The Interdisciplinary Center, Herzlia

Efi Arazi School of Computer Science M.Sc. program

Efficient Automated Signatures Extraction Implementation

M.Sc. Final Project

By Golan Parashi

Under supervision of Prof. Anat Bremler-Barr (IDC)

March 2016

Abstract

This work describes a code implementation of a tool for zero day attack signature extraction based on the work "Automated signature extraction for high volume attacks"[1]. The code implementation offers a more correct and faster implementation than the code used to initially verify the work in [1] - the new code implementation offers an increase in throughput and offer more correct signatures.

Given two large sets of messages, *P* of messages captured in the network at peacetime (i.e., mostly legitimate traffic) and *A* captured during attack time (i.e., contains many attack messages), the tool extracts a set *S* of strings, that are frequently found in *A* and not in *P*. Therefore, a message containing one of the strings from *S* is likely to be an attack message. This tool finds popular strings of variable length in a set of messages, using a modified implementation [4] of the Heavy Hitters (Finding Frequent Items) algorithm [3]. This implementation is used as a building block to extract the desired signatures.

Using the attack signatures found by the tool in conjunction with a network traffic-filtering device, a yet unknown attack could be automatically detected and stopped within minutes from attack start time.

The development focused on creating a fast implementation in order to achieve high throughput, which is very important when operating in large traffic networks environment. The development methodology included repeated inspection of code sections, by using CPU/Memory profilers and static code analysis tool. These tools helped finding issues in the code. Specifically, the CPU profiler helped finding code sections with high latency. Once an issue was found it was resolved. A performance evaluation was a major part of the development lifecycle.

The tool is offered as a command line utility and a website was created in order to make it accessible for testing.

Contents

Introduction 4

Background 5

Related Work 7

Algorithm Drawbacks 8

Implementation Improvements 10

New Implementation 14
- **Programming Environment** 14
- **Heavy Hitters (Finding Frequent Items) Algorithm** 14
- **Development Methodology** 15

New Code's Structure 17

Analyzing Traffic 18

Using the Tool 21
- **Commands** 21
- **Running The Tool** 22

Evaluation 24
- **Test Environment** 24
- **Traffic Captures** 25
- **Measurements** 26

Web Application 27

Summary and Conclusions 29

Bibliography 30

Acknowledgment 31

Chapter 1

Introduction

Signature extraction is used to mitigate various network attacks, such as Distributed Denial of Service (DDoS). Identifying signatures for unknown DDoS attacks is extremely difficult due to the seemingly legitimate content found in the packets that comprise the attack. Leading industry experts confirm that the signatures found in recent DDoS attacks are usually a bi-product of the attack tools that are used by the attackers. These tools often leave some footprint caused unintentionally by the program, such as a short string or some anomaly in the packet content structure. These subtle signatures are not identified by the current automated defense mechanisms, but rather by a manual process, which may take hours or days. Clearly, in order to stop such unknown attacks while they are occurring, such signatures must be extracted quickly, hence automatically.

Automatic signature extraction is very important to mitigate zero day attacks for which no known signatures exist. In general, security companies provide systems that offer several layers of defense against high-volume attacks. When all layers of defense fail, the attacked customer contacts the security company's support team to alert them and get their assistance in stopping the attack. Since the automated mechanisms were not able to identify or stop the attack, the attacked customer is in need of manual assistance. This manual assistance may include several stages, one of which is identifying attack signatures by a human expert. The attack mitigation process can therefore take hours or even days.

This tool automatically extracts zero day attack signatures based on the work in [1]. The tool takes as input two samples of traffic collected during peace and attack time. The peacetime traffic sample may be collected periodically and the attack traffic sample can be collected once an attack has been identified. There are systems for automatically identifying DDoS attacks such as Park et.al [2].

The tool is generic in that it can be easily adapted to solving other network problems with similar characteristics. This is because it does not make any assumption on traffic characteristics such as client behavior, address dispersion, URL statistics and so forth.

The development of this tool was focused on creating a fast implementation in order to achieve high throughput, which is very important for large traffic networks.

Chapter 2

Background

Heavy Hitters

The tool tries to find a set of frequent attack signatures, which is a variant of the classic Heavy Hitters problem. The Heavy Hitters problem is defined as follows: given input of *N* items, find those that occur most frequently where their frequency (count) greater then ϕ*N*. There are several algorithms for dealing with the classic Heavy Hitters problem; this tool uses the algorithm described by Metwally et.al [3].

The error rate ε of this algorithm is $\varepsilon = \frac{N}{n_v}$, where *N* is the number of items in the input and n_v is the number of the most frequent items we would like to get. This means that each counter in the algorithm's output is at most ε higher than the actual count of the value as it appeared in the input. The algorithm runs in O(*N*) time, makes only a single pass over the input and requires constant space.

The tool makes use of the algorithm [3] implementation described by Cormode et.al [4]. The Cormode implementation assumes integer items. Thus, the reference implementation was modified to hold variable-length buffers. After the modification, the algorithm still performs in O(*N*) time, and requires constant space - data structure space is constant up to buffer length during runtime.

Double Heavy Hitters (DHH) Algorithm

The base work [1] for this tool describes the signature extraction for peacetime traffic:

1. Using first Heavy Hitters component, HH_1, to find frequent k-grams (k-length strings) in a packet.

2. Using second Heavy Hitters component, HH_2, to find frequent variable-length strings in the entire input. The strings are built from consecutive frequent k-grams (see Figure 1) found by the HH_1 component in the former step.

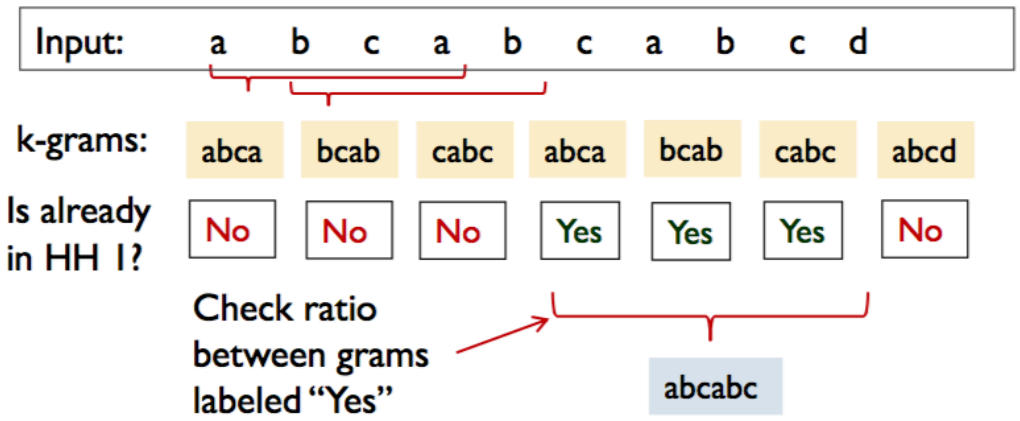

Figure 1. An example of the process of creating varying length strings from consecutive k-grams.

Triple Heavy Hitters (THH) Algorithm

The base work [1] for this tool describes the signature extraction for attack traffic:

1. Use *Double Heavy Hitters* (DHH) to extract frequent variable-length strings in the entire attack traffic (makes use of HH_1 and HH_2 components).

