Hindawi
Security and Communication Networks
Volume 2019, Article ID 8467081, 22 pages
https://doi.org/10.1155/2019/8467081

Research Article

Efficient and Transparent Method for Large-Scale TLS Traffic Analysis of Browsers and Analogous Programs

Jiaye Pan, Yi Zhuang, and Binglin Sun

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 200016, China

Correspondence should be addressed to Yi Zhuang; zy16@nuaa.edu.cn

Received 3 April 2019; Revised 16 August 2019; Accepted 20 September 2019; Published 27 October 2019

Guest Editor: Surya Nepal

Many famous attacks take web browsers as transmission channels to make the target computer infected by malwares, such as watering hole and domain name hijacking. In order to protect the data transmission, the SSL/TLS protocol has been widely used to defeat various hijacking attacks. However, the existence of such encryption protection makes the security software and devices confront with the difficulty of analyzing the encrypted malicious traffic at endpoints. In order to better solve this kind of situation, this paper proposes a new efficient and transparent method for large-scale automated TLS traffic analysis, named as hyper TLS traffic analysis (HTTA). It extracts multiple types of valuable data from the target system in the hyper mode and then correlates them to decrypt the network packets in real time, so that overall data correlation analysis can be performed on the target. Additionally, we propose an aided reverse engineering method to support the analysis, which can rapidly identify the target data in different versions of the program. The proposed method can be applied to the endpoints and cloud platforms; there are no trust risk of certificates and no influence on the target programs. Finally, the real experimental results show that the method is feasible and effective for the analysis, which leads to the lower runtime overhead compared with other methods. It covers all the popular browser programs with good adaptability and can be applied to the large-scale analysis.

1. Introduction

Currently, the incidents of malware attack occur so frequently as to cause the serious loss of data and property to internet users. Malicious code has presented new forms such as ransomware, phishing, and coin mining. Web browsers are the important sources of malware to infect the target computers. For example, users download and install software bundled with malicious codes from the third-party website, or users encounter phishing attacks and access the fake page, or the web browser loads a web page with vulnerability exploitation codes and triggers the infection of malware [1]. Meanwhile, the incorrect configuration of legitimate applications may also cause the exfiltration of privacy data. For terminal users, web browser is an important application in their work and life. Consequently, it is crucial to inspect the content of web pages in order to create a secure internet environment for computer users.

Moreover with the development and popularity of the Secure Socket Layer (SSL) protocol [2] and its subsequent version Transport Layer Secure (TLS) [3], lots of websites and client applications adopt the HTTPS protocol instead of HTTP [4]. It is based on TLS which can prevent the transmission of data from advertising hijacking and packet manipulation. This also becomes the barrier of security information and event management (SIEM) systems. Although the TLS protocol is very secure in theory, there are many kinds of problems in its practical uses. The software which implements the protocol may have vulnerabilities, such as those of *OpenSSL* [5, 6], and some insecure algorithms and cipher suites may cause the cipher text to be cracked [7, 8, 9]. Moreover, there may be Man-In-The-Middle (MITM) attacks in the authentication process, for reasons such as counterfeit certificates [10], certificate verification bugs or lack of security consciousness [11, 12]. Actually the problems led by improper deployment of TLS protocol are more complicated than we thought, especially in the browser application and HTTP protocol [13, 14]. The security of TLS protocol is continuously improved in the confrontation between attack and defense, which has been

shown in the draft of TLS version 1.3 [15]. Meanwhile, there are many enhanced mechanisms for TLS applications [16], some of which are widely used in practice, such as HTTP Strict Transport Security (HSTS) [17], Public-Key Pinning Extension (PKPE) [18], Certificate Transparency (CT) [19], and so on.

In this situation, the firewall based on packet filtering cannot make deep analysis on the packet content. One better solution is to choose, model, and classify the plain information in the TLS connections with training, such as the handshake fingerprints [20] and then to pick out the malicious traffic in the real environment [21]. This method can only discover the abnormal TLS connections, but fails to identify the web page which contains malicious payload because the packet is not decrypted. Another solution is to interpose a TLS proxy in the communication, which is more general in security software and devices. First, adding a trusted Certificate Authority (CA) in the system and configuring the network proxy, when the browser makes a TLS handshake, it will generate a specific certificate for the request domains, and it is signed with the installed certificates beforehand [22]. In this case, trusted and auditable connections will be established. Actually, this method is very suitable for web browsers but not general since many software implement the TLS protocol in a custom manner, also with the certificate or public key pinning. Besides, proxy will delay the network transmission which can be detected by both the client and server, and the incorrect proxy configuration may cause serious security problems [22, 23]. Function hook is also an approach to network data analysis, but it would interrupt the workflow of the program. Additionally, hooks will meet the challenge of program version diversity. There are also some good tools for manual analysis which depend on proxies, such as *Fiddler* and *mitmproxy* [24, 25]. *Wireshark* is also helpful when the master key of TLS session can be obtained and imported [26, 27], but it depends on the special configuration of the target program and also lacks the full automation. The plugins of browsers can also help to analyze the web page content, which do not have the capability of raw packet manipulation and have compatibility issues. From the perspective of attacks, the aggressive methods have limitations, and they will be ineffective due to vulnerability patches. Nowadays, in that the security of TLS is improved, the attack approach cannot make a persistent traffic analysis. It is common that the browser triggers the vulnerability and malware installation when users are working on the internet; hence, traffic analysis needs to be deepened in order to block the malicious code ahead.

In general, the existing methods are inefficient for the large-scale analysis in real time, which also depend on the modification of the target program or system. The analysis is limited to the decryption and isolated from the data analysis in the system perspective. In accordance with the above situations, we deeply research the mechanism of different browsers and analogous programs. Then, we propose the hyper TLS traffic analysis method, named as HTTA, which attempts to analyze the TLS traffic from a new perspective. It aims at the efficiency, transparency, large-scale, automation, and real-time analysis. The key idea is that it extracts the session information from the process space of browsers and then correlates it with multiple types of data and works in real time, so that further correlation analysis can be performed on the decrypted packets. The proposed information extraction method and data analysis pattern can cover the popular web browsers on multiple platforms and overcome the challenge of real-time analysis, without depending on the trusted certificates and hooking methods.

In summary, the main contributions of this paper are as follows:

Firstly, we propose a new efficient automated method and its corresponding framework for large-scale TLS traffic analysis of the browsers; it is also suitable for programs which have the homogeneous crypto infrastructure. The method collects and correlates multiple types of online data with noninvasive mode, which can perform real-time transparent analysis. It carries both efficiency and security and can be deployed flexibly.

Secondly, we propose a new session information extraction method based on the session cache and the characteristic of low fragmentation heap, which can cover the popular web browsers. Additionally, for coping with the version diversity of programs, the instruction similarity matching method with the constraint of graph path is further proposed to locate the key variable and function, to help extract the relevant session information.

Thirdly, we give a prototype implementation on Windows platform and analyze its essential links. The framework has a simple structure for fast installation and deployment, and it can be used in the further development of the automated analysis system for TLS traffic in the real environment.

Finally, this proposed method is verified by experiments with multiple types of browsers in reality. The experimental results demonstrate that the method can perform precise and efficient analysis with the lower performance overhead and also make no interruption to the workflow of the target program at the same time.

The rest of the paper is organized as follows. The related work is reviewed in Section 2. Section 3 describes the proposed method and analysis approaches in detail. Section 4 shows the key points of implementation. The conducted experiments and results are demonstrated in Section 5. Section 6 gives the discussions, and Section 7 concludes the work.

2. Related Work

There are a lot of research work about the attack and defense techniques of browsers and traffic analysis, and some related work with this paper are as follows.

2.1. Protocol Crack and Attack. Duong et al. proposed the BEAST attack which could obtain the plaintext passed

between the browser and server. It utilized the weakness of cipher block chaining (CBC) mode in the TLS protocol [8]. Rizzo et al. proposed the CRIME attack [7], which tried to guess the key request data such as cookies in the TLS channel exploiting the weakness of TLS compression method. Fardan et al. proposed the Lucky Thirteen attack which used a timing attack against the CBC encryption mode [9]. Padding Oracle [28], BREACH [29], so on are similar attacks. These attacks are complex, most of which need to inject the attack code into the victim's browser and send large requests to the web server. They also depend on the specific protocol versions.

2.2. Proxy and Traffic Analysis. Marlinspike et al. designed and implemented SSLStrip which could intercept the HTTP request before it redirected to HTTPS [30]. Liang et al. discovered the HTTPS deployment problems in CDNs, which could cause MITM attacks [31]. Jia et al. proposed the browser cache poisoning (BCP) attack [32], which amplified the threat of MITM attacks by poisoning the browser cache for persistence after the user ignores the warning of illegal certificates. Sherry et al. proposed BlindBox [33], which was a deep packet inspection (DPI) system based on encryption traffic, whereas a new protocol and encryption scheme needed to be designed. Similarly, searchable encryption is widely discussed in the cloud storage, outsourcing, and so on [34]. But it may have difficulties to fully secure network communications, for example, the encryption efficiency is low, and anyone on the network route can search the content by keyword guessing. Even though, from the perspective of traffic analysis, it is difficult to understand the semantics of the whole communication through some searches only.

Durumeric et al. probed into the impact on HTTPS by security software and network devices [23]. It pointed out that web servers could detect these behaviors, and the interception might weaken the security of original TLS connections. Carnavalet and Mannan designed a framework for analyzing the TLS proxies on the client, which could uncover the security risks introduced by these interception tools [22]. However, these research studies showed that the network proxy might cause the security problems, and similarly malicious code could also disrupt the proxy configuration with the same method. In this paper, HTTA does not act as a TLS proxy so as to avoid raising security problems. It is transparent to the both sides of the communication.

