JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 537-569 (2017)
DOI:10.1688/JISE.2017.33.2.15

SigBox: Automatic Signature Generation Method for Fine-Grained Traffic Identification

KYU-SEOK SHIM, SUNG-HO YOON, SU-KANG LEE AND MYUNG-SUP KIM+
Department of Computer and Information Science
Korea University
Sejong, 30019 Korea
E-mail: {kusuk007; sungho_yoon; sukanglee; tmskim}@korea.ac.kr

The continual appearance of new applications and their frequent updates emphasize the need for automatic signature generation. Although several automatic methods have been proposed, there are still limitations to their adoption in a real network environment in terms of automation, robustness, and elaboration. To address this issue, we propose an automatic signature generation method, so called SigBox, for fine-grained traffic identification. Using a modified sequence pattern algorithm, this system extracts three types of signatures: content, packet, and flow signature. A flow signature, the final result of this system, consists of a series of packet signatures, and a packet signature consists of a series of content signatures. A content signature is defined as a distinguishable and unique substring of the packet payload. By using the modified sequence pattern algorithm, we can improve the system performance in terms of automation and robustness. In addition, the proposed method can generate an elaborated signature for fine-grained traffic identification by using flow-level features beyond those of the packet level. In order to verify the feasibility of our proposed system, we present the results of experiments based on ten popular applications according to three defined metrics: redundancy, coverage, and accuracy. In addition, we show the quality of the generated signatures as compared to those produced by existing methods.

Keywords: traffic identification, traffic classification, automatic signature generation, sequence pattern algorithm, Apriori algorithm

1. INTRODUCTION

As high-speed Internet services have become more widespread and Internet-based applications have become more diverse, network management has become an increasingly important and essential process in network operations. The primary objective of network management is to maximize the use of network resources and to protect network devices from any external or internal threats. In order to achieve this goal, network operators establish appropriate network policies and apply them to the target network in a timely fashion. To establish the appropriate network policies, traffic identification must be performed before the operator applies the policy to the target network, because the policies that block/adjust the target traffic are based on the identification results [1-5].

Traffic identification is defined as the act of ascertaining which application or service is contributing to the network traffic by using a signature, which is a distinguishable unique pattern representing a particular application's traffic. Based on the traffic signatures, network traffic is named according to its corresponding application. The signature

Received December 23, 2015; revised April 5, 2016; accepted June 1, 2016.
Communicated by Ying-Dar Lin.
* This research was supported by a Korea University Grant.
+ Corresponding author: tmskim@korea.ac.kr

generation phase is the most important first step in traffic identification, because the accuracy of the entire process depends on that signature. The identification results are also utilized in numerous areas of network management, including traffic engineering, network planning, QoS planning, and Service Level Agreement (SLA) management.

In general, signature inspectors, such as network administrators or security experts, manually analyze the traffic of the target application to extract the unique pattern of the target application in order to generate a traffic signature. However, this primitive method requires difficulties especially in protocol semantic analysis and deep packet inspection whereas processes are time-consuming. Therefore, the quality of the signature could vary depending on the proficiency of the inspectors. In addition, these manual and laborious processes face challenges, given the continual appearance of new applications and their frequent updates [6-8].

These circumstances have led to several automatic signature generation methods being proposed [6-17]. However, we question whether these methods are actually automatic methods or otherwise because they require additional pre-/post-processing necessitating the intervention of operators. The pre-processing includes ordering and grouping input traffic data and the post-processing includes selecting of the final signature from several candidates. To find the appropriate grouping, ordering, and threshold values, existing methods must perform a repetitive generation process in which the input order and various types of threshold values are changed. This limitation raises challenges for generating signatures automatically in a real network environment.

To overcome the limitations of the previous methods, we propose a new method for generating signatures automatically. This method does not need any pre-processing for calibrating the order of the input data, predefining of configurable threshold values, or understanding of the protocol sematic analysis. In our method, we use a modified sequence pattern algorithm [18, 19], which is widely used in genetic engineering [20], to find sequences that represent the target application. By virtue of the nature of this algorithm, we can extract all the possible sequences observed in the input hosts' traffic. The modified sequence pattern algorithm extracts candidate sequences and measures a candidate's support value, that is, the ratio of hosts having the sequence to the total hosts, by increasing the length of the sequence from length 1. Increasing the length of the sequence, it eliminates insufficient candidate sequences under a predefined support value 1, which means that only candidate sequences observed in all hosts' traffic are accepted, in the early stage. Therefore, this algorithm can find all the possible sequences observed in all hosts' traffic.

This system, SigBox, generates three types of signatures automatically: content, packet, and flow signatures. The goal of this system is to "extract the signatures automatically" that were used in traffic identification by methods such as DPI. The term "Automatically" as defined here, refers to automatic extraction of three types of signatures (content, packet, and flow signatures) from traffic data without human intervention when a user manually enters particular traffic data in the system. As the previous methods manually extracted signatures, some problems were discovered, so in this paper we are trying to solve such problems and improve the quality of extracted signatures. Also, differently from previous methods, instead of simply comparing two strings in finding a common string, all of the existing strings are compared at once without pre- or post-processing. Therefore, compared to previous methods the proposed method is both fast and

accurate in signature extraction.

The content and packet signatures are intermediate results, and the flow signature is the final result. The flow signature consists of a series of packet signatures, and the packet signature consists of a series of content signatures. The content signature is defined as a distinguishable and unique substring of the payload consisting of continuous characters or hexadecimal values. In the generation process of all types of signatures, we apply the same sequence pattern algorithm. The only difference between the three types of signatures is the definition of the item that comprises the sequence. When this method generates a content signature, the item is a character or hexadecimal value of the packet payload. The content signature is considered an item when generating a packet signature. Finally, we consider the packet signature to be an item when generating a flow signature. By using the modified sequence pattern algorithm, we can improve system performance in terms of automation and robustness. In addition, this system can generate elaborated signatures for fine-grained traffic identification by using flow-level features beyond the packet level.

The remainder of this paper is organized as follows. In Section 2, we explain the existing automatic signature generation methods and describe the sequence pattern algorithm. We propose SigBox, an automatic signature generation method using the sequence pattern algorithm, in Section 3. In Section 4, we describe the evaluation of the effectiveness of the proposed method by testing it on ten popular applications. Finally, Section 5 concludes this paper.

2. RELATED WORK

In this section, we introduce the existing automatic signature generation methods. We also provide the definition, history, and utilization of the original sequence pattern algorithm.

2.1 Existing Automatic Signature Generation Methods

A number of methods have been proposed to address the need for automatic signature generation. In this section, we categorize existing methods in terms of purpose and type of signature and evaluate some methods that have a similar approach to our proposed method under three metrics.

2.1.1 Categories of automatic signature generation

Existing automatic signature generation methods are categorized in terms of their purpose and signature type. From a purpose perspective, the methods are divided into two categories: worm and application signatures. Methods for generating worm signatures involve binary classification to distinguish between normal and abnormal traffic [9-13]. Autograph [10] is a method that generates a worm signature by selecting TCP suspicious flows. First, it reassembles the packet payload, making it a single contiguous block according to a content-based payload partitioning technique. After the payload-partitioning phase, Autograph measures the frequency of the substrings and proposes the

most frequent ones as candidate signatures. Finally, Autograph discards unsatisfactory candidates using a predefined blacklist. In contrast, methods for generating application signatures are multi-classification methods to identify traffic according to each application [4, 6, 7, 14-16]. Choi *et al.* [16] proposed an automatic signature generation method that can identify the mobile application. They divided the traffic into HTTP and non-HTTP. In the case of HTTP traffic, their method generates signatures using the Host and User-agent fields of the traffic, while in the case of non-HTTP traffic, it applies the longest common subsequence (LCS) algorithm to extract a common substring as a signature.

Another means of categorizing existing automatic signature generation methods is to divide them by signature type. There are model-based and string-based signatures. The model-based signature is usually utilized in machine learning methods [15, 17]. ACAS [17] applies three machine learning algorithms to automatically generate a signature, in the form of a model, for a range of applications. This method uses the first N bytes of the payload for training based on several machine-learning algorithms. The model-based signature generated by the method is used for classifying application traffic. In contrast, the string-based signature is used to describe the pattern of the packet payload [6, 7, 9-14]. Shingh *et al.* [9] proposed a method for generating a string-based signatures using overlapping fixed-length content blocks. By sifting through network traffic for content strings that are both frequently repeated and widely dispersed, their method could automatically identify new worms and their precise signature. The SigBox method proposed in this paper is categorized as a method for generating an application signature in a string-based form.

2.1.2 Evaluation of automatic signature generation method

In order to evaluate existing methods, we define three metrics: automation, robustness, and elaboration. Table 1 shows an explanation of the three metrics.

Table 1. Metric for evaluating automatic signature generation method.

Metric	Explanation
Automation	• Is pre-processing required to calibrate configurable threshold value? • Is post-processing required to determine the final result?
Robustness	• Does consistent quality of signature remain regardless of the configuration? • Are the same results from the same input traffic ensured?
Elaboration	• Are various features used to represent the unique pattern of the target application? • Is the complicated behavior of the target application reflected?

In order to evaluate the automation degree, we examined whether it requires pre-processing to calibrate a configurable threshold value or post-processing to determine the final result. We changed the configuration threshold values and checked whether the quality of the results of the method is consistent, regardless of the configuration, to evaluate the robustness of the method. Finally, we checked whether the features used to reflect the complicated behavior of the target application in terms of elaboration. We chose two methods, LASER [6] and Autosig [7], which are approaches similar to our method, SigBox. According to the metrics listed above, we evaluated these two methods.

Table 2. Evaluation of LASER and Autosig.

Method	Metric	Evaluation
LASER	Automation	• Requires pre-processing for input traffic grouping • Requires post-processing for determining final signature
	Robustness	• The quality of the signature varies according to grouping • The results vary according to the order of the input traffic
	Elaboration	• Focuses only on packet-level features, such as packet payload
Autosig	Automation	• Requires several prior experiments for calibrating threshold values
	Robustness	• The quality of the signature varies according to the threshold
	Elaboration	• Focuses only on packet-level features, such as packet payload

LASER [6] automatically generates an application signature, in the form of a sequence of substrings, in the payload of a packet by using a modified version of the longest common subsequence (LCS) algorithm. The inputs of this algorithm are two distinct byte streams of packet payloads that belong to two flows. In order to improve the system's performance in terms of execution time and accuracy, this method considers only the first N packets of a flow and groups these packets by their size, since large packets are not likely to carry the same kind of information as small ones. Finally, the method compares two inputs to obtain the longest common subsequence between them, and then compares it with other subsequences iteratively to refine it.

Table 2 shows an evaluation of LASER in terms of the three metrics. In terms of the automation metric, this method requires pre-processing for grouping input traffic and determining the N value. In addition, it requires post-processing for determining the final result because of the nature of the LCS algorithm. The final result of the LCS algorithm is the length of the longest common subsequence, and therefore, this method requires an additional phase to determine the final result from several candidates. For example, the LCS of "ABC" and "ACB" is both "AB" and "AC." Therefore, it is necessary to select a final result from these candidates. In terms of the robustness metric, LASER extracts the longest common subsequence from only two designated sequences selected from all input sequences. Hence, the order of the input sequence affects the quality of the signature. Finally, LASER focuses only on packet-level features, such as the packet payload. Therefore, this method cannot reflect the complicated behaviors of certain applications.

