Applied Mathematics & Information Sciences
An International Journal

A Fast Regular Expressions Matching Algorithm for NIDS

Meng-meng Zhang[1]*, *Yan Sun*[2] *and Jing-zhong Wang*[3]

[1]School of Information Engineering, North China University of Technology, Beijing 100144, China
[2]School of Electrical Engineering and Computer Science, Washington State University, Pullman, Washington 99164C2752, U.S.A
Email: zmm@ncut.edu.cn

Received: 8 Jul. 2012; Revised 4 Oct. 2012; Accepted 6 Dec. 2012
Published online: 1 Mar. 2013

Abstract: In this paper, we propose a new algorithm to accelerate the searching speed in network intrusion detection system (NIDS) and we implement our algorithm in Snort, a popular open-source intrusion detection system. The algorithm is based on the fact that normal data stream rarely matches any virus signature and different packets need to check different keys. The algorithm does not need preprocessing and can check multiple characters in parallel. Experimental results show that our implementation is faster than original NFA/DFA based algorithms to deal with the same real packet traces while consuming an order of magnitude less memory.

Keywords: NIDS, matching, Snort

1. Introduction

Computer network security [1] is gaining popularity among network practitioners, with organizations investing more time and money to protect their valuable information. Security has also recently attracted considerable attention from network researchers due to the importance of network security has grown tremendously. A number of devices have been introduced to improve the security of a network, and the network intrusion detection systems (NIDS) are among the most widely deployed such systems [2, 3]. A network intrusion detection system (NIDS) is an intrusion detection system that tries to detect malicious activity such as denial of service attacks, port scans or even attempts to crack into computers by monitoring network traffic. The NIDS does this by reading all the incoming packets and trying to find suspicious patterns. But existing methods such as Non-deterministic Finite Automata (NFA), Deterministic Finite Automata (DFA) and Boyer-Moore string search algorithm have limitations in this particular application.

Many important services in current networks are based on payload inspection, in addition to headers processing. NIDS as well as traffic monitoring and layer-7 filtering require an accurate analysis of packet content in search of matching with a predefined data set of patterns to identify specific classes of applications, viruses, protocol definitions, etc. Traditionally, the data sets were constituted of a number of signatures to be searched with string matching algorithms, security signatures have been specified as string based exact match, but the exact string matching is not expressive enough to detect malicious patterns, so nowadays more expressive regular expressions are used to describe a wide variety of payload signatures [4]. For example, in the Linux Application Protocol Classifier [5], all protocol identifiers are expressed as regular expressions. Similarly, the Snort [6], which is an open-source network intrusion detection system, has evolved from no regular expressions in its rule set in April 2003 to 5,549 out of 8,786 rules contain at least one Perl Compatible Regular Expression (PCRE) as of November 2010. Another open-source intrusion detection system, Bro [7], also uses regular expressions as its pattern language. These are used in firewalls and devices by different vendors such as Cisco [8].

Packet content scanning is crucial to network security and network monitoring applications. Modern network devices need to perform deep packet inspection at high speed for security and application-specific services. Boyer-Moore string search algorithm, which is a particularly efficient

* Corresponding author: email: zmm@ncut.edu.cn

string searching algorithm and has been the standard benchmark for the practical string search literature, is widely used in deep packet inspection, but it has two critical limitations: first Boyer-Moore algorithms speed is not fast enough because it needs to pre-process each target string (key) and it cannot search multiple keys in parallel; second, it only performs exact string matching. Finite Automata (FAs) [9–13] are the most popular methods used to implement regular expressions matching these years, but they require complex preprocessing to construct FAs and require a large amount of memory.

Nondeterministic Finite Automata (NFA) are representations which require more state transitions per character, thus having a time complexity for lookup of O(m), where m is the number of states in the NFA; on the other hand, they are very space-efficient structures. Instead, Deterministic FAs (DFAs) require only one state traversal per character, but for the current regular expression sets they need an excessive amount of memory. When checking a particular packets payload, only a very small subset of the rule set need to be considered, so DFAs waste most of the memory. For these reasons, such solutions do not seem to be proper for implementation in real deep packet inspection applications, which are required to perform on line packet processing at high speeds.