2.3. Memory Data Extraction. Dolan-Gavitt et al. proposed the virtual machine introspection (VMI) framework Tappan Zee (North) Bridge which could analyze the memory data [35]. The keyfind plugin is used to search master keys in the memory, and it tries to decrypt and validate the packet by using the each 48-byte data as a master key. Similarly, Taubmann et al. proposed TLSkex which could also extract master keys of TLS connections based on VMI technology [36], and it used a brute force together with some heuristic approaches based on searching in a memory snapshot. Feng et al. proposed ORIGEN [37], which was applied to get the data structure profiles in the new versions of the software based on the knowledge on its old version. It is helpful to solve one of the problems in our framework. We also propose another solution to solve the location problem of pivotal functions and variables.

2.4. Page Content Analysis. Vadrevu et al. proposed a new web browser with the forensic engine named ChromePic [38], which could record and reconstruct the process of common web attacks based on Chromium. Jayasinghe et al. proposed a novel dynamic approach to detect drive-by download attacks [1], and it can monitor the bytecode generated by a browser in real time with low performance overhead. Studies based on the page content analysis fall on the next step of our research, and some can be integrated into the proposed framework in this paper.

In general, compared with the existing research work, we mainly focus on the efficient automated TLS traffic analysis in the large scale, which will recover the original text in the encryption channel and make further data analysis. Moreover, we unwrap the network packets by correlating multiple types of runtime data, and will deal with the effect of decryption and the real-time analysis problems. The proposed method in this paper is applicable and scalable, which can be easily deployed in practice.

3. Method Description

3.1. Architecture Overview. Traffic encryption mechanisms improve the capability of data protection, while they weaken the security data analysis system. To address this, flexible and efficient schemes are needed for TLS traffic acquisition and correlation. Two questions should be considered which also motivates the research. One is: how to defend the vulnerability exploitations and malicious codes before the browser parses and renders the page? The thorough method is to analyze the content of network traffic continuously and extract the traffic characteristics to the host or network intrusion prevention system. The other is: how to improve the efficiency and reduce the impact on the target system while analyzing? It preferably needs the noninvasive approaches to ensure the security and real time.

The overall architecture of HTTA is shown in Figure 1; it collects multiple types of data associated with the target programs in the hyper mode, for example, from the kernel space, hypervisor, or hardware device. The acquired data include processes, threads, active network connections, file operations, and TLS sessions on the target operating system. The network traffic is also collected and filtered, which can be performed in a bypass mode if there are no packet interception requirements. The noninvasive method is used which means it does not modify the executable module and configuration of target program, and also it does not intercept the original workflow which may lead to the failure of communication. Transparency means that the target program and remote server cannot directly detect the presence of the analysis. There is another notable point, as seen from Figure 1, that the dotted lines denote the appropriate interactions with the target system in the special case. For

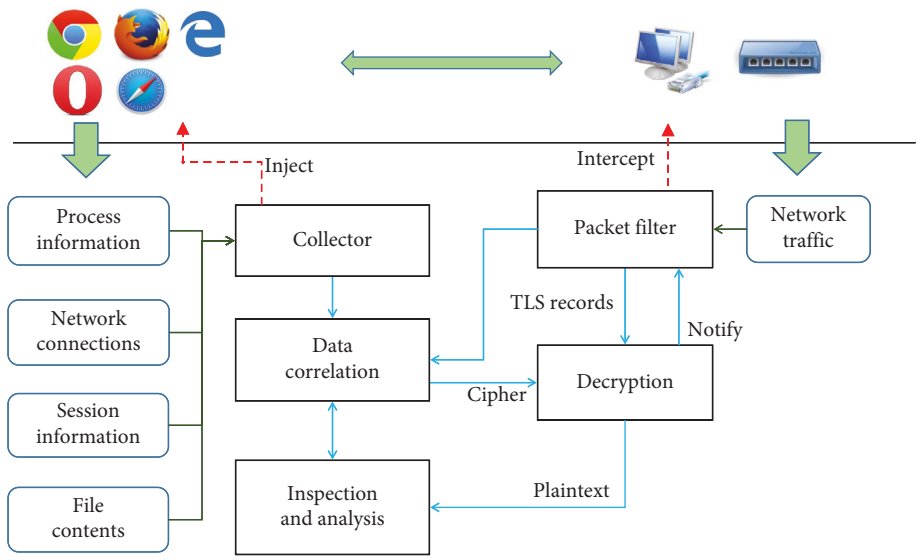

Figure 1: Architecture of HTTA.

example, inject a thread into the space of target process to accomplish aided task. For the most cases, it is not requisite.

The process information contains the process name, the executable file path, loaded modules, and process/thread environment block (PEB/TEB). According to the information, we can determine the target program together with its version and crypto infrastructure. The network connections mainly include the IP addresses and ports of target process referring to the TCP/IP protocol, which can be directly correlated with the network packets. The above data are acquired from the kernel space of the operating system; there have already been lots of research studies about that. The session information includes the essential key and related parameters for decrypting the traffic, which exist in the memory space of the target process, and we call it *TLS session information* (TSI). Actually, the former information is also in the memory space, which is managed by the system kernel. File handles refer to the files opened by the target program, and the file content can be further correlated with the decrypted network packets. The data correlation module first correlates the target program with the TLS traffic at the level of TCP/UDP stream, and later the correlation results will be sent to the inspection module for decryption and analysis. The packet filter module identifies the needed network traffic and can also perform preliminary analysis to extract the fingerprint of the crypto library. Besides, as denoted by the dotted line in Figure 1, the module can decide whether or not to pend packets of the specific stream for a while until the extraction module obtains the associated TSI. This will be discussed below in this paper.

For further discussion, the proposed method also has good extensibility and compatibility. It is suitable for not only web browsers but also other programs, while the browser programs are widely used and relatively stable. We can construct the unified TSI extraction patterns for browsers, but it is difficult to cover other unknown programs, and then individual analysis is needed in advance. HTTA is not limited to the platform; it can be applied to Windows, Linux, and others. It supports all the TLS versions, including old SSL and the newest TLS 1.3. Because the essence of symmetric encryption mechanism has not been changed, the handshake protocol changes enormously. The encryption mode such as stream cipher and block cipher has no influence on the method. Moreover, HTTA is convenient for engineering extension and deployment, for example, modules can be moved from the user space to kernel space, or to the outside of the system. On the other hand, it is helpful to migrate between operating platforms, for example, avoiding the platform interdependency of packet capture and memory access module. The number of external interfaces is reduced in order to keep it independent. In fact, we only need two interfaces which are memory read and network access.

3.1.1. Aided Packet Interception. In the scenario of all the traffic can be decrypted in time, to ensure that we additionally introduce an additional interaction procedure between the packet filter and information extraction modules, as shown in Figure 1. Because in particular situations, for example, the configuration of cache time has been modified, the session information will not be cached for a long time. When the packet filter module detects that the TLS handshake is finished, it notifies the information extraction modules and queues the next packets at the same time, as are shown in steps 1~3 in Figure 2. Information extraction module then extracts the required data, after which it notifies the packet filter module again, as are shown in steps 4~5 in Figure 2. The packet filter module continues to forward the previous packets in the waiting queue, as are shown in steps 6~7 in Figure 2. Therefore, under this mechanism, it is for sure that information extraction modules can obtain the session information through the control of network packet forwarding. With the possible little cost of network delay, we should avoid the TCP timeout.

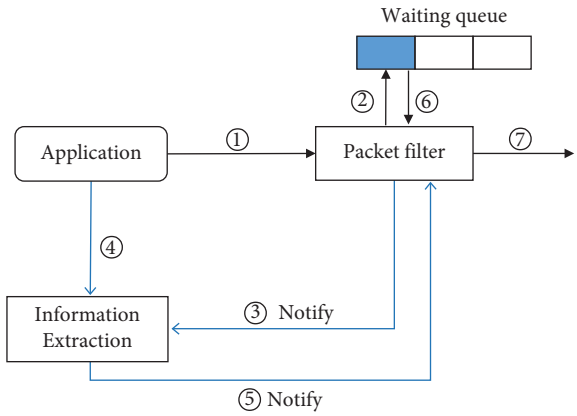

Figure 2: Additional interaction of the extraction and filter module.

In this mode, the session information and subsequent packets will be transferred to the decryption and analysis module for further process. Another waiting point can be set when packet filter module waits for the notification of analysis module and then decides whether to drop the target connection or not. It does not occur in all the packets. Because a TLS record often consists of multiple TCP packets, the decision can be determined in the last packet of the record, even in the last packet of stream, for better optimization. Moreover, if HTTA works in the offline analysis mode, there would be no interception delay.

3.1.2. Deployment and Operating Modes. For different application scenarios, there could be three deployment modes of HTTA. In the first scenario, the analysis framework is installed on the local system, and it can be extended based on further development or integrated with other security analysis components. The second scenario is in the virtualized environment; the analysis framework is deployed in the hypervisor component and then can analyze the TLS traffic of programs on the VMs. In the third scenario, it is deployed on the access point of internal network just as the firewall.

As shown in Figure 3, in the first case, HTTA is divided into the user mode and kernel mode module. The TSI extraction and packet capture module are in the kernel mode; data decryption and analysis can be in the user mode, which is convenient for further development according the specific requirement. In the second scenario, all modules are stayed in the hypervisor, and it is out of the band and fully transparent to the target system. Furthermore, data from different VMs can converge into one process node in this case. For the above cases, the network packets are collected in the kernel of the target system, which would intervene the target communication to some extent, but it only forwards the packet without unwrapping and wrapping. In the third scenario, a proxy module should be deployed on the target computer, and then the TSI obtained by the proxy module will be sent to the local firewall together with the network traffic. At this moment, the packets can be collected by port mirroring of the network device, which does not introduce any direct interceptions. For different deployment modes, the core logic of HTTA is the same, so the prototype below concentrates on the first deployment mode.

Additionally, HTTA can be configured with audit mode or interception mode which is suitable for traffic audit and real time analysis. In the audit mode, the framework only considers the integrity of TSI. While real time is necessary in the intercept mode, it can block and replace the target content. For accomplishing the TLS packet filtering according to the designed framework, there are several influence factors. First, the TSI extraction works based on the dynamic environment, not on the offline memory dump, and the precise should be ensured. Second, TSI and network packet could be correlated since they are obtained in the separated process. Finally, correlation and decryption should be finished in a short time for real-time analysis. We will give further description and discussion in this paper and then analyze and validate that by real experiments.

