PAIRING STRINGS TO THEIR MOST SPECIFIC REGULAR EXPRESSION MATCH OUT OF A SET OF REGEXPS

Donald Miner

Adviser: Dr. Richard Chang

Computer Science Departmental Honors Thesis

University of Maryland: Baltimore County

May 22, 2006

ABSTRACT

An efficient method is given that solves the problem of mapping a list of strings to their most specific matches from a list of Perl compatible regular expressions (regexps). The assigning of context labels to a file system, using a list of regular expressions as way to determine them, done by the *setfiles* program in an SELinux system, is the inspiration for this problem and is used as the sample dataset on which tests are performed. Graphs are generated to describe set relationships between pairs of regexps, specifically: subset, superset and disjoint. These graphs are used to infer regular expression matches so that the actual match does not have to be done. Also, they are used to determine specificity between regular expressions. Through experimental testing, this algorithm is faster than the simplistic approach of matching each regular expression to each string, while also correctly assigning the most specific match.

INTRODUCTION

The problem posed is as follows: given a list of Perl compatible regular expressions, which we will refer to as the REGEXPS, and a list of strings, which we will refer to as the STRINGS, match every string in STRINGS with its *most specific match* in REGEXPS. The term *most specific regular expression* is an important part of the problem and is defined as *MS(S,R)*:

Let S be a string and R = { x | x matches S and x ∈ REGEXPS }

MS(S,R) = { y | y ∈ R and there does not exist an x ∈ (S – y) such that x ⊆ y}

That is, the most specific match out of a string's set of matches are the ones that are not the supersets of any of the other matches. For example:

S = "`/usr/bin/ls`*"; R = { x="*`^/.*`*", y="*`^/usr/.*`*", z="*`^/.*/bin/.*`*", m="*`^/usr/bin/ls`*" }*

⇒ MS(S,R) = "`^/usr/bin/ls`*"*

In this case, *m* is the most specific regular expression match because it is not the superset of any other regular expression match in *R*. *x*, *y* and *z* are all supersets of *m* because for all strings that *m* will match, they will match, too. Therefore, they cannot be considered most specific matches. Given the example that only consists of *x, y, z* but not *m*, *MS(S,R)* would be both *y* and *z.*

As well as proposing a correct solution to this problem, we propose one that finds the solution in an efficient manner. We decrease the overall run time by quickly calculating set relationships (subset, superset, disjoint) between all the regexps and leveraging the information to deduct regular expression matches. Deducting the regular expression matches, apposing to simply performing them, is faster for several reasons that will be discussed later on. Because the use of several heuristics in order to better the running time as well as the difficulty of calculating theoretical running times in today's regular expression engines, we will state running times for the results as time elapsed. All tests were performed on the same computer, under similar conditions and with the same data for STRINGS and REGEXPS.

In the rest of this paper we will discuss:

- SELinux and *setfiles*, the inspiration for solving this problem.
- An overview of the approach taken to solve the problem.
- How to determine whether a regexp is a subset, superset or is disjoint from another regexp.
- A description of the graphs used to map out the set relationships and how to generate them.
- How to use the graphs to determine the final solution efficiently.
- An outline of the results, including running times.

SELinux – The Original Problem

Security-Enhanced Linux (SELinux) is an open source project sponsored by the National Security Agency which attempts to secure the Linux kernel. It incorporates the principles of mandatory access control and the principle of least privilege in the Linux kernel through the use of system libraries and tools. Most of the work being done to complete SELinux is done by volunteers and contracting companies under the direction of the NSA and is still considered in development by many. Red Hat Linux created a bare version of the security policies which is currently enabled by default in Red Hat Linux's Fedora Core 5 and Red Hat Enterprise Linux 4. Also, other Linux distributions are currently starting to integrate SELinux as well, such as Ubuntu, Debian and Gentoo.

In a SELinux system, each file is considered an object. In order to distinguish different classes of files so that they may be targeted by security policies, they are given a label called the file's *file context*. For example, the SSH daemon executable is labeled `sshd_exec_t`. Meanwhile, Apache's executable is labeled `httpd_exec_t`. Since the files have a label, rules can be created for these objects, such as a rule that allows the cron process to read crontab files. When installing SELinux, the file system must be labeled. This is done with the *setfiles* program which takes the *file contexts* file and traverses the file system, labeling each file. The file context file contains (regexp, label) pairs and for each file, setfiles attempts to match its filename with its most specific match. Once the most specific match is found, it is labeled the label associated with that regexp.

