Automatic Signature Generation: Report On The Initial Implementation

Todd Heberlein

This report describes the current status of our Automatic Signature Generation project. The primary goal of the effort to is quickly identify signatures for a new attack with the theoretically lowest possible false positive rates. The report covers the goals of the project, concept of operations, the core technology, and the current state of the implementation. The most important point for this report is we now have a running implementation, and in fact we are on our second-generation implementation and have designed our third-generation that can scale to support terabytes of data.

Table of Contents

1 Introduction 1
2 Goals of the Project 1
3 Concept of Operations 2
4 Suffix Tree Concept 3
5 Experimental Results 4
5.1 Scaling Factors 4
5.2 Moving to Support Larger Data Sets 5
5.3 Example Runs 5
6 Conclusions 7
7 References 7

List of Figures

Figure 1: Example of a Suffix Tree 3
Figure 2: Attack Data and Potential Signatures 4
Figure 3: Testing Candidate Signatures 4
Figure 4: Number of Link Nodes 5
Figure 5: Suffix Tree Size When Processing Relatively Non-compressible Data 6
Figure 6: Suffix Tree Size When Processing Compressible Data 6

1 Introduction

This report describes our Automatic Signature Generation project, the technology we use, the current level of implementation, and experimental results. The concepts and technology we use are relatively unique in the field of intrusion detection, so Sections 2, 3, and 4 provide background information and describe our overall goals for the project, the concept of operations, and the core technology used. Readers who have read previous papers on our efforts can probably skip these sections.

Section 5 describes the most recent accomplishments. The key point is that we now have an implementation running of the key technology. The code processes packets from a file and builds a suffix tree. Our implementation is based on the Ukkonen algorithm [Ukko 95], which processes the data in linear time with respect to the data; similarly the size of the tree is linear with respect to the size of the data. We have run the algorithm on a number of packet sources, and we present two experimental runs that demonstrate how the behavior of the suffix tree is affected by the type of data being processed.

Finally Section 6 summarizes the work and Section 7 provides references to relevant material.

2 Goals of the Project

The Automatic Signature Generation (ASG) project is primarily motivated by the following observations: (1) signature-based network IDS sensors often generate a large number of false alarms engendering a general distrust amongst their users, and (2) signatures cannot be generated quickly enough to play a role in fighting fast moving new attacks such as worms.

The false positive issue creates several problems. First, the high numbers of false positives creates additional workload on analysts. Second, the high false positive rates may encourage analysts to ignore certain reports or to simply turn off the signature responsible for the false positives. In either case, this may allow actual attacks to go unnoticed. Third, because of experience with the high false positive rates, network administrators are hesitant to use intrusion protection mechanisms available to many systems. Thus, while many network sensors can stop attacks from reaching their targets, operators refuse to use the capability.

So, our first goal is to create better signatures with extremely low false positive rates. In order to achieve better false positive rates we need to test candidate signatures against large sets of appropriately selected traffic, and this must be done so that it is not a burden to the analysts.

The second problem, the long time to create and deploy new signatures, causes intrusion detection systems (IDS) and intrusion prevention systems (IPS) to miss their sweet spot in capability – the time between when an attack first starts to circulate and the time a patch is widely applied. It is during this limited window of time that sensors offer the greatest value to the operators. For example, Cisco has created Cisco Network Admission Control (CNAC) technology that allows network devices to block network access to systems that are not patched [Cisc 03] [Leyd 03] [Robe 03], and as technologies such as this are fielded, the value of detecting attacks against vulnerabilities for which there are patches will diminish because most systems will be patched.

Thus our second goal is to greatly accelerate the time to create a signature so they can be fielded while they are most valuable. This is particularly important in the case of fast moving automated attacks such as worms.

Both goals are intrinsically linked. If we can quickly measure a candidate signature's false alarm rate against a large representative data set, we encourage the creation of better-tested signatures, and if the testing of candidate signatures is fast enough, we can use automated techniques to test large sets of candidate signatures and deploy the best signature (or signatures) in IDS and IPS devices.

3 Concept of Operations

In Section 2 we introduced the two core problems we propose to address: (1) false positive rates and (2) time required to generate quality signatures. In this section, in order to further illustrate the problems and examine how our proposed technology will address them, we look at four concepts of operations: (1) improving existing approaches, (2) automating existing approaches, (3) protecting critical services against zero-day attacks, and (4) protecting an enterprise against fast-moving, automated attacks such as worms.

