# Enhancement of Wu-Manber Multi-pattern Matching Algorithm for Intrusion Detection System

Soojin Lee and Toan Tan Phan(✉)

Cyber Security Lab, Department of Computer Science and Engineering,
Korea National Defense University, Nonsan, Republic of Korea
cyberkma@gmail.com, pttoan.it@gmail.com

**Abstract.** Intrusion Detection System (IDS) is a monitoring system that is the most commonly used today. IDS monitors and analyzes network traffic to detect and prevent malicious behaviors. The main process of IDS is pattern matching, which typically accounts for about 70% of IDS processing time [9]. Wu-Manber [11] is one of the fastest pattern matching algorithms [3] which is commonly used in IDSs. It uses hash techniques to build the hash tables based on the shortest patterns. However, the difference between patterns often degrades the efficiency of the algorithm. In this paper, we propose an improved Wu-Manber algorithm that reduces dependence on the shortest patterns by combining Bloom filters. The experimental results show that our algorithm reduces the matching time by 10% in the worst case and 78% in the best case compared to the original Wu-Manber algorithm, and also reduces the memory usage by 0.3%.

**Keywords:** Intrusion detection systems · Pattern matching · Network security
Wu-Manber · Bloom filters

## 1 Introduction

Intrusion Detection System is a monitoring system that is the most commonly used to monitor and analyze network traffic in an effort to detect and prevent malicious behaviors. IDSs are classified based on detection approach consisting of signature-based and anomaly-based. The signature-based detection uses a set of rules (or signatures) to detect the intrusions, while the anomaly-based detection uses the machine learning techniques to detect anomaly behaviors (such as zero-day attacks). Snort is an open signature-based intrusion detection system that is the most commonly used.

Several algorithms are used in IDS such as Aho-Corasick [1] (AC), Boyer-Moore [6] (BM), and Wu-Manber [11] (WM), in which Wu-Manber is one of the fastest algorithms on average. It is a hash-based algorithm which uses the concept of "shift bad characters" from BM algorithm [6] to get the maximum shift distance in case of pattern mismatch. WM [11] uses only first m characters of each pattern, with m is the length of the shortest pattern, to build three hash tables: SHIFT, HASH, PREFIX. These hash tables then are used in the searching phase. However, according to fact survey results of the currently latest Snort 2.9 database, the shortest pattern is 3 characters and the


P. Cong Vinh et al. (Eds.): ICCASA 2017/ICTCC 2017, LNICST 217, pp. 69–78, 2018.
https://doi.org/10.1007/978-3-319-77818-1_7

longest pattern is 516 characters, and 51% of all the patterns is greater than 9 characters. If the algorithm is run on only the first m characters of each pattern (in this case m is 3), then it will take a lot of time to match remaining characters, assuming the first m characters matched. Most previous research about the improvements of WM did not address this issue, and they were only interested in the pattern prefix part.

In this paper, we propose an enhanced Wu-Manber algorithm that focuses on the remaining characters of the patterns (called the pattern *suffix* part). We use Bloom filters instead of the PREFIX table to evaluate quickly a few characters of the *suffix* part that are capable of matching the incoming string before searching in the HASH table. The experimental result shows that our algorithm skipped a significant number of unnecessary accesses to the HASH table. Therefore it brings high-performance improvement in terms of time and memory usage compared to WM algorithm. Section 2 reviews the related background to better understand the problem. Section 3 surveys the related works. Section 4 describes the detailed structure of the proposed algorithm. Section 5 includes our experimental results. Finally, Sect. 6 concludes this paper.

## 2 Background

### 2.1 Snort and Pattern Matching Algorithms

Snort is a signature-based intrusion detection system that is the most commonly used in defense systems today. It is an open source software-based tool using a rule-driven language, where each rule consists of headers and options. The headers consist of the protocols, IP addresses, port, etc., The option fields contain the messages, contents, sid, etc., in which the contents are the signatures of attacks that were collected from the monitoring systems. The currently latest Snort 2.9 version includes 44 rule groups with a total of about 17476 rules consisting of dos, dns, ftp, web, icmp, trojan, etc., Fig. 1 represents a sample Snort rule. Several pattern matching algorithms are used in Snorts as Boyer-Moore (BM), Aho-Corasick (AC), and Wu-Manber (WM). These algorithms are classified into either single or multiple pattern matching. The single pattern matching algorithms simply match only one pattern at the moment as BM algorithm. While the multi-pattern matching algorithms, such as AC and WM, match multiple patterns at the moment. WM is a hash-based algorithm which its average performance is better than AC.

```
alert udp $EXTERNAL_NET any -> $HOME_NET 5093 (msg:"ET DOS
Possible Sentinal LM Amplification attack"; dsize:6; content:"|7a 00 00 00 00
00|"; sid:2021172; rev:1;)
```