2. Use a third Heavy Hitters component, HH_3, to extract frequent sets of variable-length strings that appear in the same packet. These sets are built from the strings found by the DHH in the former step. In this step we are actually identifying common combinations (sets) of signatures that appear in the same packet.

Chapter 3

Related Work

General

Previous work for automated signature extraction has been used mostly as a tool for identifying computer malware such as worms and viruses. Therefore, the signature generation process of the previous works [5, 6, 7, 8, 9] done on malware identification was based on the use of traffic that is *known* to be malicious, as opposed to the work in [1] that deals with the scenario in which the suspicious traffic cannot be detected a-priori, but rather, the suspicious traffic contains some unique prevalent content which needs to be identified. Meaning, attack traffic is analyzed, parts of it may be malicious and others may be legitimate.

Most of the previous works were done for fixed length signatures. There are some works that do generate variable length signatures but with some pitfalls: one work [8] used space complexity linear to the input, searching for longest common substring using suffix trees, which is not scalable when dealing with large amount of data. Another work [7] used minimum and maximum length restriction for the signatures to avoid false positives, which means that the algorithm cannot yield short signatures.

To this date there are no known tools for automatically extracting variable length signatures from traffic that contains both legitimate and unknown malicious packets.

Existing Algorithm Implementation

The only implementation that existed for the algorithm in [1] is a tool that was created by Shir Landau Feibish, one of the authors of [1]. That implementation was used to verify the correctness of the work and since it was a proof of concept it was not tuned for performance.

Chapter 4

Algorithm Drawbacks

The algorithm for signature extraction described in [1] contains several drawbacks:

1. The algorithm merge *k-grams* to longer signature if $\frac{k_gram_count}{signature_string_counter} > r$ where: k_gram_count is the count of the current evaluated *k-gram,* $signature_string_counter$ is the count of the previous *k-gram* and r is a minimum limit set as a parameter to the algorithm.

   If the above condition is not met, it means that the *k-gram* does not appear frequent enough and it won't be merged with the previous *k-gram*, meaning, stop building current signature.

   However, this condition does not deal with the situation in which the *k-gram* appears much more frequently than the previous one – no upper limit – meaning that the ratio is very high so it easily greater then r. This leads to building incorrect longer signature since the *k-grams* get inappropriately merged.

   Example:
   k-gram length is 4, the input contains packets of host names on the web: www.site.com, app.bbb.com, etc. A very common prefix is ".com", which means that ".com" *k-gram* count is very high.

   Let's assume the code encounters the following host for few times: "s.aaa.com". "s.aaa.co" is the current signature string and when ".com" *k-gram* is examined and because its counter is very high, the ratio $\frac{k_gram_count}{signature_string_counter}$ is much higher than the r threshold, thus, ".com" *k-gram* is concatenated to the currently built signature string and we get false signature "s.aaa.com".

   To fix this issue the following range threshold check should be used:

$$\left(\frac{1}{r}\right) > \frac{k_gram_count}{signature_string_counter} > r$$

   This range check prevents building incorrect long signature in which the *k-gram* is much more frequent then the currently built signature string.

2. The algorithm accumulate signature that appear in the same packet into a signature set. The decision whether to add a signature to the set, is done in a similar way described in the former paragraph, and may cause in a larger then desired sets of signatures: $\frac{counter_2}{strings_counter} > r$. The solution for this condition is also to add an upper bound as follows: $\left(\frac{1}{r}\right) > \frac{counter_2}{strings_counter} > r$

3. In the pseudo code for *DoubleHeavyHitters*:
    a. In the main section of the function, line 6 should be changed from *"if string == empty then"* to "if s_{temp}== empty then".

4. In the pseudo code for *TripleHeavyHitters* (in a newer revision of the paper):
    a. The pseudo code for *InputToHH3* is missing a call to *HH3.Update(*S_{temp}*)* .
    b. A division by zero was identified while deciding whether a string should be added to the set of signatures by examining the ratio between two strings in the following way: $\frac{counter_2}{strings_counter} > r$. This is because $strings_counter$ is initialized with a zero value and not assigned a value until this code line evaluation.

Note: All the above drawbacks are resolved in the new implementation.

Chapter 5

Implementation Improvements

The only implementation that existed for the algorithm in [1] is a tool, which was created by *Shir Landau Feibish*, one of the authors of [1]. That implementation was used to verify the correctness of the work and was not efficient in several areas such as memory management (redundant memory allocation/de-allocation, unnecessary memory copy, inefficient string allocation) and data structure selection (inefficient lookups). Note that this was a proof of concept implementation (POC) and as such was not designed to be highly efficient.

Listed below are the key improvements between the POC code and the new code implementations:

More Accurate Results

1. Correct signatures
    - In POC code, when a *k-gram* ends the process of merging *k-grams* into longer signature, it is not used as the initial value for next signature.

      Example:
      k-gram length is 3 and stream is as follow: "attack**S**ite"

      "**S**it" k-gram ends signature merging: "attackSi"
      "**S**it" k-gram is skipped
      Next signature starts with "ite" *k-gram*

    - New code does not skip the ending *k*-gram and thus produce more correct signatures. In the above example, the next signature will start with "Sit" *k-gram,* thus, "Site" can be a signature.

2. Longer signatures
    - POC code stop building longer signatures if a *k-gram* was already seen in the same packet.

      Example:
      k-gram length is 3 and stream is as follow: **att**ackSITE**att**ack

      If the conditions are correct, the longest signature that would be extracted is: "**attack**SITEat" because of the second appearance of **"att"** *k-gram*, which was already seen in the packet.

    - New code continues to build signature even if a *k-gram* was already seen in the same packet. Note the already seen *k-gram's* count is not updated because of the

additional appearance in the packet. In the above example, if the conditions are correct, the longest signature that would be extracted is: "**att**ackSITE**att**ack"

Faster

3. Throughput
    - POC code: 32Mbs.
    - New code: on some inputs, more then 1.1Gbs. See "Evaluation" section.

4. Data structure lookups
    - POC code performs inefficient data structure lookups by extensively using *std::map<>.find() - O(log n)* complexity.
    - New code uses more efficient data structures with better performance. An example of such data structure is *std::unordered_map<>.find()* - average *O(1)* complexity.

5. White-list signature filtering (filtering signatures before updating HH_2 component during attack analysis):
    - POC code **iterates** over all the *white-list* signatures and performs a **substring search** on each one of them for the candidate signature – O(n).
    - New code creates a hash table with all *white-list* signatures strings and their substrings and performs a hash table lookup to check existence – O(1).

6. Updating *k-gram* in HH (Heavy Hitters) component only if not previously added for the current packet (prevents over counting of an item)
    - POC uses *std::map<>* data-structure to track if the an item was already added into the HH component for the current packet. This mechanism is used before updating HH_1 and HH_2 components. Map operations run in *O(log n)*.
    - New code uses an additional field in the HH component to record the **last packet** the item was counted for. Then, when the item is updated in the HH component, an indication if the item was already seen in this packet is returned. The code takes advantage on the fast an efficient data-structure of the HH component (incorporates hash table) in which the update operation perform in *O(1)*.

