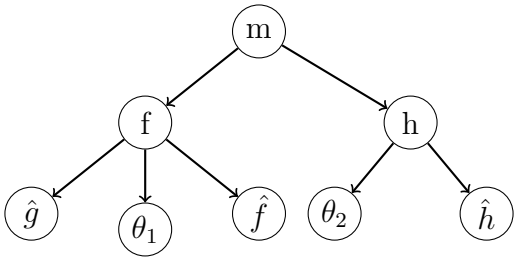
To try and overcome Level 2's limitations, Level 4 considers subterm structure. We define the notion of a depth-d skeleton: maintain a queue of fresh variables called skel_vars (add new variables when emptied, persisting between clauses). Given a term t, we explore it in a DFS fashion. We rename variables **that have not already been renamed** on a first-come-first-served basis from skel_vars, identical to Level 0. We approximate the term if it exceeds depth d (terms start at depth 1). At depth-$(d+1)$, if a variable appears we rename as normal but if a function symbol appears we replace it with its own special constant. The resulting term is the depth-d skeleton of t. Due to skel_vars maintaining its order, two terms have the same skeleton iff they are equal modulo renaming variables that have not already been renamed.



(a) Depth-1 skeleton of t_2.



(b) Depth-1 skeleton of $t_2[s \to \pi]$.



(c) Depth-2 skeleton of t_1.

Figure 10: Skeleton examples for t_1 and t_2. Denote skeleton variables by θ_i and replacement constant for symbol f by $\hat{f}$. s has already been renamed elsewhere to π in (b).

Given a depth d we recurse on any children that cause the term to exceed d in depth. If a term is not too deep (or if we have already recursed on all deep children), we build its depth-d skeleton. We then rename its variables (exploring in a DFS approach), drawing from a container that is associated with its skeleton. The intermediate data structure is a HashMap that associates skeletons with containers. Below are the first few steps when applying this algorithm to t_1 with $d = 1$.

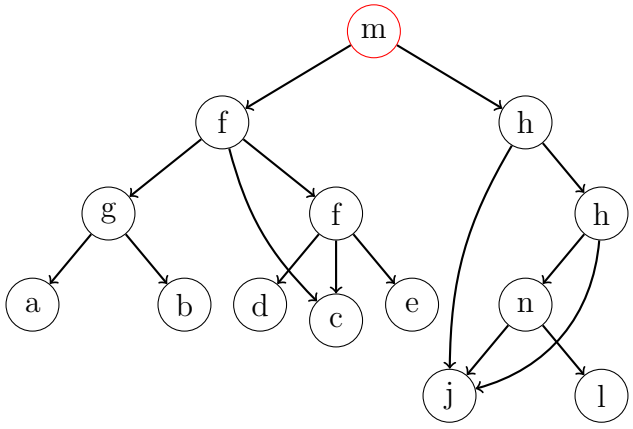


Figure 11: (a) Start at root - both children too deep, so recurse.

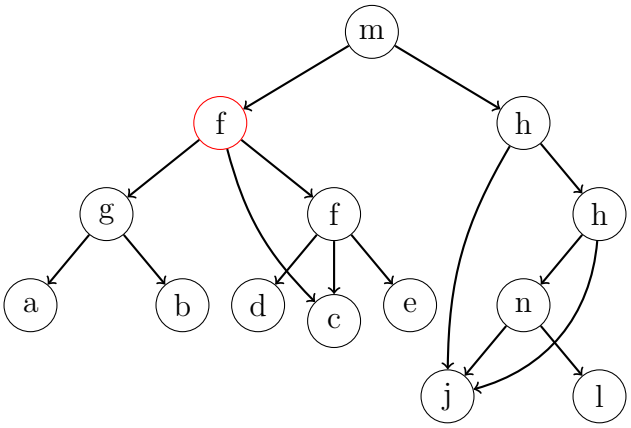


Figure 11: (b) 2 of 3 children too deep – recurse on each.

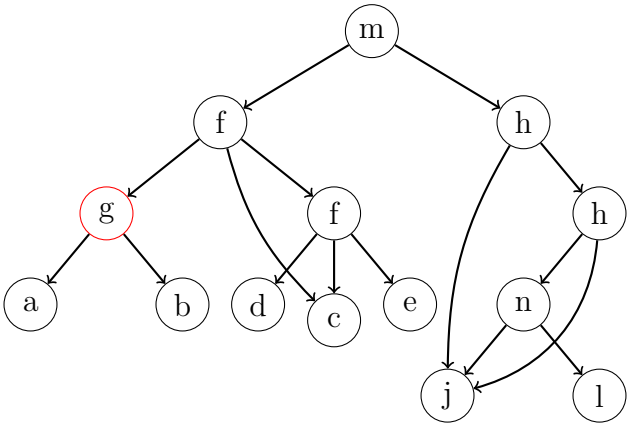


Figure 11: (c) Not too deep - generate skeleton (rooted at g) and rename variables accordingly.