The original problem and our inspiration for finding a better solution is the process of labeling a file system with setfiles program. To find a file's label, setfiles gets the file's filename and traverses the list of file contexts, which are approximately sorted in most specific to least specific. The first regular expression match it finds for the filename, it stops and labels the file that label. There are several problems with this approach, in both accuracy and running time. The main problem is the current implementation approximates the most specific match by using heuristics to compare the specificity of two regexps to sort the file contexts. These heuristics do not always give the correct result and may cause files to be labeled incorrectly. We, too, will be using heuristics, however, our heuristics will always produce a correct result and would rather state that it cannot reach a conclusion rather than guessing incorrectly. Also, this list cannot be traversed in any intelligent way and matching must be done with a linear search. We propose that the different approach of placing the the file contexts in graphs will produce faster running times than if they were in a linear list. Another problem is in the case where a file has two matches where the one is not any more specific than the other (see the first example with *m* removed in the introduction). Since setfiles stops at the first match it finds, no warning will be given and no collision will be detected. Our proposed algorithm will be able to find these errors during the formulation of the solution and report them to the user.

There is a large amount of information within the regexps that can be used. In a normal run of setfiles, many matches are done which could have been figured out without a regular expression match. We give a method that is faster because we leverage the set relationship information which can be found between the regexps. As well as being faster, it is much easier to find places where the solution may have inconsistencies. Our method is not a blind search through a list; all possible solutions, apposed to only the first solution, are found and if the algorithm does not know what to do, the user will know.

Regular Expression Background

We refer to both regular expressions and regexps seemingly interchangeably, however is important to note the difference of terminology. A *regular expression* in the context of automata theory and mathematics is an algebraic description that describes nondeterministic and deterministic finite automata and defines the regular languages. *regexp* is a syntax devised to match strings and perform string manipulation that used in utilities such as grep, sed and scripting languages such as Perl. Descriptions of regexp syntax can be easily found on the Internet. It is important to note that regexps do not describe the regular languages, as regular expressions do. For example, regexp allows the user to recall previously matched text further down in the match with a regexp like "`(.*)\1`". This regexp describes the language that is often called *sqaures*: the first half of the string is repeated in the second half. Squares is known to not be regular and since regexp is able to describe this language, it exceeds the limitations of regular languages. Since regexp are not regular, some methods of determining set relationships may not function correctly. Currently, no regexp in the file contexts describes a non-regular language. Therefore, we make the assumption that we may to use the methods and tools only available for use with regular expressions with the regexps in the file contexts.

GENERAL APPROACH AND THE USAGE OF SET RELATIONSHIPS

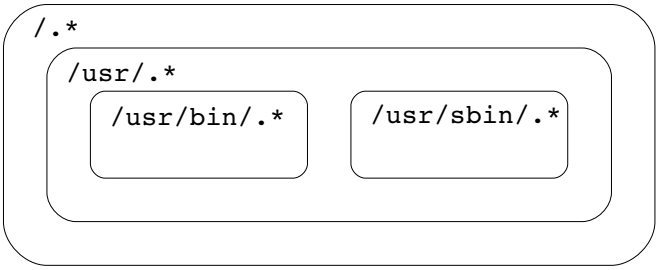
Perhaps the most simplistic approach, called NAIVE, that produces a correct solution is to match each regexp in REGEXPS to each string in STRINGS and then, for each file's list of matches, select the most specific regexp. We named this method NAIVE because it simply starts trying to find the solution, without pre-processing any information. In order to find the most specific match in a set of solutions, each regexp has to be checked to see if it the superset of any other regexp. The increase in speed in our faster algorithm, called FASTER, comes from not only using the set relationships to solve the solution, we are also able use them to infer regular expression matches. This way a large majority of the costly regular expression matches do not have to be made. Instead of NAIVE's approach of computing the set relationships at the end of the process, before doing regular expression

matching, FASTER will generate several graphs that each describe a different kind of relationships in REGEXPS.