7. Hardware Acceleration Usage
    - POC code does not use any hardware acceleration.
    - New code use hardware acceleration and uses special CPU instructions to compute CRC32 codes. CRC32 is used in hash functions throughout the code. Benchmarks observe performance that is 2-3 times faster than the fast SlicingBy8 software implementation [11].

8. Memory Allocations/De-allocations/Copy
    - POC code contains code that performs unnecessary string allocations, string copies and string de-allocations. As an example, let's review the usage of the *LCU_Update* function, which is the most called function that is used to populate Heavy Hitters data structures. This function (and its callees) is responsible for about 84% (*Figure 2*) of the execution time. The POC implementation has the following function signature:

*int LCU_Update(LCU_type *lcu,* ***std::string newitem****, bool isShift, int index)*

The signature defines *newitem* as a string class parameter, which is copied by value and not by reference. This means that for each call to the function, a copy of the input string is created, memory is allocated for it, the string is copied, and when the function exits the memory is de-allocated. This creates an inherent performance hit of code execution since it performs unnecessary system calls, which make a switch between user-mode and kernel-mode.

- New code passes object by pointer/reference and not by value, thus, eliminates unnecessary calls to object construction and destructions and all its side effects. As solution for the above function signature is: ***unsigned char * newitem*.**

Additional Features

9. Support for non-text protocols
    - POC code works only with strings. This limits the tool to only work with text protocols
    - New code works with buffers and does not assume anything about the content type. This allows the tool to process both text and non-text protocols.

10. Streamlined Runtime Environment
    - POC code is user interactive and requires the user to input information as the tool is running. As such, it cannot be used as a component of a larger system.
    - New code was designed to run in a streamlined environment and running parameters are passed upon startup. This makes the tool usable as a component in a larger system. The web application uses the tool as one of its components.

Better Code

11. Modularity & Maintainability
    - POC code is monolithic, not well structured, contains lots of code snippets used for testing which are embedded in the code and make it really difficult to read and maintain.
    - New code is modular and well structured, stripped to the minimum and structured in a very readable manner that is easy to follow and maintain.

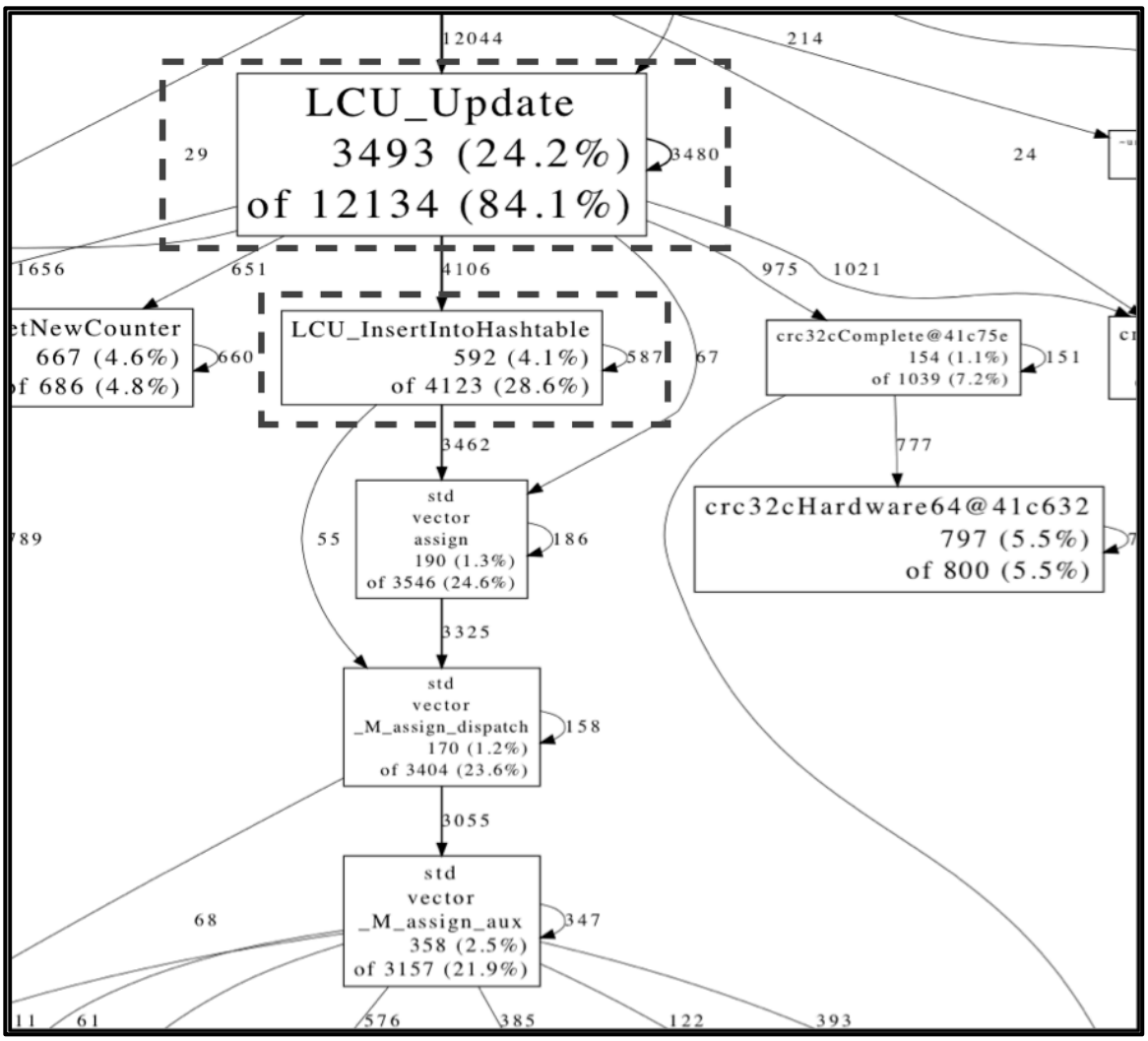

Figure 2. gperftools [10] call-graph output showing: 84.1% execution time for *LCU_Update and its* callees; 28.6% execution time for *LCU_InsertIntoHashtable.*

Chapter 6

New Implementation

This section describes the following topics in the new implementation:

1. *Programming Environment* – OS and programming language selection.
2. *Finding Frequent Items Algorithm* – Algorithm selection and modification.
3. *Development Methodology* – Development focusing on high performance and stability.

Programming Environment

1. Development and testing were done on Linux-based platforms (Debian and Mac OS X).
2. C++ was chosen as the programming language for this project. C++ is a well-proven language that allows fine control over program execution and memory management. It is an *Object Oriented Language* with, thus, enjoying the benefits of this paradigm, especially code reuse via inheritance. It has a rich built-in standard library for common operations and data structures. It has very good performance over VM languages and is richer then the C language.
3. It was decided not to use C++ features, such as polymorphism, that may slow down code execution. C++ *class templates* are used instead.
4. The code makes extensive use of *inline functions* to avoid function call overhead.

Heavy Hitters (Finding Frequent Items) Algorithm

The tool makes use of the Heavy Hitters algorithm [3] implementation described by Cormode et.al [4].

In order to support variable length signatures, the reference implementation was modified to use (variable-length) buffers instead of integer numbers. After the modification, the algorithm still performs in O(*N*) time, and requires constant space - data structure space is constant up to buffer length during runtime.