3.1.3. Challenges and Countermeasures

(a) *TSI Extraction*. There are some searching techniques in the memory, such as Magic number and debug symbols. However, it will cause high performance overhead when searching in the large address range. To overcome this, we propose an optimized method. Some browsers support log mode which can output TLS master key into a file [26], but this should modify the runtime configuration, and it is not general. Finally, we propose an efficient method of extracting TSI in the process memory of browsers, which covers all popular browsers. It will bring convenience to information extraction together with data structure analysis.

(b) *Version Change*. The minor version of browsers may change frequently, and it will cause the binary layout changing because of recompilation. The critical data extraction depends on the reverse engineering of target binary program. In fact, most applications including web browsers are not obfuscated. HTTA only depends on several binary features, if we can locate the address of some functions and then can extract TSI rapidly. We can choose some stable functions that reference the target variable, and expect they will be invariant if the program version updates. We adopt the method of semiautomation to solve the changing problem of the version with consideration of practice.

(c) *Real Time and Overhead*. In HTTA, the TSI extraction thread runs in the kernel space or out of the system, and it will continuously access and control process memory better. The TSI extraction module cooperates with the packet filter module, and it works when the connection is established and stops if the connection is closed; it can reduce the rate of CPU usage. On the other hand, we can properly delay the speed of packet forwarding by the packet filter module, and consequently extend the duration

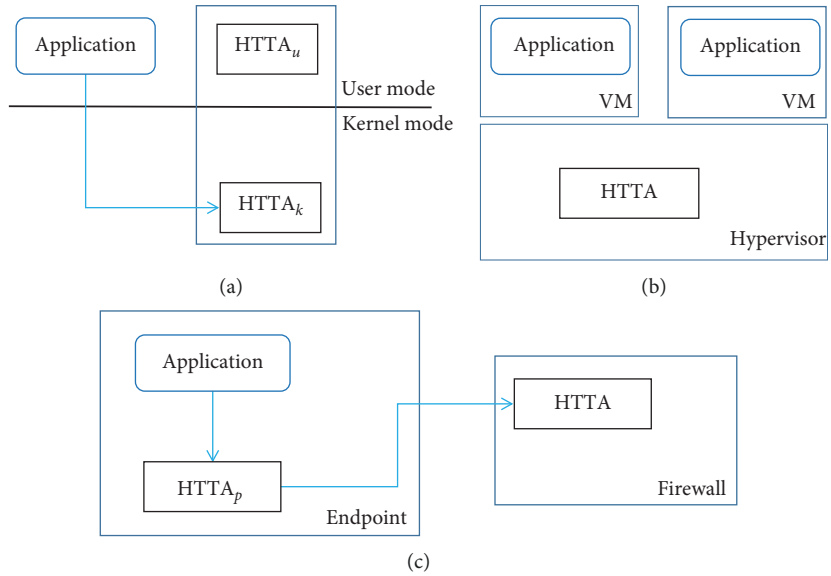

Figure 3: Different deployment modes for HTTA.

of target TLS connection. Then, the extraction modules will have enough time to obtain the session information.

3.1.4. Security Issues and Analysis. No additional trusted root certificates are installed in the system; therefore, the related security risks do not exist, and, for example, the corresponding private key is stolen or cracked. The proposed solution only inspects the network traffic in real time without preserving the data, which will be deleted after the analysis. The potential risk is that the decrypted data can also be accessed by the malicious code if it would be. In this case, we can migrate the analysis code into the kernel which can prevent most of the nonrootkit malware. On the other hand, the memory protection method can also be used such as Intel Software Guard Extensions (SGX). In fact, if the malware comprises the target system, it may directly extract the sensitive data from the target application or other resource storage more easily.

Meanwhile, the solution does not directly intervene the execution flow of target program, and it reduces the impact of uncertainties. Network packet forwarding is the only intervention. When packets are collected by the kernel module, the module should be minimized and safely developed. In the audit mode, it has a minimal effect on the network communication of target program. In the interception mode, additional daemon threads can be created to watch the process of packet forwarding, so that it can handle the connection timeout in time due to the exception of the analysis system.

3.2. Approaches of the TSI Extraction. The procedure of TSI extraction contains two parts: one is to extract the classes or structures which are related to the target TSI, and the other is to extract the internal member information of the structures above. We can extract these data from the process memory space of programs, but need to obtain the structure location and member offsets of TSI, depending on the binary or source code analysis preliminarily. For example, we can obtain the TLS session structure according to the source code of *OpenSSL*, which is defined as *SSL_session*. It contains the master key which is the critical data for the encryption and decryption according to the TLS specification, and then we can decrypt the packets based on that. So, the main goal in the extraction is the data structure like above.

According to the stipulations of the RFC documents, TLS protocol consists of key exchange and data encryption transmission, and the latter uses symmetric encryption mechanism. In other words, the both ends of communication hold the symmetric key information. The client end also holds the key information while the connection is alive. Moreover, the TLS protocol supports the session resumption for improving efficiency, using *session ID* (SID) or *session ticket* to identify the connection. So, the client program would cache the session information in the memory for resumption with the above ways.

3.2.1. Overview of Typical Browsers. We focus on the browser programs on the Windows platform, which is widely used and vulnerable in the developing countries. There are abundant applications but a few web browsers with high market penetration on personal computers, which are *Chrome, Firefox, IE, Edge, Safari,* and *Opera* [39]. Other browsers, such as *Maxthon, 360*, and *Sogou*, are all built on the core engines above.

First, although the network modules of browsers are slightly different, they all adopt the multiprocess architecture in the newest version. For *Chromium*-based browsers, the main process is in charge of network communication, so the TLS data transfer is in the separated process, and *Firefox* is similar. The development of *Safari* on Windows has been stopped. The version on Mac OS uses a separate process to

process network communication. Unlike other browsers, the network data are processed separately in each render process in *IE* series.

Second, the web browsers have different TLS implementations. Chrome and Opera based on *Chromium* use the *BoringSSL* which is a fork version of *OpenSSL* [40, 41], and *Firefox* maintains its own implementation named network security services (NSS). *Safari* uses Apple Secure Transport and *coreTLS* libraries. *IE* and *Edge* have part implementation in the *schannel* and *wininet* library. Most of the other browsers are the integration of the above-mentioned facts. In fact, other forks of *OpenSSL* such as *LibreSSL* are similar in the aspects of this paper [42]. The basic information of TLS implementation of some web browsers on Windows is shown in Table 1.

As the analysis shown above, although there are many differences between the TLS implementation for kinds of browsers, the TLS session information is managed by the specific module loaded in the memory. Therefore, we can extract the session data from the process memory space. It is also suitable for other programs which are built on the analogous crypto libraries of TLS.

3.2.2. Data Extraction. Due to the session resumption mechanism, modern web browsers always link the session structures together and cache them in the relative fixed memory region. Then, these chained structures are referenced by some global variables, in which the popular browsers and derivatives all have such characteristics. So, if we can locate the addresses of target global variables, then the session information can be extracted rapidly by traversing the hierarchy structures. We describe the TLS cache management for the main browsers below.

Chrome does not use the internal cache method provided by *BoringSSL* or early in *OpenSSL*. It manages the TLS cache externally, which defines the class *SSLClientSessionCache* [40]. When the TLS handshake initializes, the browser tries to look up existing session in the cache list and insert new session into the cache list when handshake finishes. The cache structure is *ssl_session_st* which contains established time, resumption information, cipher parameters, and so on. The cache list uses *std::list* to manage the session structures. Class *SSLClientSessionCache* is referenced in the class *SSLContext* which is defined as a singleton. This implies that the *SSLContext* pointer is stored in the binary as the global variable. *Opera* and other browsers which are built on *Chromium* have the same case.

Firefox defines the static pointer cache in *sslnonce* file [43], and it points to a double-link list structure. The structure is *sslSessionID* which contains accessed time, session ID, master key, and so on. It should be noticed that the master key is not stored in the form of plaintext and encrypted through the Public-Key Cryptography Standards (PKCS). So, we need to obtain the symmetric key used by PKCS beforehand and then decrypt the extracted session information.

Safari defines the variable *_gSessionCache* in the Security and coreTLS libraries, which points to a double queue. The cache structure also contains the master key and session resumption information.

The TLS interfaces of *IE* and *Edge* are encapsulated in the *schannel* library, among which the *SslContextList* variable points to the TLS cache information in the process memory of the browser. When the TLS handshake between the client and server is finished, the system derives the read key, write key, and other parameters from the master key and then stores them into the C*SslUserContext* class which is linked by *SslContextList*. Meanwhile, the link list *GlobalObjectList* and *GlobalServerInfoList* in the *wininet* library caches the current socket information and also have relations with *SslContextList*. A sketch is shown in Figure 4, the needed data is stored in the structure *CSecureSocket* and *CSslUserContext*. Therefore, we can get precise ports and key information in the memory for the *IE*, *Edge*, and other *IE* core-based browsers.

As known from the analysis, the popular browsers all have the global variable which points to the TLS session cache information. So, if we can locate the variable first, all existing TSI can be traversed rapidly.