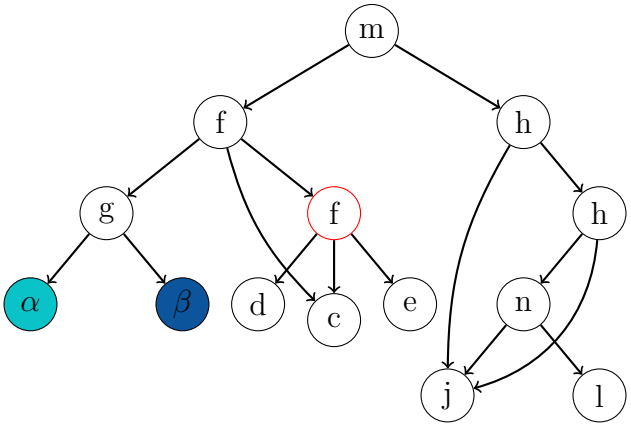
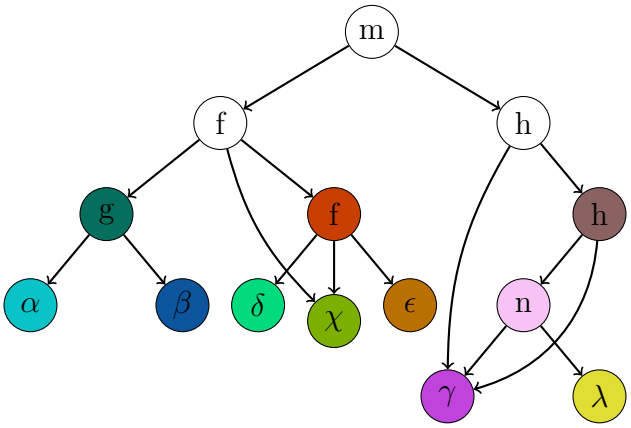
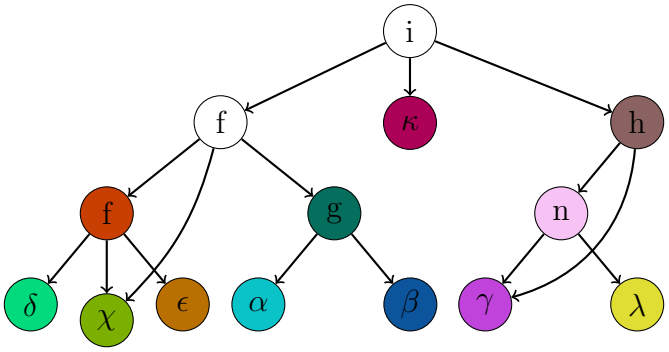


Figure 11: (d) Recurse on next child that is too deep.

Setting $d = 1$, we produce the renaming $\sigma_4 = [a \to \alpha, b \to \beta, d \to \delta, c \to \chi, e \to \epsilon, j \to \gamma, l \to \lambda, w \to \delta, x \to \chi, v \to \epsilon, y \to \alpha, z \to \beta, s \to \kappa, u \to \gamma, t \to \lambda]$.



(a) $t_1\sigma_4$



(b) $t_2\sigma_4$

Figure 12: Terms t_1 and t_2 from different clauses.

This is because both terms have the following depth-1 skeletons - $f(\theta_1, \theta_2, \theta_3)$, $g(\theta_1, \theta_2), n(\theta_1, \theta_2)$. This gives us $|\operatorname{st}(t_1\sigma_4) \cup \operatorname{st}(t_2\sigma_4)| = 17$. We note if $d = 2$, while both terms still match on the depth-2 skeleton $h(n(\theta_1, \theta_2), \theta_1)$, we get less overall sharing as now they don't match on $f(\theta_1, \theta_2, \theta_3)$ and $g(\theta_1, \theta_2)$, but rather disagree on $f(g(\theta_1, \theta_2), \theta_3, f(\theta_4, \theta_3, \theta_5))$ and $f(f(\theta_1, \theta_2, \theta_3), \theta_2, g(\theta_4, \theta_5))$.

3.6 Level 5 - Parent Symbol Frequency

Level 5 focuses on the frequency of parent function symbols while not making any approximations, unlike Level 3. It consists of three passes. The first pass counts the number of times each function symbol appears (counts are continued between invocations). The second pass associates each variable with their most frequent parent symbol - with respect to how often that symbol appears regardless of its

children. For example, say variable x has parent symbols f and g at different points, it will be associated with whichever appears more often. The third pass then renames each variable from a container corresponding to their associated function symbol. The intermediate data structure is identical to Level 3 - a frequency and container per function symbol.