Autosig [7] also generates an application signature automatically, which extracts multiple common substring sequences from input flows as the application signature. First, it divides the payload of a set of flows into short substrings called shingles. After extracting all the relevant, common shingles, Autosig merges them whether they are neighbors or overlapping. Next, a substring tree is constructed to create all the possible combinations of substrings. These combinations are considered as signatures.

Table 2 shows an evaluation of Autosig in terms of the three metrics. Although this method can generate a signature automatically without determining the order of the input traffic, several thresholds must be determined by performing a repetitive prior experiment. From this viewpoint, the automation level is low. In terms of robustness, the configurable threshold values affect the quality of the signature. The final result of the method could vary according to the configuration. Finally, Autosig, like LASER, focuses only on packet-level features, such as packet payload. Therefore, this method cannot

reflect the complicated behaviors of certain applications.

In contrast to LASER and Autosig, the performance of the SigBox method proposed in this paper is good in terms of the three metrics. SigBox does not need any prior experiment for calibrating threshold values. The original sequence pattern algorithm, the primary element of SigBox, needs the minimum support value as a threshold; however, we modify this phase. We consider the minimum support to be the number of hosts. Thus, we extract a common subsequence observed in all input hosts' traffic without any threshold values. In terms of robustness, this method always generates the same signature set from the same input traffic, regardless of the order of input. Finally, SigBox uses not only packet-level features in the content and packet signature, but also flow-level features in the flow signature. Therefore, we can generate a elaborated signature reflecting the complicated behaviors of certain applications.

2.2 Sequence Pattern Algorithm

A sequence pattern algorithm was first introduced in [18]. This algorithm, a type of data mining technique, detects time-series patterns in a database containing sequences of values or events. This technique very closely resembles association rule mining [21], in that both are similar processes for discovering frequent patterns in large datasets; however, the objective of association rule mining is to extract concurrent patterns from the same transaction, while that of a sequence pattern algorithm is to extract patterns with a certain order from different transactions [22].

The support threshold is very important for this algorithm. The measure is the ratio of sequences having the target subsequence to the total sequences. A minimum support predefined by the operator is used, because the number of possible subsequences is very large, and operators have different interests and purposes. Thus, we have to prune out uninterested patterns using the minimum support during the early stage.

Several sequence pattern algorithms have been proposed over the last few years. AprioriAll [18] is based on the Apriori property that any subsequence of a frequently occurring sequence is frequent. This method generates candidate sequences and checks the support value of each candidate to determine frequently occurring sequences. PrefixSpan [23] examines the prefix subsequences and projects their corresponding postfix subsequences into databases. This method does not need a candidate generation phase, but only that the postfix subsequences are recursively projected into the database according to the prefix. SPADE [24] is a method for determining frequent sequences using efficient lattice search techniques and simple joins. This method uses combinatorial properties to decompose the original problem into smaller sub-problems, which can be independently solved in the main memory. MEMISP [25] requires only one pass over the database, or at most two passes for a very large database, and avoids the need to generate candidates and project an intermediate database as well. SPIRIT [26] is a method that uses mining user-specified sequential patterns with regular expression constraints. In order to reduce the system overhead, this method eliminates uninterested and potentially useless patterns in early stage.

Currently, the sequence pattern algorithm is utilized in various domains. This algorithm can be used to analyze DNA sequences of two different organisms [20, 27]. In addition, it is applied in various application fields, such as geographic patterns [28],

marketing [29-31], and telecommunications [32, 33]. The basic idea of this algorithm is similar to signature generation, which finds commonly observed substrings in input traffic. Thus, we modify the sequence pattern algorithm such that it is suitable for signature generation.

3. SIGBOX: AUTOMATIC SIGNATURE GENERATION METHOD

SigBox is an automatic signature generation method for fine-grained traffic identification. This method does not require protocol semantic analysis or configurable threshold values to achieve complete automatic generation. In addition, it uses both packet-level and flow-level features to generate elaborated signatures capable of fine-grained identification.

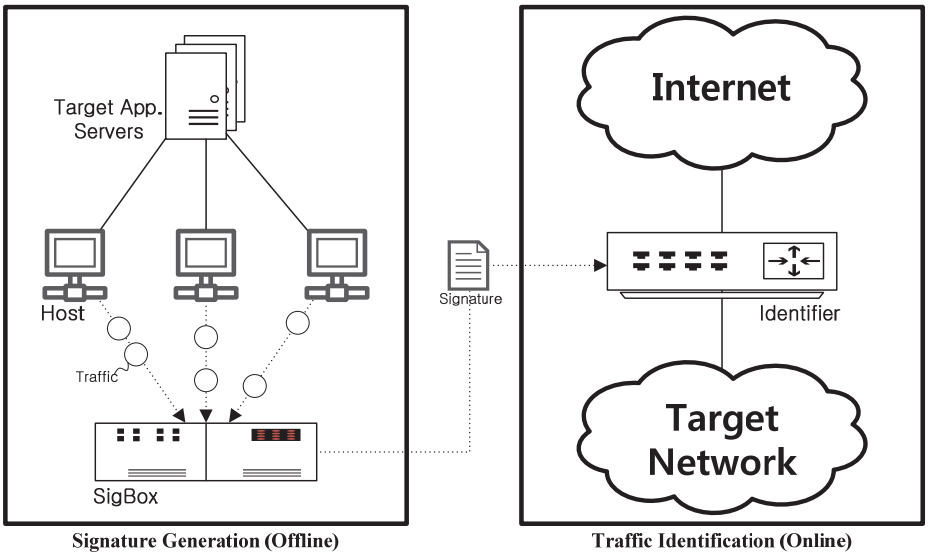

Fig. 1. Abstraction of SigBox process.

Fig. 1 shows the position of SigBox in the traffic identification process. In order to identify traffic, the traffic identifier located between the Internet and the target network inspects all traffic in real time. Typical identifiers are Snort [34] and Bro [35]. For this purpose, signatures should be loaded in the identifier in advance. The signature generation is operated in the offline mode. SigBox is deployed in this process. After determining the target application, the traffic of the application is collected from several different hosts, and then, the collected traffic set is input to SigBox without any configurable threshold values. SigBox outputs signatures as the result. The resulting signature set is loaded in the traffic identifier.

In the following sections, further details are provided. First, we define the three types of signatures: content, packet, and flow signature. Second, we explain the modified sequence pattern algorithm that is a core element of our method. Finally, we present an

explanation of the overall process, and then, we give details of each module of SigBox.

3.1 Signature Definition

In SigBox, we compartmentalize the signature as three types: content, packet, and flow signature. The content signature is defined as a unique substring, continuous characters of hexadecimal values, in a packet payload representing particular application traffic. The packet signature is defined as a sequence of content signatures in a packet. The flow signature, the final result of SigBox, is defined as a sequence of packet signatures in a flow. These three types of signatures have the relation of inclusion. Thus, a flow signature consists of packet signatures, and a packet signature consists of content signatures.

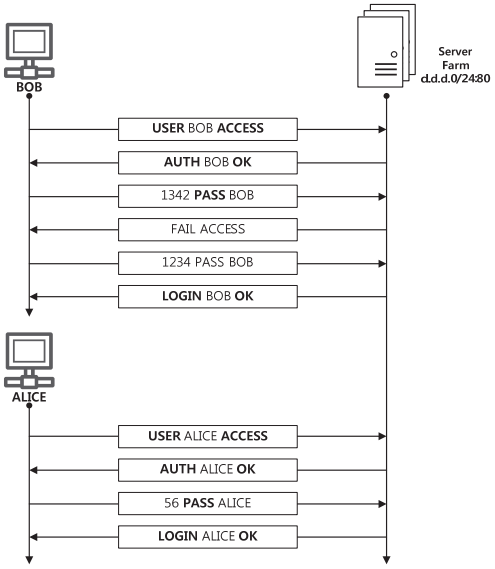

Fig. 2. Example traffic toward application server.

Fig. 2 shows example traffic toward a certain server farm from two different hosts when they start using the Internet application. Usually, Internet application provides their service via a server farm to reduce the system overhead. Two different hosts, BOB and ALICE, communicate with certain application servers located in the server farm. The rectangle located between the server and host represents a packet payload, and we suppose that all packets of a host are engaged in a single flow. As shown in Fig. 2, there is a unique sequence of substrings, marked in bold, in a flow when different hosts communicate with an application server. We extract this pattern as a signature.

Table 3 shows sample signatures based on the traffic example shown in Fig. 2. In order to support intuitive understanding, we borrow the description of Snort [34], which is widely used in network intrusion systems. The content and packet signatures are the intermediate results of SigBox, and the flow signature is its final result. The content sig nature consists of a single content. The packet signature consists of sequence of the con-

Table 3. Signature example.

Signature type	Num. of signature	Example
Content signature (intermediate)	6	tcp any any → d.d.d.0/24 80 (label: "TEST"; content: "USER"; offset: 0; depth: 4;); tcp any any → d.d.d.0/24 80 (label: "TEST"; content: "ACCESS"; offset: 9; depth: 8;); tcp d.d.d.0/24 80 → any any (label: "TEST"; content: "AUTH"; offset: 0; depth: 4;); tcp d.d.d.0/24 80 → any any (label: "TEST"; content: "OK"; offset: 9; depth: 5; tcp any any → d.d.d.0/24 80 (label: "TEST"; content: "PASS"; offset: 3; depth: 6;); tcp d.d.d.0/24 80 → any any (label: "TEST"; content: "LOGIN"; offset: 0; depth: 5;);
Packet signature (intermediate)	4	tcp any any → d.d.d.0/24 80 (label: "TEST"; content: "USER"; offset: 0; depth: 4; content: "ACCESS"; offset: 9; depth: 8;); tcp d.d.d.0/24 80 → any any (label: "TEST"; content: "AUTH"; offset: 0; depth: 4; content: "OK"; offset: 9; depth: 4); tcp any any → d.d.d.0/24 80 (label: "TEST"; content: "PASS"; offset: 3; depth: 6;); tcp d.d.d.0/24 80 → any any (label: "TEST"; content: "LOGIN"; offset: 0; depth: 5; content: "OK"; offset: 10; depth: 4;);
Flow signature (final)	1	tcp any any → d.d.d.0/24 80 (label: "TEST"; content: "USER"; offset: 0; depth: 4; content: "ACCESS"; offset: 9; depth: 8; flowbits: set, mark-1;); tcp d.d.d.0/24 80 → any any (label: "TEST"; content: "AUTH"; offset: 0; depth: 4; content: "OK"; offset: 9; depth: 4; flowbits: isset, mark-1; flow bits: set, mark-2;); tcp any any → d.d.d.0/24 80 (label: "TEST"; content: "PASS"; offset: 3; depth: 6; flowbits: isset, mark-2; flowbits: set, mark-3;); tcp d.d.d.0/24 80 → any any (label: "TEST"; content: "LOGIN"; offset: 0; depth: 5; content: "OK"; offset: 10; depth: 4; flowbits: isset, mark-3;);

tent signatures. Finally, the flow signature consists of individual packet signatures. Therefore, the number of signatures decreases from the content to the flow signature. Using a *flowbits* keyword, the flow signature allows a conjunction with each packet signature. For example, the second signature of the flow signature is activated only if the *flowbits* mark-1 is set in advance. Therefore, the last signature of the flow signature is activated when all prior packet signatures are activated in consecutive order. Each packet signature in the flow signature represents a pattern of a single packet consisting of content signatures. For example, the first signature of the packet signature example identifies a certain packet as "TEST" application if the packet has a sequence of content signatures such as "USER" and "ACCESS" with designated positional information and toward one of d.d.d.0/24 server farm using tcp/80 port.