Additionally, if the session cache data is flushed or the lifecycle of session is too short, we may not obtain the corresponding TSI in time. As a supplement to satisfy the special requirement, we also propose the rapid TSI extraction method based on the *low fragmentation heap* (LFH) mechanism [44], to locate the target in the limited memory region. The operating system provides a special memory management scheme for the memory allocation with small sizes which is named as LFH on Windows. Actually, some applications or libraries also have the similar custom implementations, such as Firefox, and we all call it LFH here. The key of LFH is that the similarly-sized memory blocks are allocated from the same memory bucket by using the bucketing scheme. As shown in Figure 5, when the allocation with specific size is committed, the memory block will be allocated via the corresponding bucket from the preallocated memory chunks. For the structure that contains the TSI, the allocation size belongs to the scope of LFH. The size of target structure nearly remains stable in different versions of the same program and is easy to obtain. When the version of target program is known, the allocation of the structure that stores TSI will be bound to the certain bucket. For example, as shown in Figure 5, the size of target structure is 400 bytes and allocated from bucket #25; then, we only focus on this bucket continuously. Therefore, when the TSI cannot be obtained by the method above, we can extract the fields such as port, session ID, or session ticket from the network packet and then match the target structure in the preacquired memory blocks of the corresponding bucket. It should be noted that the LFH bucket should be activated before it takes over the allocation of the corresponding size for the LFH of operating system. The bucket is activated if the number of allocations for the bucket allocation size has reached a small value, and it can be easily satisfied for browser programs. It can be forcedly activated in the worst case by injecting the specialized thread, as shown in Figure 1.

Furthermore, if the target structure that contains TSI is unknown, and the reverse engineering and debugging are difficult to perform on the target program, the brute force

Table 1: Overview of related modules of browsers on Windows.

Browsers	Number of network processes	Related libraries
Chrome	1	chrome.dll
Firefox	1	nss3.dll/softokn3.dll
IE	≥1	schannel.dll/wininet.dll
Edge	≥1	schannel.dll/wininet.dll
Opera	1	opera_browser.dll
Others	1 or ≥1	chrome.dll or schannel.dll/wininet.dll

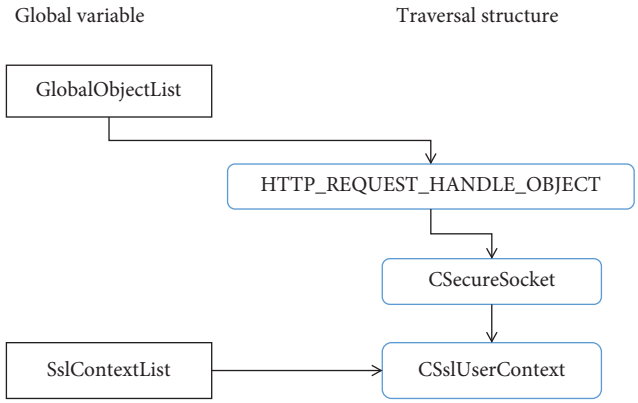

Figure 4: Extraction by the traversal for IE.

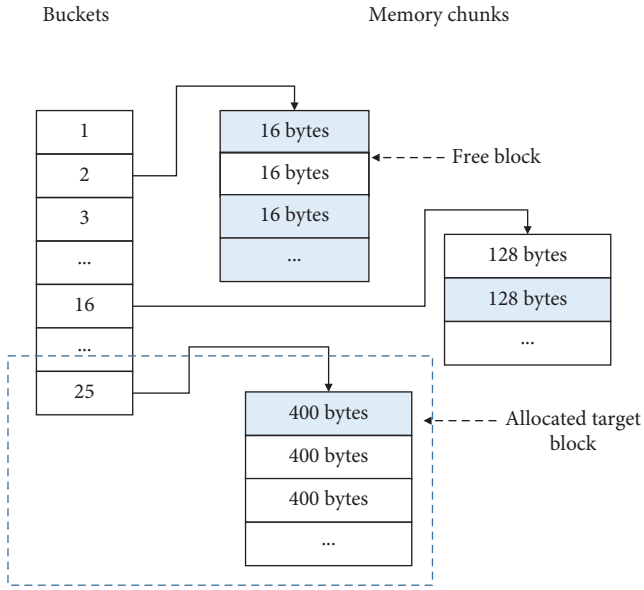

Figure 5: Example of memory allocation via the LFH.

method can be used first to determine the size and location of the target structure [36]. Then, the TSI can be rapidly obtained according to the LFH for the subsequent analysis in the wide range.

3.2.3. Locating the Variable and Structures. We divide the acquisition of global variables that point to the sessions into two cases, one is that the binary program has abundant debug symbols, and the other is contrary. If the debug symbols of the binary program can be accessed, such as *IE*, *Edge*, and *Firefox*, we could quickly obtain the offset value of the target variable according to the debug symbol. In that case, the impact of version change can be reduced. For the latter case, we should extract the offset to the module base by reverse engineering. One simple method is to locate the target address by the reference of constant strings; it is effective in some cases, but not universal.

Then, we introduce the semiautomated method to ease the burden of reverse engineering because the variable can be located through the functions which reference it. When the address of related function is known in one version of the program, we could obtain the similar information from other versions. Two approaches can be applied, one is the graph matching based on the *control flow graph* (CFG) of the target function [45], and the other is proposed here, named as *instruction similarity matching with the constraint of graph path* (ISMCP). The former is widely used in the research of malware detection and binary program similarity detection [46, 47]. But in this paper, we only find the specific function and do not make comparisons on the entire binary program. It can make an accurate match when the CFG of target function has few changes in the new versions. The latter chooses the key path of the CFG to calculate the path similarity between different versions, with the purpose of picking out the target function when the CFG has major changes. It is fuzzy compared with the former. In the first place, both the two methods statically disassemble the target program and generate the CFG for all functions of the program.

(1) Locating the function by CFG matching. CFG is a directed graph, $G = (V, E)$, where V is the vertex set of the graph and E is the edge set of the graph, $E = \{v_i, v_j \mid v_i, v_j \in V, v_j$ may execute after $v_i\}$, the vertex in the graph indicates a program basic block which has no jump instructions. Graph G contains all the possible execution paths of the program. The CFG analysis is widely used in compiler optimization and program analysis. Here, this method builds and models the CFG of target function and then locates new target based on the graph isomorphism check.

Definition 1 (subfunction CFG (SCFG)). The CFG G_f of subfunction F is a tuple.G = (V, E, VE, VX, LABEL_V, LABEL_E), where V is the set of basic blocks, $E \in V \times V$ which is the set of edge, VE denotes the entry block of the function, and VX is the exit block. LABEL_V is the label function of vertexes, which maps each basic block into a

positive integer. Similarly, LABEL_E is the label function of edges.

Then, we define LABEL_V as LABEL_V = SIZEOF $(v)\mp r$, $v \in V$, where the function SIZEOF is used to calculate the total instruction bytes of the basic block and r is an error parameter which is introduced to extend the match range. We do not give the definition of _E, in order to reduce the matching restrictions. Furthermore, one can limit the vertex number of graph to increase the accuracy in the matching, or match with the subgraph G_f' instead of the graph G_f to increase the coverage in the other versions of target program.

When we have the CFG G_{f1} of target function in the specific version of binary program, then attempt to find G_{f2} in the other version, G_{f1} and G_{f2}, is isomorphic. Graph isomorphism is a nondeterministic polynomial time (NP) problem, but CFG has some particular favorable conditions. It has fixed entry vertex, and each vertex has at most two outgoing edges, while the jump tables are an exception. Actually, the case of jump tables can be recognized easily. The VF2 algorithm is adopted [48], and the computation time based on CFG matching will be short in practice. When the change between different versions is slight, the CFG match method can locate the target function precisely. But the graph isomorphism restricts the relationship of edges, and it would expose the limitations when some edges of the CFG of the corresponding function are broken in the new version of program.

(2) Locating the function by ISMCP. For the purpose of locating the target function in the new version when the CFG of target function has major changes, we propose the ISMCP method which can be helpful to extract the target function.

Definition 2 (path in SCFG). Path P is an order $v_1, v_2, \ldots, v_k$, where v_i belongs to vertex set V, v_i, v_{i+1} belongs to set E, $1 \le i \le k$, and vertexes in the order are different from each other. k is the path length.

Definition 3. Path similarity,

$$\mathrm{PATH\_SIM}(S, T) = \sum_{i=0}^{\mathrm{PATH\_LEN}(s)} \frac{\mathrm{LCS\_LEN}(\mathrm{STR}(vs_i), \mathrm{STR}(vt_i))/\mathrm{MAX}(\mathrm{LEN}(\mathrm{STR}(vs_i)), \mathrm{LEN}(\mathrm{STR}(vt_i)))}{\mathrm{PATH\_LEN}(S)}, \quad (1)$$

where S and T are two paths, PATH_LEN is the function of calculating path length, and $\mathrm{PATH\_LEN}(S) \le \mathrm{PATH\_LEN}(T)$, $vs_i \in S$, $vt_i \in T$, $1 \le i \le \mathrm{PATH\_LEN}(S)$. LCS_LEN is a function which calculates the length of the longest common string (LCS) [49], LEN calculates the length of the string, MAX returns the maximum value of arguments, and function STR converts the basic block into a byte sequence.

While the program version updates, some codes of the specific function may also change in the new version, which would cause the change of CFG. As shown in Figure 6, the two CFGs are not the same, but there is the same execution path, *020112*. If the path length is short, we can also measure the similarity of two paths by computing LCS, instead of direct comparison. We expect that the function of the new version preserves part execution features compared with the old one, and they can be found by the matching of instruction sequences. If the similarity between the new and original functions is low, we consider that they do not have the correspondence. In that situation, we need to choose the function in the new version as the template instead, expect to still find the corresponding one in the subsequent versions of the program.

The detailed steps of ISMCP are described as follows:

Step 1: selecting the matching paths. First, we choose some matching paths in the target function of the initial version, which can represent the vital execution flow of the function; meanwhile, take the entry block of the function as the starting point of paths and confirm the path length k (suggest $k \le 12$). If k is too large, the capability of matching will decrease, and the path number will grow exponentially. When paths are confirmed, we continue to label the edges of each path, and the label function is as follows:

$$\mathrm{LABEL\_}E(v_i, v_{i+1}) = \begin{cases} 0, & \text{if the jump condition is FALSE,} \\ 1, & \text{if the jump condition is TRUE,} \\ 2, & \text{others.} \end{cases} \quad (2)$$

If the jump condition is FALSE from block v_i to v_{i+1}, label v_i, v_{i+1} is marked as 0 and marked as 1; on the contrary, in other cases, it is marked as 2. Finally, we get the set of path templates $\mathrm{PT} = \{P_i \mid 1 \le i \le k\}$. Figure 6 shows two same paths which belong to different CFGs.