To speed up the pattern matching speed, some hardware solutions have been proposed [14–17], and they are mostly based on Finite Automaton (FAs). In this paper, we implement our algorithm based on software, but we argue that our algorithm also efficient if implemented in hardware such as FPGA, and we will focus on that in the future work.

To the best of our knowledge this paper presents a novel and different approach to the pattern matching problem: we divide the packets payload into fixed- length blocks and performs the pattern matching for every block one by one. For each block, only a small number of comparisons are required based on the fact that normal data streams rarely match any virus signature in NIDS and most of the keys we need to compare are short. And multiple keys are checked in parallel to accelerate the processing speed and reduce the data dependency between instructions. For the unfixed-length regular expression keys, our algorithm is used as a hash table to exclude most of the packets from further check. Our algorithm only need a small amount of memory to store frequently used data, which stored in Cache, so most of time the CPU does not need to access main memory. The latency of memory access is usually hundreds of times longer than the CPU execution clock cycle, so our algorithm speeds up the pattern matching mainly because of seldom memory access. The remainder of the paper is organized as follows. In section 2 NIDS and Snort rule set are discussed. Section 3 describes our algorithm and section 4 compares our algorithm with other algorithms. Then in section 5 the implementation and simulation are shown. Finally, conclusions and future works are exposed in Section 6.

2. NIDS and Snort

A network intrusion detection system (NIDS) is an important security tool for network administrators to protect their networks. It enables them to monitor the networks by inspecting packets in real time and detecting malicious attacks such as unauthorized accesses, port scans, and denial of service (DoS) attacks. A NIDS classifies packets using a rule (or signature) database in order to determine whether packets are malicious. Snort uses a simple language to define rules to describe network behaviors. Each rule consists of five mandatory fields and numerous option fields. The mandatory fields include protocol type (e.g., TCP, UDP), source/destination IP addresses and port numbers, all of which are part of a packet header. A straightforward way to check whether a packet matches any of the rules is to search the rule database in a brute force manner: testing each rule against the packet one by one. It is easy to implement but time-consuming. To reduce the number of rules to examine, Snort builds a tree structure called rule tree as shown in Figure 2.1 to store and organize all the rules. For each rule, the mandatory fields are stored in a rule tree node (RTN) and the option fields are stored in an option tree node (OTN). An OTN is associated with the corresponding RTN. If there are multiple rules that have the same mandatory fields, only a single RTN is created and OTNs share it.

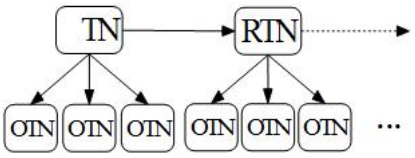


Figure 2.1 A simple rule tree in Snort

3. Proposed Fast Regular Expressions Matching Algorithm

A. Related works on Regular Expressions Matching

Our proposed fast regular expressions matching algorithm focus on the pattern matching over packets payload. When a rule matches all the mandatory fields in a packet header, our pattern matching engine performs string pattern matching over payload, and the keys are stored in the corresponding OTN set.

The traditional DFA based NIDS has three main limitations: first they fail to exploit the fact that normal data streams rarely match any virus signature; second, DFAs

are extremely inefficient in following multiple partially matching signatures, and third DFAs require long latency to be reconstructed when the rule set is updated. We propose mechanisms to solve these drawbacks and demonstrate that our solutions can implement a NIDS much more efficiently.

When the string pattern matching over payload is need, only a small number of keys need to be checked. Typically, on microprocessors, the key matching isperformed by first converting the given key into a corresponding NFA or Deterministic Finite Automaton (DFA) which is then used to search input text characters. While a DFA can process

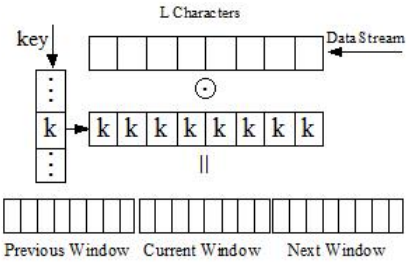


Figure 3.2 Architecture of the Fast String Matching

each character in constant time (i.e. it requires $O(1)$ time), the number of DFA states, for an n character key, can be $O(2^n)$ [18] , which in some cases can significantly degrade performance. Some recent methods use all the keys to construct big DFAs in order to check multiple keys in parallel and can check one character each time but almost need memory access each time, which will increase the searching latency. Unfortunately, most of packets rarely match any keys, so we want to exclude the unmatched packets fast and check the payload of a packet in parallel instead of checking one character each time.