Table 4 shows a detailed explanation of the signature components. A signature is composed of a header and a body to distinguish one target application from another. The header is a packet header engaged in the inspection. If it is impossible to specify the header component, we write "any." The body consists of a label and a sequence of contents. Each content has a range of inspection, such as offset and depth. In the case of the flow signature, we use a *flowbits* keyword to indicate the conjunction with a prior or later packet signature.

3.2 Sequence Pattern Algorithm for SigBox

In this section, we describe the modified sequence pattern algorithm. The original algorithm targeted purchase history data of a market to find sequential purchase patterns

[18]. In order to suit our purpose, which is to find common subsequences in traffic, we modified this algorithm in the sequence and support aspects.

Table 4. Signature components.

Component		Explanation
Header	L4Prot	OSI layer-4 protocol such as TCP, UDP, or ICMP
	srcIP	Source IP address using CIDR
	srcPort	Source port number
	dstIP	Destination IP address using CIDR
	dstPort	Destination port number
Body	Label	Target application name
	Content	Continuous bytes, include printable and non-printable character
	Offset	Start offset from which to search the content within a packet
	Depth	Range to search the content from a given offset
	Flowbits	Conjunction with prior or later signature

The ultimate aim of the sequence pattern algorithm is to find frequent subsequences in a set of input sequences. When extracting the three types of signature mentioned above, the algorithm applied is exactly same, and only the input sequences and composed item are different.

$$SequenceSet = \{S_1, S_2, \ldots, S_s\} \tag{1}$$

$$S_i = \{host_{id}, \langle I_1 I_2 I_3 \ldots I_n \rangle\} \tag{2}$$

Eqs. (1) and (2) show a set of sequences, which is one of the inputs for the sequence pattern algorithm. *S*, an element of *SequenceSet*, consists of host id, a series of items. The item is varied depending on the type of signature. In the case of extracting the content signature, the content sequence is the payload of the packet. Therefore, the item of the sequence is a one byte character or hexadecimal value of the packet payload. In the case of extracting a packet signature, the packet sequence is a series of content signatures located in the same packet payload. Therefore, the item of the packet sequence is an individual content signature. In the case of extracting a flow signature, the flow sequence is a series of packet signatures located in the same flow. Therefore, the item of the flow sequence is an individual packet signature.

$$support = \frac{Number\ of\ support\ hosts}{Total\ number\ of\ hosts} \tag{3}$$

The usage of support, one of this algorithm's inputs, is to maximize its performance. Thus, when increasing the length of a subsequence candidate, this algorithm eliminates certain candidates having an unsatisfied minimum support value in the early stage. Therefore, we can improve execution time and memory usage. However, SigBox does not use the original definition of support, which is the ratio of the number of sequences, including the target candidate subsequence, to the total number of sequences. Because the purpose of our method is to extract a subsequence contained in traffic generated by

all hosts when they use a target application, we modify the definition of support and the calculation method, as shown in Eq. (3). We redefine the support as the ratio of the number of hosts having sequences that contain the subsequence to the total number of hosts, to find the subsequences contained in the traffic generated by all the hosts. A sequence contains the host id as an element in order to calculate its support. Therefore, when the support is set as 1, SigBox can extract a subsequence observed in all input hosts.

Algorithm 1 shows the subsequence extraction process. First, we extract length-1 subsequences from all the sequences and store them in the length-1 subsequence set, L_1 (Alg. 1, Lines 1-5). From the length-1 subsequences, we extract all length-k candidate subsequences by increasing the length until no newer subsequences or candidates to be extracted (Alg. 1, Lines: 6-16). This iteration process consists of two parts. First, we eliminate candidates that do not satisfy the minimum support (1.0) after obtaining the support value from the calculation function described in Algorithm 2 (Alg. 1, Lines 8-13). Second, we extract length-k candidates by using length-(k–1) candidates, as described in Algorithm 3 (Alg. 1, Lines 14). As a final step, the relation of inclusion between subsequences is checked; if the relation is found, the included subsequences are deleted (Alg. 1, Line 18).

Algorithm 1: Pseudo algorithm for subsequence extraction

Procedure: *subsequence Extractor*
Input: *SequenceSet*, *MinSupp*
Output: *SubSequenceSet*

```
01: for each sequence S in SequenceSet do
02:     for each item i in sequence S do
03:         L1←L1 ∪ i;
04:     end
05: end
06: k←2
07: while Lk-1 ≠ ∅ do
08:     for each candidate c in Lk-1 do
09:                 supp←calSupport(c,SequenceSet);
10:         if (supp < Minsupp) then
11:             Lk-1←Lk-1–c;
12:         end
13:     end
14:     Lk←extractCandidate(Lk-1);
15:     k++;
16: end
17: SubSequenceSet←∪kLk;
18: deleteSubset(SubSequenceSet);
19: return SubSequenceSet;
```

Algorithm 2 shows the calculation method of the support of subsequences. This algorithm receives a subsequence and a set of sequences and outputs the support of the subsequence. First, the host ids of all the sequences are stored (Alg. 2, Line 2). Next, the

algorithm checks whether the subsequence is included in a sequence, for all sequences in the input set. If a sequence includes the subsequence, it stores the corresponding host id (Alg. 2, Lines 3-12). Finally, this algorithm returns the support value, calculated by dividing the number of support hosts by the number of total hosts (Alg. 2, Line 14). In this algorithm, we use the naïve pattern-matching algorithm to help provide an intuitional explanation of our algorithm. In the implementation of our system, we used advanced matching algorithms, such as Rabin-Karp [36] and Boyer-Moore [37]. Thus, there is room to improve the performance of our system.

Algorithm 2: Pseudo algorithm for support calculation

Procedure: *calSupport*
Input: *candidate*, *SequenceSet*
Output: *support*

```
01: for each sequence S in SequenceSet do
02:     totalhost ← totalhost ∪ S.host_id;
03:     for k = 1 to size of (S, ⟨I1I2I3…In⟩) do
04:         p←k, q←1;
05:         while (S, ⟨Ip⟩ = = candidate.⟨Iq⟩) do
06:             p++, q++;
07:         end
08:         if (q = = size of (candidate.⟨I1, I2, I3, …, In⟩)) then
09:             supphost←supphost∪S.host_id;
10:             break
11:         end
12:     end
13: end
14: return supphost/totalhost;
```

Algorithm 3 shows the extraction of the candidate subsequence. The length-(k–1) subsequence set, provided as input data, is utilized in order to extract the length-k subsequence set, L_k. All possible pairs of $L_{k\text{-}1}$ are compared (Alg. 3, Lines 1-10). If k–1, the length of the input subsequence, is 1, we simply combine two subsequences as a length-2 subsequence (Alg. 3, Lines 3-5); otherwise, we compare two subsequences to discover whether these two subsequences have a joinable common element. If one subsequence, excludes its first item, then it the same as another subsequence, whereas excluding its last item, means the two length-(k–1) subsequences are combined to form a single length-k subsequence (Alg. 3, Line 6-8). For example, length-3 subsequences "USE" and "SER" are combined into length-4 subsequence "USER," because "USE," excluding the first item "U," and "SER," excluding the last item "R," have a common subsequence "SE."

Algorithm 3: Pseudo Algorithm for Candidate Extraction

Procedure: extractCandidate
Input: $L_{k\text{-}1}$
Output: L_k

```
01: for each candidate x in L_{k-1} do
02:     for each candidate y in L_{k-1} do
03:         if ((k – 1) = = 1) then
04:             L_k ← L_k ∪ ⟨x.⟨i_1⟩ y.⟨i_1⟩⟩;
05:         end
06:         else if ((x.⟨i_2⟩ = = y.⟨i_1⟩) && (x.⟨i_3⟩ = = y.⟨i_2⟩) && …
08:                 (x.⟨i_{k-1}⟩ = = y.⟨i_{k-2}⟩)) then
07:             L_k ← L_k ∪ ⟨x.⟨i_1, i_2, i_3, …, i_{k-1}⟩ y.⟨i_{k-1}⟩⟩;
08:         end
09:     end
10: end
11: return L_k;
```

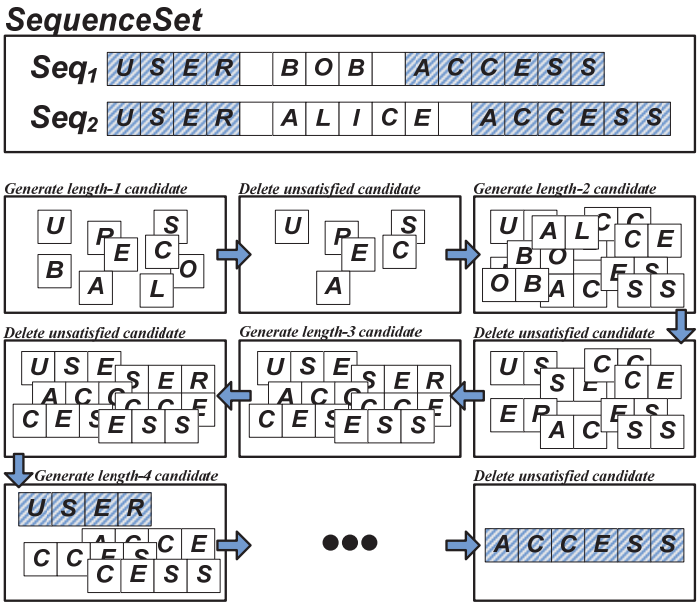

Fig. 3. Example of sequence pattern algorithm.

Fig. 3 shows an example of the sequence pattern algorithm when extracting a content signature. The box located at the top of the figure shows the input set of sequences. Because this example shows the generation of a content signature, the item composing the sequence is a one-byte character in a packet payload. The total number of sequences is two and the total number of hosts is two. We suppose that Sequences 1 and 2 have different host ids. Because a minimum support value of 1.0 is used, we extract the subsequences that are included in all hosts. First, we extract the length-1 candidate subsequences from all sequences, and calculate their support. Only subsequences satisfying the minimum support value of 1.0 are stored in the length-1 subsequence set, L_1. After extracting the length-1 subsequences, we extract all possible length-2 candidate subsequences using the length-1 subsequences. For example, we extract length-2 subsequence

"US," using the length-1 subsequences "U" and "S." For the length-3 candidate subsequence extraction, we combine two subsequences if they both have a common element. For example, the length-3 subsequence "USE" is extracted because "US" and "SE" contain "S" as a common element. Thus, we repeat the process of extending the subsequence length until there are no more subsequences. Finally, we check the relation of inclusion between the generated subsequences, and then, included subsequences are eliminated from the subsequence set. For example, "USER" includes the "USE" subsequence. Therefore, we delete the "USE" subsequence from the set of subsequences. Finally, "USER" and "ACCESS," which are highlighted, are extracted as results.