3.7 Introducing Randomness

A final route we explored was introducing randomness. We want an algorithm that selects the next variable to use randomly (in a way that is still structure-preserving) instead of using queues or stacks. We then run this algorithm several times for a set of clauses and take the renaming that leads to the best compression. We implemented randomised versions of Level 0 and Level 1.

4 Benchmarking the Algorithms

4.1 Outlining the Benchmark

To benchmark the algorithms, we took a selection of protocols and ran ProVerif, asking it to take snapshots of the clauses every 5000 generated. This gave us a range of sizes and protocol types. The snapshots range from 1KB to 26MB. While these appear small, the snapshots simply list subterms in a text file, but in reality represent many GB of RAM as ProVerif associates lots of information with each variable and function symbol.

For each snapshot we calculated the number of subterms with tree representation (ProVerif's current implementation), DAG representation and after applying each of the six approximation algorithms.

4.2 Compression Results

By using DAG representation, we already experience compression before running the approximation algorithms.

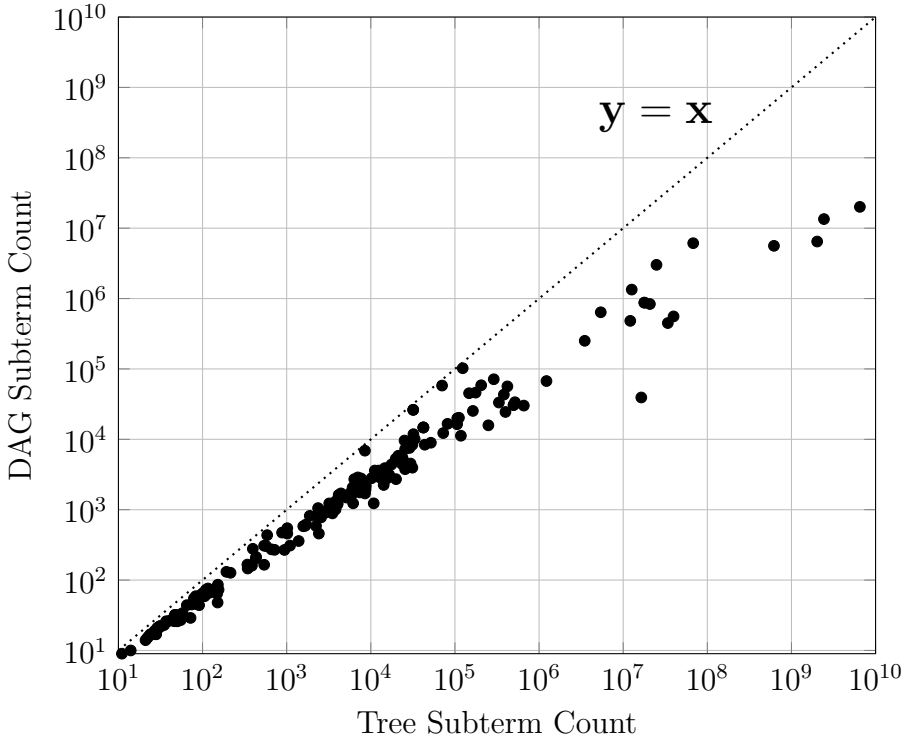


Figure 13: Comparing memory between DAG and Tree representations.

We see that this compression improves as file sizes increase, even before any variable renaming.

We now run the approximation algorithms (for this benchmark, we set $n = 10$ for Level 3, $d = 1$ for Level 4 and chose queues over stacks for our containers; we examine these choices later). We generate the following heatmap of results.

