Experiences with Model Inference Assisted Fuzzing

Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen
Mika Seppänen, Kimmo Halunen, Rauli Puuperä, Juha Röning
University of Oulu, Computer Engineering Laboratory
ouspg@ee.oulu.fi

Abstract

In this paper we introduce the idea of model inference assisted fuzzing aimed to cost effectively improve software security. We experimented with several model inference techniques and applied fuzzing to the inferred models in order to generate robustness attacks. We proved our prototypes against real life software, namely anti-virus and archival software solutions. Several critical vulnerabilities were found in multiple file formats in multiple products. Based on the discovered vulnerabilities and the positive impact on the security we argue that our approach strikes a practical balance between completely random and manually designed model-based test case generation techniques.

1 Introduction

Software has bugs, and the bugs often have security implications. 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 anomalous input is constructed either manually or randomly with the hope of catching the vulnerabilities prior to wide exploitation.

The classic work by Miller et al. demonstrated the suitability of random testing for disclosing security critical input parsing errors [15, 16]. In 1999 the PROTOS project [1] developed an approach to systematically test implementations of protocols in a black-box fashion. "PROTOS Classic" approach produced several highly successful test suites. [10, 18–20]. Lately fuzzing has become a buzzword in information security. Many recent public disclosures of vulnerabilities have been based on various degrees of fuzzing [2] [3] [4] [5].

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 software. Unfortunately while powerful, manual test design has some bottlenecks. It requires some kind of format specification as a basis, and e.g. poorly documented formats must be reverse-engineered before test designers can write a model-based test suite for the format. The human factor also brings in the danger of tunnel vision, as the power of manually designed cases is largely dependent on the expertise and imagination of the designer. On the other hand, blind random fuzzing has a considerably lower entry barrier, but is hindered by the impossibility of efficiently addressing a virtually infinite input space in finite time.

Our hypothesis is that to be able to effectively test various areas of a program, the testing input data must have a fairly correct structure. One way of generating this kind of data is to use known valid input as basis and change random parts of it to random values. However, this approach has the major drawback of not being able to make wholesome structural changes to the data without knowing how or what it represents. Assuming that some knowledge about the structure of the data is available, it is possible to make mutations also in the structural level. These mutations often have complex consequences in the resulting data, making them extremely improbable to emerge from a random fuzzer.

The PROTOS Protocol Genome Project was initiated in 2003 to be a continuation of PROTOS Classic. One

[1] http://www.ee.oulu.fi/research/ouspg/protos/

[2] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-3741

[3] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-2754

[4] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-6353

[5] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-3493

of the goals was to essentially produce a technique and a general tool that automatically creates effective test cases from arbitrary valid data examples. The tools could then be used to complement manual model-based test case designs. We call the resulting approach "model inference assisted fuzzing". The idea is to automatically build a model describing the structure of some given training material, and use the model as to generate similar data for robustness testing.

Model inference for network protocols has previously been studied in the reverse engineering context [5, 7]. The Discoverer tool by Microsoft Research demonstrated the feasibility of model inference for some protocols. Inferring grammars, specifically context free grammars, from arbitrary input data has been extensively studied in data compression research [12, 13].

Purely random testing was deemed ineffective based on previous experiences, and precise testing based on a manually designed model is known to work well. What kind of curve lies between the ends? Our prototype tools have tested several points in between, and the results have been surprising.

Our approach can be split into four main phases. In the first phase samples of data that represent the tested format are collected for training. The second phase is model inference, where a description of the training material is formed. The third phase is using the model, or possibly a mutated version of it, to produce data that resembles the training material. In the last step the produced data is used as input for the test subjects, which are monitored for anomalous behavior. In the beginning of this paper we concentrate on the second and third phases, model inference and fuzzing, and describe how these phases are implemented in some of our prototype tools. Then in the following sections, we discuss experiences on the effectiveness of the produced test cases when pitted against real life software, namely anti-virus and archival tools, in the format distributed to relevant vendors. Multiple critical vulnerabilities were found in several products. Finally, we argue that our approach strikes a practical balance between completely random and manually designed model-based test case generation techniques.