The idea for using set relationships in FASTER came from observing the structure of the file contexts. They often followed a directory by directory pattern, for example:

regexp	*Label*
`/.*`	`root_t`
`/usr/.*`	`usr_t`
`/usr/bin/.*`	`bin_t`
`/usr/sbin/.*`	`sbin_t`

Observe they consist of a fixed length string followed by any number of characters. Subset is easy to determine in this case as several of them share the same prefix. However, the sbin and bin directory do not share the same prefix, thus are disjoint.



subset relatonships: `/usr/bin/.*, /usr/sbin/.*` $\subseteq$ `/usr/.*` $\subseteq$ `/.*`

disjoint relationships: `/usr/bin/.*` $\cap$ `/usr/sbin/.*` $= \varnothing$

This case is very obvious and the vast majority of file contexts do follow either this pattern or one of a handful of others. We believe that part of the success of FASTER is due to this structure. Exploiting these patterns for faster running times will be discussed in depth when we introduce heuristics for determining set relationships.

A regular expression match can be costly since they often times have to be compiled, depending on the scripting language, and have to be ran through complex engines. The running time of having to perform matches over tens of thousands of comparisons adds up. With the set graphs, in the case where enough information has been gathered, determining if a string will match a regexp could take constant time. This is done by referencing the the relationships determined available to us in the set graphs. Several inferences can be made by knowing if a regexp matches a string. For example:

`^/.*/bin/.*` *matches* `/usr/bin/ls`

`^/.*` $\supseteq$ `^/.*/bin/.*`

$\Rightarrow$ `^/.*` *matches* `hello world`

A detailed list of inferences will be given later on. As more is known about which regexps match a string, more

inferences can be made and future matches will be able to be inferred rather than actually matched. These inferences usually take much less time than computing the simplest of regular expression matches. However, making the inferences is not the hard part, calculating the set relationships are.

We first present a way to calculate a set relationship between two regular expressions *A* and *B*, no matter what the they may be. We first convert these regular expressions to non-deterministic finite automata (NFA) by using Thompson's Algorithm. We then perform the following operations on them to determine the relationship between *A* and *B*:

- *Subset:* $A \subseteq B \equiv A \cap \overline{B} = \varnothing$
- *Superset:* $B \subseteq A \equiv B \cap \overline{A} = \varnothing$
- *Disjoint:* $A \cap B = \varnothing$

The theoretical running time of this approach is slow. This is due to the fact that in order to create the complement on an NFA, it must be first converted to a deterministic finite automaton (DFA). Then, once in DFA form, the finishing states are switched to non-finishing states and visa versa. The conversion of a NFA to a DFA may result in a DFA with exponential number of states, in reference to the number of states in the original NFA. Therefore, both generating and traversing a DFA that is the result of a NFA to DFA conversion has a worst case exponential running time. Considering there are over a thousand regexps in REGEXPS, if speed is a concern, this approach is probably no the best solution to the problem.

Not only is this unfavorable because of the slow running time, but it would be very difficult to implement. Most regexp engines today, such as the one used in Perl, uses a backtracking algorithm (depth-first search) not NFA's. One would have to make his own or use an old NFA regular expression engine to perform these operations. Even then, the regular expression syntax used would have to have lose some features such as back references because NFA's cannot support what regexp can do since regexp do not describe the regular languages. We opted to not implement this.

The positive aspect of this approach is that the set relationship are guaranteed to be discovered every time with one-hundred percent correctness. All other faster attempts to solving this problem would merely be approximations. Sacrificing this level of correctness and methods with shorter running times to determine set relationships is both more practical and more feasible to implement.

A more practical approach is to use heuristics to attempt to determine the relationships. These methods are implemented so that they are guaranteed to be correct if they give an answer, but have the option to fail and return nothing. These heuristics provide shortcuts by parsing and analyzing the structure of the regexps and

without the use of state machines to determine a relationship in polynomial running time. For example, one of the most simple heuristics we used in testing is one that checks for fixed strings at the beginning of the regexps. If the two regular expressions have a difference in their fixed string prefix, the two are disjoint. If they are the same, the heuristic is not able to decide. The example showing how `/usr/bin/.*` is disjoint from `/usr/sbin/.*` earlier in this section demonstrates this reasoning. The list of heuristics we used, given string representations of regexps *A* and *B* can be found in the appendix. Each heuristic attempts to characterize a single pattern we noticed in REGEXPS. Although together the heuristics provide absolute correctness when a relationship is found, sometimes the relationship cannot be determined and the relationship is left ambiguous. If the methods are not good enough and this happens a lot, this ambiguity can result in an non-satisfactory solution and running time could increase later on. The overhead of perfoming FASTER is considerably higher than NAIVE so if there are not enough relationships found, not as many inferences can be made and running time will suffer.