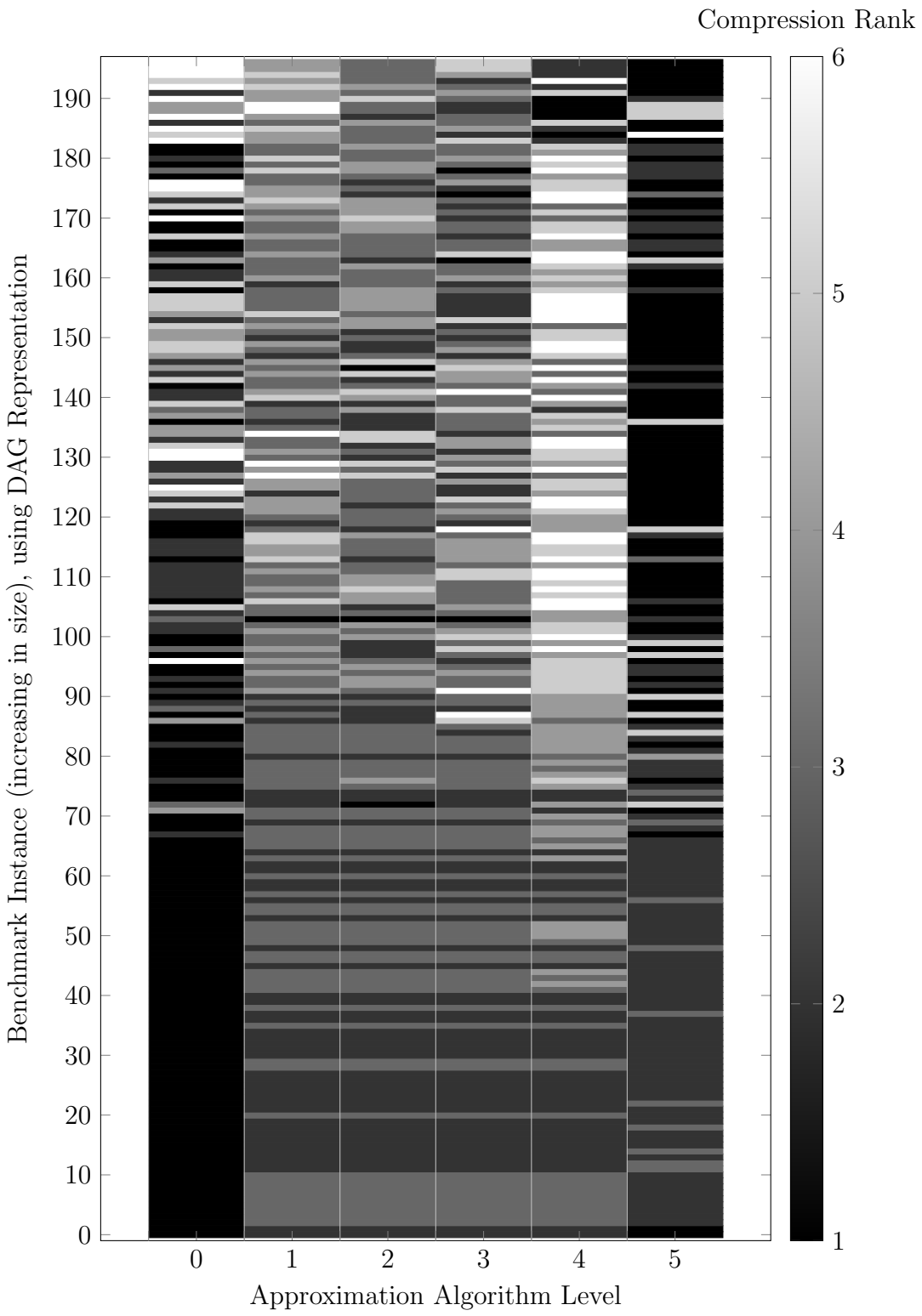


Figure 14: Heatmap ranking each approximation algorithm for each protocol.

The benchmark instances are sorted in increasing order of their DAG representation size - **the darker a cell is, the better the corresponding algorithm**

placed compared to others. We note the instances have the following subterm counts: 0-62: < 100, 63-94: $< 1,000$, 95-150: $< 10,000$, 151-180: $< 100,000$, 181-189: $< 1,000,000$, 190-194: $< 10,000,000$, 195-196: $< 20,000,000$. The heatmap shows that in the vast majority of instances 0-100, Level 0 provides the best compression, while for the larger instances Level 5 performs best.

Practically, the absolute compression achieved is more important than the relative rankings of the algorithms, especially as files get larger. To measure this we calculated the weighted average compression. For a protocol p, define its DAG subterm count as p_d and its compressed subterm count from Level i by p_i. Define by $\mathcal{P}_{i,j}$ the set of protocols where $10^i \leq p_d < 10^j$. We use the following formula to calculate the weighted averages in the table below:

$$\mathrm{W}(i,a,b) = \frac{\sum\limits_{p \in \mathcal{P}_{a,b}} p_d p_i}{\sum\limits_{p \in \mathcal{P}_{a,b}} p_d}$$

DAG Size	$\mathrm{W}(d,a,b)$	**L0**	**L1**	**L2**	**L3**	**L4**	**L5**
$[0, 10^2)$	46	18	24	24	24	24	22
$[10^2, 10^3)$	605	160	185	185	188	208	172
$[10^3, 10^4)$	5253	901	902	895	898	1011	744
$[10^4, 10^5)$	40668	4592	4985	4943	4909	5354	4106
$[10^5, 10^6)$	681825	44021	38496	37925	38137	36427	36204
$[10^6, 10^7)$	5390605	184794	148341	147799	146414	146001	111030
$[10^7, 10^8)$	17421572	230684	148664	146809	150732	138390	78056

Table 1: Weighted averages split by DAG subterm counts (2nd column is uncompressed).