B. Proposed Fast Regular Expressions Matching Algorithm

First, lets consider exact string matching. We divide the payload of a packet into same-length blocks. Assuming that the length of a block is L bytes, and need three L-bit long Windows to record the temporary result. The Window used to record current being checked blocks result is called Current Window (CW), the record generated by the previous block is called Previous Window (PW), and the result generated by current block and used for the next block matching is called Next Window (NW). The architecture of the proposed string matching engine is shown in Figure 3.1.

Our engine checks the blocks on by one as following steps:

Step 1: For the first block, the three windows are set to be zero and set a counter "CNT" to be L for initialization.

Step 2: If CNT is L, NXOR the current block with L duplicated first character of the key, and store the L-bit result into CW, or NXOR the current block with L duplicated next character of the key then take AND operation with the CW and store the result into CW.

Step 3: If this is the last character in the key and the new CW is not zero, then a match has been found, or go to step 4.

Step 4: If the PW and CW are all zero, shift {CW, NW} CNT bits right, then copy the NW to PW, set CNT to be L and move in the next block for further check.

Step 5: If the PW is all zero and CW is all zero except the least significant bit, shift {CW, NW} CNT bits right, then copy the NW to PW, set CNT to be L and move in the next block for further check.

Step 6: If PW is not zero, or CW is not zero or one, shift {PW, CW, NW} one bit right and CNT count down 1.

Step 7:Go back to step 2.

A simple example is shown in Figure 3. In this example,$L = 8$, key="key", block 1 and block 2 contain a string "Tmnorrkeyinokruk". From (b) to (c), a new block moves in and {CW, NW} shifts 7-bit right ($CNT = 7$). Some windows are omitted in Figure 3.2. if the content of the window is "00000000", and a match is found in (e).

Above sequence is a subset of our algorithm, for we do not consider the wild cards in regular expressions and we assume the length of key is less these two problems in section 3.3. From the theory and experimental Results, we can conclude:

a) If a block does not contain the first character of key, only once check is needed for this block, and experimental results show that when $L = 8$, approximate 83 percent of blocks only need to check 1 to 3 times for each block.

b) The longer of the block length the better performance we can get, because the times of comparison in a block only depend on the longest match in this block and the previous one. So the times of comparison in a large combined block is roughly the same as times of comparison in the longest times of these small blocks.

c) This algorithm does not require preprocessing and rarely access memory for does not need to save or restore much data.

d) We can check multiple keys simultaneously. Match one block with keys one by one and only need to save the temporary result of each key, and they can be stored in registers or cache to avoid time consuming memory access. And this also can reduce the data dependency during the instruction execution of a single key.

We need to consider the worst case even though it seems cannot happen ever. For the worst case, every block needs L comparisons with one key, but only this special key can match the worst case condition. Even though this needs much more comparisons, our algorithm still can achieve high packet throughput for these simple instructions have much shorter latency compared with memory access. And we will discuss the detail of the worst case later.