**Fig. 1.** Sample rule of Snort

### 2.2 Wu-Manber Algorithm

Wu-Manber [11] is a hash-based algorithm using "shift bad characters" technique in order to get the maximum shift distance when it finds a mismatch. The database of WM is a set of same length patterns which each pattern is first m characters of original pattern (where m is the length of the shortest pattern [11]). Each pattern is divided into two parts: *prefix* and *block*. WM uses three hash tables: SHIFT, HASH, and PREFIX. SHIFT table contains the maximum shift distance (called *shift-value*) of each block in pattern set. HASH is a hash table of the blocks that its *shift-value* is zero. Each index of HASH is a linked list of all patterns of the same block. PREFIX is a hash table of the *prefixes*. PREFIX is used to quickly check the appearance of a pattern in the linked list of HASH table. WM consists of two main phase: preprocessing and matching phase. In preprocessing phase, the algorithm constructs three hash tables consist of SHIFT, HASH, and PREFIX. In matching phase, WM scans on the incoming string to get blocks. The blocks then are hashed to get results by the same hash functions of the preprocessing phase. The results then are searched in the HASH table for retrieving the corresponding entities. The detailed processes are described as follow:

(a) **Preprocessing phase**

First, WM finds the minimum length m of the shortest pattern. The algorithm then uses only the first m characters for each pattern (known as a representative pattern) to build three hash tables. To construct SHIFT table, each pattern of size m is divided into multiple substrings of size B, called X. The substrings X then are computed the corresponding *shift-value*s and are stored in the SHIFT table. The shift-values are computed as following: For each pattern P*i* to compute the shift-value for each substring X, there are two possibilities. If X does not appear in any pattern, then its shift-value is (m − B + 1) characters. This value also is the default shift-value to build the table [11]. The second case, if X appears in some of the patterns, when the rightmost position of X in the pattern, called q, is located, and the *shift-value* is (m − q).

The HASH table contains only the blocks that its shift-value is zero. Each index of HASH is a linked list of (full) patterns of the same block. The PREFIX table is used to quickly check the appearance of a pattern in the linked list of HASH table. Experimentally, a block size of 2 or 3 is a favorable choice [11]. Figure 2 shows that the SHIFT, HASH, and PREFIX are constructed from a pattern set P = {GetInfo, passwd, password=, sicken, ficken}, where m is 6 and B is 3.

(b) **Matching phase**

In this phase, the algorithm uses a sliding window of size m to scan on the incoming string to get blocks. Each block then is mapped into the SHIFT table to get the *shift-value*. If the *shift-value* is greater than zero, then the window is slid by the length of *shift-value* and repeats the process. Otherwise, if the *shift-value* is zero, then the substring of the window might match to one of the patterns. When the HASH and PREFIX tables are checked to determine the matched pattern. Table 2 represents the matching phase.

**Table 1.** SHIFT table

| Block | Shift | Block | Shift |
|---|---|---|---|
| Get | 3 | swo | 0 |
| etI | 2 | sic | 3 |
| tIn | 1 | ick | 2 |
| Inf | 0 | cke | 1 |
| pas | 3 | ken | 0 |
| ass | 2 | fic | 3 |
| ssw | 1 | others | 4 |
| swd | 0 | | |