Determining the set relationship between two regexps is perhaps the most complex part of solving the problem as they lay the foundation of eliminating comparisons when assigning the regexps to the strings. The faster, less accurate heuristic approach is what we used in the tests.

SET GRAPH

The first step of FASTER is to pre-process the set information. We generate several graphs, one for each set relationship. Each graph contains one node for each regexp in the dataset. Two nodes will contain an edge, sometimes directed, in the graph corresponding to that type of a relationship. Given regexps *A* and *B*, and edge will occur in the following graphs:

- SUBSET – If $A \subseteq B$, there will be a directed edge from *A* to *B*.
- DISJOINT – If $A \cap B = \varnothing$ there will be an edge from *A* to *B*.
- OTHER – If $A \cap B \subset$ B and $A \cap B \subset$ A there will be an edge from *A* to *B.* This means there is overlap between the languages *A* and *B.* There exist strings that match *A* but not *B*, strings that match *B* but not *A* and strings that match both *A* and *B.*

We generate the graphs by simply matching every regexp to every other regexp, determining the relationship and then storing the result in a bit-vector. Originally, we used a heterogeneous list data structure was used to easily code set operations such as the intersection of two regexp. However, the running time of performing such operations is much worse than blindly setting a bit in a bit-vector and a complete run ended up taking over a day to complete. The structure of this graph is intended to minimize the amount of time to make inferences later

on. Once the graph is generated, retrieving a category of relationships for a node can be done in linear time. We achieve this by storing lists of regexps for each relationship for each regexp. When asked for a specific relationship for a specific regexp, the list is simply returned.

At the completion of the matching process, we may take steps to find relationships through transitivity. For example: if $A \subseteq B$ *and* $B \subseteq C$ *then* $A \subseteq C$. It is possible that the relationship between A and C was left as unknown and would have been missed. We decided making inferences and percolating relationships during this process is not worth it, as every regexp would have to be checked for every new relationship discovered. The best way to avoid this is to make the heuristics as transitive as possible. In the tests ran, only a couple of relationships would have been found out of the tens of thousands so this step was deemed unnecessary.

The time needed for the creation of this graph is minimal, only taking approximately seven seconds to complete with the our heuristics. This running time is greatly affected by the number of regular expressions and the complexity of the heuristics. Also, if a set graph were to be used for actual deployment, the graph could be pre-compiled and packaged. If accuracy was important, the slowest approach with the automaton conversions to determine set information could be used and compiled, eliminating the user having to perform this step. For all these reasons, what the graph actually does is far more important than the running time of generating it.

ASSIGNING

The next step in FASTER is to assign each string to their most specific match in REGEXPS. At this time the set graph has been generated and can be used. Each string can avoid several regular expression matches by making the following inferences, given regexps *A* and *B,* an edge *(A,B)* and string *S*:

- If *A* matches *S* and $(A \rightarrow B) \in$SUBSET then *B* matches *S*

  $\Rightarrow B \notin$MS(S) because *A* is more specific than *B*
- If *A* matches *S* and $(A,B) \in$DISJOINT, then *B* does not match *S*.

  $\Rightarrow B \notin$MS(S) because *B* does not match *S.*
- If *A* does not match string *S* and $(B \rightarrow A) \in$SUBSET, then *B* does not match *S*.

  $\Rightarrow B \notin$MS(S) because *B* does not match *S.*

When each string starts traversing the list of regexps, every regexp has the possibility of being its most specific match. We represent this with a bit-vector where each bit corresponds to a regexp. Then, as each regular expression match is done, we make all the inferences possible and update the bit-vector with the eliminations we make each step. This is done by retrieving the regexps that correspond to the relationship needed to make an

inference and toggling each regexp's respective bit. For example, while searching for the most specific match for `/usr/bin/ls` we checks against `/usr/bin/.*`:

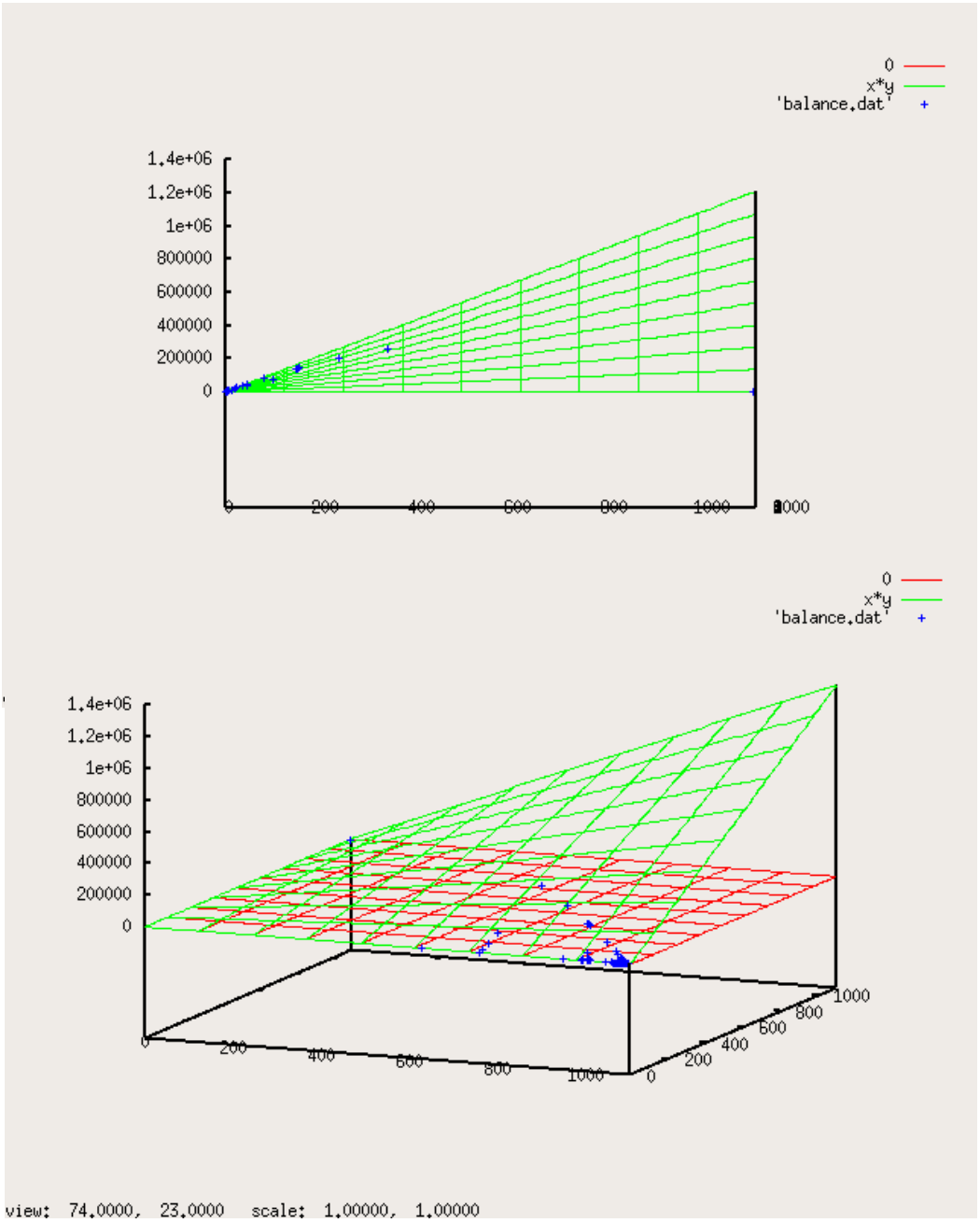
1. Check to make sure `/usr/bin/.*` matching has not been eliminated, yet. If it has, skip over this regexp.
2. Since no inference can be made, we must perform a regular expression match. The match returns True.
3. We are able to infer that no regexps that are disjoint with `/usr/bin/.*` could possibly match `/usr/bin/ls`. We look up `/usr/bin/.*` in DISJOINT and for each regexp returned, eliminate it as the most specific match by toggling its bit in the bit vector.