2 Model Inference

The first task is to choose a formal system for the model. You could easy roll your own domain specific language, data types and operations or objects, but one should also remember that mathematics provides several well understood systems. The important thing is that the system is expressive enough to represent the kind of information that will be inferred. The next step is implementing structure inference. This can be anything from a simple pass computing probabilities to an unsolvable problem, solution to which is only approximated. For many systems, like Markov Models, there are many existing libraries and tools [1–3].

When choosing the formal system, the most important point to keep in mind is the balance between the system's expressiveness and the ability to infer models that take advantage of the system's full power. Generally it might be said that the more expressive a formal system, the harder such inference becomes. Dealing with simpler systems tends to be easier and therefore has less overhead for experimenting with.

It is useful to view the structure inference as an optimization problem. A trivial model based on training data simply lists each case of the training data. This gives no additional insight to the structure or semantics of the data. Fuzzing based on an empty or trivial model corresponds to naive random testing. At the other end of the spectrum lies the hypothetical perfect model for the format that the training samples represent. The problem is to advance from the initial trivial model to better ones based on some evaluation metric. We often use the "Minimum Description Length" [22] principle to evaluate the models. This in essence means that we consider the best model to be the one that represents the input data in the most compact form, as such a model expresses the most regularity in the input. Should this be a good metric, a consequence is that compression algorithms can be useful in robustness testing, as the data structures they infer can be used as models.

2.1 Experiences

In our first prototype tools we assumed that the inferred model should closely follow the human written specifications of the valid data. To this end we built an embedded domain specific language called "functional genes" to house the information. It was easy to define common structures of communication protocols by writing the descriptions manually, but the power came with a price. The language was Turing complete, which made good and lightweight inference hard.

Some of our early structure inference tools used only a subset of their expressive power, namely that of *Context Free Grammars* (CFGs) [9]. We have since then started to use CFGs as models in most of our tools. CFGs excel at describing syntactic structure, which subsumes lexical structure, but they cannot be used to define everything. Although not universal, they are pleasant to work with. It turns out that for our purposes CFGs strike a good balance between simplicity and power.

Although there are good pre-existing algorithms such as SEQUITUR [17] for building context free grammars from tagged training data, we decided to roll our own algorithms for the inference step. Early on we used suf-

Figure 1: Context Free Grammar (CFG) inference illustrated

fix trees [23] and suffix arrays [14] for identifying most common shared substring of data elements (e.g. bytes) between different training data specimens. Each appearance of the selected substring was then replaced in the source data with one placeholder tag (called a "non-terminal") that could be later expanded back. The process was repeated until the data could not be further compacted. Later, we have started favoring a simpler method whose variations we collectively call MADAM that does the same thing but just for digrams found from the training material. Later in our research we found out that a very similar linear time inference algorithm called RE-PAIR has been described earlier in data compression context [12]. While its implementation is fairly simple compared to e.g. SEQUITUR, turns out it produces grammars with comparable quality in terms of the Minimum Description Length principle.

Figure 1 illustrates the CFG inference process. In this case three natural language samples are fed into the inference algorithm, and in the end a context free grammar is produced. The original samples are represented by non-terminals 0, 1 and 2. Non-terminals 3-5 represent reoccurring substrings, for example non-terminal 4 expands to the substring "HEY". When using natural language as samples general rules, such as periods being followed by whitespace and certain digrams being very common, tend to be found. Our hypothesis is that the same applies to file formats - the inferred CFGs model actual data structures.

We have also experimented with finite state machines, n-gram probabilities, and other model based systems, as well as traditional random testing. An interesting and currently open question is how much better, if at all, the more computationally expensive models are at providing good input coverage.

3 Fuzzing

Once a model has been inferred, the generation of randomized test cases, i.e. fuzzing, may finally commence. This is usually a simple procedure compared to the previous steps. The model inference step often builds generalizations, which may allow generating more than the training material. This is already one source of fuzzed data. In addition to this we have introduced two kinds of structural mutations; *global mutations* arise when mutating the model prior to the data generation phase, and *point mutations* are inserted during the generation process.

