Model Inference Guided Random Testing of Programs with Complex Input Domains

Revision : 1.44

Aki Helin, Joachim Viide, Marko Laakso, Juha Röning
University of Oulu, Computer Engineering Laboratory
Linnanmaa BOX 4500, FIN-90014 University of Oulu, Finland
ouspg@ee.oulu.fi

ABSTRACT

The obvious need for networked software to survive malicious input has promoted robustness testing where exceptional input is either manually or randomly designed with the hope of catching the vulnerabilities prior to wide exploitation. Manual test design is subject to human errors, the language of undocumented proprietary protocols is out of the reach of the designer, and even with documentation lack of human resources may become a bottleneck. Conversely, blind random fuzzing is hindered by the impossibility of addressing infinite input space in finite time. As a compromise between pure randomness and human design we have developed an intelligent random testing methodology. The technique is based on generating testing material for programs by mutating an automatically inferred structural model of their proper input data. Our technique is applicable to programs that process input, provided that samples of the input data are available. The technique can be used in a black box manner to automatically produce test cases. It can also be extended with domain specific knowledge in a natural way. We argue that our approach strikes a practical compromise between completely random and structure aware test case generation techniques.

1. INTRODUCTION

Security vulnerabilities infest information technology. The programs we use process information from various sources and use a plethora of encodings and protocols. Input processing routines are among the most exposed areas of a program, which is why they should be especially reliable. This is rarely the case. The obvious need to survive malicious input has drawn attention to robustness testing where exceptional input is either manually or randomly designed with the hope of catching the vulnerabilities prior to wide exploitation.

The classic work by Miller et al. demonstrated the effectiveness of random testing for disclosing security critical input parsing errors.[18, 17] The PROTOS project[2] developed an approach to systematically test implementations of protocols in a black-box fashion. *PROTOS classic* approach produced several highly effective test suites. The most famous of them so far being the SNMP suite, which affected over one hundred vendors and raised considerable interest e.g. from the critical infrastructure protection perspective.[11, 3] Lately fuzzing has become a buzzword in information security. Many recent public disclosures of vulnerabilities have been based on various degrees of fuzzing.

Our previous work in robustness testing of protocol implementations has shown that manually designed structural mutations and exceptional element values are an efficient way to expose errors in networked software. Unfortunately manual test design is subject to the same human errors as the original programming task of the tested software. Furthermore, the languages of undocumented proprietary protocols are out of the reach for human test designers, and even with documentation the lack of human resources may come a bottleneck. Blind random fuzzing on the other hand is hindered by the impossibility of addressing an infinite input space in finite time.

During the PROTOS project the suspicion emerged that there could be a fairly small set of structural building blocks that most real-life protocols actually use. A corollary would be that many protocol implementations, considering their general quality, could share the same kinds of vulnerabilities. And indeed, the test case designers soon began to spot what they called *death zones* from the protocol specifications; similar parts between specifications for different protocols which seemed to be systematically implemented particularly sloppily. Eronen and Laakso have identified some possible reasons for this.[9]

The hypothetical structural building blocks were called *protocol genes*, and based on this idea the PROTOS Protocol Genome Project was initiated in 2003. The main motivation was to study the existence of protocol genes, and to find ways to identify and exploit them. The real ulterior motive was to get rid of the most time consuming phases of PROTOS classic testing. The goal was to essentially produce a technique and a general tool to automatically create effective test cases from arbitrary valid data examples, which would in the long run complement the manual test design approach.

The resulting techniques can be seen as instance of what we call *model based fuzzing*. The idea is to automatically build a model describing the structure of some given training material, and use the model to generate similar data to be used as robustness testing material. By using a higher level description of the data, the fuzzer is able to make changes to structure as well as the content of some training material.

In the PROTOS classic approach the model is built manually, and in traditional random testing the model can be considered to be a trivial one.

The approach can be split to three main phases. Firstly the concept of a structure is made concrete by selecting a language in which to represent the models. Any inferred structure will then be representable as an expression in the chosen language. The second phase is model inference, the task of which is to build a meaningful or otherwise interesting description of the training material. The last phase is using the model, or possibly a mutated version of it, to produce data that resembles the training material. This data can then be used to test the target programs.