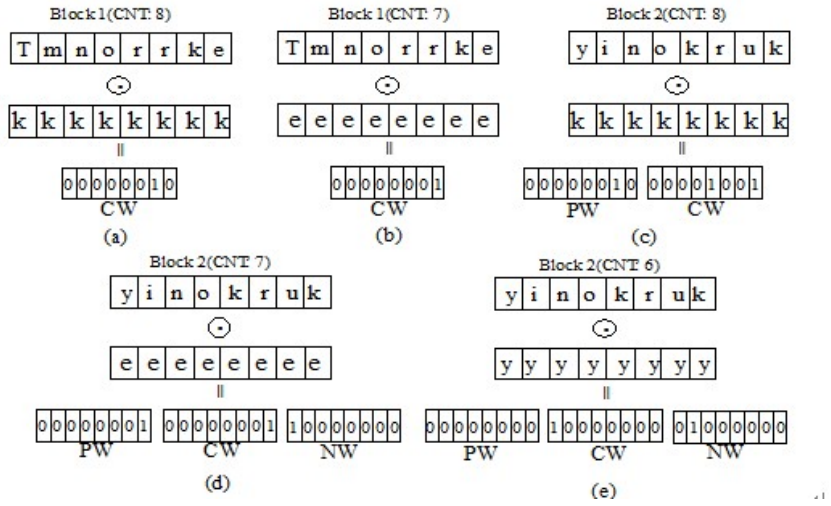


Figure 3.3 Architecture of the Fast String Matching

C. Special Cases

The algorithm above is based on some assumptions, but we need to consider all situations in practice, so we will discuss these in this section.

1. Long Key Problem

If the length of a key is larger than the length of the block, we can use double windows (or more windows) to deal with and calculate using the similar way. Actually, we cut the long key into several L byte pieces and compare them with one block separately, then multiple results are passed on to the next block comparison.

2. Non-Exact Character

In the recent rule sets, some characters are not "exact", but the length of the characters is "exact", such as "!a" and "a|b", which mean "is not character a and character a or character b" separately. For these characters, we can simply modify the operations used in the comparisons. For example, we can use XOR operation instead of NXOR to represent "!a", and use (NXOR a)|(NXOR b) to represent "a|b".

3. Variable Length Key

Some regular expressions have variable length, such as Kleene Star (*), which means matches zero or more occurrences of the regular expression. So when there are random numbers of characters in the key, we have two methods to deal with.

The first is that we can find the fixed-length prefix of the key as a hash value, and some of the packets will be excluded using the hash value as a new key. Thats because a packets payload does not match a part of the key must not match the whole key. And we can prove that we can find at least one-character long exact prefix in any regular expression because the first character in every regular expression must not be marked by a Kleene Star. For example, a regular expression "$a*bc+d$" is equal to "$bc+d$", for the "$a*$" is meaningless, and we only need to check bc, which is the max exact prefix, as the hash value to perform string matching over packets payload. When the first match is found, further match checking is needed. But we only need to check the reminder part of the regular expression instead of the whole one to reduce the states of DFAs. So we construct a simple DFA for the remainder part of the regular expression, and use the result created by the hash value to perform further checking. Only the strings follow the matched hash value need further checking and finish this possible match checking when the start state in the DFA is active. So only a small part of the packets payload goes into the DFA when a packets payload need further checking.

We can also build a big DFA engine to combine all these variable length keys instead of several small DFAs. But experimental results show that this method is worse than the previous one, thats because: (1) Building several small DFAs is faster than building a big combined DFA; (2) After checking the prefix, only a small part of the packets payload need further checking and among them only a small part of a packets payload goes into the DFA; (3) Some small DFAs can be reused by other packets and easy to be stored in Cache.

The second method uses counters instead of DFAs. This avoids using DFAs but need to preprocess the regular expressions. The preprocessor divides the regular expressions in to several pieces to separate the exact length parts and variable length parts. For example, abc+d" can be divided into three pieces: "abc", "c*" and "d". Using the method discussed above When checking "abc" and "d", and the only difference is how to check"c*". Our engine checks "c*" matching after checking the string "abc", and the difference is that the result of the NXOR operation should be stored in a Temporary Window (TW) instead of updating Current Window(CW), and the three windows (PW, CW and NW) stalled during this matching. Then the matching engine compares character d using NXOR operation and the result is called T. The i th bit of the CW can be updated using the following equation:

$$CW(i) = T(i)\&(CW(i)|(TW(i+1)\&CW(i+1)) |(TW(i+2)\&TW(i+1)\&CW(i+2))|...|(TW(L-1)\&...\& TW(i+1)\&CW(L-1))) \quad (1)$$

We need to use counters to record the length of character "c" between "abc" and "d" if the regular expressions have length require of "c", such as matching N or more occurrences of the character "c". Table 1 shows the supported regular expressions (RegExp).

