Protocol Genome Project: Structure Inference (draft)

$Id: structure.ms,v 1.3 2005/05/10 12:58:19 aki Exp aki $
http://www.ee.oulu.fi/research/ouspg/protos/genome/
ouspg@ee.oulu.fi

ABSTRACT

This document presents some of the current structure inference techniques used in the Protocol Genome Project. The project aims to develop tools and techniques for processing communication protocols. The underlying idea is that by taking advantage of shared structural properties of these protocols - their genes - these tools can be made simpler and more efficient.

1. Overview

The techniques presented in this paper are based on the third prototype of the Protocol Genome Project, representing about four man-months of work.

The Protocol Genome Project aims to develop techniques and tools for processing current communication protocols. Although our prototype tools are being developed for processing packet capture files, the techniques can be applied to processing other data sets such as files or sentences of a language.

Our main goal for the first half of 2005 is to produce a traffic fuzzer. This involves processing a piece of some traffic, inferring structure from it and then producing or modifying traffic based on that knowledge. Fuzzers are useful for example in protocol implementation testing.

The nontrivial part of writing a fuzzer is structure inference. Our current approach is based on techniques borrowed from bioinformatics, parsing, logic programming, artificial intelligence and design of some traditional algorithms. These techniques are intended to be sufficient for the purposes of producing a simple fuzzer. They will be refined later to produce a more realistic protocol reverse engineering tool.

We have split the problem to three parts. Firstly the concept of a structure is made concrete by having a language to describe them. This makes a structure simply an expression of the structure description language. The remaining parts are structure inference, in other words building expressions of the language from data, and fuzzing, meaning building or modifying data based on the inferred structure. This document describes how the first two tasks are currently handled. The third task is trivial and useful only if the preceding ones are.

We use the Scheme programming language to make rapid prototyping and debugging easy. Functional programming style ensures that parts of programs can be changed without affecting how the rest of the program behaves. This allows us to easily explore with different techniques. The current prototypes come bundled with a simple (slow) Scheme interpreter that is sufficient for running them. One of the freely available high quality implementations will be used for the actual programs.

2. Functional genes

We will first define what is meant by a structure. If something is to be regarded as a useful structure, one has to be able to somehow recognize and use it. An overly general definition would be that a structure is something for which there is an accepting algorithm.

Usually the structures that are being searched, interpreted or otherwise used are simple and do not require a general purpose language to describe them. Also, additional power in in a structure description language means higher cost for using it. Subsets of more general languages are often used, because they can be guaranteed to have some useful properties, and the languages themselves can be made simpler by using a special purpose syntax. For example regular expressions and BNF representation of context free grammars, which are subsets of formal grammars, are commonly used to describe syntactic structure.

Much of the structure we need to describe and process is syntactic and could be easily described with grammars. The rest however involves some kind of input evaluation which would require horrible grammatical constructions. To address this shortcoming we developed a language called functional genes. We started from simple lambda expressions that handled the troublesome parts, and then built typical grammar operations from them.

The resulting language has simple semantics, can be implemented and embedded easily and can be processed symbolically within programs. Within a program the gene expressions are compiled to functions that do something with the structure. The separation of the language and implementation is useful, because special purpose parsers and fuzzers can be compiled from gene expressions simply by using a different compilation strategy.

We currently use gene expressions in the prototype as a structure description language and an embedded language to build parsers. In this document by structure we mean something that can be described in this language. A more detailed description of functional genes is in [4].

3. Structure inference

Structure inference is currently the main part of the project. Due to tight schedule we have had to make a number of compromises to be able to produce the fuzzer in time. Refined versions of the presented techniques may be used as components in our later programs, but as such they are not yet sufficient for realistic protocol reverse engineering.

3.1. Expert modules

The data processed by the structure inference program can be thought of as a list of byte lists. They may be packets, files or other pieces of data. An expert module is a part of the program that looks for some kinds of structures from this data. If some interesting patterns are found, the expert module proposes them as interesting ones.

Our current prototype currently uses three expert modules.

3.1.1. The fall-back expert

A gene expression that matches any input data sources can be constructed trivially. The expression essentially lists all of the data and thus does not give any additional insight to its structure. The fall-back expert constructs either this or some other trivial gene expression from any input data. This expert is only invoked when no other structures are found.