In this paper we describe how these phases are implemented in one of our prototype tools, and discuss initial experiences on the effectiveness of the produced test cases when pitted against real life software. Finally we argue that our approach strikes a practical compromise between completely random and structure aware test case generation techniques.

2. REPRESENTING GENES

The first task in building a random model based testing framework is selecting a knowledge representation system. The purpose of this system is to store the result of structure inference. Since the structure inference step is usually computationally complex, it is useful to have an external or otherwise storable representation for the result, so that it can be saved and reused. The system should provide a way to easily represent the kinds of structures that will be inferred, along with means of processing and using them.

In many cases the result can simply be stored in an ad-hoc data structure. However, a well-defined language may prove to be useful in structure inference and fuzzing. There are many formal languages which are suitable to this task. In our case the requirement was, that the language should be able to easily define the shared building blocks in packets of communication protocols; the protocol genes.

2.1 Formal Grammars

Our first approach for expressing protocol genes was based on formal grammars[10], namely the regular and context-free subsets. These subsets of grammars are widely used in form of regular expressions and BNF based syntax definitions. Many subsets of grammars have well known propertis, such as matching complexities and techniques, associated automata, normal forms and in some cases inferring complexities. They are an attractive structure representation system, because they can be easily processed symbolically, have clean and simple semantics and can describe various syntactic structures efficiently.

One early prototype of ours, called *RegExpert*, produced incrementally more precise Unix-style regular expressions from input data. Figure 1 shows a regular expression produced from a bunch of GIF images. An eye trained to the ways of regexps may see that even the early prototype finds some structural information: the magic identifier string ("GIF87a") and most of the header.[1]

```
GIF87a.{1}\^C.{1}\^B.{1}\000\000\000\000\000\ß\ß\ß.*\,\000\000\000
\000.{1}\^C.{1}\^B\000\^B\■\<8C>\<8F>\l'\Ë\í\^O\č\<9C>\t'\Ú\<8B>\ş\■\ij\û
\^O\<86>\âH\<96>\æ\<89>\ę\ê\Ê\■\î\^K.{2}L\Œ\ö\<8D>\ç\ú\Î\œ\■\^O\^L<87>.{3}
\<88>L\*\<97>\Ì\ę\ó<8D>J\ğ\Ô.{2}\<8A>\Íj\ů\Ü\■\œ\^K\^N\<8B>.*\Œ.*\ì\■.{3}
\Ë\ç\ô\ž\ý\<8E>\Ï.{3}\¿\ß\^O\^X\(8.*\È.*\Ø.*\è.*\)9IYi.*\Ù.*\é\ù.*\.*\:
JZjz.*\<8A>.*\ž\Ê\Ú\ê\ú.*\.*\;K\[k\{.*\■.*\Ë\Û\ë.{4}\<L.*\ñ.*\ij\Ì\Ü.*\ì.*\-

\=M.{2}\}\<8D>\<9D>\■.*\¡.*\í\ý\^M\^^.{3}\^n\~\<8E>.*\<9E>.*\Î.*\SS.*\■.*
\■.*\■.*\‘.*\■.*\Î.*\■.*\<.*\■.*\<99>.*\■.*\ĕ.*\■.*\■.*\ë.*\Û.*\■.*
\Ï.*\■.*\^?.*\^?.*\<83>.*\<85>.*\<87>.*\<88>.*\<8A>.*\<8C>.*\Í.*\■.*
\<95>.*\<89>.*\<9A>.*\<9A>.*\■.*\ą.*\d'.*\■.*\■l.*\■l.*\ñ.*\■.*K.*\■.*\■.*
\¿.*\■.*\Â.*\■.*\<8C>.*\Ì.*\Ì.*\Ë.*\■.*\<9D>.*\Í.*\Ý.*\■.*\Ý.*\■.*\é.*\î.*
\■.*\õ.*\■.*\000.*\■.*\).*\■.*\Â.*\■.*6.*\■.*\].*\■.*\".*\■.*\d'.*\■.*V.*
\■.*\■.*\ä.*\■.*\■.*\Ï.*O.*\■.*\^Q.*\^Q.*\■.*R.*\ò.*\/.*\■.*S.*\?.*\^T.*\■.*
U.*u.*\■.*l.*6.*\■.*w.*7.*\■.*\<8F>.*Y.*\ź.*y.*\:.*\;.*\ij.*\ü.*\ü.*\ü.*
\=.*\;.*
```

Figure 1: A regular expression an early prototype RegExpert produced from GIF images.