Figure 2 illustrates two data generation runs using the grammar inferred in Figure 1: a regular one without any mutations and a mutated one with a point mutation. The regular run produces one of the original input samples. In the mutated version one expansion of rule 3 is skipped, and the fuzzed output `GABBA_HEY` is produced. Other possible mutations would be e.g. replacing rule 3 with rule 4 (`GABBA_HEYHEY`) or duplicating it (`GABBA_GABBA_GABBA_HEY`).

In case of our functional genes, the fuzzing part consisted of compiling the model into a reverse parser, which when called, generated the data conforming to the model instead of parsing it. With CFGs the generation step is the regular derivation process of formal grammars. Both mutation types described above were easy to implement, but we ended up not using global mutations. Point mutations were done by picking a probability, a *fuzz factor*, and with that probability altering each step of the generation process, e.g. by skipping the step or doing it twice. The higher the fuzz factor, the higher the rate of mutation.

Different strategies for choosing the fuzz factor can be applied depending on the desired goal. Using a large fuzz factor yields something we refer to as *shotgun testing*, i.e. heavily mutated test cases with point mutations applied liberally. On the other hand, narrowing down the number of mutations can be used to pinpoint an uncovered error, and tested programs may start outright rejecting too heavily fuzzed data. According to our experiences, varying the fuzz factor between test cases seems to give better results than keeping the factor constant.

Random testing cannot be effective when programs reject just about any change to data before processing it.

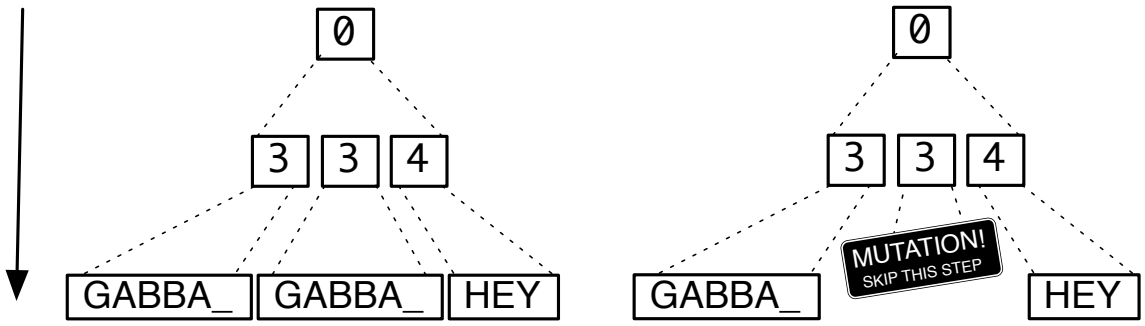

Figure 2: Regular and mutated data generation runs using the CFG inferred in Figure 1

Programs can use for example a checksum over a certain part of the data to decide the validity of input. However, these kinds of features can be added to the injection or as special cases to the structure. This approach could be continued to make fuzzing, and also structure inference, a human-assisted process. This kind of hybrid testing may well become the sweet spot of robustness testing.

4 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 and network protocols. When seeking a topic for a large scale test of the effectiveness of our approach, anti-virus software solutions presented themselves as a tempting target. This was because anti-virus tools by definition process input from potentially malicious sources, and usually run at high privileges, increasing the impact of a potential compromise. Anti-virus tools are commonly installed organisation-wide on all able computers, even critical and high-profile ones. Indeed, usage of anti-virus tools is commonly mandated by organisational policy, contract and other administrative and/or legal requirements. For example the US "Health Insurance Portability and Accountability Act" (HIPAA) [4] legislation is commonly interpreted to mandate use of anti-virus software.