Improve existing approaches to generating signatures for IDS systems. A typical approach for generating a signature for a newly discovered attack is to have an analyst examine the attack, generate a candidate signature, optionally test the signatures against an archive of past data, and deploy the signature in fielded systems. The testing of a single candidate signature against large amounts of past traffic (e.g., terabytes) may take hours, and if the analyst needs to perform several iterations of testing to refine the signature, the process may take days.

Our approach pre-processes the terabytes of archived network traffic so that testing a candidate signature against the archive will take about a second. This greatly improved turn-around time on testing should (1) increase the probability that an analyst will test a signature before deploying it, (2) allow the analyst to quickly refine the signature so that he can generate higher quality signatures, and (3) improve the time from detection of a new attack to deployment of a high-quality signature.

Automatically generate signatures for IDS systems. In some cases a human analyst in the loop may not be possible, or an organization may want to reposition the role of the human analysts. For example, some organizations may not have analysts available 24 hours a day, 7 days a week, or the attack may exploit an application that the analyst is not familiar with. In these cases, the system can automatically generate candidate signatures with the lowest possible false alarm rates.

We can achieve automatic signature generation because the testing of a candidate signature is so fast. We perform this by taking a sample of the attack and generating all possible candidate signatures (i.e., all possible substrings) and testing each against the terabytes of data. A set of candidate signatures with the lowest false positive rates will be presented to the analyst for review.

Protect mission critical services. While the traditional approach for creating signatures for IDS sensors may take a long time, creating, testing, and deploying a patch for a vulnerability in an application or service often takes much longer. Thus, in order to allow a mission critical service to continue operating in the face of a zero-day attack (e.g., an attack against an unknown vulnerability for which there is no known patch available) we need to stop the attack before it reaches the vulnerable service.

Fortunately, several companies are starting to deploy deep inspection firewalls, firewalls that can examine the content of packets and terminate sessions containing data matching a signature. By combining our ability to automatically and quickly generate high-quality signatures for new attacks with deep inspection firewalls positioned in front of the vulnerable service, we

create the capability to allow the mission critical service to operate with minimum impact on legitimate activity until a fix for the vulnerability can be found.

Stop the spread of automated attacks. Since about 1993 attackers have been embedding their attacks into software that can scan large swaths a network in seconds to minutes, and since about 2000 attackers have been embedding their attacks in self-propagating code that can infect much of the Internet in minutes to hours. Thus, even if a patch is available, there is no time to apply the patches to the many vulnerable systems, and if it is a zero-day attack, all the applications are vulnerable.

However, as in the previous case, by combining the quick and automatic generation of high-quality signatures with deep inspection firewalls distributed throughout an enterprise, we can quickly deploy a stop-gap measure (signature in a firewall) that will protect most of the machines from the automated attack until patches can be created, tested, and widely deployed.

4 Suffix Tree Concept

In this section we demonstrate the approach through a trivial example. For a more realistic example, the suffix trees are too complex to be reasonably displayed on a standard page.

Figure 1 shows a suffix tree for our fixed text "xabxac". This text represents traces of past network activity. Each path from the root to a leaf node represents a suffix in the original text T, and at each leaf is an index indicating the position in the text T for that suffix. For example, the left-most branch of our tree represents the suffix "bxac", and the index "3" tells us that this suffix begins at the third letter in the original text. We also have several numbers in parentheses, one at each leaf node and one at each branching location. These numbers will tell us how often a particular pattern is found in the original text. For example, the top right branch begins with the string "xa" before hitting a branch labeled with "(2)". This tells us that the string "xa" is found in the original text two times. For our purposes, these numbers tell us the false positive rates for a candidate signature.