Step 2: generating the byte sequences. We should convert the basic blocks into integer values. Before this, we map all the instructions of the architecture to 16-bit integers, and build a mapping table in advance. Then, instructions of each basic block will be translated to the byte sequence, as shown in Figure 7. This can preserve the semantic feature of the target function and ignore the influence of specific registers and memory addresses.

Step 3: calculating the path similarity. Export all CFGs of functions in the target executable and traverse all the CFGs successively based on depth first search algorithm. In the traversal process of each CFG, we check the edge label of each path until the end of the path. For one path, if all the label values are matched with the

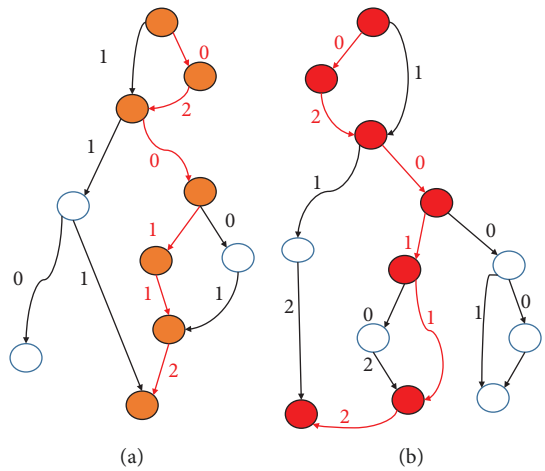

Figure 6: Two paths of different CFGs having the same labels.

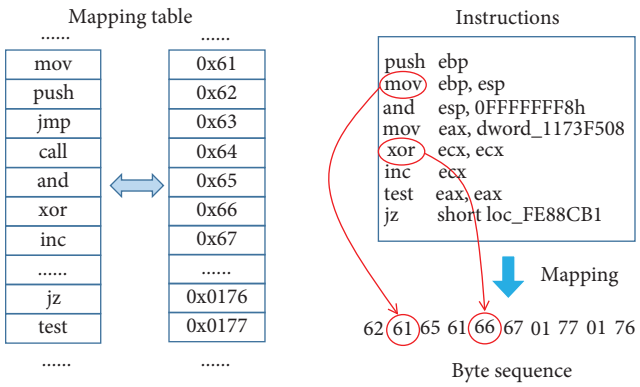

Figure 7: Example of instruction mapping.

path template, we calculate the path similarity of both. As mentioned above, the LCS method can also be used in the matching. Otherwise, we continue to traverse until all path templates are checked or the entire CFG is walked through. The detailed procedure is shown in Algorithm 1.

Step 4: sorting all similarity values. When all the CFGs in the target binary program are traversed, lots of similarity values will be generated. Then, we sort these values in descending order and show the results together with the address of each function. Finally, we pick out the top similarity values of each path template for further analysis.

We expect that the above method can assist with the rapid locating of the target function when part of the code changes in different versions of the program, especially when the structure of CFG changes greatly. The complexity of path similarity generation algorithm is $O(M*N*2^k)$, where M is the total number of the CFG in target program, N is the vertex number of the CFG, and k is the length of path. In reality, the CFG structure is sparse, so when k is small and the root node is fixed, the method can be executed efficiently.

3.2.4. Offset of the Structure Member. Another problem is to extract the members of structure when the starting address of chained structures is located. It is easier for programs which are open source or have abundant debug symbols. One can clearly see the internal variables and understand the logic, or calculate the member offset in a structure according to the variable type of the source code. For the common binary program, static manual reverse analysis is a time-consuming job. Similar to the location of the target structure, we can first determine the function that references the member or parses the entire structure. Then, the specific version of target program is selected as the initial template to obtain the information of other versions. On this basis, we can rapidly examine the changes of member offsets when the program version updates.

Another aided method is to debug and analyze the target program dynamically. So, we can develop and set a local TLS server program based on the modified *OpenSSL*, so that the TSI of each connection would be known, and it can be replayed in addition. The target program is then started and connected to the local server. When the TLS handshake completes, the address of corresponding TSI is also determinate. Then, in the memory space of the browser, we can recursively search the byte sequence of the member of target structure to get the offset value. The procedure can be automatically accomplished. The example diagram is shown in Figure 8; we locate the master key parameter in the session structure.

There may be the recursive search since sometimes the byte array is referenced by a pointer or referenced by the pointer to pointer. The more layers the recursion have, the more uncertain and complicated it would be. For example, in the structure *ssl_session_st*, the buffer of the master key locates in the memory region of the current structure, but the ticket member is in the buffer referenced by the pointer *tlsext_tick* of the structure [40]. In general, when the offset information in the specific version is known, it would have few changes in other versions. Based on this assumption, the extraction of member offset of the structure can be automated to some extent.

```
Input: The set of CFGs in target binary program, CFG_SET. The set of path templates, PT
Output: The set of path similarity values, SV_SET
1: function PathSimGenerator(CFG_SET, PT)
2:   for graph in CFG_SET
3:     Initialize an array variable, stack
4:     Append first node of graph to stack
5:     while stack is not empty
6:       Pop the top item n from stack
7:       Load instruction sequence of n
8:       Get current path path_c and its label sequence label_c from stack
9:       if the length of stack is equal with length of path_c and p in PT has the same label with path_c then
10:          s_value = PATH_SIM(p, path_c)
11:          Insert s_value into SV_SET
12:          Remove n from stack
13:        else
14:          Load neighbors of n into n_nbs
15:          if b in n_nbs is not visited then
16:            if b is not in stack then
17:              Append b to stack
18:              Mark b as visited
19:            end if
20:          else
21:            Remove n from stack
22:            Mark all items in n_nbs as not visited
23:          end if
24:        end if
25:      end while
26:   end for
27:   return SV_SET
28: end function
```

Algorithm 1: Generation algorithm of path similarity.

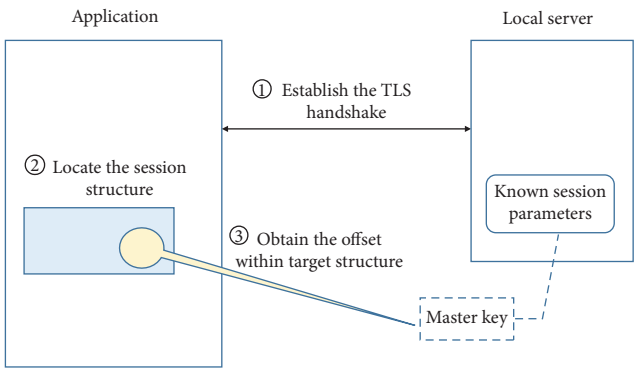

Figure 8: Locate the structure member by dynamic debugging analysis.

Besides, we should extract enough information from the target structures in order to achieve the analysis of TLS traffic. For the cipher mode of CBC, the master key or read/write key is needed, and the hash message authentication code (HMAC) is also needed. The initialization vector is the last bytes of the last cipher text which can be obtained in data packets. For the Galois counter mode (GCM) mode, the master key is enough; otherwise, another value *Salt* is needed, comparing with the CBC mode. Because *Salt* is an implicit nonce while the explicit nonce is transferred through the network; both are then combined into a single nonce value according to the TLS specification. The additional authentication data (AAD) can also be obtained from the network packet.

3.3. Data Correlation. First, all kinds of acquired data would be correlated in order to accurately decrypt the TSL traffic of the target program. Then, the unwrapped plaintext can be correlated with the files that the target process has opened, to examine if there exists the data exfiltration. The fields that data correlation depends on are shown in Figure 9, where the field plaintext represents the decrypted content of the network packet. Besides, for the case that the process information is fuzzy, the fingerprint extracted from the handshake packets can also be used to determine the crypto library loaded by the target process.

The TLS session is established upon the TLS connection; therefore, one TCP stream may contain several TLS sessions, but each data packet only corresponds to one TLS session. We deal with the packets according to the TCP stream and build the mapping to TSI for each segment of the TCP stream. As shown in Figure 10, the handshake messages are the dividing point. When the TLS handshake is finished, the decryption task of the current session will be started. Moreover, due to the session resumption mechanism, one TSI can also correspond to many TCP streams. As denoted

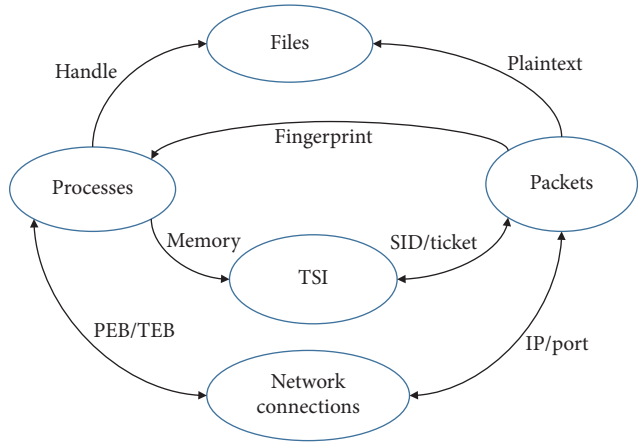

Figure 9: Correlations of multiple types of data.