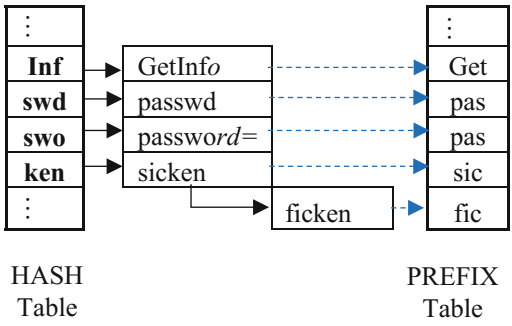


**Fig. 2.** HASH and PREFIX table

### 2.3 Bloom Filter

Bloom filter [5] is a hash vector representing for multiple strings which can easily exclude negative matches. It uses less memory space but quickly queries for membership. The Bloom filter computes several hashing functions on each string to get hash results. The results then are indexed into the Bloom vector of size m. The bits at the index position are set to 1. To check the existence of a new string in the string set, the Bloom filter uses the same hash functions to compute the hash results of this string and then checks the corresponding bits in the Bloom vector to determine whether the new string exists or not. If 100% of the hash bits are set, then the new string might exist. Otherwise, the new string does not exist. Bloom filter has no false negatives but has a probability of false positives.

**Table 2.** Matching phase

| Step | LoggedGetInforootpassword=toor | Shift | Output |
|---|---|---|---|
| 1 | **Logged***GetInforootpassword=toor* | 4 | |
| 2 | Logg**edGetI***nforootpassword=toor* | 2 | |
| 3 | Logged**GetInf***orootpassword=toor* | 0 | GetInfo |
| 4 | LoggedG**etInfo***rootpassword=toor* | 4 | |
| 5 | LoggedGetIn**foroot***password=toor* | 4 | |
| 6 | LoggedGetInforo**otpass***word=toor* | 2 | |
| 7 | LoggedGetInforoot**passwo***rd=toor* | 0 | password= |
| 8 | LoggedGetInforootp**asswor***d=toor* | 4 | |
| 9 | LoggedGetInforootpassw**ord=to***or* | 4 | |
| 10 | LoggedGetInforootpassword=**toor** | End | |

## 3 Related Work

Based on WM, many variants were proposed to overcome the limitation of WM algorithm. In 2009, an improved WM algorithm based on address filtering named as AFWM was proposed by Zhang et al. [4]. Based on the address pointers of the patterns, the Prefix table in AFWM is utilized to filter the linked list of possible matching patterns. The patterns in the linked list are sorted in ascending order according to the address pointers. The advantage of the address filtering algorithm is that it avoids traversing the whole linked list.

In 2015, another improved WM algorithm based on a data structure of AVL tree is implemented by Bhardwaj and Garg [10]. The improved algorithm eliminates the Prefix table, construct two Shift table and uses nonlinear data structure of AVL tree. The results show that the algorithm has better performance as compared to WM and the variants. However, the improvement is no efficiency of memory usage due to the use of two SHIFT table. Moreover, experimental data is no generality for network attacks.

In 2016, new modified WM algorithm based on Bloom filters is implemented by Aldwairi et al. [2, 3]. The algorithm uses Bloom filter instead of the PREFIX table of WM to exclude the unnecessary HASH table searches. The Bloom filters are performed by computing the hash functions on the prefix part of the patterns. In searching phase, the Bloom filter computes the same hash functions for the prefix part of each window, when the *shift-value* of the block is zero, and checking the corresponding bits in the Bloom vector to determine whether the substring might exist or not. If the hash bits appear, then the HASH table will be searched. Otherwise, the HASH table is skipped.

Generally, most previous approaches [7, 8, 12] only focused on the prefix part which is too small than the remaining characters of the patterns. Therefore they could not achieve high-efficiency.

## 4 Proposed Algorithm

We are interested in the remaining characters, and called suffix, in each pattern. Accordingly, a pattern consists of three parts: *prefix*, *block*, and *suffix*, as shown in Fig. 3. As explained above, WM uses only the first m characters for each pattern, including *prefix* and *block* to build three hash tables: SHIFT, HASH, and PREFIX. After searching in the SHIFT table, if the *shift-value* is zero, the HASH table is accessed to the corresponding position of the block. In fact, however, the probability of finding the matched patterns in a linked list of the patterns is very low (around 5%). There is about 95% of the patterns do not match, while they are still accessed in the HASH table when the *shift-value* is zero. The PREFIX table is used to reduce the unnecessary accesses to the HASH table. However, even if the *prefix* is found, comparing the remaining characters (*suffix*) in original patterns also take too long time. Moreover, in the worst case, when all the *prefixes* are the same (or not exist), there is almost no hope for performance enhancement.