3.3 System Architecture

SigBox automatically generates signatures for fine-grained traffic identification from target Internet application traffic by using the modified sequence pattern algorithm after inputting the traffic of the target applications from two or more hosts. This system generates content signatures, then generates packet signatures, and finally generates flow signatures. After generating the flow signature, it refines the signature in the header and positional information aspect. The system architecture of SigBox consists of three phases according to signature type, as shown in Fig. 4.

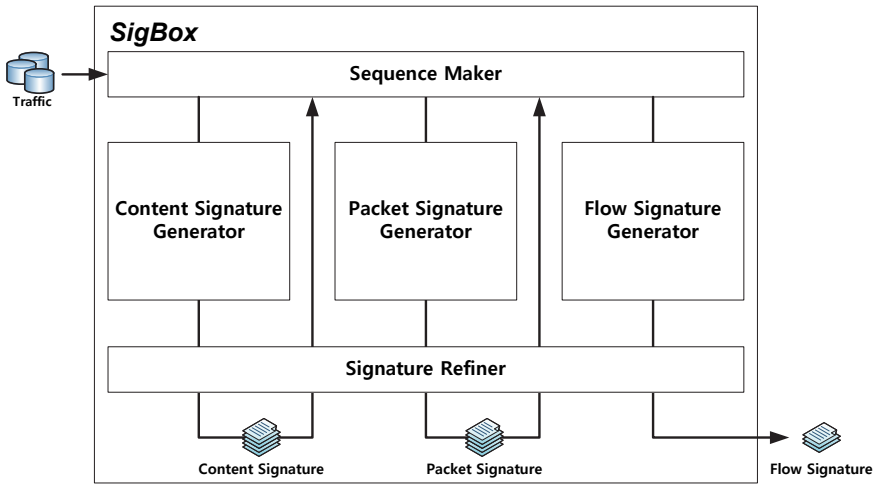

Fig. 4. System architecture of SigBox.

In the sequence maker module, traffic is loaded in the form of flows and sequences are constructed for each generator. For the content signature generator, this module makes *ContentSequences* consisting of one byte of packet payload. For the packet signature generator, this module makes *PacketSequences* consisting of content signatures located in the same packet. For the flow signature generator, this module makes *FlowSequences* consisting of packet signatures located in the same flow.

The content signature generator module extracts content signatures from the set of *ContentSequences*. The sequence consists of one byte of characters or hexadecimal val-

ues as an item in the packet payload. Therefore, this module extracts a series of characters or hexadecimal values observed in all hosts' traffic as a content signature. The packet signature generator module extracts packet signatures from the set of *PacketSequences*. The sequence consists of a series of content signatures located in the same packet. Therefore, this module extracts a series of content signatures observed in all hosts' traffic as a packet signature. The flow signature generator module extracts the flow signature from a set of *FlowSequences*. The sequence consists of a series of packet signatures located in the same flow. Therefore, this module extracts a series of packet signatures as a flow signature if it is observed in all the hosts' traffic.

The signature refiner module sets the header and positional information of the content. This module yields the position, such as offset and depth, of content by matching input traffic. In addition, the header information, such as the IP address, port number, and L4 protocol, is set.

3.3.1 Content signature generation

In order to a generate content signature, this module constructs a set of *ContentSequences* from input traffic and extracts content signatures from the sequences using the sequence pattern algorithm. Finally, it completes a content signature by operating the signature refiner.

$$Packet = \left\{ \begin{array}{l} packet_{id}, host_{id}, header, payload \\ | \; header = \{L4\,\mathrm{Pr}\,ot, srcIP, srcPort, dstIP, dstPort \end{array} \right\} \tag{4}$$

As shown in Eq. (4), a packet consists of a packet id, a host id, header, and payload. The header consists of the layer-4 protocol, source IP/port, and destination IP/port. The host id denotes the identifier of the host that generates the packet. The payload consists of a series of characters or hexadecimal values.

$$ContentSequence = \left\{ \begin{array}{l} sequence_{id}, host_{id}, \langle I_1 I_2 I_3 ... I_n \rangle \\ | \; I \text{ is one byte character or hexadecimal value} \end{array} \right\} \tag{5}$$

After composing a set of packets, the sequence maker reconstitutes a packet as a *ContentSequence*. The *ContentSequence* consists of a sequence id, a host id, and a series of one byte characters or hexadecimal values as shown in Eq. (5). All the components of the *ContentSequence* are inherited from the corresponding packet. Table 5 shows an example of *ContentSequence* set based on Fig. 2.

After constructing the set of *ContentSequences*, SigBox extracts the content signatures using the sequence pattern algorithm described in Section 3.2. When extracting the content signature, we consider a *ContentSequence* as a sequence and a one byte character as an item.

$$Content\ Signature = \left\{ \begin{array}{l} signature_{id}, \langle I_1 I_2 I_3 ... I_n \rangle \\ | \; I \text{ is one byte character or hexadecimal value} \end{array} \right\} \tag{6}$$

Table 5. Example of ContentSequence.

$sequence_{id}$	$host_{id}$	*ContentSequence*
1	BOB	<USER BOB ACCESS>
2	BOB	<AUTH BOB OK>
3	BOB	<1342 PASS BOB>
4	BOB	<FAIL ACCESS>
5	BOB	<1234 PASS BOB>
6	BOB	<LOGIN BOB OK>
7	ALICE	<USER ALICE ACCESS>
8	ALICE	<AUTH ALICE OK>
9	ALICE	<56 PASS ALICE>
10	ALICE	<LOGIN ALICE OK>

Table 6. Example of content signature.

$signature_{id}$	*Content Signature*
1	<USER>
2	<ACCESS >
3	<AUTH>
4	<OK>
5	<PASS>
6	<LOGIN >

The content signature consists of a signature id and a series of one byte characters or hexadecimal values as shown in Eq. (6). Table 6 shows an example of content signature using the example in Fig. 2.

3.3.2 Packet signature generation

The packet signature generator constructs a set of *PacketSequences* using the above content signatures, and extracts packet signatures from the sequences using the sequence pattern algorithm.

$$PacketSequence = \left\{ \begin{matrix} signature_{id}, host_{id}, \langle I_1 I_2 I_3 ... I_n \rangle \\ | \ I \text{ is content signature} \end{matrix} \right\} \tag{7}$$

The sequence maker reconstitutes a set of *PacketSequences* by referring to the packet set and content signature set. The *PacketSequence* consists of a sequence id, a host id, and a series of content signatures as shown in Eq. (7).

Table 7 shows an example of *PacketSequences.* By referring to the packet and content signature set, this module makes a set of *PacketSequences*. For example, the first sequence "<<USER><ACCESS>>" is composed of two content signatures "<USER>" and "<ACCESS>".

$$Packet\ Signature = \left\{ \begin{matrix} signature_{id}, \langle I_1 I_2 I_3 ... I_n \rangle \\ | \ I \text{ is content signature} \end{matrix} \right\} \tag{8}$$

Table 7. Example of PacketSequence.

$sequence_{id}$	$host_{id}$	*Packet Sequence*
1	BOB	<<USER><ACCESS>>
2	BOB	<<AUTH><OK>>
3	BOB	<<PASS>>
4	BOB	<< ACCESS>>
5	BOB	<<PASS>>
6	BOB	<<LOGIN><OK>>
7	ALICE	<<USER><ACCESS>>
8	ALICE	<<AUTH><OK>>
9	ALICE	<<PASS>>
10	ALICE	<<LOGIN><OK>>

After constructing the set of *PacketSequences*, SigBox extracts the packet signatures using the sequence pattern algorithm. A packet signature consists of a signature id and a series of content signatures, as shown in Eq. (8). Table 8 shows an example of packet signatures using Fig. 2.

Table 8. Example of packet signature.

$signature_{id}$	*Packet Signature*
1	<<USER><ACCESS>>
2	<<AUTH><OK>>
3	<<PASS>>
4	<<LOGIN><OK>>

3.3.3 Flow signature generation

The flow signature generator constructs a set of *FlowSequences* using the above packet signatures and extracts flow signatures from the sequences using the sequence pattern algorithm.

$$Flow\ Sequence = \left\{ \begin{matrix} sequence_{id}, host_{id}, \langle I_1 I_2 I_3 ... I_n \rangle \\ | \ I \text{ is packet signature} \end{matrix} \right\} \tag{9}$$

The sequence maker reconstitutes the set of *FlowSequences* by referring to the packet set and packet signature set. A *FlowSequence* consists of a sequence id, a host id, and a series of packet signatures as shown in Eq. (9).

Table 9. Example of FlowSequence.

$sequence_{id}$	$host_{id}$	*FlowSequence*
1	BOB	<<<USER><ACCESS>><<<AUTH><OK>><<<PASS>><<<ACCESS>><<<PASS>><<<LOGIN><OK>>>
2	ALICE	<<<USER><ACCESS>><<<AUTH><OK>><<<PASS>><<<LOGIN><OK>>>

Table 9 shows an example of *FlowSequences* using the example in Fig. 2. By referring to the packet set and packet signature set, this module makes a set of *FlowSequences*.

$$Flow\ Signature = \left\{ \begin{array}{l} signature_{id}, \langle I_1 I_2 I_3 ... I_n \rangle \\ | \ I \text{ is packet signature} \end{array} \right\} \quad (10)$$

After constructing the set of *FlowSequences*, SigBox extracts the flow signatures using the sequence pattern algorithm. The flow signature consists of a signature id and a series of packet signatures as shown in Eq. (10). Table 10 shows an example of a flow signature using the example in Fig. 3.

Table 10. Example of flow signature.

$signature_{id}$	*Flow Signature*
2	<<<USER><ACCESS>><<AUTH><OK>><<PASS>><<LOGIN><OK>>>

3.3.4 Signature refiner

The signature refiner module completes the signature by adding the positional information of the content and header of each signature. This module sets the content positional information, such as *offset* and *depth*. The *offset* information indicates the starting position for searching certain content within a packet payload and the *depth* information indicates how far the searching must continue from the *offset* location. Thus, we simply search for the content from the *offset* to the *offset* + *depth* in the packet payload. First, we initialize the *offset* value to the maximum size (1,500) of the packet payload and the *depth* value to 0. Next, we inspect all payloads of the input packet and obtain the *offset* and *depth*. If the payload includes the content, we obtain the starting position as *offset* and the ending as *depth*. If the position is already assigned, we choose the minimum value between the *offset* and the new *offset*. In the case of *depth*, we choose the maximum value. Finally, the *offset* is outputted as it is, and the *depth* is outputted after subtracting the *offset* from it.