The Cormode implementation was chosen mainly for the following reasons:

1. Efficiently pre-allocates the necessary data structure in memory at initialization time. While using the data structure, no new memory is allocated. This is highly beneficial since memory allocation and de-allocation are costly operations due to the system calls that require switch between user-mode and kernel-mode.

2. Uses a data structure (combination of a hash table with a list of doubly linked lists of items) that allows efficient operations such as item lookup, update, insert, and find item with lowest count in O(1).

Overall structure is a doubly linked list of groups (of items), ordered by counter value. Each group represents a collection of items with equal counters. Each group has a list of items (doubly linked list), and a "difference" property, which is the value between group's counter and the counters in the previous group, or, for the first group, the value itself. For quick lookup, an open hash structure is used to hold all monitored items. Each item has a pointer to its object in linked-list structure.

Items in a group have a lower count than the items in the following groups, thus, when a new item need to be inserted to the data structure, any item in the first group can be quickly identified and its memory re-used for the new item.

Development Methodology

The development methodology included the following steps:

1. Write/Fix code.
2. Run static code analysis tool.
3. Run CPU and Memory profilers.
4. Analyze detected issues (code issues, slow code, memory issues).
5. Return to step 1.

After a new feature implementation or a code change or, several tools were run on the code and during runtime. These tools include static code analysis (*cppcheck* [12]), CPU profiler (*gperftools* [10]) and memory profiler (*Valgrind* [13]). These tools help detect issues such as performance and memory management issues. After an issue was detected, a solution was devised and a code fix was employed. These steps then repeated.

These steps ensure a high quality code and help achieving one of the most important goals of this tool – high performance. For example, *gperftools* indicated that the use of *LCU_InsertIntoHashtable() function* is very time consuming and responsible for about 28.6% of the code running time (See Figure 1). Further examination showed that the hotspot is in the call to *std::vector::assign() function* from the C++ standard library, which consumed 24.6% of code running time. The *std::vector* class was used for storing buffers in the Heavy Hitters data structure. As a result, a more efficient class for storing buffers was developed (*SimpleBuffer class*) which improved overall performance (Figure

3). After using the new data structure *LCU_InsertIntoHashtable() function* consumed only 5.3% of the code running time.

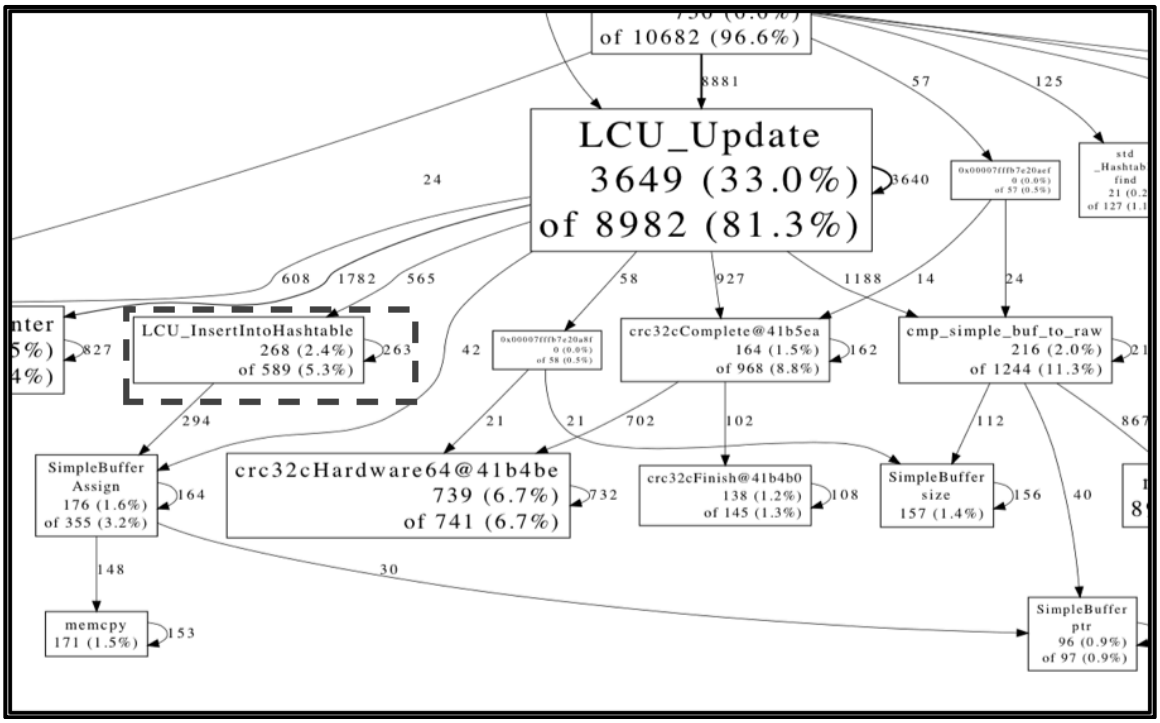

Figure 3. gperftools [10] call-graph output showing: 5.3% execution time for *LCU_InsertIntoHashtable.*

Chapter 7

New Code's Structure

The tool is implemented as a command line application, which enables execution of several commands such as analyzing peace traffic, analyzing attack traffic and printing signatures. Each command has its own set of parameters (for more information see "Using The Tool" section).

The tool is built from the following key modules:

1. *main* – The entry point of the tool that is responsible for getting input from the command line and executing the correct command using the supplied parameters.

2. *cmd_line_options* – Responsible for command line parsing and parameter validation. The module provides default values for optional parameters and enforces mandatory parameters.

3. *dhh* – Implements the DHH and THH algorithms for peacetime and attack traffic.

4. *lcu_buffer* – Implements the modified Heavy Hitters algorithm implementation that supports variable-length buffers.

5. *pcap_pckt_iterator* – Implements a packet iterator over a PCAP file.

6. *line_iterator* – Implements a packet iterator over a text lines file.

7. *simple_buffer* – A simple buffer to replace *std::string* for holding buffers in Heavy Hitters implementation (*lcu_buffer*) to speed up assignment to the data structure.

8. *signatures* – A data structure for holding signature information.

9. *crc32* – A hardware implementation for calculating CRC32 codes. These codes are extensively used in hash functions for hash tables used in this tool. If no hardware support is available a software implementation is provided.

10. *utils* – Utility functions for performing various common tasks such as counting signatures in traffic file, storing signatures into files, printing signature files and more.

Chapter 8

Analyzing Traffic

Analyzing Peacetime Traffic

The tool gets a sample of peacetime traffic and uses the DHH algorithm to find a set of frequent signatures. These signatures are then categorized into two signature lists, *whitelist* and *maybe-whitelist* as follows:

- A peace signature is in the *whitelist* if its frequency is higher than *peace-high* threshold. Signatures in this list are not considered an attack signature.

- A peace signature is in the *maybe-whitelist* if its frequency is above *peace-low* threshold and lower then *peace-high*.

The lists*, whitelist* and *maybe-whitelist*, are used for filtering attack signatures that found when analyzing attack traffic.

Analyzing Attack Traffic

The tool gets a sample of attack traffic and uses the DHH algorithm to find a set of frequent signatures (Figure 4). In this case, the DHH algorithm does not insert strings into its HH_2 component if the string is equal or a substring of any signature in the *whitelist* (to reduce false positives). The output of the DHH is a list of attack signature candidates.