4. Comparison with Boyer-Moore string search algorithm

The BoyerCMoore string search algorithm is a particularly efficient string searching algorithm, and it has been the standard benchmark for the practical string searching. Usually, the Snort use Boyer-Moore string search algorithm to perform pattern matching, but this algorithm needs to preprocess each key and needs to check one character each time even though can skips over some of the characters in the payload. Furthermore, this algorithm gets faster as the key being searched for becomes longer, but most keys are short in the Snort rules.
Compare with Boyer-Moore string search algorithm, our algorithm have following advantages:
a) Our algorithm does not need to preprocess each key.
b) The length of packets payload shifted in is fixed (one block), but the length of packets payload shifted in for Boyer-Moore algorithm depends on the previous checking result.
c) The above two "static" properties make our algorithm more easy to be implemented in hardware, such as in FPGA.
d) The length of packet shift in is fixed instead of depends on the previous checking result, and this makes our algorithm more easy to check multiple keys in parallel.
e) Our algorithm checks the characters in parallel instead of one by one. Our algorithm is more scalable.

For Boyer-Moore string search algorithm, the worst-case to find all occurrences in a payload needs approximately $3*N$ comparisons, and N is the number of characters in the payload. For our algorithm, the worst-case to find all occurrences in a payload needs approximately N comparisons. In the worst-case situation, the length of the key is equal to the length of the block and all the characters except the last one of the key is the same as content of each block, for example, the key is "abc", the length of block is 3 and the payload is "abdabeabdabf...".

5. Implementation and Simulation

We implement our algorithm in Snort 2.8.3.1., and use the intrusion detection evaluation data set from the MIT DARPA for performance comparison. We compare our algorithm with the original detection engine of Snort, the start-of-the-art NFA-based regular expression matching algorithm and DFA-based algorithm proposed in [19] . All the experimental results reported were obtained on PCs with 3 GHz CPU and 3 GB memory. The results are shown in Table II. Our algorithm is 17 times faster than the NFA-based algorithm and consumes a half of the memory used by the NFA-based algorithm.

Compared to DFA-based algorithm, our algorithm has approximate 7 times performance improvements and reduces the memory usage to one-ninth.

6. Conclusion and future work

In this paper, we propose a new algorithm to accelerate the searching speed in NIDS and we implement our algorithm in Snort. Our algorithm based on the fact that normal data stream rarely matches any virus signature and different packets need to check different keys. Compared with Boyer-Moore string search algorithm, our algorithm has many advantages in some special applications. Our algorithm does not need preprocessing and can check multiple characters in parallel. Experimental results show that our implementation is faster than original NFA/DFA based algorithms to deal with the same real packet traces while consuming less memory. Even though some papers have proposed many methods to improve the NFA/DFA algorithms to meet the requirements of high throughput and low memory consumption, our algorithm solves these problems through a novel way. Further, based on analysis discussed in section 4, our algorithm is easy to be implemented in FPGA, so configuring our algorithm onto ASIC/FPGA to further increase the throughput are being worked out.

Table 1 Syntaxes currently supported by our Regular Expression Engine

Syntax	Description	Support
abc123..	All ASCII alphanumerics, match a single occurrence of themselves	Well
.	Dot: Matches any character other than .	Well
RegExp	Groups regular expressions, so syntaxes can be applied.	Well
RegExp1RegExp2	Concatenation: Regular Expression 1, followed by Regular Expression 2.	Well
RegExp1/RegExp2	Union: Regular Expression 1 OR Regular Expression 2.	Well
RegExp*	Kleene Star: Matches zero or more occurrences of the regular expression.	Not well, need additional processing
RegExp+	Plus: Matches one or more occurrences of the regular expression.	Not well, need additional processing
RegExp?	Question Mark: Matches at most one occurrence of the regular expression.	Well, but need additional processing
RegExpN	Exactly matches N occurrences of the regular expression.	Well
RegExpN,	At least: Matches N or more occurrences of the regular expression.	Not well, need additional processing
RegExp,M	At most: Matches M or less occurrences of the regular expression.	Not well, need additional processing
RegExpN,M	Between: Matches if the regular expression occurs between N and M times.	Not well, need additional processing
NRegExp	Anchor: Matches the regular expression at position N from the beginning of the payload.	Well

Table 2 Comparison between different algorithms

Algorithms	Average Memory Consumption(KB)	Average Throughput (Gbps)
NFA-based	1763	0.124
DFA-based	7532	0.319
Our approach	827	2.108

7. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (61103113) and BJNSF (KZ2010 10009008).