In addition, this module sets the header of the signature by grouping packets including the signature. If a component of the header is unique, then we use it; else, we set the component to "any." For example, a set of packets containing a certain content has a unique destination port, 80. We use the destination port, 80, for the dstPort component of the signature. For the IP address, we attempt to extract a unique value by decreasing the Classless Inter-Domain Routing (CIDR), for example, from 32 to 24 or from 24 to 16. Thus, we first attempt to extract a unique IP address using CIDR 32. If the resulting value is not unique, we try C class using CIDR 24. For example, if a certain content originated from two different IP addresses, "z.z.z.1" and "z.z.z.2," we fail to extract a unique value using CIDR 32, and therefore we attempt to use CIDR 24 instead. When we have done this, we extract the IP address used in the header as "z.z.z.0/24."

Fig. 5 shows an example of the signature refiner. In order to set the position of the content "PASS," this module inspects three packets containing the content in their payload. The first and second packets contain the content from the fifth to ninth position in their payload, and the third packet contains the content from third to seventh position in their payload. Therefore, the offset of the content is 3, which is the minimum value between 5 and 3. The depth of the content is 6, which is the difference between 3 and 9. Next, this module sets the header of the signature. It Inspects the three packets and de-

termines dstPort as 80, because the three packets have the same dstPort. However, the srcIP and srcPort are described as "any" because they are different in these packets. Finally, the dstIP is calibrated by using CIDR 24.

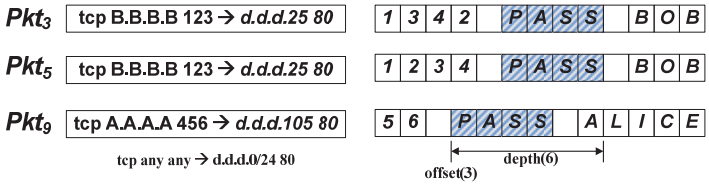

Fig. 5. Example of signature refiner.

4. EVALUATION

In this section, we present the details of the experimental results that validate the feasibility and demonstrate the performance of our proposed method. We define three metrics to evaluate signatures generated by the SigBox method. Under these metrics, we compared our method with other methods, LASER [6] and Autosig [7].

4.1 Traffic Trace

In order to collect the pure traffic (ground truth) of the target application, we deployed a traffic measurement agent (TMA) [6] on five selected hosts that were generating the target application traffic, as shown in Fig. 6. This agent continuously monitors the network interface card (NIC) and records the host's socket data and process information in a log file. Finally, the agent periodically transfers the log to a designated server called a traffic measurement server (TMS). Thus, we can obtain the absolute ground truth traffic of the target applications.

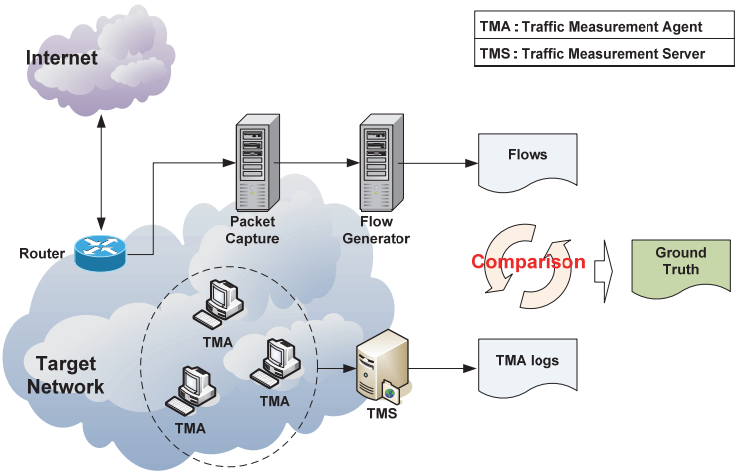

Fig. 6. Ground-truth traffic collection using TMA.

To achieve an objective experimental analysis, we selected ten popular Internet applications. The application types included video streaming, post office protocol, file transfer protocol, file sharing, messenger, game, and music streaming. The operation methods included peer-to-peer (P2P), server-client (SC), and encryption. When we selected the target applications, we referred to experiments in related studies, because our evaluation included a comparison analysis with the related studies, as described in Section 4.5. In addition, we collected two sets of traffic using each application, one for signature generation (SG) and the other for traffic identification (TI), on five different hosts. Thus, we automatically generated signatures from the SG trace set, and identified the TI trace set using the generated signature set. Table 11 shows the list of selected applications with their type, operation method, and statistical volume. The top six applications in Table 11 (Afreeca, BitTorrent, POP3, FTP, Fileguri, and Skype) were used in related studies. The other four applications were selected because they are currently popular in real networks.

Table 11. Selected applications.

Application	Type	Operation			Usage	Flows	Packets	Bytes(K)
		P2P	SC	Encryp.				
Afreeca	Video streaming	○	○		SG	1,470	110,667	95,894
					TI	1,447	117,069	100,741
BitTorrent	File sharing	○			SG	2,503	116,664	98,906
					TI	2,208	113,604	98,838
POP3	Post office protocol		○		SG	600	45,700	34,200
					TI	600	7,200	600
FTP	File transfer protocol		○		SG	2,005	41,605	4,798
					TI	1,851	40,899	4,545
Fileguri	File sharing	○	○		SG	831	34,713	26,679
					TI	1,230	26,633	14,744
Skype	Messenger	○	○	○	SG	956	23,033	7,254
					TI	829	19,793	6,370
KakaoTalk	Messenger		○	○	SG	282	54,900	76,177
					TI	451	78,076	107,506
DiabloIII	Game		○	○	SG	273	6,924	2,362
					TI	282	7,030	2,420
Melon	Music streaming		○		SG	770	76,072	64,482
					TI	944	115,054	99,926
Nateon	Messenger	○	○	○	SG	578	25,094	16,781
					TI	635	27,181	18,634

SG: Signature Generation, TI: Traffic Identification

4.2 Experimental Results

We generated signatures automatically from the SG trace set using SigBox. As previously mentioned, in our system we do not configure any threshold values. Although the core algorithm of SigBox is derived from the sequence pattern algorithm, which requires minimum support values as a threshold, our system sets the value as 1.0. This

means that the system extracts common patterns as signatures if the patterns are observed in all hosts' input traffic. Table 12 shows the experiment results in terms of execution time and number of signatures.

In all applications, SigBox generated signatures automatically in about 20 seconds. Because SigBox eliminates unsatisfied candidates in the early stage and considers only satisfied candidates when increasing the length of candidates, the generation execution time was reduced. Usually, a signature generation process is operated in offline mode, and therefore 20 seconds is completely acceptable execution time. However, this method is a method based on automatic signature generation. Rather than fast creation of the signatures, the system's origin goal is creation of accurate signatures, regardless of time. The time for creating signatures differs from one another due to difference in size of the input traffic data. As the future work we plan to reduce the signatures creation time of the system as much as possible, through the application of an optimal sequence pattern algorithm.

The number of signatures decreases as the type of signature changes from content signature to flow signature. As mentioned in Section 3, SigBox generates three types of signature. First, SigBox extracts continuous characters from the packet payload. Then, it converts the pattern to a content signature. A sequence of content signatures becomes a packet signature. Finally, a sequence of packet signatures becomes a flow signature, which is the final result of SigBox. Therefore, the flow signature type has the lowest signature count. In the real network environment, the number of signatures affect the overhead of the identification system. The system's overhead increases with the number of signatures. Therefore, using a flow signature can improve the performance of the identification system.

Table 12. Experimental results: execution time and number of signatures.

Application	Execution time (sec.)				Number of Signatures		
	Content Sig. generator	Packet Sig. generator	Flow Sig. generator	Total	Content Sig.	Packet Sig.	Flow Sig.
Afreeca	4.76	3.06	1.22	9.04	172	138	15
BitTorrent	2.89	0.07	0.03	2.99	56	37	16
POP3	0.03	0.01	0.01	0.05	7	7	1
FTP	0.12	0.01	0.03	0.16	7	7	1
Fileguri	0.05	0.01	0.01	0.07	90	45	18
Skype	1.32	0.03	0.01	1.36	40	17	16
KakaoTalk	0.19	0.01	0.01	0.21	16	10	8
DiabloIII	0.30	0.05	0.03	0.38	35	28	20
Melon	1.06	0.24	0.13	1.43	48	31	28
Nateon	19.82	0.09	0.04	19.95	17	8	18

Table 13 shows sample signatures of target applications generated using SigBox. Afreeca, a video streaming application, generates Internet traffic under HTTP and a proprietary protocol. The first signature generated by SigBox is extracted from the HTTP traffic. The destination IP described in the signature header is 58.229.158.0/24. The CIDR of the destination IP is calibrated by the signature refiner, as described in Section

Table 13. Experimental results: signatures.