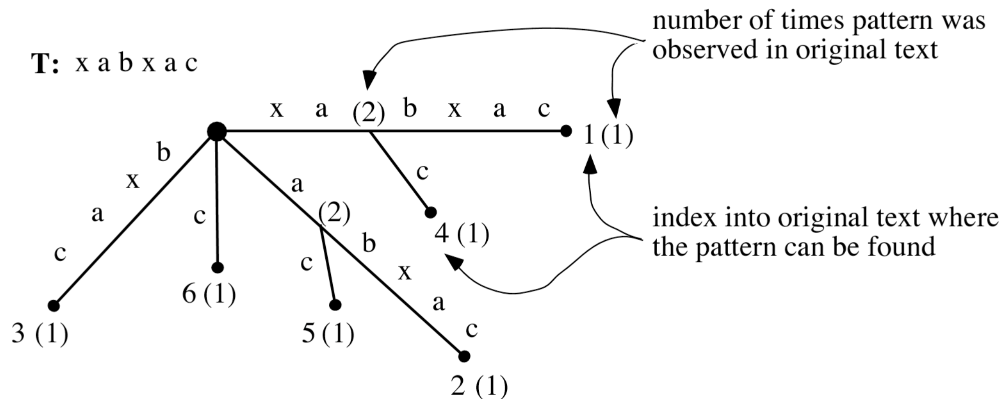

Figure 1: Example of a Suffix Tree

Figure 2 shows a captured attack "xax". How the initial attack is captured is outside the scope of this proposal. A honeypot may have been used, or a system such as ASIM that captures and stores all packets for a limited time (somewhat like an airplane flight recorder) may have captured the data. For illustration purposes, we will look at all possible signatures of at least two characters. For this attack we have two candidate signature: "xa" and "ax". These will be tested against our suffix tree of historical activity.

Historical Data: **x a b x a c**   Attack: **x a x**   Candidate Signatures: **x a**
a x

Figure 2: Attack Data and Potential Signatures

Figure 3 shows how the candidate signatures are tested against the historical data in order to identify false positive rates. Only the relevant portions of the suffix tree are shown in the figure. On the left side of the figure we show the testing of the first candidate signature, “xa”. We test the signature by traversing the suffix tree with the same pattern. As shown, “xa” follows the top-most right branch. Because we reach the end of the candidate signature while we are still in the tree, we know the pattern does in fact exist in the historical data. Furthermore, by following the current tree segment to the next branching location or leaf node (whichever comes first), we arrive at a number in parentheses telling us how frequently the pattern occurs in the data. In this case, the pattern “xa” occurs twice in the historical data.

On the right side of Figure 3 we test the second candidate signature, “ax”. Once again, we start traversing the tree beginning with the branch that begins with the character “a”. We immediately hit a branching location, but none of the branches follow with the next letter in the candidate signature, ‘x’. Because, by starting with the root of the suffix tree we could not find the pattern “ax” in the suffix tree, we know this pattern does not exist in the historical data. The string “ax” should be our signature for detecting the attack “xax”.

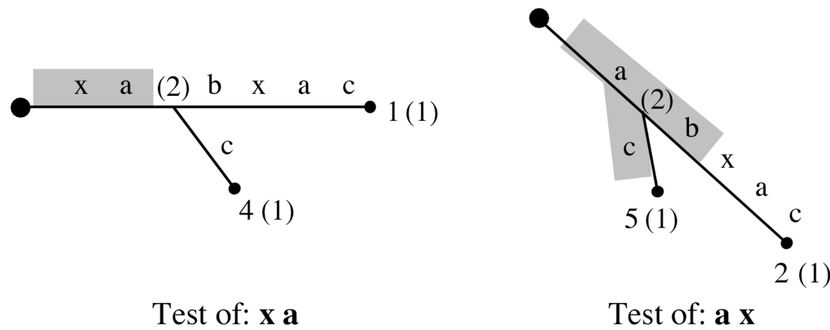

Figure 3: Testing Candidate Signatures

5 Experimental Results

5.1 Scaling Factors

The key to the entire project is that whether you have 100 bytes of test data or 100 terabytes of test data, to test a candidate signature of 20 bytes against that test data only requires 20 compares into the data structure. And the more bytes of test data you have the greater your prediction of false positive rates for a candidate signature will be.

Two concerns that we had about our proposed approach were (1) how quickly we could build the suffix tree and (2) how large the suffix tree would be compared to the original test data. Fortunately Esko Ukkonen developed an algorithm that, with respect to the number of test bytes, can build the suffix tree in linear time. Furthermore, the size of the suffix tree is also linear with respect to the data [Ukko 95]. This roughly means that if the suffix tree for 100 MBytes of data was X bytes in size, then the suffix tree for 200 MBytes of data would be about 2X bytes in size.

We measure a suffix tree’s size by the number of link nodes in the tree. For example, the tree in Figure 4 has eight link nodes, where a grey oval identifies each link node. While our graphic representation implies that links are of different sizes depending on the number of bytes in them, in the implementation each node is of equal size, and pointers (or indexes) indicate the

location in and span of the original data that the node represents. In the Ukkonen algorithm the number of link nodes (grey ovals) are proportional to the size of the original data that the tree represents.