3.1.2. The gene pool expert

A gene pool expert contains a pool of hand-written gene expressions. The pool may contain for example genes for null terminated strings, ipv4 headers, length-value pairs and other common structures.

This expert when given input data finds all the occurrences of all the genes in the pool from the data, and proposes the matched ones as interesting structures. This expert is used to find known interesting structures.

3.1.3. The substring expert

The main shortcoming of the gene pool expert is that it is only concerned with structure within one data source. Often structures that are not interesting by themselves turn out to be such when one looks at the whole.

The substring expert finds interesting frequently occurring substrings from all of the input data. These substrings are scored by the number of their occurrences and their lengths. When the search has finished, gene expressions are built for the strings and proposals consisting of the gene expressions and their occurrences are returned.

3.2. The controller

The role of the controller is to manage the input data and the expert modules. A controller is first given some data to process. This can be for example lines of a text file or packets from of packet capture file.

The controller uses a divide and conquer approach to process the data and build a gene expression by combining expressions of the components. It gives the data to all the expert modules and collects the proposals. If there are no proposals, the fall-back expert is invoked. Otherwise the most interesting proposal is selected based on an evaluation function.

Once the currently most interesting pattern is selected, the data is partitioned to data sources that did not contain a match of the pattern, and the before and after parts of each data source that have matched. These three partitions are processed recursively and the resulting gene expressions are combined with the proposed one using a catenation and a union.

The actual combining step can apply a number of simplifications to avoid constructing a huge gene expression. For example nested catenations and unions can be combined and empty genes can be removed from a catenation.

4. Suffix trees and -arrays

The most difficult part of the project is keeping time and space complexities of each step in acceptable bounds. A simple layered, heuristic and inherently parallel structure inference program would probably fit in few thousand lines of code. It would also be absolutely useless even for nontrivial inputs.

Our first prototype consisted essentially of a gene pool expert and a controller, and simply iterated all of the remaining input data at each step to find. This is obviously inefficient, since a gene usually has to inspect the same sequence of bytes many times. One way to solve this problem would be to have the genes memoize the inputs. There is however a more general and elegant technique.

Suffix trees and -arrays were developed for processing genomes. Suffix trees can be thought of as radix trees into which each suffix of the input data is inserted. Suffix arrays were developed as a more space efficient alternative to suffix trees. A suffix array of a string of length n contains a permutation of the numbers [0..n], denoting the lexicographically sorted suffixes of the input starting from each position.

The reason why a this data structure is useful is that once it is computed for some input data, certain queries about the data can be answered efficiently. In our case both the gene matching and shared substring searching operate directly on the suffix array. This makes it possible to effectively process each occurrence of each unique byte sequence in the input in parallel.

Construction of suffix arrays is important for both bioinformatics [2] and data compression [3]. Currently the most efficient algorithms for constructing them have linear time and space requirements. Our current prototype uses an algorithm described in [1]. It was modified to be able to process multiple data sources.

When the controller has selected a proposal and partitions the data based on it, the data in the suffix array changes. Instead of recomputing the suffix array for each new stage the ordering of the previous one is used to build the new ones. Therefore the suffix array needs to be constructed only once.

5. Future directions

The presented techniques will be refined for a while, after which a fuzzer will be made that uses the inferred structure. If we were to continue in this direction, learning expert modules and evaluation functions would be the next logical steps. The most obvious extension to controller would be a decent post-processing step after combining the partitions. This would for example recognize shared gene expressions found in separate partitions as instances of the same structure.

Although these additions would improve the quality of inferred structure, there is still a fundamental problem in our current approach. In most interesting cases the input consists of layers, where structure should be found found from the interpretation of another structure.

Our current plan is to experiment with the presented techniques, and then move to more general structure inference.

References:

[1] N. Jesper Larsson, Kunihiko Sadakane *"Faster Suffix Sorting"* Technical Report LU-CS-TR:99-214, Dept. of Computer Science, Lund University, 1999

[2] M.I.Abouelhoda, S. Kurtz, E. Ohlebusch *"The enhanced suffix array and its applications to genome analysis"* Proc. 2nd Workshop on Algorithms in Bioinformatics, volume 2452 of LNCS, 2002

[3] Sadakane, K. *"A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation"* Data Compression Conference, 1998. DCC ’98. Proceedings

[4] The usual suspects *"Protocol Genome Project: Functional Genes (draft)"* Our project web site.