App.	Signature
Afreeca	tcp any any → 58.229.158.0/24 80 (label: "AF"; content: "GET /CONFIG/?type=xml"; offset: 0; depth: 21; content: "User-Agent: afreecaplayer"; offset: 32; depth: 25; content: "Host: collector1.afree ca.com"; offset: 59; depth: 28; flowbits: set, AF-1-1;); tcp 58.229.158.0/24 80 → any any (label: "AF"; content: "HTTP/1.1 200 OK"; offset: 0; depth: 15; flowbits: isset, AF-1-1;); tcp any any → any any (label: "AF"; content: "|02 02 10 27 4E 02 00 00 5C 27 00 00|"; offset: 0; depth: 12; flowbits: set, AF-2-1;); tcp any any → any any (label: "AF"; content: "|02 02 BE 1B 00 00 00 00 BC 19 00 00|"; offset: 0; depth: 12; flowbits: isset, AF-2-1; flow bits: set, AF-2-2;); tcp any any → any any (label: "AF"; content: "|02 02 24 CB 00 00 00 00 26 C9 00 00|"; offset: 0; depth: 12; flowbits: isset, AF-2-2; flow bits: set, AF-2-3;); tcp any any → any any (label: "AF"; content: "|02 02 C1 1B 28 00 00 00 EB 19 00 00|"; offset: 0; depth: 12; flowbits: isset, AF-2-3; flow bits: set, AF-2-4;); tcp any any → any any (label: "AF"; content: "|CB 04 00 00 00|"; offset: 3; depth: 5; flowbits: isset, AF-2-4; flow bits: set, AF-2-5;); tcp any any → any any (label: "AF"; content: "|00 00 00 00|"; offset: 1; depth: 34; flowbits: isset, AF-2-5);
Bit-Torrent	tcp any any → 67.215.246.204 80 (label: "BT"; content: "GET /onboarding/bittorrent"; offset: 0; depth: 26; content: "Host: bundles.bittorrent.com"; offset: 80; depth: 28; content: "User-Agent: BTWebClient"; offset: 110; depth: 23; flowbits: set, BT-1-1;); tcp 67.215.246.204 80 → any any (label: "BT"; content: "HTTP/1.1 301 Moved Permanently"; offset: 0; depth: 30; flowbits: isset, BT-1-1;); tcp any any → any any (label: "BT"; content: "|13| BitTorrent protocol"; offset: 0; depth: 20;); udp any any → any any (label: "BT"; content: "d1:ad2:id20"; offset: 0; depth: 12;);
POP3	tcp any 110 → any any (label: "P3"; content: "+OK"; offset: 0; depth: 4; flowbits: set, P3-1-1;); tcp any any → any 110 (label: "P3"; content: "CAPA|0D 0A|"; offset: 0; depth: 6; flowbits: isset, P3-1-1; flow bits: set, P3-1-2;); tcp any 110 → any any (label: "P3"; content: "-ERR Invalid command|0D 0A|"; offset: 0; depth: 22; flowbits: isset, P3-1-2; flow bits: set, P3-1-3;); tcp any any → any 110 (label: "P3"; content: "AUTH |0D 0A|"; offset: 0; depth: 7; flowbits: isset, P3-1-3; flow bits: set, P3-1-4;); tcp any 110 → any any (label: "P3"; content: "-ERR Invalid command|0D 0A|"; offset: 0; depth: 22; flowbits: isset, P3-1-4; flow bits: set, P3-1-5;); tcp any any → any 110 (label: "P3"; content: "USER"; offset: 0; depth: 5; flowbits: isset, P3-1-5; flow bits: set, P3-1-6;); tcp any 110 → any any (label: "P3"; content: "+OK"; offset: 0; depth: 4; flowbits: isset, P3-1-6; flow bits: set, P3-1-7;); tcp any any → any 110 (label: "P3"; content: "PASS"; offset: 0; depth: 5; flowbits: isset, P3-1-7; flow bits: set, P3-1-8;); tcp any 110 → any any (label: "P3"; content: "+OK"; offset: 0; depth: 4; flowbits: isset, P3-1-8;);
FTP	tcp any 21 → any any (label: "FT"; content: "220"; offset: 0; depth: 3; flowbits: set, FT-1-1;); tcp any any → any 21 (label: "FT"; content: "USER anonymous|0D 0A|"; offset: 0; depth: 16; flowbits: isset, FT-1-1; flowbits: set, FT-1-2;); tcp any 21 → any any (label: "FT"; content: "331"; offset: 0; depth: 3; content: "password.|0D 0A|"; offset: 22; depth: 50; flowbits: isset, FT-1-2; flowbits: set, FT-1-3;); tcp any any → any 21 (label: "FT"; content: "PASS"; offset: 0; depth: 16; flowbits: isset, FT-1-3; flowbits: set, FT-1-4;); tcp any 21 → any any (label: "FT"; content: "230-|0D 0A|"; offset: 0; depth: 6; flowbits: isset, FT-1-4;);
Fileguri	tcp any any → any any (label: "FG"; content: "GET /?p2pmethod=search&keyword="; offset: 0; depth: 31; content: "&extension="; offset: 34; depth: 58; content: "Freechal P2P/1.0"; offset: 140; depth: 62; flowbits: set, FG-1-1;); tcp any any → any any (label: "FG"; content: "HTTP/1.1 200 OK"; offset: 0; depth: 15; flowbits: isset, FG-1-1;); tcp any any → any any (label: "FG"; content: "POST /FgDownLog.php"; offset: 0; depth: 19; content: "User-Agent: Fileguri"; offset: 117; depth: 20; content: "Host: kpi.fileguri.com"; offset: 145; depth: 22; flowbits: set, FG-2-1;); tcp any any → any any (label: "FG"; content: "HTTP/1.1 200 OK"; offset: 0; depth: 15; flowbits: isset, FG-2-1;); tcp any any → any any (label: "FG"; content: "sfiltersite.candlemedia.co.kr"; offset: 143; depth: 180;);
Skype	tcp any any → 111.221.123.231 80 (label: "SK"; content: "GET /ui/"; offset: 0; depth: 8; content: "User-Agent: Skype|A2 E2| 7.2"; offset: 107; depth: 23; content: "Host: ui.skype.com"; offset: 132; depth: 18; flowbits: set, SK-1-1;); tcp any any → any any (label: "SK"; content: "HTTP/1.1 200 OK"; offset: 0; depth: 15; flowbits: isset, SK-1-1;); tcp any any → any 443 (label: "SK"; content: "api.skype.com"; offset: 111; depth: 234;); tcp any any → any 443 (label: "SK"; content: "fig.skype.com"; offset: 317; depth: 13;);
Kakao Talk	tcp any any → any 5223 (label: "KT"; content: "|00 01 00 00 05 00 00 00 02 00 00 00|"; offset: 0; depth: 12;); tcp any 443 → any any (label: "KT"; content: "Thawte SSL CA"; offset: 181; depth: 13; content: "*.kakao.com"; offset: 318; depth: 11;);
DiabloIII	tcp any any → 121.254.166.21 80 (label: "DB"; content: "GET /service/d3/alert/ko-kr"; offset: 0; depth: 27; content: "User-Agent: BNET_COMPANY_BNET_PROJECT_BNET_CURRENT_VERSION_FUL L_TOKEN"; offset: 38; depth: 69; content: "Host: kr.launcher.battle.net"; offset: 109; depth: 28; flowbits: set, DB-1-1;); tcp 121.254.166.21 80 → any any (label: "DB"; content: "HTTP/1.1 200 OK"; offset: 0; depth: 15; flowbits: isset, DB-1-1;); tcp any any → 121.254.166.38 443 (label: "DB"; content: "kr.battle.net"; offset: 131; depth: 844; flowbits: set, DB-2-1;); tcp 121.254.166.38 443 → any any (label: "DB"; content: "kr.battle.net"; offset: 962; depth: 13; content: "tw.battle.net"; offset: 977; depth: 13; content: "see.battle.net"; offset: 992; depth: 14; content: "forums.battle.net"; offset: 1008; depth: 17; flowbits: isset, DB-2-1;);
Melon	tcp any any → any 9568 (label: "ML"; content: "GET /_music128?"; offset: 0; depth: 15; content: "User-Agent: HttpAsyncTemp1"; offset: 378; depth: 26; content: "Host: "; offset: 406; depth: 6; content: ":9568"; offset: 422; depth: 10; flowbits: set, ML-1-1;); tcp any 9568 → any any (label: "ML"; content: "HTTP/1.1 200 OK"; offset: 0; depth: 15; flowbits: isset, ML-1-1;); tcp any any → 211.234.237.25 80 (label: "ML"; content: "GET /static/web/resource/script/w1/0e/7/unsg936jxt.js"; offset: 0; depth: 53; flowbits: set, ML-2-1;); tcp 211.234.237.25 80 → any any (label: "ML"; content: "HTTP/1.1 304 Not Modified"; offset: 0; depth: 25; flowbits: isset, ML-2-1;);
Nateon	tcp any any → any 5004 (label: "NO"; content: "NCPT 1 "; offset: 0; depth: 7; flowbits: set, NO-1-1;); tcp any 5004 → any any (label: "NO"; content: "NCPT 1 "; offset: 0; depth: 7; flowbits: isset, NO-1-1; flow bits: set, NO-1-2;); tcp any any → any 5004 (label: "NO"; content: "PRVK 2"; offset: 0; depth: 6; flowbits: isset, NO-1-2; flow bits: set, NO-1-3;); tcp any 5004 → any any (label: "NO"; content: "PRVK 2"; offset: 0; depth: 6; flowbits: isset, NO-1-3; flow bits: set, NO-1-4;); tcp any any → any 5004 (label: "NO"; content: "CRPT "; offset: 0; depth: 5; flowbits: isset, NO-1-4; flow bits: set, NO-1-5;); tcp any 5004 → any any (label: "NO"; content: "CRPT "; offset: 0; depth: 5; flowbits: isset, NO-1-5;);

3.3.4. Thus, two more servers located in the 58.229.158.0/24 server farm generate traffic matched with the signature. In the content of the signature, some HTTP keywords are included, such as the URI, User-Agent, Host, and Response fields. All contents include not only the pattern but also its positional information. The second signature is extracted from the proprietary protocol traffic. The last packet signature of the flow signature is "|00 00 00 00|." One of the related systems, LASER, has the same signature. However, using only the signature consisting solely of several "|00|"s causes the accuracy of the result to be degraded, because the hexadecimal value "|00|" is usually used for traffic padding in a real network. Thus, our flow signature overcomes the uncertainty because we combine it with other packet signatures, and even their position.

BitTorrent, a file sharing application, generates Internet traffic under HTTP and a proprietary protocol. The first signature is extracted from the HTTP traffic. Because of the content "BTWebClient," we realized that the signature corresponds to a BitTorrent client. For sharing files with other peers, this application uses a proprietary protocol. SigBox generates various signatures, which include the well-known signature of BitTorrent "|13| BitTorrent protocol," presented in related studies. By specifying the position of each signature, we can improve the accuracy of the result as compared to other methods.

POP3 and FTP are used for sending e-mails and files to remote hosts. Because these two protocols use common words, such as "USER" and "PASS," as protocol keywords, the signatures of other methods, extracted from only one packet payload, may incorrectly identify other traffic. In other words, the POP3 signature "USER |20|" generated by Autosig incorrectly identifies FTP traffic. However, there is a low possibility that our signature will yield an incorrect identification, because the flow signature is a combination of packet signatures.

LASER presents only one signature "Freechal P2P" for Fileguri. This string could be located anywhere, even in another application. Therefore, this signature could incorrectly identify the traffic of another application. However, the first signature of SigBox includes not only the content but also other contents, such as "GET /?p2pmethod=search&keyword=" and "&extension=." Thus, the signature generated by SigBox can identify traffic more accurately.

Skype is a typical encrypted application for communicating with remote hosts. Its traffic is encrypted using SSL/TLS, and therefore, it is known that it is impossible to extract a payload signature. However, when a Skype client negotiates with issuers for a certification process, it generates plain text traffic under the SSL/TLS protocol in the early stage. Our signatures are extracted from the negotiations process traffic.

4.3 Evaluation Metrics

To evaluate our signatures generated using SigBox, we applied three metrics, redundancy, coverage, and accuracy, as defined below. Recall that we focus only on the performance of signature, and not on the quality of identified traffic.

Redundancy is defined as the traffic overlapped by the signature set of the target application, as shown in Eq. (11). Redundancy has a value between 0 and 1, where 0 is the best value, which indicates that all the signatures of a set identify traffic exclusively. If the redundancy of a certain signature set is close to 1, this indicates that the signature

set includes unnecessary signatures, which identify overlapped traffic. As the number of signatures increases, so does the system's overhead. Therefore, the redundancy value should remain low.

$$\text{Redundancy} = \frac{\text{volume of traffic identified by more than two signatures}}{\text{volume of traffic identified by signature set}} \tag{11}$$

The second metric is coverage. Coverage is defined as the ratio of the traffic identified by the signature set of the target application, as shown in Eq. (12). Coverage has a value between 0 and 1, where 1 is the best value, which indicates that the signature set identifies the total traffic of the target application. As a low coverage value causes false negative detection, we have to increase the coverage of the signature.

$$\text{Coverage} = \frac{\text{correctly identified traffic of target App}}{\text{total traffic of target App}} \tag{12}$$

Accuracy is defined as the ratio of traffic correctly identified by the signature set of the target application, as shown in Eq. (13). It has a value between 0 and 1, where 1 is the best value, which indicates that the traffic identified using the signature set is correct. As low accuracy causes a false positive detection, we have to increase the accuracy of the signature.

$$\text{Accuracy} = \frac{\text{correctly identified traffic of target App}}{\text{identified traffic}} \tag{13}$$

4.3.1 Redundancy evaluation

We measured the redundancy of the signature set using Eq. (11). Table 14 shows the number of signatures and the redundancy of each signature set. For all applications, the redundancy was under 0.2. This results indicates that the possibility of the generated signature set to overlap identification is low. Therefore, the number of signatures generated by SigBox is reasonable. For example, the number of signatures of Diablo III is 20, but the redundancy value is 0, which means that all signatures of Diablo III exclusively identified target traffic.