Our proposed algorithm uses the Bloom filters instead of the PREFIX table in order to achieve higher performance. The Bloom vector selection can ensure the efficiency of the memory and the hash functions. The detailed processes of two phases are described as follows.

### 4.1 Preprocessing Phase

During constructing the HASH table, we insert the Bloom vectors into each index of the HASH table. Each Bloom vector consists of 16 bits and is computed by two hash functions, called *pre-hash* and *suf-hash*. When a pattern is inserted into a linked list of the HASH table: First, the *pre-hash* function hashes the *prefix* part of the pattern. The result then is modulo 5 to get the final result, called *h1*. Second, the *suf-hash* function hashes first *m* characters of the pattern *suffix* part (*from m + 1 to 2 m*). The result *r* then is modulo 11 and plus 5 as the following equation: ((r mod 11) +5) to get the final result, called *h2*. As Fig. 4 represents formatting of a Bloom vector. Finally, the bits of the Bloom vector corresponding to *h1, h2* are set to 1. There are *k* Bloom vectors corresponding to the size of the HASH table. Each Bloom vector represents a linked list of the patterns in the HASH table. Figure 5 represents a new HASH table structure.

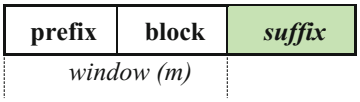


**Fig. 3.** Pattern format

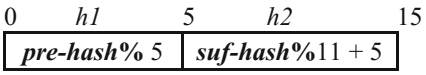


**Fig. 4.** Bloom vector format

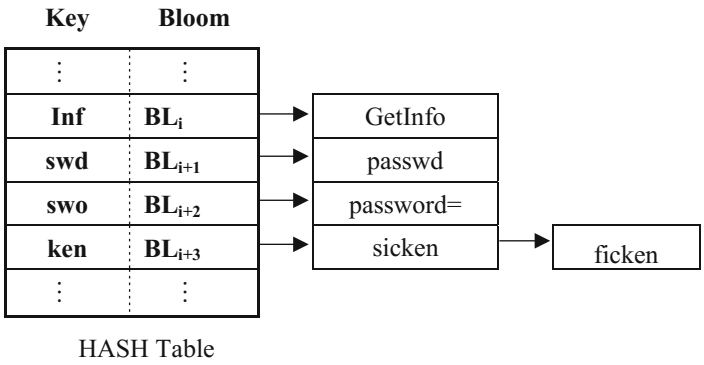


**Fig. 5.** New HASH table structure

### 4.2 Matching Phase

Similar to original WM, our algorithm also uses a sliding window of size m to scan on the incoming string T to get blocks. Each block then is mapped into the SHIFT table to get the *shift-value*. If the *shift-value* is greater than zero, then the window is slid by the length of *shift-value* and repeats the process. Otherwise, if the *shift-value* is zero, then the substring of the window might match to one of the patterns. After then, the *pre-hash* and *suf-hash* functions are used to hash the substring. The *pre-hash* computes the *prefix* of the block and *suf-hash* computes the next m characters following the block. Two hash results will then be mapped into the Bloom vector at the corresponding index of the HASH table. The *pre-hash* result will be searched in the first 5 bits of the vector and the *suf-hash* result is the remaining 11 bits. There will be three cases as follow:

- The *pre-hash* result does not appear (bit 0): ignoring all the patterns.
- The *pre-hash* result appears (bit 1), but the *suf-hash* result does not appear (bit 0): Only search the patterns of size less than 2 m characters, ignoring other patterns.
- Both *pre-hash* and *suf-hash* results appear (bit 1): searching all the patterns.

Then sliding the window by 1 character and repeating the matching process until the end of the string.

## 5 Experiment and Performance Evaluations

Our main goal is the improvement of WM multi-pattern matching algorithm of the Snort. Therefore, our algorithm is built on the database of the Snort and implemented in C++ using Microsoft Visual Studio 2013 as IDE. The performance evaluations were done by comparing to original WM algorithm in the aspects of the preprocessing time, matching time, memory usage and the number of the HASH table accesses. Our algorithm is described as Bloom-Wu-Manber (BWM) algorithm.