The downside of using grammars is that they can not easily represent many simple structures used in protocols and file formats. These structures, such as length-payload pairs and checksums, can be considered as leaks from semantics to syntax. Creating definitions of them using grammars alone is possible, but the resulting system would lose the simplicity appeal. To this end, the grammar formalism is often extended with operations for handling other kinds of tasks. Attribute grammars[13] provide a well defined way of extending grammars with semantics. A similar approach was adopted in the former PROTOS classic model representation system.

2.2 Functional Genes

Instead of adding extensions to grammars, we decided to add grammar-based operations as an extension to another system. *Functional genes* is a small domain-specific language developed at early phases of our project. It provides a way to build declarative structure descriptions by using grammar-style rules as well as purely functional program code. The language consists of a small set of simple primitive operations, from which aggregate structures can be built. The language is implemented as a subset of the Scheme[12] programming language.

A functional gene is an expression obeying a simple grammar. Each functional gene defines a structure in some given data, and gives an interpretation of its meaning. The genes can be processed symbolically in the inferring phase, and later they can be evaluated for example to extended parsers or fuzzers of the specified structure.

In addition to being useful as a language for storing intermediate results of structure inference, manually written functional genes offer a convenient way to express simple parsers, in which case they are evaluated to fairly standard code for backtracking parsing functions. Figure 2 shows a handwritten example functional gene representing well formed IPv4 packets. The gene is defined as a sequential structure of fields with different bit widths. The interpretations of

[1] Coincidentally - and a bit ironically - the GIFs used were wood spectrum images which our neighbouring group working on image pattern recognition used in a completely different context. The resulting regular expression worked, without modifications, quite well for filtering away most GIFs that were not wood spectrum images from the same source.

```
(let-structure
  ((ip-version (integer 4))
   (header-length (integer 4))
   (service-type byte)
   (total-length (integer 16))
   (identification (integer 16))
   (skip zero-bit)
   (DF-bit bit)
   (MF-bit bit)
   (fragment-offset (integer 13))
   (time-to-live byte)
   (protocol byte)
   (header-checksum (integer 16))
   (source-address word)
   (dest-address word)
   (options
     (repeat (- header-length 5) word))
   (payload
     (repeat
       (- total-length (* header-
length 4)) byte)))
  payload)
```

Figure 2: A functional gene representing IPv4 packets

these fields can be named and used in other parts of the definition. In the example, the definition begins by assigning the names `ip-version` and `header-length` to integer-interpretations of the first two 4 bit sequences some data. The names `integer`, `byte` and `word` refer to previously defined or primitive functional genes, whereas `repeat` is an operation for composing joint structural definitions. At the end of the structure its final interpretation - in this case the payload of an IPv4 packet - is specified.

3. MODEL INFERENCE

Once a suitable knowledge representation system was chosen, we can proceed to infer structure from the data. This is the model inference step, which is a tough nut to crack. Ideally the program should be able reasonably to deal with common file formats, network protocols, as well as natural language and weather data statistics.

3.1 General Principles

We will assume that the training material can be encoded as an initial model, that is, an expression of the structure representation language. In our fuzzing context, the initial model describes a set of files containing the valid program inputs. The structure inference step can be specified as the task of finding a more interesting model that does not conflict with the initial one. One approach would be to grow a new model altogether, for example by using genetic programming techniques. We have mainly focused on applying property-preserving transformations to the initial model. In both cases the process, conceptually or in practice, consists of a rapidly expanding tree of possibly better models.

An important subproblem is that of model selection. Given two models, one should be able to decide which of them is more interesting. One common approach to solving this problem is to use MDL (Minimum Description Length) principle[19]. It is often useful to equate learning, or inferring, structure from some data with the ability to compress it. A good model will generally require less space than the initial one, since it can describe redundancy in data by using higher-level concepts. The MDL principle uses the amount of information required to represent the model as the scoring method. In other words, it provides a formalised version of the Occam's razor. One of the most useful properties of this strategy is its tendency to protect from overfitting a model.