Table 14. Evaluation results: redundancy.

App.	Num. of signature	Redundancy
Afreeca	15	0.03
BitTorrent	16	0.05
POP3	1	0.00
FTP	1	0.00
Fileguri	18	0.13
Skype	16	0.19
KakaoTalk	8	0.04
DiabloIII	20	0.00
Melon	28	0.06
Nateon	18	0.02

4.3.2 Coverage evaluation

Table 15 shows the evaluation results in terms of coverage. As mentioned above, coverage is defined as the ratio of traffic identified by the signature set of the target application, as shown in Eq. (12). We measured coverage in two different trace sets, SG and TI. Our objective was to verify that the coverage of the signature is maintained at a certain level in different trace sets. First, we generated signatures from the SG trace set, and identified SG and TI trace sets using the signature set. The results show that all the signature sets had a similar level of coverage in both trace sets. Therefore, we can guarantee that SigBox can generate general and robust signatures of target application regardless of the particulars of the input data.

Some signature sets show a low coverage value. For example, the coverage of Skype signature is 25.84% for flow, 41.30% for packet, and 58.83% for byte. As mentioned, Skype is operated under the TLS/SSL protocol. Although we generated its signatures from negotiation phase traffic, these signatures could not identify data traffic that was encrypted by the TLS/SSL protocol. Thus, the Skype signature set covers only the negotiation traffic of Skype.

Also, the coverage of most applications does not reach 100% ratio. The reason is that not all the traffic flows contains signatures. These traffic flows could be encrypted flow, multimedia data flow, and so forth. The proposed automatic signature generation method can extract only the common string that appears in the input traffic flows. So, the proposed method cannot extract signatures which covers all the traffic flows for an applications including encrypted flow or data flow. The generation of signatures for encrypted traffic is beyond the scope of this paper. But, we set the identification of encrypted traffic on top of our future research topics.

Table 15. Evaluation result: coverage.

App.	Coverage (SG)			Coverage (TI)		
	Flow	Packet	Byte	Flow	Packet	Byte
Afreeca	77.96%	63.53%	61.19%	80.03%	77.48%	74.06%
BitTorrent	75.95%	67.54%	67.13%	71.92%	76.45%	78.58%
POP3	100%	100%	100%	66.67%	86.11%	83.33%
FTP	91.97%	98.47%	98.94%	95.41%	99.12%	99.35%
Fileguri	40.55%	51.13%	39.67%	36.26%	62.73%	49.91%
Skype	25.84%	41.30%	58.83%	25.09%	33.57%	48.09%
KakaoTalk	94.68%	82.18%	85.81%	93.13%	95.51%	96.76%
DiabloIII	93.41%	97.60%	97.89%	92.91%	96.74%	96.31%
Melon	91.43%	64.12%	60.41%	82.84%	46.29%	43.27%
Nateon	54.50%	78.86%	80.02%	50.39%	68.35%	69.51%

4.3.3 Accuracy evaluation

Table 16 shows the evaluation results in terms of accuracy. As mentioned above, accuracy is defined as the ratio of traffic correctly identified by the signature set of the target application, as shown in Eq. (13). We generated a signature set from the SG trace

set and identified the TI trace set using the signature set. As a result, the accuracy of all the signatures, except for Afreeca and Skype, is 100%. This indicates that the quality of the signatures generated by SigBox is very reasonable.

Table 16. Evaluation results: accuracy.

App.	Accuracy		
	Flow	Packet	Byte
Afreeca	86.17%	79.38%	78.96%
BitTorrent	100.00%	100.00%	100.00%
POP3	100.00%	100.00%	100.00%
FTP	100.00%	100.00%	100.00%
Fileguri	100.00%	100.00%	100.00%
Skype	95.18%	99.60%	99.87%
KakaoTalk	100.00%	100.00%	100.00%
DiabloIII	100.00%	100.00%	100.00%
Melon	100.00%	100.00%	100.00%
Nateon	100.00%	100.00%	100.00%

The accuracy of some signature sets is slightly low. The accuracy of the Afreeca signature is 86.17% for flow, 79.38% for packet, and 78.96% for byte. In order to analyze the reason for the accuracy degradation, we checked the accuracy of each signature for Afreeca.

tcp any any → any 80 (*label: "AF"; content: "GET /"; offset:0; depth:5; content:"image"; offset:5; depth:13;*)

The signature above causes the accuracy degradation of the Afreeca signature set. The signature consists of one packet signature. The packet signature consists of two content signatures. This signature was generated from the following payloads shown in Table 17.

Table 17. Afreeca traffic payload.

Afreeca traffic
GET/images/player/txt_vodtop_s.gif HTTP/1.1|0d||0a|Accept: */*|0d||0a|Accept-Language: ko-KR|0d||0a|Accept-Encoding: gzip, deflate|0d||0a| User-Agent: Mozilla/4.0 (compatible |3b| MSIE 7.0|3b| Windows NT 6.1|3b| WO W64|3b| Trident/7.0|3b| SLCC2|3b| .NET CLR 2.0.50727|3b| .NET CLR 3.5.3072 9|3b| .NET CLR 3.0.30729|3b| Media Center PC 6.0|3b | .NET4.0C |3b| .NET4.0E|3b| InfoPath.3)|0d||0a|Host: www.afreeca.com|0d||0a|Connection: Keep-Alive|0d||0a|Cookie: PCID=14301031179927722146235|3b| NO WCOM_RESOLUT ION=undefined*undefined|3b| NOWCOM_COLOR=24|0d| |0a||0d||0a|
GET/svc/image/U03/clix_adcontent/spacer HTTP/1.1|0d||0a|Accept: */*|0d||0a| Referer:ht tp://www.afreeca.com/ad/broad_default_AD.htm|0d||0a|Accept-Langu age:ko-KR|0d||0a|A ccept-Encoding: gzip, deflate|0d||0a|User-Agent: Mozilla/4.0 (compatible|3b| MSIE 7.0|3b| Windows NT 6.1|3b| WOW64|3b| Trident/7.0|3b| SLCC2|3b| .NET CLR2.0.50727|3b| .NET CLR 3.5.30729|3b| .NET CLR 3.0.30 729|3b| Media Center PC6.0|3b|. ET.0C|3b|.NET4.0E|3b| InfoPath.3)|0d| |0a|Host: i1.daumcdn.net|0d||0a|Connection:Kee-A v|0d||0a||0d||0a|

The highlighted strings in Table 17 are the sources of the signature. Within these payloads, the generated signature reflects Afreeca traffic well. However, the signature incorrectly identified traffic of other applications because the content described in the signature included common words.

Table 18 shows the traffic payload of Nateon. The signature of Afreeca incorrectly identified the highlighted string in Table 18. In order to prevent this conflict situation, we plan to apply black list filtering technology.

Table 18. Nateon traffic payload.

Nateon traffic
GET/plugin/images/ipml/quickLaunch/CID36_RQL3.png HTTP/1.1|0d||0a|User -Agent: NateOn/5.1.14.0 (3688)|0d||0a|Host: ipmlimg.nateon.nate.com|0d||0a| Cache-Control: no-cache|0d||0a|Cookie: UD3=u0d16bc6842ac97e50f6235b11ef 07fb|3b|UA3=MTAwMT U3NTYxMDY=|7c||7c||3b| pcid=141870852352994792 |3b|NateMain=isub=Y|7c|2015031 6|7c|3&HPClose=Y%7C20150317%7C2|3b| L OGIN=saveid=off&iplevel=2|0d||0a||0d||0a|

4.4 Comparison Evaluation of Signature Types

SigBox outputs three types of signature: content, packet, and flow signature. In this section, we compare the performance of each type of signature in terms of redundancy, coverage, and accuracy, and verify the feasibility and superior performance of the flow signature as compared to other types of signature.

Table 19. Comparison of signature types: redundancy.

Application	Redundancy		
	Content Sig.	Packet Sig.	Flow Sig.
Afreeca	0.99	0.80	0.21
BitTorrent	0.16	0.12	0.06
POP3	0.83	0.83	0.00
FTP	0.99	0.99	0.00
Fileguri	0.90	0.38	0.15
Skype	0.45	0.20	0.17
KakaoTalk	0.59	0.12	0.02
DiabloIII	1.00	0.96	0.00
Melon	0.99	0.94	0.04
Nateon	1.00	0.95	0.02

Table 19 shows the change in the redundancy value over the three signature types. As mentioned, the redundancy metric, the value of which is between 0 and 1, represents the ratio of overlapped traffic identified by more than two signatures in the same signature set. If a signature set has a high redundancy value, this means that it includes unnecessary signatures as its members. Thus, a redundancy value of 0 indicates the best signature set, which means that all the signatures of the set exclusively identify traffic.

From the content to flow signature type, the redundancy of the signature generated by SigBox declined in all applications. In particular, the gap between the packet and flow

signature type was extremely wide. According to the signature count comparison analysis described in Section 4.2, the content and packet signature types include more signatures that are unnecessary signatures for identification. As a result, the flow signature type is better than other signature types in terms of redundancy.

Table 20. Comparison of signature types: coverage and accuracy.

Application	Coverage			Accuracy		
	Content Sig.	Packet Sig.	Flow Sig.	Content Sig.	Packet Sig.	Flow Sig.
Afreeca	99.03%	98.96%	80.03%	20.70%	24.25%	86.17%
BitTorrent	84.10%	83.88%	71.92%	15.97%	18.53%	100%
POP3	100%	100%	66.67%	51.18%	51.18%	100%
FTP	97.84%	97.84%	95.41%	100%	100%	100%
Fileguri	99.27%	99.27%	36.26%	22.18%	25.56%	100%
Skype	30.64%	29.55%	25.09%	9.65%	9.65%	95.18%
KakaoTalk	96.67%	96.67%	93.13%	21.18%	21.18%	100%
DiabloIII	97.16%	97.16%	92.91%	27.21%	27.21%	100%
Melon	100%	100%	82.84%	23.88%	23.88%	100%
Nateon	97.80%	97.64%	50.39%	20.36%	21.61%	100%

Table 20 shows the change in the coverage and accuracy of the three signature types. In the flow signature type, the coverage slightly decreased. The reason for this degradation is that the flow signature type has more constraints than other signature types. However, the accuracy of the flow signature type was far superior to that of the other signature types. Thus, adding more constraints, the flow signature type could identify traffic correctly in terms of fine-grained identification. As a result, the flow signature type, the final result of SigBox, shows the best performance of the three signature types. Although coverage degradation occurred, the other two metrics were good.