The list of attack signatures candidates is then inspected and filtered to build a list of *black signatures*, which are the attack traffic signatures, as follows:

- If a candidate signature frequency in attack traffic is less than the *attack-high* threshold it is filtered out.

- If a candidate signature is equal to or a substring of a signature in the *maybe-whitelist* and its frequency in attack traffic is less than the sum of a *delta* and the frequency (of the signature in the *maybe-whitelist)* in peacetime traffic then the candidate signature is filtered out.

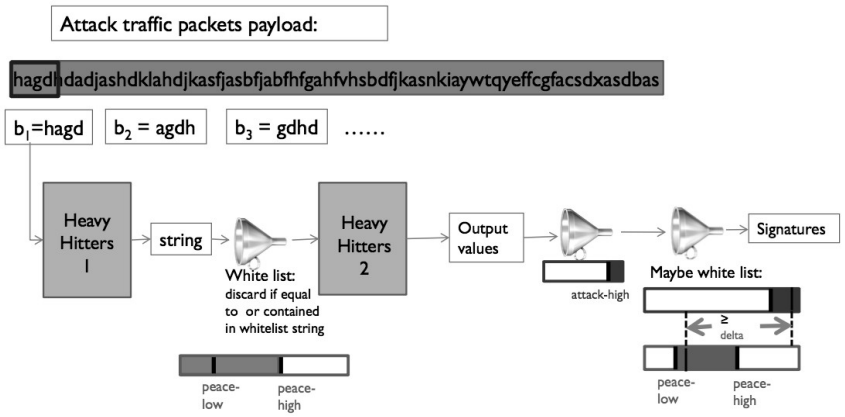

Figure 4. Attack traffic analysis (DHH).

In certain circumstances, it is very important to minimize the amount of the attack signatures. For example, when they are used in network components to filter out traffic. Keeping the number to the minimum means that the filtering process will be faster and less resource consuming (CPU and memory). Also, fewer signatures may contribute for less false-positive ratings of the signatures.

To further minimize the number of attack signatures (Figure 5), the tool finds which signatures commonly appear together in the same packet and puts them into signature sets. These signature sets are then used to calculate individual *black signature* coverage rate of all the packets. This means that a signature with a higher coverage rate filters more packets.

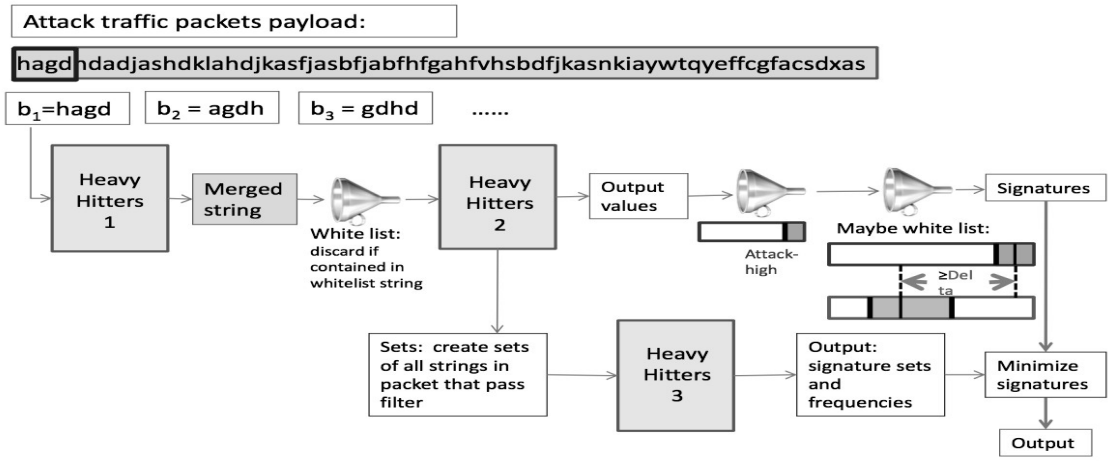

Figure 5. Attack traffic analysis with signature minimization (THH).

Example of minimizing signatures using the signature sets is depicted in Figure 6. In this example, there are six different types of attack packets (six signature sets). There are also four *black signatures* extracted by the tool: "*bad", "guy", "really"* and "*mean"*. Since either the signature *"bad"* or *"guy"* appear in all of the different attack packets, they alone can be used to stop the attack, hence the number of signatures can be minimized without creating false negatives. *"bad"* coverage rate is 71% of the six different packet types and *"guy"* coverage rate is 65%. The minimization process done by a greedy algorithm listed in Figure 7.

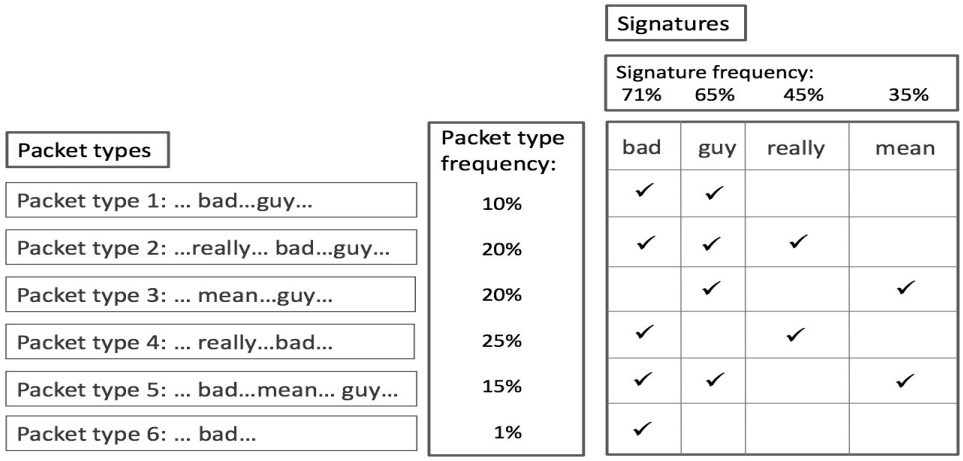

Figure 6. Minimizing attack signatures using signature sets [14].

Procedure MinimizeSignatures

Data: list L_{sigs} of n_{HH_2} signatures, list L_{sets} of n_{HH_3} sets of signatures

Result: the final list of signatures

// Initialize the *cover_rate* of all signatures to be zero

list $L_{final} = empty$;

for $i = 1 \rightarrow n_{HH_2}$ **do** $L_{sigs}[i].cover_rate = 0$;

Sort L_{sigs} by frequency;

i = 0;

while $i < n_{HH_2}$ *and* L_{sets} *not empty* **do**

  for $j = 0 \rightarrow L_{sets}.size()$ **do**

    if $L_{sets}[j]$ *contains* $L_{sigs}[i]$ **then**

      $L_{sigs}[i].cover_rate+ = L_{sets}[j].frequency$;

      remove set $L_{sets}[j]$ from L_{sets};

for $i = 0 \rightarrow n_{HH_2}$ **do**

  if $L_{sigs}[i].cover_rate > 0$ **then** $L_{final}.insert(L_{sigs}[i])$;

return L_{final};

Figure 7. *MinimizingSignatures* algorithm [14].