This approach is very basic, and there are a couple ways we improve the running-time. First of all, there is an order of which to check the regexps in order to attempt to make the most inferences possible. The reason why this approach is faster is a lot like how binary search is faster than linear search on a dataset of size N: binary search eliminates N/2 of the data each step and linear search only eliminates one piece of data. The regexps that have the probability of eliminating the most regexp in one step should be checked first. There are two sets of inferences that can be made, which gives us a binary choice: one for if the string matches the regular expression and one for if it does not. If the string matches a regexp, then all supersets and all disjoints may be removed from the set of possible most specific matches. On the other hand, if the string does not match the regexp, then all subsets may be removed from the set of possible most specific matches. In order to reward regexps that will eliminate more, we devised the ranking function:

$$\text{RANK}(A) = (|A.\text{supersets}| + |A.\text{disjoints}|) \times |A.\text{subsets}|$$

This scoring ranks the regexp with the highest number of relationships as well as the most balanced relationships so that they are checked first. The maximum rank attainable would be (|REGEXPS| / 2) x (|REGEXPS| / 2): a even split between the two and no regexps are left out. There are no regexps in the sample data that came close to achieving this rank. On the other hand, there are several regexps that are ranked 0. These are the ones that are completely one-sided, containing either only subsets or only supersets and disjoints, such as `/.*` and all fixed string regexps. These getting a rank of 0 is a good thing, since the only regexps they will be eliminating when being checked are themselves. Other rankings were tried, such as the addition of the two sides, but did not work as good as the multiplication. This approach was rewarding ones with the most relationships but did reward balance of the binary choice. The problem with this is that as more regexp are eliminated, the eliminated ones should no longer count in this ranking. To be consistent, the ranking would have to be recalculated

every time, but for the sake of running time, the original ranking is used throughout the entire process. What this means, however, is that as more and more eliminations are made, the ranking becomes less accurate over time.



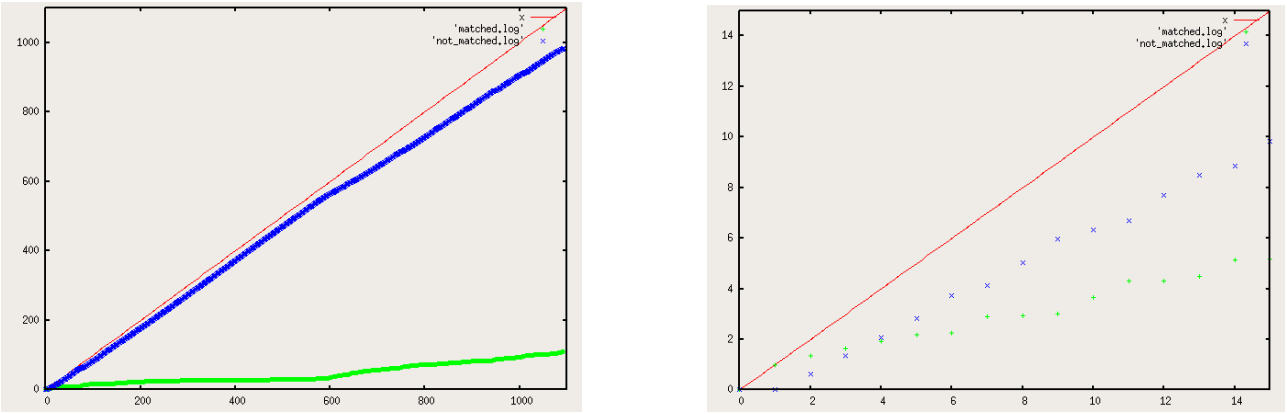
*Blue points laying on the green plain x*y depict the the ranks of file names.*

To compensate for the lack of accuracy as time goes on we switch the algorithm to a brute force method and checks all regular expressions that have not been eliminated yet to avoid the overhead of attempting to make inferences. The optimum time to switch over is at most the point at which no new inferences are being made, with this set of data being the 78th ranked regexp. At this point in time approximately 90% of the remaining regexps have been eliminated.

RESULTS

The method described was completely tested with a script written in the Python programming language. The datasets used were actual datasets that would be used for the original *setfiles* problem: an actual file context file with 1094 regexps from a recent SELinux system and a list of 110675 files in an Ubuntu Linux desktop system. All tests were ran on the same system in approximately the same conditions to ensure that tests are consistent with each other.

The two main parts of FASTER are generating the subset graph and finding the solution. In comparison to the overall running time, the generation of the subset graph is by far the shortest part, running for only approximately seven seconds of the overall 110 second running time. The rest of the 103 seconds is consumed by doing the solution assigning. In comparison, NAIVE using the same datasets, takes approximately 160 seconds. NAIVE takes 40 seconds longer, 145% of the running time, than FASTER. The reduction in running time is probably most due to the improved algorithm only performing 12 million total regular expression matches apposed to the 120 million that a one to one match would be doing. The other 90% of the comparisons are inferred, making the overall running time much faster.