Lower numbers correspond to better compression and all levels make significant improvements compared to no compression (2nd column). Despite its strong performance for small inputs, Level 0 performs poorly for larger files, meaning it doesn't produce meaningful compression. Level 5 performs the best as the inputs get larger, making it the best choice.

Levels 3 and 4 were parameterised by n and d respectively. We now examine

these parameters.

4.3 Choosing n

For Level 3, we tested $n = 0$ (Level 1), 1, 3, 5 and 10. The results were noisy and often produced equal compression. Any differences were negligible with no clear parameter choices producing consistently better results (Figure 15).

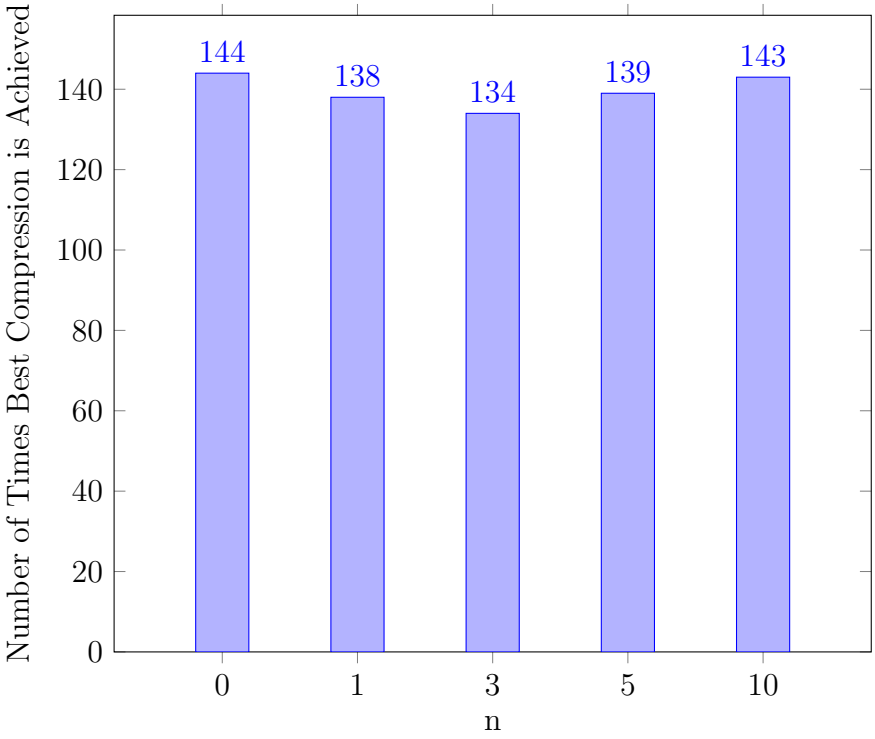


Figure 15: Choosing n for Level 3.

We conclude this algorithm was weak, as changing n had no impactful consequences.

4.4 Choosing d

For Level 4, increasing d led to worse compression, emphasised as the number of subterms grew.