Chapter 9

Using the Tool

The following describes the command line interface of the tool. It lists and explains the various commands and their options and explains the output of the tool.

Commands

1. *analyze-peace:* Analyze peacetime traffic.
2. *analyze-attack:* Analyze attack traffic.
3. *print-signatures* Print a signature file. Used for testing.
4. *count-signatures* Count the signatures in a traffic file. Used for testing.
5. *pcap2text* Convert PCAP file to text file. Used for testing.

To see all available commands and their options, open a terminal window and type: ./auto-sig-gen. This will print out the following:

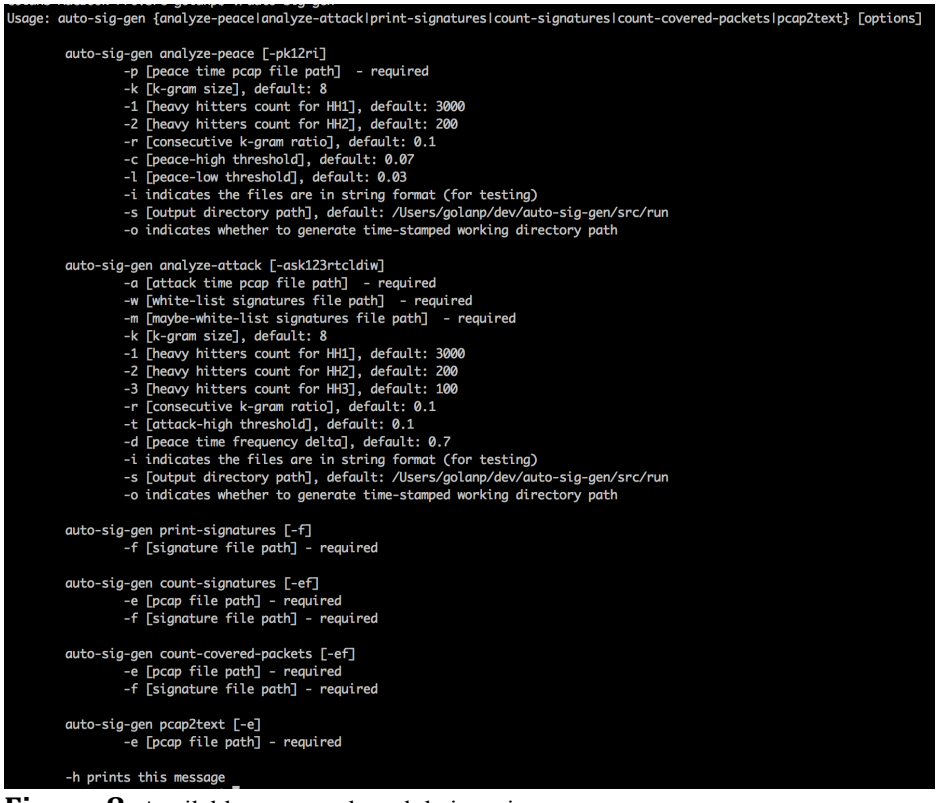

Figure 8. Available commands and their options.

Options Description

-p Path to PCAP file containing peace traffic.
-a Path to PCAP file containing attack traffic.
-w Path to a file containing *white-list* signatures.
-m Path to a file containing *maybe-white-list* signatures.
-k *k-gram* size in bytes. This indicates the minimum signature length and is used in DHH/THH algorithms.
-r Ratio between the current string and the next *k-gram*. Used to determine if the *k-gram* should be concatenated to the string to compose a longer string.
-1 Number of the most frequent *k-grams* stored in HH_1 component.
-2 Number of the most frequent variable-length strings stored in HH_2 component.
-3 Number of the most frequent string sets to store in HH_3 component.
-c *peace-high* threshold used to categorize whitelist and *maybe-whitelist* signatures.
-l *peace-low* threshold used to categorize *maybe-whitelist* signatures.
-t *attack-high* threshold used to filter candidates for attack signatures.
-d *delta* used to filter candidates for attack signatures that appear in *maybe-whitelist* signatures.
-e Path to a PCAP file used in various utility commands.
-f Path to a signature file used in various utility commands.
-i Flag indicating that input files are Line-Feed delimited files and not PCAP files.
-o Flag indicating to set newly generated time-stamped working directory.
-s Set working directory to the specified path.

Running The Tool

Each run creates various output files in the "*/run"* output directory under the current working directory (a different directory path or an auto generated dated directory name can be used). The output directory may include signature files (binary and CSV formats), log files and more.

Analyzing Peacetime Traffic Example

```
Golans-MacBook-Pro:src golanp$ ./auto-sig-gen analyze-peace -p /Volumes/auto-sig-ram-disk/pizza_peace_request.pcap
run_options = {
        "kgram_size": 8,
        "n1": 3000,
        "n2": 200,
        "n3": 100,
        "r": 0.1,
        "attack_high_threshold": 0.1,
        "peace_high_threshold": 0.07,
        "peace_low_threshold": 0.03,
        "peace_time_delta": 0.7,
        "peace_time_file_path": "/Volumes/auto-sig-ram-disk/pizza_peace_request.pcap",
        "attack_time_file_path": "",
        "white-list_signatures_file_path": "",
        "maybe-white-list_signatures_file_path": "",
        "work_dir": "/Users/golanp/dev/auto-sig-gen/src/run",
        "use_pcaps": true
}
** peace time traffic analysis - duration: 77565 usec, throughput: 204.599 Mbps, packets: 2347
** counted signatures in source file  - completed in: 335400 usec
** stored 200 peace signatures in: /Users/golanp/dev/auto-sig-gen/src/run/peace_signatures
** stored 200 peace signatures in: /Users/golanp/dev/auto-sig-gen/src/run/peace_signatures.csv
** stored 57 white signatures in: /Users/golanp/dev/auto-sig-gen/src/run/white_list_signatures
** stored 20 maybe white signatures in: /Users/golanp/dev/auto-sig-gen/src/run/maybe_white_list_signatures
```

Figure 9. Running *analyze-peace* command.

After running the command the output directory contains several files:

- *peace_signatures* – peace signature.
- *peace_signatures.csv* – peace signature in CSV format.
- *white_list_signatures* – *whitelist* signatures.
- *maybe_white_list_signatures* – *maybe whitelist* signatures.
- *run.log* – last run output.