The blue points depict the average number of regexps inferred and the green points depict the average number of regexps that had to be checked during a complete run of FASTER.

The point at around rank 600 cannot be explained. We believe that it probably has something to do with how the data is organized in the first place, as all regexps after rank 78 have the same rank of 0. Also interesting from these graphs is the crossover point; here shown at the 4th regexp. If this were binary search, theoretically, rank 2 would have a 50% rate of being checked, rank 3 would have a 25% chance, etc. Being that the 50% mark is at rank 4, the data is not providing an exact 50/50 binary choice, however it is close enough that it makes the ranking worth it.

CONCLUSION

Perhaps a simpler approach should have been used to implement this part of SELinux. It appears that regular expressions are too complex and allow too much freedom in creating the file contexts. A better implementation would use a syntax that would be easy to find the most specific match without much calculation. However, to solve the problem at hand for what it is, our method is superior in running time and accuracy by looking at the results. It is obvious that it is worth it to think before trying to find the solution rather than starting the matches right away. The reduction of running time by looking at the information available inside of the regexps themselves is significant. Even if our procedure used the slower yet more accurate way to compile the graph to make the results even more accurate, it could be created before hand and packaged, making the running time negligible. From conceiving a naive approach and bettering it, a faster and more correct method than the original is used to implement the *setfiles* procedure.

FUTURE DIRECTION

Using the file structure

Much like how the regexps were analyzed and relationships were found, we could do the same for strings. Along with the set graphs described above, a file structure tree would be created as well. The basic inference that can be made is, with strings *S* and *T* and regexp *A*:

If *A* failed to match *S* before getting to the end of *A*

and if *S* is a prefix of *T*

then *A* will not match *T.*

The hard part is "... before getting to the end of..." We were thinking about modifying an existing regexp engine to do this, but did not seem worth it. I do feel that if this were to be implemented, it could lower the running time of FASTER even more, as inferences are being made on both the REGEXP side and the STRINGS side.

APPENDIX

HEURISICS USED:

Given regular expressions A and B...

Supplemental Functions

- *regexp.FIND({characters}):*

  # Returns the true if the characters are found

- *regexp.FIND_FIRST({characters}):*

  # Returns the index of the first instance of a character being found

- *regexp.FIND_LAST({characters}):*

  # Returns the index of the last instance of a character being found

- *PREFIX(A):*

  # Simply returns all the fixed non-meta characters at the beginning of a regexp

  return A[0 ... A.FIND_FIRST({ '(', '.', '[' })]

Heuristics

- *CHECK_EQUAL(A,B):*

  # Checks two regexps to see if their strings are identical

  if A = B: return EQUAL

  else: return UNKNOWN

- *CHECK_PREFIXES(A,B):*

  # Checks two regular expressions to see if they have different fixed character prefixes

  if |PREFIX(A)| > |PREFIX(B)| and PREFIX(B) != PREFIX(A)[0 ... |PREFIX(B)|]:

  return DISJOINT

  else if |PREFIX(A)| > |PREFIX(B)| and PREFIX(B) != PREFIX(A)[0 ... |PREFIX(B)|]:

  return DISJOINT

  else: return UNKNOWN

- *CHECK_SUFFIXES(A,B):*

  # Checks two regular expressions to see if they have different fixed character suffixes

  return CHECK_PREFIXES(REVERSE(A), REVERSE(B))

- *CHECK_STAR(A,B):*

  # Checks to see if two regular expressions that both have a . to see if they share a prefix*

  Apre = A[0 ... A.FIND_LAST({'.'})]*

  Bpre = B[0 ... B.FIND_LAST({'.'})]*

  if |Apre| > |Bpre| and Bpre = A[0 ... B.FIND_LAST({'.'})]: return SUPERSET*

  else if |Apre| < |Bpre| and Apre = B[0 ... A.FIND_LAST({'.'})]: return SUBSET*

  else: return UKNOWN

- *CHECK_FIXED(A,B):*

  # Checks to see if one of the two regular expressions is a fixed string, if it is do a

  # regular expression match.

  if !A.FIND(ALL_META_CHARACTERS) and B matches A: return SUBSET

  else if !B.FIND(ALL_META_CHARACTERS) and A matches B: return SUPERSET

  else: return UNKNOWN

TESTING COMPUTER SPECS:

AMD 2800+, 1024 MB of RAM

Python 2.4.2, Linux 2.6.12-10-386, Ubuntu 5.10