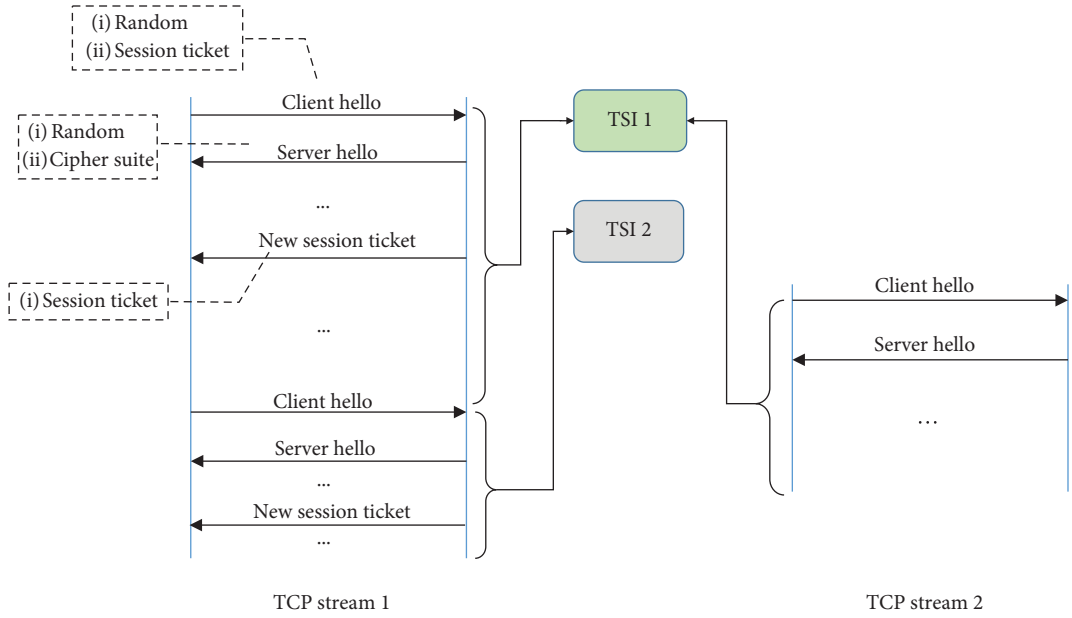

Figure 10: Correlation between TCP streams and TSIs.

in the dotted box in Figure 10, some fields would be extracted from the handshake messages, and after that the further correlation continues.

Since the TSI and network packets are obtained, respectively, and the TSI may not contain all elements needed by decryption, next we discuss how to correlate them in time. According to the RFC document, for each TLS record, the integrity needs to be verified at both ends of communication. With the cipher key, we can determine the corresponding TLS record by trying decrypting and verifying the record. At worst, we need to verify each record with all the extracted cipher keys, but the cost will rise when the number of keys increases. In the offline mode, it is still an effective method, though. In fact, we should consider the following cases.

(1) TSI contains the IP address and port. It is the perfect case because all the packets and TSI can be directly correlated by the IP and port. Of course it should be limited in a reasonable time period because the client port may be reused when the associated TCP connection is closed.

(2) TSI contains the resumption information. The basic resumption information includes session ticket and session ID, and the ticket is more widely used by servers than session ID currently. With the resumption information, the TSI can be correlated with the packet easily in most cases. The resumption information can be extracted from the TLS handshake packet *Server Hello* and *New Session Ticket* and the extension of *Client Hello*. However, there would be many connections with the same resumption information but corresponding to different TSIs. Because the session key derivation depends on the random numbers of the handshake packets, which may be updated in the new TLS session. In this case,

we should try to decrypt and verify TLS record by the set of TSIs which contains the same resumption information.

(3) TSI contains the time information. If the TSI only contains the last accessed time, meanwhile it does not contain any resumption information. In this case, we can set a time period and try to decrypt and verify the record with the TSI within this period. As is shown in Figure 11, when one record occurs, the period around the current time is chosen as the validation window. Therefore, in the short period of time, the number of TSIs used for trying is limited.

(4) TSI contains nothing except the key. Since there is no assistant data, we need to perform decryption and verification on each TLS record with all the TSIs. As is mentioned above, it will cause network delay. Therefore, when the TSI is extracted we should append the acquired time, and then the correlation will be converted to the case in the last paragraph.

4. Prototype Implementation

We implement a prototype of HTTA on Windows platform, in order to verify its feasibility. The extraction of global cache and structure member offset belong to the preliminary work before the framework running. We develop some plugins based on the *IDA Pro 6.8* tool [50]. As is shown in Figure 12, the prototype contains a kernel driver which captures the packets and extracts the TSI, an application in user mode that manages the I/O with kernel driver and decrypts the packets.

Information extraction module works as a kernel thread, which currently supports three types of browsers, *Chrome*, *Firefox*, and *Edge*. It also processes the additional encryption of Firefox. We can distinguish applications from each other by reading their process names. For *Chrome* and *Firefox*, the TSI is maintained in the parent process which has the same process name with child processes. It works on multithread mode, which creates several work threads beforehand. Multiple threads can cope with several browsers simultaneously. Moreover, TSI is stored in the render process for *Edge*, and multiple threads can improve its efficiency. We use hash table to store the extracted TSI and remove the duplicated item based on master key or send key. For the sake of performance we build several hash tables which takes port, ticket, session ID, and time as the hash key, respectively.

The packet filter module is built on the Windows Filter Platform (WFP) [51]. By taking advantage of the character of the FwpsFlowAssociateContext function that it only processes the packets of the target processes and registering the callback function in FWPM_LAYER_STREAM_V4 layer, the module can process the payload directly without maintaining the TCP sequence. All the packets are captured and cloned and then sent to the uploading buffer. Packet filter module hangs up the packets, waits for the notification, and then decides to drop it or reinject into the protocol stack.

The decryption module is implemented based on the *OpenSSL* library, which has multiple work threads and

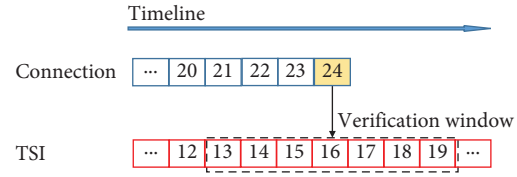

Figure 11: Verification with the TSI in a short period.

creates separate buffer queue for each TCP stream. It also parses the packet and extracts some parameters such as random number, and then derives key information based on TSI and required parameters. In practice, there is a special case that the client may send application data immediately after *Client Key Exchange* message in an initial handshake before it receives the *New Session Ticket* message from server at that moment. Hence, the sent payload cannot be correlated with TSI by the ticket. To cope with this situation, we create an additional buffer queue to decrypt them after the correlation is finished.

5. Experiments and Results Analysis

The experiments are performed on the desktop computer composed of *Intel i7-6700 @ 3.40 GHz CPU, 24 GB memory* and *Windows 10 (1607) 64-bit*. For the kernel driver testing, we also install VMware Workstation and create virtual machines with it. The configuration of virtual machine is *4 cores CPU, 8 GB memory* and *Windows 10 64-bit* operating system. Meanwhile, another virtual machine is created for a local gateway, which has 2 core CPU, 1 GB memory, and Ubuntu 16.04 64-bit operating system. The main test browsers are 64 bit *Firefox* (65.0.1), *Chrome* (72.0.3626.81), and *Edge* (38.14393.1066.0), which can represent most of the cases.

5.1. Evaluation of the TLS Session Lifecycle. First, we evaluate the lifetime of TLS sessions in the memory for different browsers in real life. The target browsers are executed with default configurations, and the network bandwidth of the experiment environment is 20 Mbps. We write a test script for automatically visiting the homepages of top HTTPS websites from Alexa, including 20 domestic and 10 foreign sites. The page will be immediately closed when it is fully loaded, and the experiment is repeated at least 10 times. Then, we count the TLS sessions by the ticket and TCP port, and also develop hook plugins for browsers to obtain the time of allocation and free for each session object. *Wireshark* is used to capture the network packets. Finally, we analyze the duration of TCP connections and correlate them with TLS sessions. Even though these websites may change dynamically and there may be errors between different tests, it will not affect the result.

In each test, about one thousand TCP connections are established in the interval of ten minutes. The result is shown in Figure 13, which is the average value based on the ten tests. It shows the lifetime distribution of TCP connections and TLS sessions in the real website access, where the left bar denotes the duration distribution of TLS sessions. Most

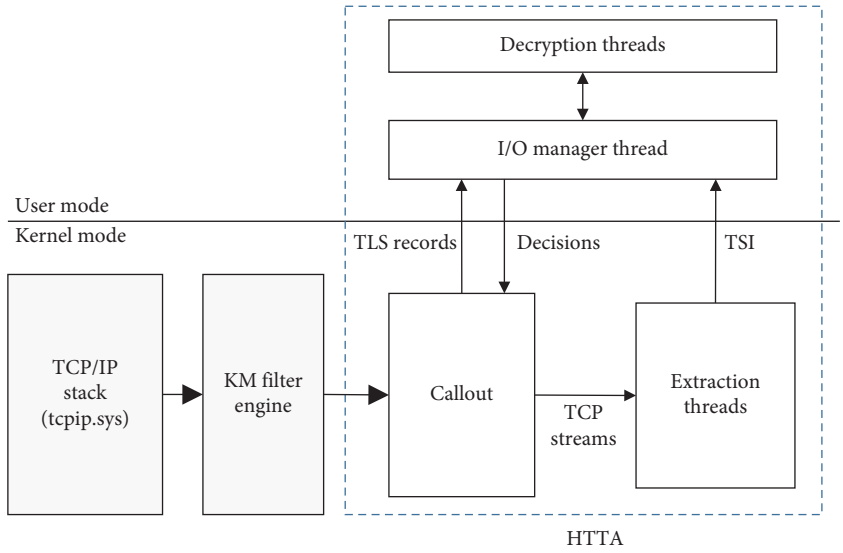

Figure 12: Prototype architecture of HTTA on Windows platform.