Analyzing Attack Traffic Example

```
Golans-MacBook-Pro:src golanp$ ./auto-sig-gen analyze-attack -t 0.10 -a /Volumes/auto-sig-ram-disk/pizza_attack_request.pcap -p /Volumes/auto-sig-ram-disk/pizza_peace_request.pcap -w
white_list_signatures -m maybe_white_list_signatures
run_options = {
        "kgram_size": 8,
        "n1": 3000,
        "n2": 200,
        "n3": 100,
        "r": 0.1,
        "attack_high_threshold": 0.1,
        "peace_high_threshold": 0.07,
        "peace_low_threshold": 0.03,
        "peace_time_delta": 0.7,
        "peace_time_file_path": "/Volumes/auto-sig-ram-disk/pizza_peace_request.pcap",
        "attack_time_file_path": "/Volumes/auto-sig-ram-disk/pizza_attack_request.pcap",
        "white-list_signatures_file_path": "white_list_signatures",
        "maybe-white-list_signatures_file_path": "maybe_white_list_signatures",
        "work_dir": "/Users/golanp/dev/auto-sig-gen/src/run",
        "use_pcaps": true
}
** attack time traffic analysis - duration: 12577 usec, throughput: 141.265 Mbps, packets: 407
** counting candidate attack signatures in attack file - completed in: 31378 usec
** stored 200 candidate attack signatures in: /Users/golanp/dev/auto-sig-gen/src/run/candidate_attack_signatures

DEBUG: peace_time_delta_filter_count=0, attack_high_threshold_filter_count=190
** extracting black signatures - completed in: 439 usec
** stored 10 black signatures in: /Users/golanp/dev/auto-sig-gen/src/run/black_signatures

DEBUG: black_in_peace_threshold=164.29
** filtering black signatures according peace source file - completed in: 17826 usec
** stored 10 filtered black signatures in: /Users/golanp/dev/auto-sig-gen/src/run/black_signatures_filtered

** minimize attack signatures by sets - completed in: 116 usec
** stored 5 minimized black signatures (after HH3) in: /Users/golanp/dev/auto-sig-gen/src/run/black_signatures_minimized

** stored 100 attack signature sets in: /Users/golanp/dev/auto-sig-gen/src/run/attack_signatures_sets
```

Figure 10. Running *analyze-attack* command.

After running the command the output directory contains several files:

- *candidate_attack_signatures*– attack signature candidates output from DHH.
- *attack_signatures_sets* – attack signature sets output from THH.
- *black_signatures* – attack signatures after threshold and *maybe whitlist* filtering .
- *black_signatures_minimized* – minimized black signatures according to their coverage rate in the signatures sets.
- *black_signatures_minimized.csv* – same as *black_signatures_m*inimized but in CSV format.
- *run.log* – last run output.

Chapter 10

Evaluation

This chapter reviews the tests used to evaluate the performance of the new implementation. It includes a description of the environment where the tests run, the traffic used for analysis and the performance results.

Test Environment

Hardware:

Processor Name	Inter Core i7
Architecture	x86_64
Processor op-mode	64-bit
Processor Speed	2.7 GHz
Number of Processors	1
Number of Cores	4
L2 cache	256K
L3 cache	6MB

Table 1. Hardware specifications.

Compiler:

Target	x86_64-apple-darwin14.5.0
Thread model	posix
gcc version	Apple LLVM version 7.0.2 (clang-700.1.81)

Table 2. Compiler specifications.

Traffic Captures

The evaluation included the following traffic capture sets:

1. Real peacetime and attack traffic captured from the same machine during normal operation time and in real attack time. A *path traversal* attack was conducted on a website that finds and recommends Italian businesses. The attack extracted data of various pizzeria restaurants.

2. Real peacetime capture from a server serving static content including on the fly image resizing captured during normal operation time. This is a large capture of 1GB. No attack in this case.

3. Real peacetime and synthetic attack captures. Peacetime traffic was captured during normal operation. For the attack traffic, during normal traffic time, a tool was used to simulate high-frequency attack (hundreds of concurrent connections). A *URL tampering* attack was directed to a real-time image resizing service and forced the service to bypass cache and actually resize the images.

4. Synthetic peacetime and attack captures. Peacetime traffic contained HTTP requests to nine different URLs. Attack traffic included 30% peace packets, 30% attack packets and 40% of second attack packets.

Tests Input Parameters

All test runs used the following input parameters:

k-gram size	8 bytes
k-gram ratio	0.1
HH1 component size	3000
HH2 component size	200
HH3 component size	100
Attack-High threshold	0.10
Peace-High threshold	0.07
Peace-Low threshold	0.03
Peace-Time delta	0.70

Table 3. Tool input parameters for the tests.

Measurements

The following table lists the measurements taken while running the tool on the above traffic captures. For each run, packet-processing measurements, latency and throughput were recorded since they are the key indicators for the tool's performance. Each capture file was analyzed 5 times and the average results are shown below:

Capture Set #	**Type**	**Number of packets**	**Capture File Size**	**Throughput (Average)**	**Latency (Average)**
1	real-peace	2347	2MB (2,080,078 bytes)	202 Mbps	78,602 usec
	real-attack	407	227kb (232,875 bytes)	144 Mbps	12,375 usec
2	real-peace	483,560	1024MB	206 Mbps	39,800,204 usec
3	real-peace	1306	1.1MB (1186751 bytes)	172 Mbps	52,805 usec
	synthetic-attack	1090	600KB (614041 bytes)	163Mbps	28,700 usec
4	synthetic-peace	1000	80KB (81807 bytes)	1227Mbps	508 usec
	synthetic-attack	10,000	946KB (968308 bytes)	358Mbps	20,640 usec

Table 4. Summary of performance evaluation on the traffic captures.

Chapter 11

Web Application

To make the system available publicly, a web application was developed. The web application was created for testing/evaluation purposes only; it was not created for production, high scale traffic rather to small size captures.

A real web service would have been designed and implemented entirely differently and would involve queues and an auto-scaled machine cluster to handle a high load of analysis requests. Each analysis request would have been recorded in a database and status and results could have been retrieved at any time. The system would handle both real-time and offline traffic captures.

The web application requires that the user upload two PCAP files for analysis, one for peacetime traffic and one for attack traffic. The user can adjust various parameters before submitting the files for analysis.

The output of the analysis is then displayed to the user and includes *black signatures* list, and *minimized black signatures* list (according to signature set coverage) and top peacetime signatures.

The website driving the web application is available in the following address:
http://golanp.wix.com/autosiggen