Even though this approach gives an intuitively sound definition for a better model, the fact that the problem is now equivalent with compression may not seem what was intended. However, assuming that one extends the model description language with domain specific knowledge, the score of a model may benefit from using the extensions. A model candidate can therefore be more interesting if it can describe data using the supplied background knowledge.

Assuming these principles, the task of writing a good model inference system would seem to be somewhat trivial; either enumerate all possible models in size order and finish with the first one that matches the training data, or start with the initial model and search the best possible model derivable from it. Rather obviously both of these approaches require exponential time and space in nontrivial cases. Using a turing-complete structure representation system, such as Scheme, finding the optimal model is not solvable[8]. If only the model size is used as scoring method, the problem is still equivalent with computing the Kolmogorov complexity[14] of the given data. Thus, one generally must resort to heuristics and make educated guesses.

3.2 Functional Gene Inference

Our first prototype of a functional gene inference engine operates by searching the contents of a model for occurrences of hand written functional genes describing common protocol structures, such as null terminated strings and length-payload pairs. In this tool, the domain specific knowledge consists these predefined structures. The latter prototypes also incorporate searching for shared content in input data, namely maximal frequently occurring substrings.

The tool starts by constructing a trivial model from the training data sources. The model is then evolved using a recursive divide and conquer approach. The tool first searches the model for occurrences of the predefined structures and frequently occurring substrings. A hand-written evaluation function, based on the MDL principle, is used to select the most interesting proposal at each division step. After the most interesting path has been selected, the model is partitioned around the current finding, and the surrounding parts are processed recursively.

When no more interesting structures are found, the conquering phase begins. The smaller submodels are recombined back into a complete model of the data. Some further model simplifications are also applied while recombining. The intention is to build a model which recognizes shared or otherwise interesting contents in training data, so that they may be mutated, repeated, deleted and permuted instead of simply modifying their contents.

3.3 Implementation Issues

The most difficult part of model inference is keeping time and space complexities of each step in acceptable bounds. If some expensive operation is to be performed frequently, it is often useful to make it faster by precomputing a suitable data structure. The intention is to represent the frequetly needed results statically in memory, so that the result can be simply looked up.

Algorithms and data structures, which may prove to be useful, can be searched from areas such as bioinformatics[22, 23], data compression[21] and artificial intelligence[20, 4, 7]. Indeed, for example the Protocol Informatics Project[1] has applied algorithms from bioinformatics to reverse engineer network protocols.

In our experience, suffix trees[24] and suffix arrays[16, 5, 15] have been useful. Once constructed, they allow parallel walking of each unique occurrence of a sequence of data in the model using simple constant- or logarithmic time lookups. Suffix arrays are especially attractive because of their low memory overhead. In our current prototypes functional genes and other techniques operate directly on a suffix array computed from the input data. This speeds many operations considerably while adding only a small constant memory-overhead. In many cases the suffix array needs only be constructed once, since the same ordering can be reused when the model changes.

A functional data structure for model representation may also prove to be useful, since it makes backtracking easy and allows automatic sharing of contents between model generations.

4. FUZZING

Once a model has been inferred, the generation of randomised test cases, i.e. fuzzing, may finally commence. This is a fairly simple procedure compared to previous steps. Basically our prototype fuzzer takes a model expression, which in our case is a functional gene, and compiles it to a reverse parser, which generates data based on requested structure instead of parsing it.

The model, since it gives some insight into the structure of the correct data, allows a fuzzer to make changes to higher level structures as well as traditional changes to known contents. Since the inferred model usually builds generalisations rather than rigidly defining only the training data, the inferred model can generate more data than the initial one, where the the extra data can be considered fuzzed. However, we have had most successful results by intentionally mutating the model itself. The model mutations contain for example duplication, removal, swapping and random alteration of functional genes.

Different kinds of strategies in producing data from the model can be applied, depending on the desired goal. Making large sets of strongly mutated data, a strategy sometimes referred to as *shotgun testing*, is a good way to expose any suspicious behavior. On the other hand making single point mutations to original inputs is a good way to narrow down what exactly causes a problem. In both cases it is useful to store some metadata describing how each fuzzed piece of data was constructed, in order to be able to examine it later on.