However, as our approach is especially suited for rapid testing of several different formats, the most relevant motivation for choosing anti-virus tools is that they parse a wide variety of different data formats. They even tend to process files inside archived content. Therefore, we decided to create a test suite for several archive formats, namely ACE, ARJ, BZ2, CAB, GZ, LHA, RAR, TAR, ZIP and ZOO. Many archive formats have a long history, which has given their implementations plenty of time to mature and harden with respect to implementation level errors. Evaluating such mature products should provide us useful feedback on the current state of implementation level robustness in general as well as the effectiveness of our approach. Moreover, while the specifications for the archive file formats are in some cases available, there are many versions and variants of many of the formats, and there are in many cases no formal easily processable specifications of the contents. Covering such file formats with testing approaches requiring manual modeling of the tested protocol/file format, such as PROTOS Classic [10], would be cumbersome.

For the training material, we gathered samples of files compressed in various archive formats, with some attempt of representativity of features (different versions, password-protected files etc.), resulting in 10-100 training files per format. A model for each of the formats was built using our prototype. At most 320 000 fuzzed files were then generated from each model and tested against a variety of anti-virus products. Getting the tests running took approximately an hour per archive format, with the majority of the time spent on gathering the training material, not counting the initial software installation and setting up a general rudimentary environment for feeding the test cases and collecting the most spectacular crashes.

During the preliminary tests our approach proved to be, to our horror and surprise, quite effective. The findings are summarized in Table 1. Five anti-virus tools were instrumented only by capturing the exceptions raised by them and counting their crashes. Two crashes where the instruction pointer was identical were considered to be the same bug. This is a very rough metric and may underestimate the real number of bugs (a program does not have to crash to have a bug), but provides a reasonable estimate for software where the source code is not available.

All in all, four test subjects were found to be vulnerable to some of the test cases [6], each failing to survive

[6] The fifth product only implemented a small handful of archive formats.

Subject	ace	arj	bz2	cab	gz	lha	rar	tar	zip	zoo
1	3	2	1	1	-	3	-	-	3	1
2	-	5	-	12	-	2	1	-	-	-
3	-	5	2	1	-	3	2	-	-	-
4	-	1	-	-	-	1	1	-	1	-
5	n/a	n/a	n/a	-	-	n/a	n/a	-	-	n/a

Table 1: Unique bugs per file format.

test cases from at least four archive formats. The behaviour of the tested anti-virus programs after receiving input from the test cases was extremely interesting and wildly imaginative. For instance, in one case feeding a fuzzed RAR data caused an anti-virus tool to start ignoring all malware it usually catches. Meanwhile, the program continued to present the impression that there was nothing wrong with it. Also several other software tools that process archived files were found to be vulnerable, such as the now-patched bzip2 archiver [7].

As a result of our tests, we concocted a CD with 1 632 691 different fuzzed archive files, during the latter half of the year 2006. The vulnerability coordination task was taken on by the Finnish national Computer Emergency Response Team [8] (CERT-FI) and the Centre for the Protection of National Infrastructure [9] (CPNI, formerly known as NISCC), who identified more than 150 vendors whose products has prior been vulnerable to similar issues. The vulnerability coordination process was done according to the constructive disclosure [11] guidelines, and as such took a considerable amount of time. The test suite [21] and a related CERT-FI and CPNI joint vulnerability advisory [8] were finally released to the public 17 March 2008. Due to the long time period during which the test set was only available to vendors, a number of potential security holes in the software packages we had found vulnerable were independently found and fixed by other parties.

5 Discussion

The presented technique has proven to be a surprisingly effective way of creating test cases causing repeatable visible software failures, considering the fact that it lacks 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 hand-made 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 and after that, 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, i.e. the semantic meaning of pieces of data. 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, while still keeping things simple. For example, we have implemented simple tools for identifying potential checksum fields and length-payload pairs from given data samples.

We have not yet extensively studied how the collection phase of the original training data should be done to maximize the coverage of produced test cases. Moreover, our monitoring of test subjects is still rudimentary and warrants further exploration. For example, the data collected from properly monitored test subjects could be used as feedback for the model generation, although this might mean stepping outside the strictly black box testing mindset. Ultimately, we aim to mature our techniques into a test case generation methodology and framework, which would be similar to the one produced in the earlier PROTOS Classic project, but easier and faster to use. We are also planning on releasing more full fledged test sets of fuzzed data in the manner established in the PROTOS Classic project and refining the vulnerability disclosure process.