### 5.1 Experimental Data

Our experimental database is the rules set of Snort 2.9. It consists of 17476 rules dividing into 44 groups, including dos, dns, ftp, web, icmp, trojan, etc., Each rule consists of *headers* and *options*. The attack signatures are the strings following keyword "*content*" of the *options*. There are totals of 40767 patterns, in which the shortest pattern is 3 characters, the longest pattern is 516 characters and 51% of the patterns are greater than 9 characters. In the case of the patterns that are shorter than size B (with B = 3) of the block, we concatenate that pattern with the previous pattern from the same rule by one space character as a delimiter. The matching process of a rule with an incoming packet is performed on both the *headers* and *options*. In our experiment, we assume that the headers were already matched, and therefore our algorithm only matches the signatures of the *options*.

We use four sample attack payload datasets that are often used to evaluate the IDSs. It consists of DEFCON20 Capture the Flag (CTF) game packet traces released in 2012, the Information Security Talent Search (ISTS12) in 2015, the all_attack_win and all_attack_unix files of FuzzDB (is like an application security scanner). Table 3 describes Snort database and the experimental datasets.

**Table 3.** Experimental and test dataset

| Snort rule database | | Test data | |
|---|---|---|---|
| Number of rules | 17476 | *Sample set* | *Payloads* |
| Number of patterns | 40767 | all-attacks-unix | 510 |
| Max_length pattern (ch) | 516 | all-attacks-win | 530 |
| Min_length pattern (ch) | 3 | DEFCON20 | 3644 |
| Patterns greater than 9 chars | 51% | ISTS12_2015 | 228030 |

## 5.2 Experimental Results

As the time and memory of the program slightly change in each running time, we executed each algorithm (consist of WM and BWM) 100 times on each test dataset to get the average results. To evaluate the effect of the Bloom filter, we compared with each other for the number of the HASH table accesses, processing time and memory usage of each algorithm. A more efficient algorithm should have fewer the HASH accesses, using fewer system resources while ensures better detection result. The detailed evaluation results are shown in the Tables 4, 5, 6. Table 4 shows the comparison results of the number of the HASH table accesses. On average, the number of the HASH table accesses of our algorithm is fewer than WM by 13.45%. However, there is a big difference between the minimum and maximum access counts due to the conflict of the hash functions in the Bloom filter.

**Table 4.** Number of HASH accesses

| Test Data | Found patterns | | HASH table accesses | | |
|---|---|---|---|---|---|
| | WM | BWM | WM | BWM | Performance |
| all-attacks-win | 39364 | 39364 | 59186 | 51980 | **12.18%** |
| all-attacks-unix | 47540 | 47540 | 72453 | 66967 | **7.57%** |
| DEFCON20 | 25032 | 25032 | 434294 | 332972 | **23.33%** |
| ISTS12_2015 | 2.0E+08 | 2.0E+08 | 3.5E+08 | 3.1E+08 | **10.74%** |

Table 5 shows that our algorithm has better processing time compared to WM. On average, the preprocessing time of our algorithm is reduced by 10% and the matching time is reduced by 9.2% compared to WM. Table 6 shows that memory usage of our algorithm is also reduced by 0.34%.

**Table 5.** Preprocessing time and matching time

| Test data | Preprocessing time (ms) | | | Matching time (ms) | | |
|---|---|---|---|---|---|---|
| | WM | BWM | Performance | WM | BWM | Performance |
| all-attacks-win | 448 | 406 | **9%** | 139 | 124 | **10.79%** |
| all-attacks-unix | 465 | 406 | **13%** | 163 | 146 | **10.43%** |
| DEFCON20 | 547 | 500 | **9%** | 980 | 847 | **13.57%** |
| ISTS12_2015 | 563 | 500 | **11%** | 1616750 | 1584380 | **2.00%** |
| Average: | | | **10%** | | | **9.20%** |

**Table 6.** Memory usage

| Test data | Memory usage (KB) | | |
|---|---|---|---|
| | WM | BWM | Performance |
| all-attacks-win | 85012 | 84688 | **0.38%** |
| all-attacks-unix | 85040 | 84712 | **0.39%** |
| DEFCON20 | 84957 | 84678 | **0.33%** |
| ISTS12_2015 | 85172 | 84940 | **0.27%** |
| Average: | | | **0.34%** |