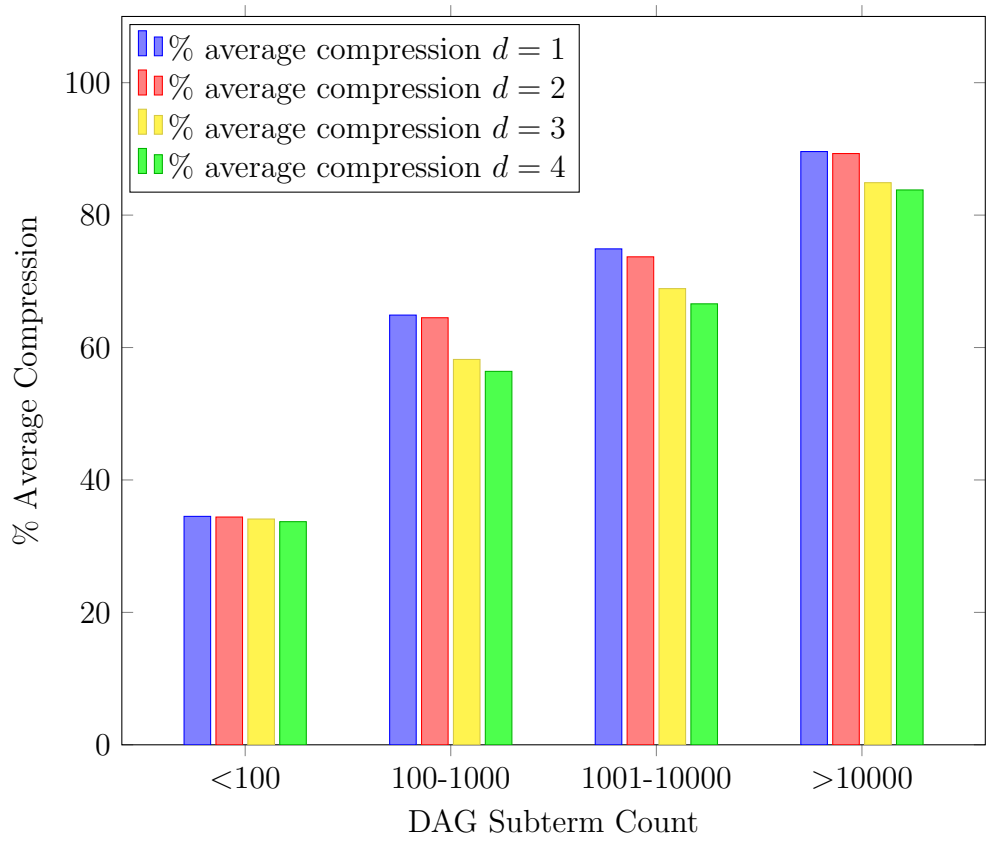


Figure 16: Choosing d for Level 4.

We hypothesise this is due to the following: consider terms (from different clauses) $f_1(g(x_1)), f_2(g(x_2)), f_3(g(x_3)), \ldots$. With $d = 1$ all x_i variables get mapped to the same fresh variable, hence all the $g(x_i)$ subterms become the same node. For $d = 2$, each term has a different skeleton (rooted at f_i), so each x_i would be renamed to something different - we saw a similar result in the explanation of Level 4. We believe this is because most of our similar terms tend to be shallow, and by increasing d we miss out on matching them.

4.5 Choosing a Variable Container

We compared stacks and queues when choosing containers. We found queues consistently produced better compression. To compare results, we ran a two-tailed paired t-test [15], to examine whether there is a statistically significant difference between the mean compression achieved. Since we have 197 entries in our dataset, the *degree*

of freedom is 196.

Level	t**-statistic**	p**-value**
0	-2.99	0.0031
1	-3.41	0.0008
2	-3.31	0.0011
3	-3.34	0.0010
4	-3.95	0.0001
5	-3.39	0.0008

Table 2: Results of two-tailed paired t-tests.

All t-statistic values are negative, indicating the average compressed count for the first set (queues) is lower than for the second set (stacks) for all levels. Since all p values are < 0.05, this difference is statistically significant.

We hypothesise queues performed better as they offer a stable variable ordering, so each time we reach a new clause we always start with the same variable. For a stack, each time a clause empties it and adds a new variable, the next clause starts from a different fresh variable.

4.6 Randomisation

Finally, we examined choosing variables randomly from their containers for Levels 0 and 1.

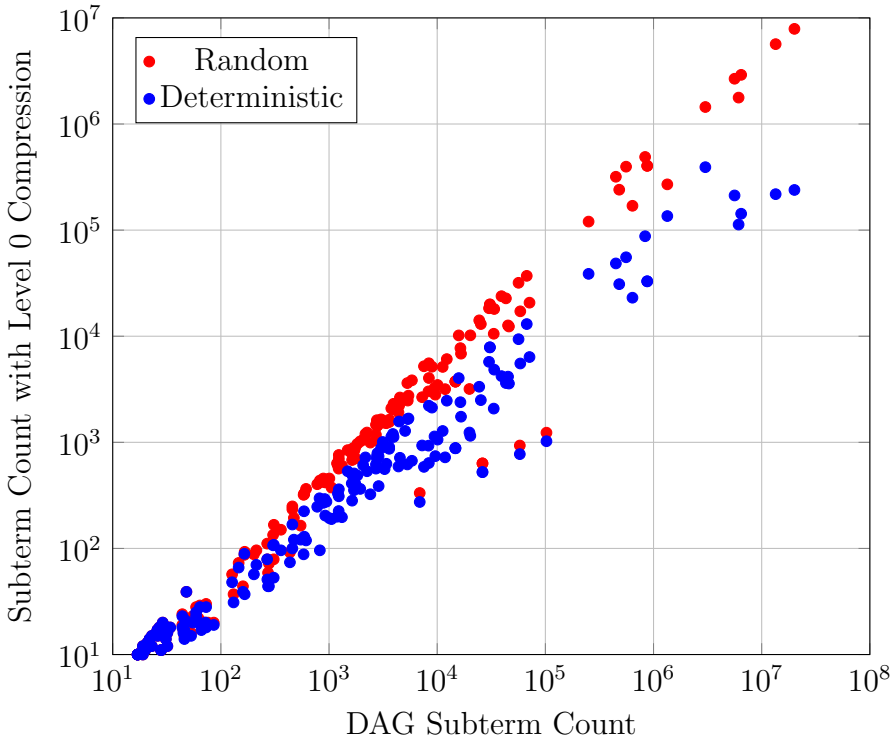


Figure 17: Level 0 random variable selection.