[7] http://bzip.org/
[8] http://www.cert.fi/en/
[9] http://www.cpni.gov.uk/

6 Conclusions

An automatically inferred model can be used as a basis to generate seemingly valid, but still anomalous data for robustness testing. The technique can be applied to any sample input to automatically produce test cases in a black box manner, and seems to be effective in revealing implementation faults in software. However, the approach in its current form should not be expected to give a good coverage of the test space. The fuzzing method is extremely rudimentary, and the context free grammars representation used does not really contain information about the actual semantics of the data.

In spite of this, 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 robustness testing material. We argue that our approach strikes a practical balance between completely random and manual test design. It can therefore complement both of these techniques. Our experiences with simple prototypes suggest that model inference assisted fuzzing is effective and should be explored and developed further. Being able to finding critical vulnerabilities in mature, security-related software is proof of this.

References

[1] Hidden Markov Model (HMM) Toolbox for Matlab. http://www.cs.ubc.ca/~murphyk/Software/HMM/hmm.html.

[2] The General Hidden Markov Model library (GHMM). http://www.ghmm.org/.

[3] The General Hidden Markov Model library (GHMM). http://htk.eng.cam.ac.uk/.

[4] Health insurance portability and accountability act. http://www.cms.hhs.gov/HIPAAGenInfo/Downloads/HIPAALaw.pdf.

[5] BEDDOE, M. A. Network protocol analysis using bioinformatics algorithms. http://www.4tphi.net/~awalters/PI/pi.pdf.

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

[7] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium* (Aug 2007), pp. 199–212.

[8] FINNISH NATIONAL COMPUTER EMERGENCY RESPONSE TEAM (CERT-FI). CERT-FI and CPNI Joint Vulnerability Advisory on Archive Formats. https://www.cert.fi/haavoittuvuudet/joint-advisory-archive-formats.html%.

[9] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[10] KAKSONEN, R. *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%/.

[11] LAAKSO, M., TAKANEN, A., AND RÖNING, J. Introducing constructive vulnerability disclosures. In proceedings of the 13th FIRST Conference on Computer Security Incident Handling.

[12] LARSSON, N. J., AND MOFFAT, A. Offline dictionary-based compression. In *DCC '99: Proceedings of the Conference on Data Compression* (Washington, DC, USA, 1999), IEEE Computer Society, p. 296.

[13] LEHMAN, E., AND SHELAT, A. Approximation algorithms for grammar-based compression. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2002), Society for Industrial and Applied Mathematics, pp. 205–212.

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

[15] MILLER, B., KOSKI, D., LEE, C. P., MAGANTY, V., MURTHY, R., NATARAJAN, A., AND STEIDL, J. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Tech. rep., 1995.

[16] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery 33*, 12 (1990), 32–44.

[17] NEVILL-MANNING, C. G. Inferring sequential structure. *PhD thesis* (1996). http://sequitur.info/Nevill-Manning.pdf.

[18] OULU UNIVERSITY SECURE PROGRAMMING GROUP. PROTOS Test-Suite: c06-snmpv1. http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/.

[19] OULU UNIVERSITY SECURE PROGRAMMING GROUP. PROTOS Test-Suite: c07-sip. http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/.

[20] OULU UNIVERSITY SECURE PROGRAMMING GROUP. PROTOS Test-Suite: c09-isakmp. http://www.ee.oulu.fi/research/ouspg/protos/testing/c09/isakmp/.

[21] OULU UNIVERSITY SECURE PROGRAMMING GROUP. PROTOS GENOME Test-Suite: c10-archive. http://www.ee.oulu.fi/research/ouspg/protos/testing/c10/archive/.

[22] RISSANEN, J. Modeling by shortest data description. *Automatica 14* (1978), 465–471. http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/.

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