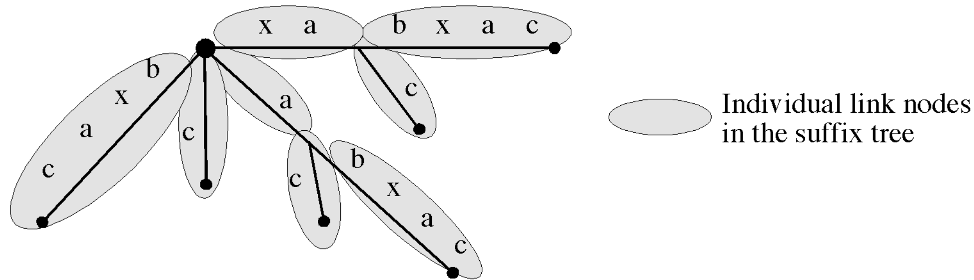

Figure 4: Number of Link Nodes

The one drawback of the implementation we use is that while the tree size is proportional to the data that it represents, there is still a relatively large constant expansion factor. For example, in the worst-case scenario for every byte of test data we need 30 bytes of data structure. The data in the worst-case scenario would be completely random and non-compressible. Fortunately in practice this will not generally be the case, and we believe that we can recognize apparently random data and treat it separately.

5.2 Moving to Support Larger Data Sets

Our current implementation developed for this phase of the project is completely in memory, and this limits the amount of data that we can represent in our suffix trees. To address this limitation we will move to a disk-based solution in the next phase of the project. To make this transition easier we have modified our current in-memory solution by removing the use of linked lists, binary trees, and arrays and are now using hash tables containing objects. In the database solution each object will map to a table, and the keys used in the hash tables will be the same keys used to link tables in the database. Since modern databases easily support terabytes of data, our new solution should scale to large data sets.

5.3 Example Runs

To demonstrate scaling behavior of our implementation of a suffix tree using the Ukkonen algorithm we ran the data on two sets of data: data from web servers and data from web clients. The data from the web server was dominated by compressed images, and compressed images provide a close approximation to random data. The data from clients to the web servers exhibit a much more regular pattern, is more compressible, and therefore creates smaller suffix trees.

Figure 5 shows the number of suffix tree nodes (vertical axis) needed to represent the number of bytes from a web server (horizontal axis). The graph represents 250 packets containing data (primarily compressed images), and the number of bytes processed and the number of nodes used to represent the current suffix tree are calculated after each packet. In other words, there are 250 data points represented by the line. The line is remarkably straight indicating very little compression was possible.