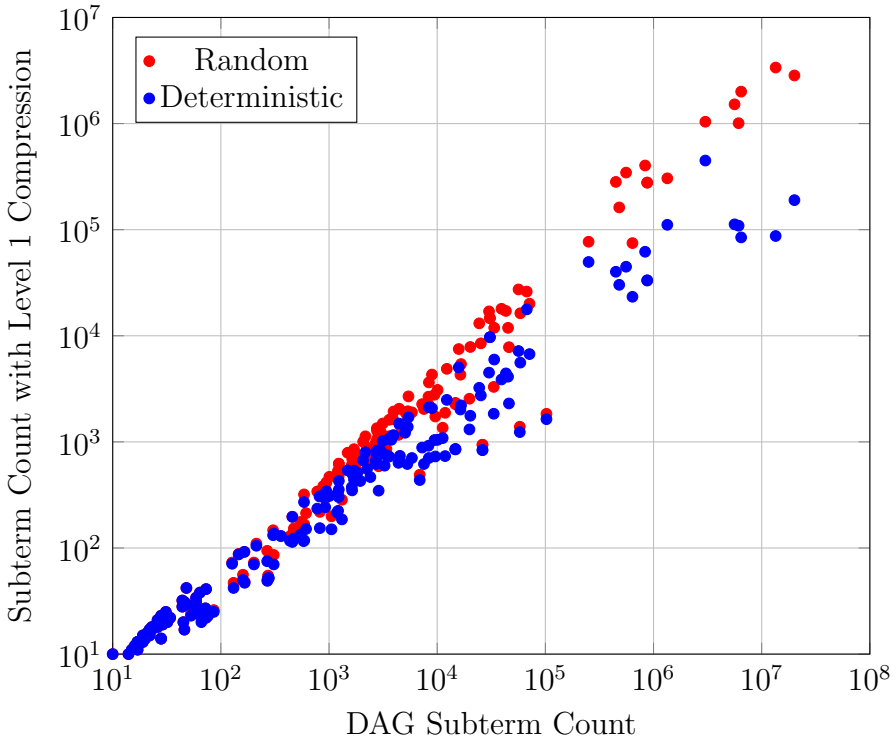


Figure 18: Level 1 random variable selection.

Figures 17 and 18 show that randomisation produces consistently worse com-

pression, reinforcing our stable variable ordering hypothesis.

4.7 Overall Progress

Table 3 shows the cumulative subterm totals for protocols sorted by DAG size, for their tree representation (Σ Tree), their DAG representation (Σ DAG), and after applying each approximation algorithm. As expected the more complex algorithms produced better compression as subterm counts got larger, with Level 5 dominating. *Note: "nodes (n)" is our unit of measurement. E.g.* $xKn \equiv x \times 10^3$ *nodes.*

DAG Size	Σ Tree	Σ DAG	L0	L1	L2	L3	L4	L5
$[0, 10^2)$	3.9Kn	2.2Kn	1Kn	1.3Kn	1.3Kn	1.3Kn	1.3Kn	1.3Kn
$[10^2, 10^3)$	42.6Kn	14.5Kn	3.9Kn	4.5Kn	4.5Kn	4.6Kn	4.9Kn	4.2Kn
$[10^3, 10^4)$	827Kn	203Kn	39.8Kn	40.7Kn	40.4Kn	40.4Kn	44.9Kn	34Kn
$[10^4, 10^5)$	23.5Mn	924.8Kn	104.6Kn	112.8Kn	111.9Kn	111Kn	121.3Kn	93.2Kn
$[10^5, 10^6)$	151.2Mn	5.1Mn	351.2Kn	318.1Kn	314.1Kn	315.8Kn	304.4Kn	290Kn
$[10^6, 10^7)$	2.7Gn	22.5Mn	995.3Kn	865.5Kn	863.4Kn	839Kn	820.5Kn	675.Kn
$[10^7, 10^8)$	8.9Gn	33.5Mn	457.3Kn	277.1Kn	273.8Kn	280.6Kn	258.4Kn	148.3Kn

Table 3: Cumulative subterm counts for different sized protocols.

Table 4 shows the average reduction in file size relative to the initial DAG representation after each approximation algorithm is applied.