References

[1] Liu Yanguo Michael. Properties for Security Measures of Software Products. Applied Mathematics & Information Sciences.vol: **1**, (2007) 129-156.

[2] Pastrana, S. Orfila, A. Ribagorda. A Functional Framework to Evade Network IDS, 2011 44th Hawaii International Conference on System Sciences (HICSS), (2011)1 C 10.

[3] Daeseob L. and Dongho W. A Study on Security Management Service System for Wireless Network Environment, Applied Mathematics & Information Sciences , VOL: **6**, (2012) 209-220.

[4] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context, CCS '03 Proceedings of the 10th ACM conference on Computer and communications security, (2003)262-271.

[5] J. Levandoski, E. Sommer, and M. Strait. Application Layer Packet Classifier for Linux. http://l7- filter.sourceforge.net/.

[6] SNORT Network Intrusion Detection System. http://www.snort.org.

[7] "Bro Intrusion Detection System". http://bro-ids.org /Overview.html.

[8] J. William and Will Eatherton, An encoded version of regex database from cisco systems provided for research purposes.2005.

[9] Y. Sun, Victor Valgenti and Min Sik Kim, Hierarchical NFA-Based Pattern Matching for Deep Packet Inspection, In Proceeding of ICCCN11. (2011) 1-12.

[10] Y. Sun and M. S. Kim, DFA-based Regular Expression Matching on Compressed Traffic. In Proceeding of IEEE International Conference on Communications (ICC'11). (2011)1-5. June

[11] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In Proc. of SIGCOMM '06. (2006)339-350.

[12] Y. Sun, H. Liu, V. Valgenti and M. S. Kim, Hybrid Regular Expression Matching for Deep Packet Inspection on Multicore Architecture. In Proceeding of the International Conference on Computer Communication Networks, (ICCCN'10). (2010)1-7. Aug.

[13] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata, In IEEE Symposium on Security and Privacy, (2008)187-201.

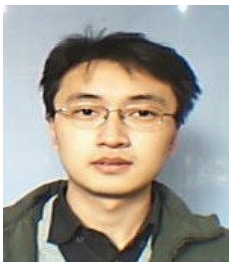
[14] I. Bonesana, M. Paolieri, and M.D. Santambrogio. An adaptable FPGA-based System for Regular Expression Matching, Design, Automation and Test in Europe, (2008)1262 C1267.

[15] P. Marco, B. Ivano, S. Marco D. ReCPU: A parallel and pipelined architecture for regular expression matching, IFIP International Conference on Very Large Scale Integration, (2007)19 - 24. Oct.

[16] N. Yamagaki, R. Sidhu, S. Kamiya, "High-speed regular expression matching engine using multi-character NFA", Inter-

national Conference on Field Programmable Logic and Applications. (2008)131-136. Sept.
[17] W. Zhang, Y. Xue, D.S. Wang, T. Song, "A multiple simple regular expression matching architecture and coprocessor for deep packet inspection", Asia-Pacific Conference on Computer Systems Architecture. (2008) 1-8.
[18] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Languages and Computability, 2nd Edition, Addison-Wesley, Nov. 2000.
[19] D. Ficara, S. Giordano, G. Procissi, etc.. An improved DFA for fast regular expression matching, SIGCOMM Comput. Commun. Rev, (2008)29 C 40.

Mengmeng Zhang received the MS degree in Electrical Engineering from Beijing Jiaotong University in 1996, and the PhD degree in Communication Engineering from University of Science and Technology of Beijing in 2011. He is currently an associate professor in North China University of China. His research interests are in the areas of network security and image video coding.



Yan Sun received his B.S. degree in Applied Physics in 2005 from University of Science and Technology, Beijing, China, the M.S. degree in 2008 in Microelectronics from University of Science and Technology of China, and PhD in Computer Science from Washington State University in 2011. He is currently a Staff Scientist in Broadcom at Santa Clara, CA. USA. His research interests include Network Security, High-Performance VLSI systems and computer architectures.

Jingzhong Wang received his MS degree in Computer Science and Technology form Inner Mongolia University of Technology, Beijing, China. He is currently a professor in North China University of China. His research interests ere in the areas of computer information security and digital image processing.