The above experimental results are based on the real dataset of Snort 2.9. As the size of the shortest pattern is equal to the size B of the block. Therefore the prefix does not appear, and this is also the worst case. In order to extend the experimental results, we modify the pattern dataset by gradually increasing the size of the shortest patterns from 4 to 6 characters. In each case, we use the test dataset of *all-attacks-win* to compare the corresponding parameters of both algorithms. Table 7 shows that our algorithm is more efficient than WM in all cases. Especially, the matching time heavily depends on the size m. When the larger the size m is, the faster the matching time is. On average, the matching time of our algorithm is reduced by 66%, and in the best case (when m is 6) it can be reduced by 78.49% compared to WM.

**Table 7.** Algorithm performance depend on the size of m

| Size of m (ch) | Num of payloads | Found patterns | Hash table access | | Preprocessing time (ms) | | Matching time (ms) | | Memory usage (KB) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | WM | BWM | WM | BWM | WM | BWM | WM | BWM |
| 3 | 40767 | 39364 | 59186 | ↓ 51980 | 448 | ↓ 406 | 139 | ↓ 124 | 85012 | ↓ 84688 |
| 4 | 37465 | 14292 | 55292 | ↓ 52947 | 420 | ↓ 375 | 224 | ↓ 109 | 84440 | ↓ 84180 |
| 5 | 32724 | 13073 | 36008 | ↓ 29447 | 397 | ↓ 353 | 244 | ↓ 073 | 83632 | ↓ 83388 |
| 6 | 29715 | 10595 | 24115 | ↓ 21772 | 391 | ↓ 343 | 265 | ↓ 057 | 83088 | ↓ 82836 |

## 6 Conclusions

In this paper, we propose an enhanced Wu-Manber algorithm for intrusion detection systems. Our algorithm uses the Bloom filters instead of the PREFIX table to reduce the number of unnecessary HASH table accesses. We focus on the suffix of the pattern because its size is very large compared to the first m characters of the pattern. The experimental results show that our algorithm is more efficient than WM in both time and memory usage. More specifically, the matching time is reduced by 10% in the worst case and reduced by 78% in the best case compared to WM. The memory usage is also reduced by 0.3% on average.

## References

1. Aho, A., Corasick, M.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**, 333–340 (1975)
2. Aldwairi, M., Al-Khamaiseh, K.: Exhaust: optimizing Wu-Manber pattern matching for intrusion detection using bloom filters. IEEE (2015)
3. Aldwairi, M., Al-Khamaiseh, K., Alharbi, F., Shah, B.: Bloom filters optimized Wu-Manber for intrusion detection. J. Digit. Forensics Secur. Law **11**(4), Article 5 (2016)
4. Zhang, B., Chen, X., Pan, X., Wu, Z.: High concurrence Wu-Manber multiple patterns matching algorithm. In: Proceedings of the International Symposium on Information Process, p. 404 (2009)
5. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13** (7), 422–426 (1970)
6. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (1977)
7. Kacha, C., Shevade, K.A., Raghuwanshi, K.S.: Improved Snort intrusion detection system using modified pattern matching technique. Int. J. Emerg. Technol. Adv. Eng. **3**(7), 81–88 (2013)
8. Yang, D., Xu, K., Cui, Y.: An improved Wu-Manber multiple patterns matching algorithm. In: The 25th IEEE International Performance, Computing, and Communications Conference (IPCCC), pp. 680–686 (2006)
9. Antonatos, S., Anagnostakis, K., Markatos, E.: Generating realistic workloads for network intrusion detection systems. SIGSOFT Softw. Eng. Notes **29**(1), 207–215 (2004)
10. Bhardwaj, V., Garg, V.: Efficient Wu Manber string matching algorithm for large number of patterns. Int. J. Comput. Appl. **132**(17), 29–33 (2015)
11. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical report TR94-17. University of Arizona at Tuscon (1994)
12. Zhang, W.: An improved Wu-Manber multiple patterns matching algorithm. In: Proceedings of the 2016 IEEE International Conference on Electronic Information and Communication Technology (ICEICT 2016), pp. 286–289 (2016)