DAG Size	L0×	L1×	L2×	L3×	L4×	L5×
$[0, 10^2)$	2.2	1.7	1.7	1.7	1.7	1.8
$[10^2, 10^3)$	3.8	3.2	3.2	3.2	2.9	3.4
$[10^3, 10^4)$	5.1	5.0	5.0	5.0	4.5	6.0
$[10^4, 10^5)$	8.8	8.2	8.3	8.3	7.6	9.9
$[10^5, 10^6)$	14.4	15.9	16.1	16.0	16.6	17.5
$[10^6, 10^7)$	22.7	26.0	26.1	26.9	27.5	33.4
$[10^7, 10^8)$	73.3	121.0	122.5	119.5	129.8	226.1

Table 4: Times smaller than DAG representation each algorithm achieves.

All algorithms performed well, with Level 5 compressing our largest DAG representations to 266^{th} of their original size. The single largest protocol had $\approx 6,500,000,000$ subterms in its tree representation. DAG representation followed

by Level 5 left $\approx 94,000$ subterms - **70,000 times smaller than ProVerif's current implementation, in a totally lossless fashion.**

5 Conclusion

We aimed to reduce the memory consumption for representing terms within Horn clauses for ProVerif, exploiting hash consing techniques, representing terms as DAGs instead of trees, and developing algorithms to rename variables to compress terms. We proved that finding the optimal compression is NP-hard and then defined how to find this compressiom by solving a 0/1 ILP problem. Finding this to be intractable, we presented six approximation algorithms. These compressed the terms, with Level 5 providing the most meaningful compression. All approximation algorithms produced significant compression and ran within an acceptable amount of time, making them suitable for introduction into ProVerif's process.

5.1 Future Work

Level 4's ability to characterise terms by structure is promising, but limited as only effective at small depths. Further research into a more sophisticated way to characterise subterm structure, potentially combining skeletons of different depths, as well as incorporating the Level 5 sorting ideas are natural next steps.

Although the algorithms performed well empirically, we did not bound the quality of the approximation achieved and finding guarantees would be useful.

References

[1] V. Cheval, S. Kremer, and I. Rakotonirina, "Deepsec: Deciding equivalence properties in security protocols theory and practice," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 529–546. DOI: 10.1109/SP.2018.00033.

[2] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of diffie-hellman protocols and advanced security properties," in *2012 IEEE 25th Computer Security Foundations Symposium*, 2012, pp. 78–94. DOI: 10.1109/CSF.2012.25.

[3] "Proverif: Cryptographic protocol verifier in the formal model." (), [Online]. Available: https://bblanche.gitlabpages.inria.fr/proverif/ (visited on 04/17/2025).

[4] B. Blanchet, *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif*. 2016, vol. 1. DOI: 10.1561/3300000004.

[5] "Verification of TLS 1.3 draft 18." (), [Online]. Available: https://github.com/inria-prosecco/reftls (visited on 04/28/2025).

[6] "A formal analysis of EKEP." (), [Online]. Available: https://github.com/google/ekep-analysis (visited on 05/05/2025).

[7] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, *ProVerif 2.05: Automatic cryptographic protocol verifier, user manual and tutorial*, Available at https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf, 2023.

[8] B. Blanchet, V. Cheval, and V. Cortier, "Proverif with lemmas, induction, fast subsumption, and much more," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 69–86. DOI: 10.1109/SP46214.2022.9833653.

[9] J.-C. Filliâtre and S. Conchon, "Type-safe modular hash-consing," in *ML '06: Proceedings of the 2006 workshop on ML*, 2006, pp. 12–19.

[10] T. Kutsia. "Introduction to Unification Theory, Speeding Up, Syntactic Unification Improved Algorithms." (), [Online]. Available: https://www3.risc.jku.at/education/courses/ss2018/unification/slides/02_Syntactic_Unification_Improved_Algorithms_handout.pdf (visited on 04/16/2025).

[11] M. Sipser, *Introduction to the Theory of Computation*, Third. Course Technology, 2013, p. 311, ISBN: 113318779X.

[12] "Gurobi: 0,1 industry standard ILP solver." (), [Online]. Available: https://www.gurobi.com/solutions/gurobi-optimizer/ (visited on 04/18/2025).

[13] "CPLEX: Industry standard optimisation tool maintained by IBM." (), [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer (visited on 04/20/2025).

[14] Y. Chen, I. E. Grossmann, and D. C. Miller, "Computational strategies for large-scale milp transshipment models for heat exchanger network synthesis," *Computers & Chemical Engineering*, vol. 82, pp. 68–83, 2015, ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2015.05.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S009813541500174X.

[15] J. H. Stock and M. W. Watson, *Introduction to Econometrics*, 4th. Boston: Pearson, 2015, ISBN: 978-0-13-446199-1.