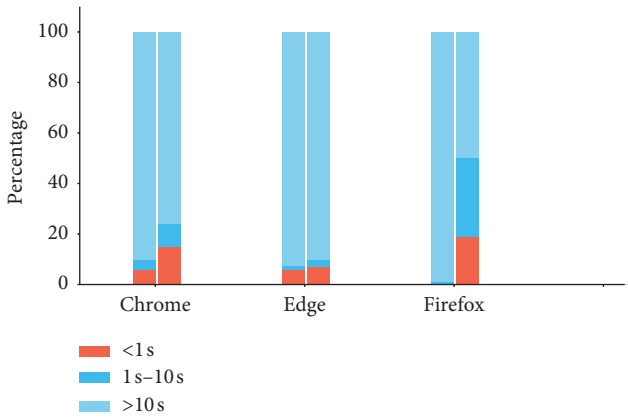

Figure 13: Duration of sessions and connections for typical browsers.

sessions can exist for a long time, so that the extraction module has enough time to obtain the TSI from the memory in the bypass mode. It also has challenges for the session that exists less than 1 s. Besides, the duration of TLS session is associated with the TCP connection in theory, which is more obvious in *Edge*. The session objects in the process memory of *IE* and *Edge* can be released in time. Other sessions with long cache time are managed by the system process. *Chrome* and *Firefox* manage sessions by themselves. Therefore, most TLS sessions exist in the memory for a long time, and the rest with short lifecycle can be released in time corresponding to the concurrent connections. As shown in the figure, in general the duration of most TLS session objects is long, especially in the case of mass data transmission, such as sending and receiving large files, so that the extraction modules have enough time to obtain them. Moreover, we find that *Chrome* do not release the session objects in time for most cases, and it may have the consideration of communication performance.

5.2. Effectiveness and Overhead Measurement. Then, we perform experiments on the real session extraction and packet decryption, in order to evaluate the effectiveness of HTTA. The extraction module extracts TSIs by traversing the cache information starting at the global variable pointer. It should be notified that the audit mode is used here, which means that we do not intercept and pend the packets; otherwise, this experiment may make no sense. We still choose three 64 bit browsers mentioned in the last section as the target programs and then install them in the virtual machine. Meanwhile, when the HTTA framework runs, the *Wireshark* and *perfmon* tool are used to capture network packets and obtain the processor time, respectively. The experiment is also repeated 10 times, while generating about one thousand connections each time, and then we calculate the average results. As shown in Figure 14, the packet decryption rates are 96.45%, 99.07%, and 95.25%, respectively. The decryption rate is over 95% for each test browser, especially near 100% for *Firefox*. Further, we investigate the reason why the decryption does not cover all the TCP connections. One is due to the statistical method, for *Chrome*, it creates multiple concurrent connections at the same time, but only one corresponding session is reserved and actually cached, which occupies about 4% in all the captured connections. For *Edge*, some connections do not come from the render process but the frame process, such as the *favicon* request which does not belong to our concerns. The rest few connections have a very short time to live, and most of that do not have the substantial data transmission, so that they are not obtained due to the very short duration of session objects in the audit mode.

In addition, as can be seen that the average CPU usage rate is low in all the tests, within 5%. Those are the counting values, while the browsers open and render the web pages. For longer period, if the idle time is considered, it would lead to a lower average processor time. The CPU occupation mainly comes from the extraction and decryption module

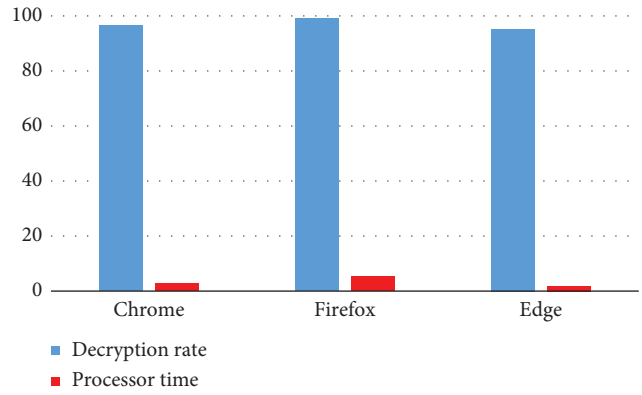

Figure 14: Decryption rate and overhead of the extraction by the global variable.

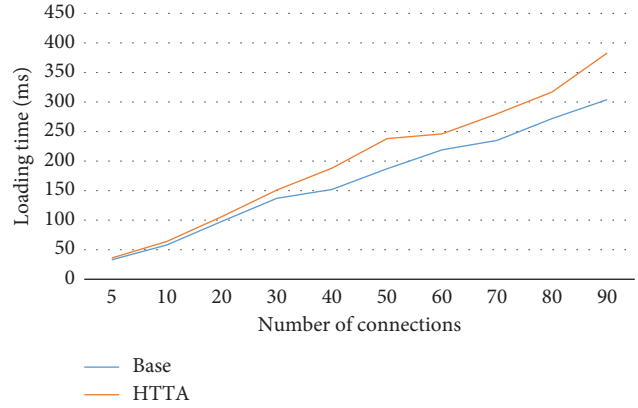

Figure 15: Overhead of the extraction by the traversal of LFH.

which are in the user space. Overhead of the packet filter module in the kernel space is negligible.

Next, we evaluate the effectiveness of the matching method based on the low fragmentation heap; consider the extreme case that all the TSIs are obtained from the heap memory by searching. As mentioned above, if the handshake packets are pending in the communication, the corresponding session information can surely be obtained in the memory space, and what needs to be concerned is the overhead and time delay. We take *Chrome* as the example and choose the *SSLClientSocketImpl* object as the target from which we can extract the TSI and TCP connection information all at once. Then, we count and observe the time delay caused by packet queuing with different number of concurrent connections in the web page, respectively. In order to avoid the impact of network transfer, a local TLS server is created. As shown in Figure 15, the page loading time does not increase obviously when the number of connections increases, and it is in the acceptable range. The main reason is that the number of concurrent connections of the browser is limited. While the number of connections grows, one traversal of the LFH chunks can obtain several target objects, and it is beneficial. Finally, the browser loads the page that generates 1000 http requests with the response size of 50 KB, and the processor time is recorded. The result shows that the processor time of entire system does not increase obviously because the extraction modules works on demand, not continuously. In fact, the search only occurs when the TSI is not obtained from the session cache.

Actually, the scenario above is common in the non-browser programs, which utilize the aforementioned TLS libraries to make secure communications. For example, some malicious programs communicate with the remote server based on the HTTPS protocol. It uses the *WinHttpSendRequest* function of the *winhttp* library to send data, which will be subsequently encrypted by functions in the *schannel* library before sending. The TSI can be easily located by magic string "Microsoft SSL Protocol Provider" in the heap memory while the handshake process is hung temporarily. So, the encrypted TLS traffic can be obtained stealthily without any preliminary analysis.

Then, we choose three structures from three browsers to evaluate the real matching effect. The structures are *sslSessionID* (0x188), *ssl_session_st* (0x198), and *CSecureSocket* (0x170), which contains the TSI information, respectively. The value in parentheses is the size of structure in the test version. In the experiment, we successively access the above-mentioned websites, the matching process continues, and the processor time is suppressed below 5%. The experimental results are shown in Table 2. There are lots of repetitive accesses to a small amount of memory blocks. After removing the duplicate addresses, the number of blocks that contain the target structures is limited. The matching process is efficient, and the time consumption can be negligible, although there are lots of nontarget blocks associated with the bucket, especially for *Edge*. One reason is that the new LFH heap is introduced on Windows 10, which is called segment heap [44], and then more memory chunks would be traversed to obtain the target structures.

5.3. Comparison with Other Methods. In the next experiments, HTTA is compared with *Fiddler* and *mitmproxy*, which are the widely used web analysis tool, to demonstrate the impact on the original communication of the method. *Fiddler* interposes a proxy to intercept the TLS connections of browsers, while *mitmproxy* is configured as the transparent mode. As is similar to the last section, we, respectively, download the page with different sizes from the local TLS server and obtain the page loading time from the development tool of the browser. As shown in Figure 16, for the transfer of small page, the page loading time is near for different methods. But there may be errors in the loading time of around 10 ms because of the counter of operating system, as shown in the interval 10 KB~100 KB. While the size of transferred packets grows, the impact on transfer delay by *Fiddler* and *mitmproxy* will increase obviously, and the impact of our method tends to be negligible. The average latency of each communication is about 5 ms, while the maximum is about 10 ms at worst. Because HTTA only needs to correlate the TSI with the connection, it only duplicates the packets and does not intercept the connection continuously. *Mitmproxy* maintains the TLS communications on both sides because of the transparent proxy; the initial handshake would cause a lot of time, and the handshake impact will be ignored while the size of packets

Table 2: Statistics of the traversal on the LFH.

Program	Number of all traversed blocks (million)	Duration	Time consumption per million blocks	Proportion of target blocks in the traversal	Number of target blocks without duplications
Firefox	338	190s	562 ms	78%	537
Chrome	326	187s	574 ms	50%	343
Edge	381	207s	543 ms	25%	363

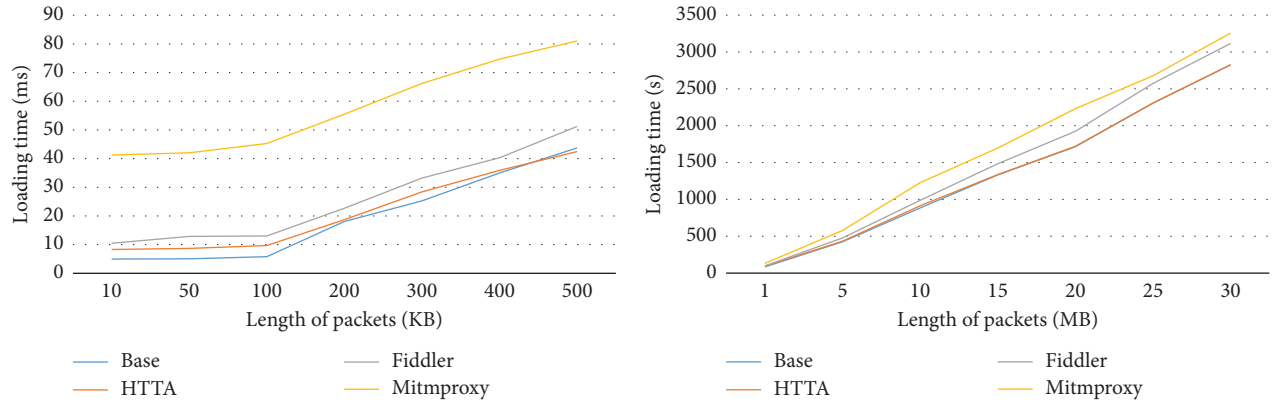

Figure 16: Comparisons with Fiddler and mitmproxy.

grows. *Fiddler* causes the lower network delay than *mitmproxy*, since it uses web proxy protocol on the client side.