4.5 Comparison Evaluation with Other Methods

We compared the performance of SigBox with other methods, LASER and Autosig. We selected six applications commonly used in other methods. Table 21 shows the coverage and accuracy comparison. In terms of both coverage and accuracy, the signature generated by SigBox showed a better performance than the other methods. As mentioned above, our signature uses flow features beyond one packet level. Therefore, SigBox generates better signatures than the other methods.

Table 21. Comparison with other methods: coverage and accuracy.

Application	Coverage			Accuracy		
	SigBox	LASER	Autosig	SigBox	LASER	Autosig
Afreeca	99.03%	14.24%	N/A	86.17%	5.57%	N/A
BitTorrent	84.10%	40.53%	57.97%	100%	100%	29.82%
POP3	100%	N/A	100%	100%	N/A	17.69%
FTP	97.84%	0%	97.78%	100%	0%	59.08%
Fileguri	99.27%	60.49%	N/A	100%	100%	N/A
Skype	30.64%	N/A	N/A	95.18%	N/A	N/A

Unfortunately, some applications are N/A for coverage and accuracy tests because we don't have both the LASER and Autosig system. We referred the experimental result of LASER and Autosig from their papers, so N/A means that the paper did not provide the measure of the application. However, we similarly experimented applications that were extracted from the LASER and Autosig systems. So as to objectively analyze and compare the experimental results, we used not only the extracted traffic but also the separately collected new traffic. Therefore, the objective was maintained after the input traffic being tested from all the three methods.

5. SNORGEN: WEB SITE FOR SNORT RULE GENERATION

We opened a Web site (http://snorgen.korea.ac.kr/) to announce our results. This Web site provides automatic signature generation for users who want to generate their own signature. After a user uploads traffic data, the Web site provides them with a signature after a short amount of time. In addition, our experimental results discussed in this paper can be found on the Web site. The signatures that are generated are represented in Snort form. Therefore, a user can use the signature in their Snort engine directly.

Fig. 7. SnorGen Web site (http://snorgen.korea.ac.kr/).

6. CONCLUSIONS AND FUTURE WORK

We proposed SigBox, which automatically generates a signature based on a sequence pattern algorithm to overcome previous limitations in terms of automation, robustness, and elaboration. This system finds unique substrings in the input packet payload and uses them to construct a content signature. The sequence of content signatures

is combined into a packet signature, and the sequence of packet signatures is combined into a flow signature. The flow signature is the final result of SigBox. To achieve this, we applied a modified sequence pattern algorithm to find a sequence commonly described in the input hosts' traffic.

Through experimentation, we validated the performance of SigBox using ten popular applications. The results show that SigBox is capable of generating signatures automatically and quickly without any predefined thresholds. In all the applications, we generated signature sets of target applications within 20 seconds. In addition, our system shows a good performance in terms of redundancy, coverage, and accuracy. The average redundancy of all the signature sets was 0.05, which indicates all the signatures of each signature set exclusively identified the target traffic. The coverage level was similar in both SG (training) trace and TI (testing) trace, which indicates that the signatures generated by SigBox are general and robust. Finally, the average accuracy of all signature sets was 98.14%, which indicates that the quality of the signatures generated by SigBox was very elaborated. A comparison evaluation of signature types showed that the flow signature type performs better in terms of redundancy and accuracy, with a slight degradation in coverage. This method even outperformed other methods, LASER and Autosig.

In future work, we plan to develop automatic post-processing techniques that will allow us to select the most meaningful signatures from those output by SigBox and to enlarge the range of generation to allow signatures that are more elaborated.

REFERENCES

1. M.-S. Kim, Y. J. Won, and J. W.-K. Hong, "Application-level traffic monitoring and an analysis on IP networks," *ETRI Journal*, Vol. 27, 2005, pp. 22-42.
2. M.-J. Choi, J.-S. Park, and M.-S. Kim, "An integrated method for application-level internet traffic classification," *KSII Transactions on Internet and Information Systems*, Vol. 8, 2014, pp. 838-856.
3. T. T. Nguyen, G. Armitage, P. Branch, and S. Zander, "Timely and continuous machinelearning-based classification for interactive IP traffic," *IEEE/ACM Transactions on Networking*, Vol. 20, 2012, pp. 1880-1894.
4. Y. Wang, Y. Xiang, W. L. Zhou, and S. Z. Yu, "Generating regular expression signatures for network traffic classification in trusted network management," *Journal of Network and Computer Applications*, Vol. 35 , 2012, pp. 992-1000.
5. B. Park, Y. Won, J. Chung, M. S. Kim, and J. W. K. Hong, "Fine-grained traffic classification based on functional separation," *International Journal of Network Management*, Vol. 23, 2013, pp. 350-381.
6. B.-C. Park, Y. J. Won, M.-S. Kim, and J. W. Hong, "Towards automated application signature generation for traffic identification," in *Proceedings of IEEE International Conference on Network Operations and Management Symposium*, 2008, pp. 160-167.
7. M. Ye, K. Xu, J. Wu, and H. Po, "Autosig-automatically generating signatures for applications," in *Proceedings of the 9th IEEE International Conference on Computer and Information Technology*, 2009, pp. 104-109.

8. X. Feng, X. Huang, X. Tian, and Y. Ma, "Automatic traffic signature extraction based on Smith-waterman algorithm for traffic classification," in *Proceedings of the 3rd IEEE International Conference on Broadband Network and Multimedia Technology*, 2010, pp. 154-158.
9. S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementaion*, 2004, p. 4.
10. H.-A. Kim, B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *Proceedings of USENIX Security Symposium*, 2004.
11. J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proceedings of IEEE Symposium on Security and Privacy*, 2004, pp. 226-241.
12. Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Proceedings of IEEE Symposium on Security and Privacy*, 2006, pp. 15 pp.-47.
13. Y. Tang, B. Xiao, and X. Lu, "Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms," *Computers and Security*, Vol. 28, 2009, pp. 827-842.
14. G. Szabó, Z. Turányi, L. Toka, S. Molnár, and A. Santos, "Automatic protocol signature generation framework for deep packet inspection," in *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*, 2011, pp. 291-299.
15. Y. Wang, Y. Xiang, and S. Z. Yu, "An automatic application signature construction system for unknown traffic," *Concurrency and Computation-Practice and Experience*, Vol. 22, 2010, pp. 1927-1944.
16. Y. Choi, "An automated classifier generation system for application-level mobile traffic identification," in *Proceedings of IEEE Network Operations and Management Symposium*, 2012, pp. 1075-1081.
17. P. Haffner, S. Sen, O. Spatscheck, and D. Wang, "ACAS: automated construction of application signatures," in *Proceedings of ACM SIGCOMM Workshop on Mining Network Data*, 2005, pp. 197-202.
18. R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the 11th IEEE International Conference on Data Engineering*, 1995, pp. 3-14.
19. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 487-499.
20. C.-M. Hsu, C.-Y. Chen, B.-J. Liu, C.-C. Huang, M.-H. Laio, C.-C. Lin, and T.-L. Wu, "Identification of hot regions in protein-protein interactions by sequential pattern mining," *BMC Bioinformatics*, Vol. 8, 2007, S8.
21. R. Agrawal, T. Imieli, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993, pp. 207-216.
22. Q. Zhao and S. S. Bhowmick, "Sequential pattern mining: A survey," ITechnical Report, CAIS, Nayang Technological University, Singapore, 2003, pp. 1-26.
23. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth,"

in *Proceedings of IEEE International Conference on Data Engineering*, 2001, pp. 0215-0215.
24. M. J. Zaki, "SPADE: An efficient algorithm for mining frequent sequences," *Machine Learning*, Vol. 42, 2001, pp. 31-60.
25. M.-Y. Lin and S.-Y. Lee, "Fast discovery of sequential patterns by memory indexing," in *Data Warehousing and Knowledge Discovery*, Springer, 2002, pp. 150-160.
26. M. N. Garofalakis, R. Rastogi, and K. Shim, "SPIRIT: Sequential pattern mining with regular expression constraints," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 223-234
27. K. Wang, Y. Xu, and J. X. Yu, "Scalable sequential pattern mining for biological sequences," in *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*, 2004, pp. 178-187.
28. J. Han, K. Koperski, and N. Stefanovic, "GeoMiner: a system prototype for spatial data mining," in *ACM SIGMOD Record*, 1997, pp. 553-556.
29. J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web usage mining: Discovery and applications of usage patterns from web data," *ACM SIGKDD Explorations Newsletter*, Vol. 1, 2000, pp. 12-23.
30. M. El-Sayed, C. Ruiz, and E. A. Rundensteiner, "FS-Miner: efficient and incremental mining of frequent sequence patterns in web logs," in *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*, 2004, pp. 128-135.
31. U. Yun, "Analyzing sequential patterns in retail databases," *Journal of Computer Science and Technology*, Vol. 22, 2007, pp. 287-296.
32. J.-Z. Ouh, P.-H. Wu, and M.-S. Chen, "Experimental results on a constraint based sequential pattern mining for telecommunication alarm data," in *Proceedings of the 2nd IEEE International Conference on Web Information Systems Engineering*, 2001, pp. 186-193.
33. P.-H. Wu, W.-C. Peng, and M.-S. Chen, "Mining sequential alarm patterns in a telecommunication database," in *Databases in Telecommunications II*, Springer, 2001, pp. 37-51.
34. M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration*, 1999, pp. 229-238.
35. V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, Vol. 31, 1999, pp. 2435-2463.
36. R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, Vol. 31, 1987, pp. 249-260.
37. R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, Vol. 20, 1977, pp. 762-772.

Kyu-Seok Shim received his B.S degree in Computer Science from Korea University, Korea, in 2014. He is currently a master's student of Korea University, Korea. His research interests include Internet traffic classification and network management.

Sung-Ho Yoon received his B.S. degree in Computer Science from Korea University, Korea, in 2009, his M.S. degree in Computer Science from Korea University, Korea, in 2011, and his Ph.D. degree in Computer Science from Korea University, Korea, in 2015. He is currently a researcher of Vehicle Components Company, LG Electronics, Korea. His research interests include Internet traffic classification and network management.

Su-Kang Lee received his B.S degree in Computer Science from Korea University, Korea, in 2014. He is currently a master's student of Korea University, Korea. His research interests include Internet traffic classification and network management.

Myung-Sup Kim received his B.S., M.S., and Ph.D. degree in Computer Science and Engineering from POSTECH, Korea, in 1998, 2000, and 2004, respectively. From September 2004 to August 2006, he was a postdoctoral fellow in the Department of Electrical and Computer Engineering, University of Toronto, Canada. He joined Korea University, Korea, in 2006, where he is working currently as an Associate Professor in the Department of Computer and Information Science. His research interests include Internet traffic monitoring and analysis, service and network management, and Internet security.