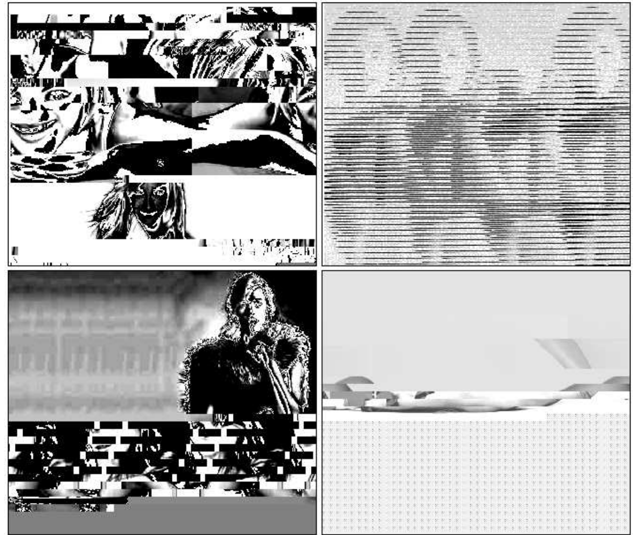

Figure 3: Some of the remotely haunting imagery of the 'Freak Show' gallery, produced by fuzzing valid GIF and JPEG images.

5. RESULTS

We have implemented and enhanced the described techniques in a series of prototypes and have used them to generate fuzzed test cases for several file formats. File formats were chosen as a good starting point for testing, because they share a lot of common ground with protocols in terms of implementation. In the tests we used a crude structure inference strategy which only finds maximal shared contents, and combines them in the manner described above. We also focused on visible and easily reproducible failure modes and did not include any low level monitoring for more esoteric and masked software failures. Nevertheless, this approach has proven to be surprisingly effective and malicious. In the following, we will briefly cataloque some trials in feeding fuzzed data to real programs. However, we will not go into specifics such as singling out tested software or test case details.

5.1 Round 1: Image files (GIF and JPEG)

The first two groups of test cases that were actually fed to implementations were for the GIF and JPEG image formats. The fuzzing for each format was done based on a few hundred valid images randomly obtained from the internet. From these images we generated around 1000 cases for both formats, which we fed to several programs by hand.

Amongst the test subjects were several popular web browser packages on various platforms, most of which we managed to crash with several generated broken images. We then constructed a HTML page gallery of some of the fuzzed pictures affectionately called 'Freak Show' (Figure 3). Loading the page caused visibly erroneous program behaviour in tested browsers, in contrast to the page simply being rendered to screen with some visual glitches.

5.2 Round 2: Office packages (DOC, RTF and XLS)

Two groups of test cases generated from valid DOC and RTF documents and XLS spreadsheets proved to be effective as well. The test cases managed to cause a wide range of visible failures in each sofware package, from resource exhaustion to to crashing and even locking up the entire operating system.

5.3 Round 3: Security software (multiple formats)

The latest development has included the testing of security software solutions to see their reactions to fuzzed data. We created test cases from several different data formats, such as executable files and RAR and ZIP compressed archives.

The caused quirks were extremely interesting and wildly imaginative. For instance, in one case feeding a fuzzed RAR data caused a security software solution to start ignoring all obvious security threats it usually catches. Meanwhile the program continued to present the impression there is nothing wrong with it.

Naturally feeding the image and office files from earlier rounds caused failures as well.

6. DISCUSSION

The presented technique has proven to be a surprisingly effective way of creating test cases causing repeatable visible software failures, considering its lack of any domain specific knowledge. Thus we argue that incorporating model inference with random test generation has the potential to overcome the inefficiencies of both random testing and handmade test suites, such as those of PROTOS classic.

Furthermore we postulate that the combination of manual test design and model inference guided random testing should be explored. The quality of the inferred model obviously depends on the available data samples; if samples lack in depth and diversity, then much of the dormant parsing functionality in software will be missed by the generated test cases. Manual test design would result in coarse partitioning of the input space, from where the machine may take over in order to systematically crunch the fine-grained details. This way the ill effects of tunnel vision and omissions as well human errors may be alleviated in test design. Perhaps this will be a way to leap beyond the pesticide paradox as stated by Boris Beizer: "Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual."[6]

The most significant limitation of the described approach is its lack of domain specific knowledge. The means of expressing, inferring and incorporating external reasoning should be developed further. A realistic tool would probably combine several independent model inference techniques in a unified framework. A sufficiently powerful structure description language could be used as the common denominator to glue the approaches together.