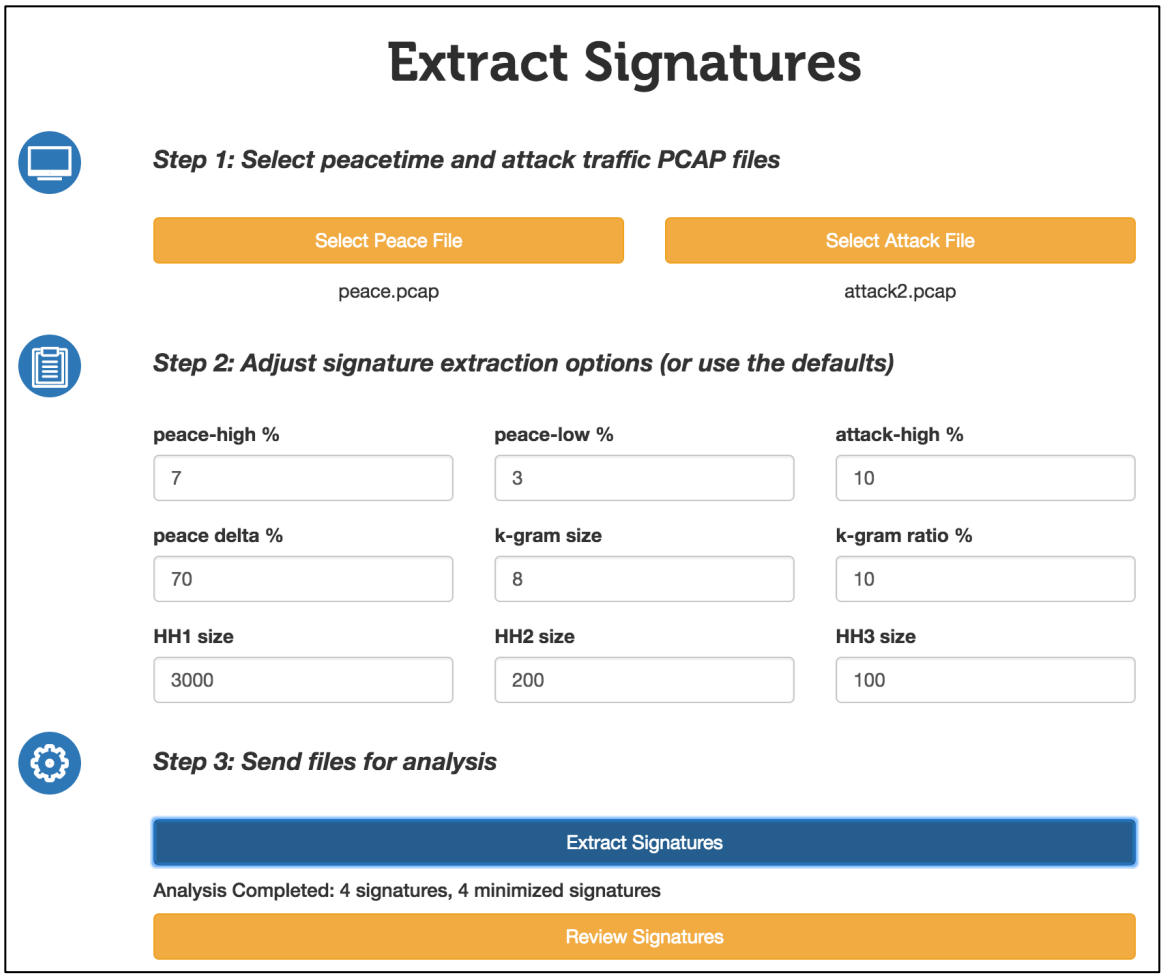

Figure 11. Screenshot of the website for driving the tool.

Chapter 12

Summary and Conclusions

This paper presented a tool for extracting zero-day attack signatures using the DDH/THH algorithm described in [1]. A web application was created in order to make this tool publicly available for testing/evaluation purposes.

The development of this tool was focused on creating a fast implementation in order to achieve high throughput, which is very important when operating in large traffic networks environment.

The tool can be easily adapted to solving other network problems with similar characteristics since it does not make any assumption about traffic characteristics such as client behavior, address dispersion, URL statistics and so forth.

Future work should be considered in two different paths:

a. Take advantage of multiple processors/cores and analyze traffic in parallel.

b. Develop a comprehensive web service for real time traffic analysis in addition to offline analysis.

Bibliography

[1] Afek, Yehuda, Anat Bremler-Barr, and Shir Landau Feibish. Automated signature extraction for high volume attacks. Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems. IEEE Press, 2013.

[2] H. Park, P. Li, D. Gao, H. Lee, and R. H. Deng. Distinguishing between FE and DDOS Using Randomness Check. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, ISC, volume 5222 of Lecture Notes in Computer Science, pages 131–145. Springer, 2008.

[3] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In T. Eiter and L. Libkin, editors, ICDT, volume 3363 of Lecture Notes in Computer Science, pages 398–412. Springer, 2005.

[4] Graham Cormode , Marios Hadjieleftheriou. Finding Frequent Items in Data Stream. 2008

[5] K. Griffin, S. Schneider, X. Hu, and T. cker Chiueh. Automatic generation of string signatures for malware detection. In E. Kirda, S. Jha, and D. Balzarotti, editors, RAID, volume 5758 of Lecture Notes in Computer Science, pages 101–120. Springer, 2009.

[6] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In 4th International Virus Bulletin Conference, Sept. 1994.

[7] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In USENIX Security Symposium, pages 271–286. USENIX, 2004.

[8] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. Computer Communication Review, 34(1):51–56, 2004.

[9] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In OSDI, pages 45–60. USENIX Association, 2004.

[10] gperftools. https://code.google.com/p/gperftools

[11] Peter Kankowski (2011). Benchmarking CRC32 and PopCnt instructions. http://www.strchr.com/crc32_popcnt.

[12] cppcheck. http://cppcheck.sourceforge.net

[13] Valgrind. http://valgrind.org

[14] Shir Landau Feibish. Automated signature extraction for high volume attacks (HH3). Revised - Minimizing signature using “Signature-Sets”.

Acknowledgment

This research was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no 259085.

תקציר

עבודה זו מתארת יישום קוד של כלי להפקת חתימות בזמן התקפות ״יום אפס״ (התקפות לא ידועות) המבוסס על העבודה "Automated signature extraction for high volume attacks" [1] . הכלי מציע מימוש נכון ויעיל יותר מהקוד הראשוני שנכתב על מנת לוודא את העבודה שב [1]. הקוד החדש מפיק חתימות נכונות יותר ומביא שיפור בתפוקה.

בהינתן שתי קבוצות גדולות של הודעות, קבוצה P של הודעות אשר נתפסו בזמן רגיל (בעיקר תעבורה לגיטימית) וקבוצה A של הודעות שנתפסו בזמן התקפה (מכילות הרבה הודעות התקפה), הכלי מייצר קבוצת מחרוזות S, אשר נמצאו בתדירות גבוהה ב A אך לא ב P. כלומר, הודעות המכילות אחד מהמחרוזות שב S עשויה להיות הודעת התקפה בסבירות גבוהה. כלי זה מוצא מחרוזות פופולריות באורך משתנה בקבוצה של הודעות, על ידי שימוש בגירסא מותאמת של האלגוריתם "Heavy Hitters״. אלגוריתם זה משמש כאבן בניין למציאת החתימות המבוקשות.

באמצעות שילוב חתימות ההתקפה שנמצאו על ידי הכלי עם התקני רשת לסינון תעבורה, התקפות חדשות שלא היו מוכרות יכולות להיות מזוהות בצורה אוטומטית ולהיחסם בפרק זמן קצר מתחילת ההתקפה.

הפיתוח התמקד ביצירת יישום מהיר ויעיל על מנת להגיע לקצבי תפוקה גבוהים, דבר הכרחי כאשר פועלים בסביבת רשתות בעלות תעבורה גבוהה. מתודולגיית הפיתוח כללה בדיקה איטרטיבית של הקוד, על ידי שימוש בכלים חיצוניים, למציאת חלקי קוד ״איטיים״ ומציאת מימושים מהירים יותר. הערכת הביצועים הייתה חלק מרכזי במחזור הפיתוח.

הכלי מוצע בתצורה של יישום שורת פקודה (Command Line Interface). בנוסף, פותח גם אתר אינטרנט על מנת להנגיש את הכלי למטרות בדיקה והתרשמות כללית.

המרכז הבינתחומי הרצליה
בית ספר אפי ארזי למדעי המחשב
התוכנית לתואר שני (M.Sc)

מימוש יעיל להפקת חתימות אוטומטית

פרוייקט גמר

מאת

גולן פרשי

בהנחיית פרופסר ענת ברמלר בר

מרץ 2016