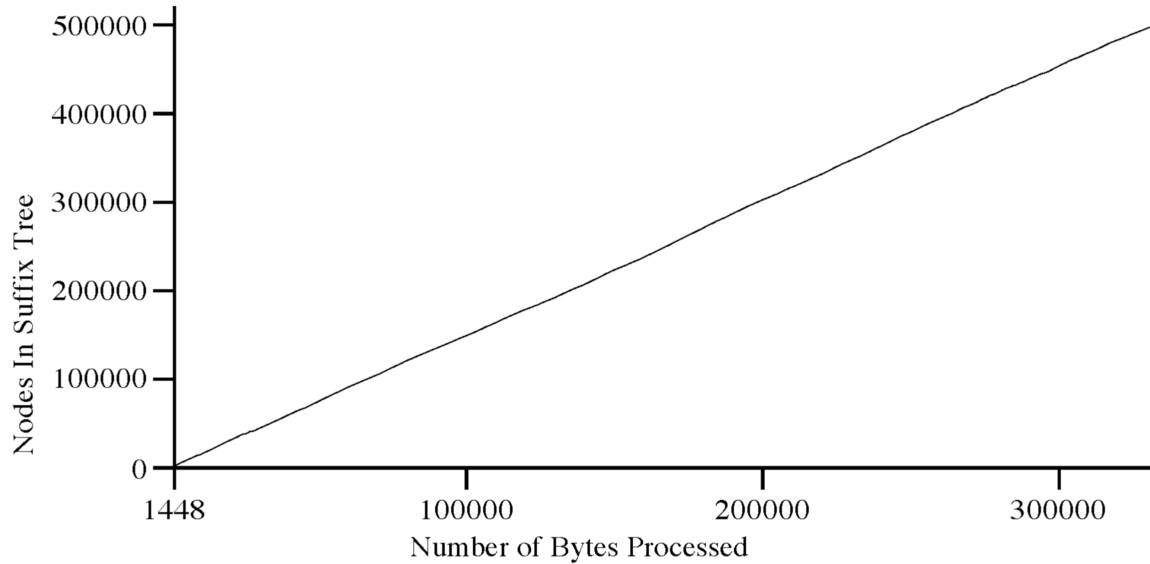

Figure 5: Suffix Tree Size When Processing Relatively Non-compressible Data

Figure 6 shows the number of suffix tree nodes (vertical axis) needed to represent the number of bytes from a web client (horizontal axis). Once again the graph represents 250 data points representing the number of bytes processed the number of nodes used after each data packet. There are two primary differences between this graph and the graph from Figure 5. First, the growth rate of the suffix tree is roughly half the rate per byte processed than it is for the server data. This is because the web client data has more repetition resulting in more "re-use" of links in the suffix tree. Second, and more noticeable, the line is not nearly as straight as it was in the previous figure. Once again, this is a side effect of the repetition in the data. When we are processing relatively similar data the graph tends to flatten out a bit, and when we processes relatively unique data the graph tends to spike up more.

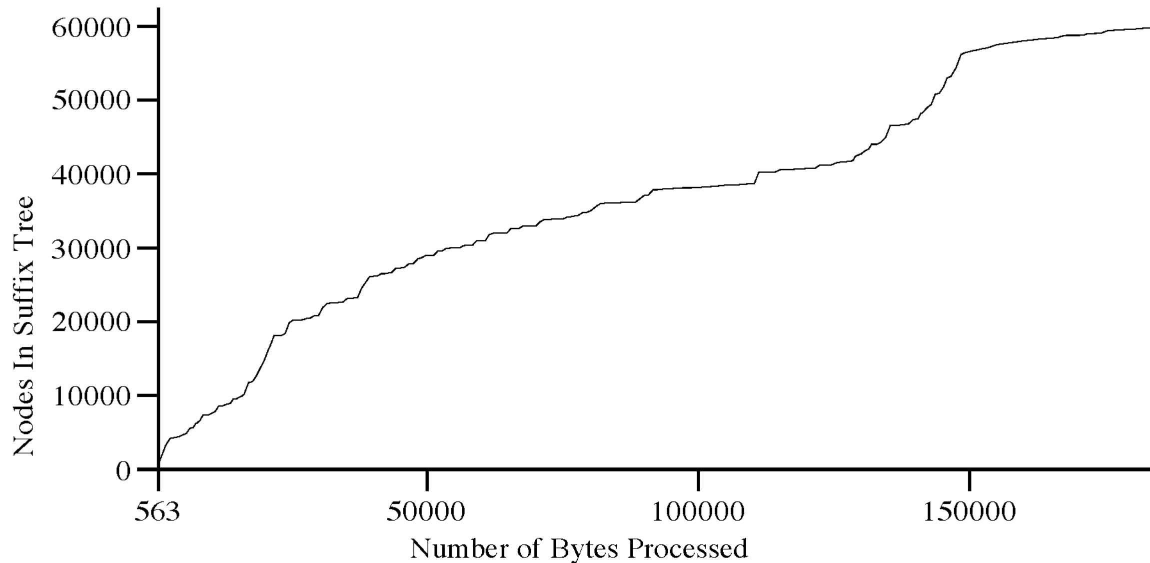

Figure 6: Suffix Tree Size When Processing Compressible Data

6 Conclusions

The automatic signature generation project will assist security analysts in finding signatures for an attack that have optimally low and predictable false positive rates. Furthermore, for fast moving attacks (e.g., worms) the proposed approach is fast enough to automatically generate signatures that can be deployed in content-based firewalls to slow or stop the spread of the worm before significant damage can be done.

The key technology to achieve these capabilities is the suffix tree, and we have implemented an in-memory version of the Ukkonen suffix tree algorithm. The Ukkonen algorithm can build a suffix tree from test data in linear time, and the suffix tree's size is also linear in size with respect to the original data. We have developed a second-generation in-memory solution that maps cleanly to database schemas. By using a database approach our solution should support terabytes of test data.

7 References

[Ukko 95] E. Ukkonen. "On-line construction of suffix trees", *Algorithmica*, 14:249-60, 1995.