The design of our initial prototypes was biased towards being able to generate effective testing material for a certain class of programs. Now that we have developed something that has proven to be effective, work will also contiue towards general purpose model inference and abuse. Ultimately we aim to mature this technique into a test case generation framework, which would be similar to the one produced in the earlier PROTOS project, but easier and faster to use. We are also planning on releasing full fledged test sets of fuzzed data in the manner established in the PROTOS project.

7. CONCLUSIONS

An automatically inferred model can be used as basis to generate correct looking data for program robustness testing. The technique seems to be effective if the inferred models succeeds in finding meaningful structures from the original data. The bottleneck is usually in model construction, which requires resources.

Our testing technique can be applied to any sample input to automatically produce test cases in a black box manner. The technique can also be extended in a natural manner with domain specific knowledge that augments its efficiency. We argue that our approach strikes a practical compromise between completely random and structure aware test design, while it can be used in conjunction with these techniques to complement them.

We believe that automatic structure analysis can make random testing a viable option, because a structural model allows a randomized fuzzer to generate more meaningful changes in robustess testing material. Our experiences with simple prototypes suggest that this approach is effective and should be explored further.

8. REFERENCES

[1] The protocol informatics project. http://www.baselineresearch.net/PI/.

[2] Protos - security testing of protocol implementations. http://www.ee.oulu.fi/research/ouspg/protos.

[3] Protos test-suite: c06-snmpv1. http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/.

[4] N. Abe. Feasible learnability of formal grammars and the theory of natural language acquisition. In *Proceedings of the 12th conference on Computational linguistics*, pages 1–6, Morristown, NJ, USA, 1988. Association for Computational Linguistics.

[5] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *2nd Workshop on Algorithms in Bioinformatics*, LNCS, 2002. http://citeseer.ist.psu.edu/abouelhoda02enhanced.html.

[6] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, USA, 1990.

[7] S. Ben-David and M. Jacovi. On learning in the limit and non-uniform (&egr;,&dgr;)-learning. In *COLT '93: Proceedings of the sixth annual conference on Computational learning theory*, pages 209–217, New York, NY, USA, 1993. ACM Press.

[8] G. J. Chaitin. *The Limits of Mathematics*. Springer-Verlag, 2003.

[9] J. Eronen and M. Laakso. A case for protocol dependency. In *Proceedings of the First IEEE International Workshop on Critical Infrastructure Protection*, Nov. 2005. http://www.ee.oulu.fi/research/ouspg/protos/sota/matine/IWCIP2005-depen%dency/.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[11] R. Kaksonen. *A Functional Method for Assessing Protocol Implementation Security*. Technical Research Centre of Finland (VTT), Espoo, Finland, 2001. Licentiate thesis. http://www.ee.oulu.fi/research/ouspg/protos/analysis/VTT2001-functional%/.

[12] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998. html://citeseer.ist.psu.edu/article/kelsey98revised.html.

[13] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968. http://dx.doi.org/10.1007/BF01692511.

[14] A. Kolmogorov. Logical basis for information theory and probability theory. *IEEE Transactions on Information Theory*, 14(5):662–664, 1968.

[15] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.

[16] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[17] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[18] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[19] J. Rissanen. Hypothesis selection and testing by the mdl principle. *Comput. J.*, 42(4):260–269, 1999.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[21] K. Sadakane. A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation. In *Proceedings of IEEE Data Compression Conference (DCC'98)*, pages 129–138, Mar. 1998. http://citeseer.ist.psu.edu/sadakane98fast.html.

[22] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjölander, R. C. Underwood, and D. Haussler. Recent methods for RNA modeling using stochastic context-free grammars. In *Proceedings of the Asilomar Conference on Combinatorial Pattern Matching*, New York, NY, 1994. Springer-Verlag.

[23] I. Salvador and J.-M. Benedí. Rna modeling by combining stochastic context-free grammars and n-gram models.

[24] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.