Another difference is that *Fiddler* captures and processes packets in the user mode, while HTTA currently captures packets in the kernel mode and sends to the upper module. The communication between the user mode and kernel mode module may cause time delay. If HTTA works on the offline mode, it would have no impact on communications.

5.4. Correlation Cost in the Extreme Case. As aforementioned, it is easy to correlate TSI with the network connection in most cases by the connection port or session resumption information. In general, when the TLS handshake is finished, the related information can be rapidly extracted, so that the decryption can be started. The time spent of extraction is less than 5 ms. However, in the absence or loss of such information, it is necessary to try to decrypt and verify one record of TLS session to determine the relationship between the two. Specific methods are aforementioned. In this experiment, we choose some popular cipher suites of TLS protocol to evaluate and take *OpenSSL* as the crypto library even though it is not the optimal. As shown in Figure 17, with the growth of key numbers the verification time does not increase drastically for different cipher suites. If the number of key materials is less than 100, the time required to achieve the correlation with TCP stream is about 5 ms. The time consumption of the GCM mode is lower than the CBC mode. In reality, few TSI is generated at the same interval of time. Therefore, if the verification is needed, it is possible to keep the time overhead within a reasonable range. In the experiment, we use TLS 1.2 version because many websites do not support TLS 1.3. Actually, the new version only changes the handshake protocol, the symmetric encryption does not change, and it would not impact the result.

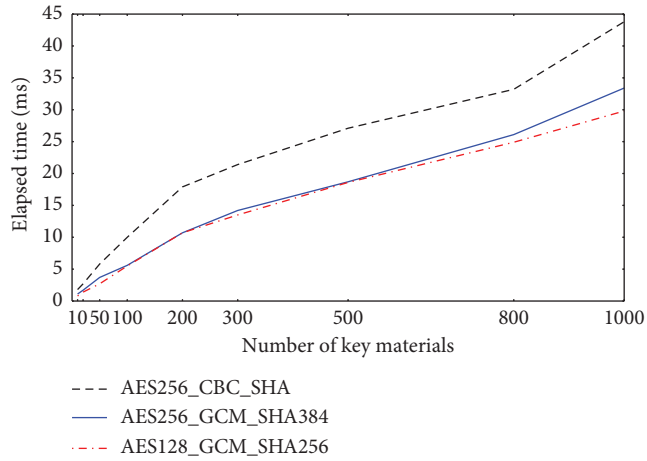

Figure 17: Elapsed time of correlation attempt for different cipher suites.

5.5. Localization of Target Objects. To support the large scale application of HTTA, we also design the helper of locating target structures and variable which can reduce the additional overhead of reverse engineering. Here, we perform the experiments to demonstrate the effect. Because there are abundant debug symbols for *Edge* and *Firefox*, we first choose *Chrome* 32-bit as the target program. To obtain the cache variable, function *SSLContext::GetInstance* is used as

the target for a test, from which the session structures can be referenced. The function in the version of *58.0.3029.110* is taken as the initial template. First, we try to get the target function by the traditional CFG matching. The base control flow graph is established, and two parameters described above, which are the matching error of the node label and the number of nodes, are also considered. Then, it matches all the CFGs in other versions of the program, so that it can quickly locate the target. The results show that it can locate the function precisely for the latter consecutive versions, while fails to find target in some prior versions. In fact, there are few subtle differences between different versions that some sides of CFG have changed. It is a challenge for applying graph isomorphism search method in that situation. Even though the complexity of graph isomorphism search is high, the actual match time is controlled because of the special constraints. Besides, we can observe that when the match error of vertex grows, the match result grows drastically. This means the fuzzy match based on graph isomorphism does not work well here.

Then, we continue to measure the ISMCP method by the next experiment, and also take the function above as the target. But we use the function in the older version *51.0.2704.103* as the template, from which we select the paths. The length of path changes from 3 to 6, and the number of each generated path locates in the range from 1 to 5. Then, the template is matched with all the functions of target version of the program by ISMCP. The result is shown in Figure 18; for each version of the program, we sort all the functions by their path similarities, from which we can see the obvious change in curvature. The target functions in different versions are in concentrated distribution, and hundreds of them rank within top 0.5%; it can still be observed clearly in the partial magnification which also bring some challenges for further analysis, although shrank the search scope. In addition, with the growth of path length, the number of generated similarity values goes down; meanwhile, the capability of locating target function also declines.

The main reason is that the CFG of the above function has a too simple structure and few vertexes. Therefore, we then choose another function *DoVerifyCertComplete*, which also changes in the different versions. Meanwhile, we can also obtain the target by one direct call reference in that function. The CFG of the function has around 20 vertexes, and structure is more complex. We choose the path length between 6 and 9, and the generated path number of each length varies from 16 to 30. The experiment steps are the same with the former one, and the result is shown in Figure 19. Now, we can see an excellent match result. For each version of the program, the top function is the target function. Most of the target functions in different versions are located within top 20, which is a better result.

Besides, the fact as shown in Figure 20, even the path number and length grows, the match time does not improve obviously. One reason is that the CFG structure is sparse, with the path length grows, the number of matched graphs will reduce, and the number of paths which can satisfy the constraint will also decrease. On the contrary, the number of matching operations increases when the path length is short. Additionally, because the module (*Chrome.dll*) that contains the target function consists of near 200 thousand functions, the process is hard to finish within the acceptable time range when the *bindiff* tool is used.

For the location of the structure members, we can use the same method. In the subsequent experiment, we choose the *SSL_SESSION_dup* function of *Chrome*, which references lots of session parameters, including master key. The version 65.0.3325.181 is used as the template to build the paths, and the path length is same as the above experiment. The result is shown as Figure 21 since the architecture of the function keeps stable in many versions; the top 10 functions are all the target function in the newer versions of target program.

Similarly, for *Firefox*, we can obtain the global cache variable by function *ssl_DestroySID*. In the next experiment, version 56.0.0 (64 bit) is used as the initial template to build the paths of ISMCP. Then, we try to locate the same function in the programs with newer version number. The result is shown in Figure 22, and the target function can be found in the top 5 values of the result for each newer version. Then, function *ssl3_FillInCachedSID* is further used in the experiment, from which we can obtain the offsets of specific session parameters. As shown in Figure 23, even though the similarities decrease in totality, the target can also be found in the top 10 of the results for version 58.0.2 and 60.0.2. But most matching values are located between the top 50 and 100 for version 65.0.1 and later, and then the difficulty of location is increased. It is better to update the matching template of the function due to the major structure change in the new versions.

Since the file size of module containing target functions is small for *Firefox*, we can perform the comparions by the *bindiff* tool. The result is not good, as shown in Figure 24. For *ssl3_FillInCachedSID,* the desired functions in the newer versions are not recognized. Function *ssl_DestroySID* is located only in version 60, due to the few changes of that version.

In fact, compared with the address change of the global variable, the member offset of target structure remains more stable, as shown in Table 3. In most cases, we only check whether the offset has changed in the new version.

5.6. Adaptation to the Program Change. The proposed method can be applied to the analysis of the popular browsers, also including other browsers with the same kernel. Meanwhile, some programs use the browser as the web component, and the corresponding traffic can also be directly analyzed, for example, *Weixin for Windows, Tencent video, Netease cloudmusic*, and so on. Besides, some client programs of cloud disk use *wininet* library for the secure communications, which can also be directly analyzed. Furthermore, the address of the target global variable may change in different versions of the program due to the compilation. So, we need to construct the offset database of the target variable in advance. For the structure member, it generally changes when the version of program has the major change.

Moreover, we investigate the variation of the target object size, which is used in the experiment of traversing the LFH. As shown in Table 4, the size can remain stable in different versions, which is sensitive to the major version

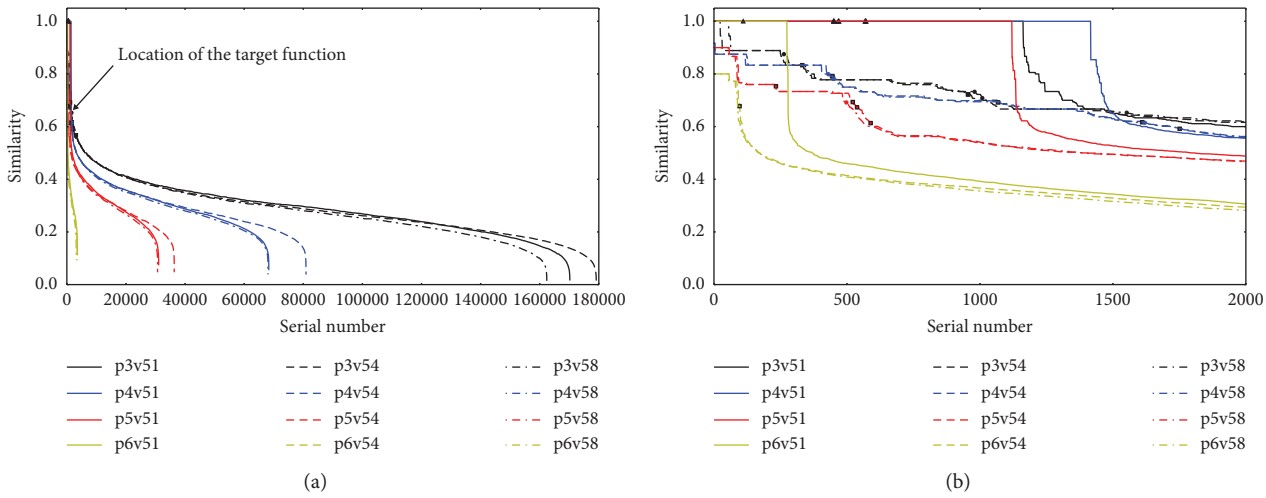

Figure 18: Result 1 of Chrome based on ISMCP. P3 denotes that the path length is 3, v51 denotes that the major version is 51, and others are similar.

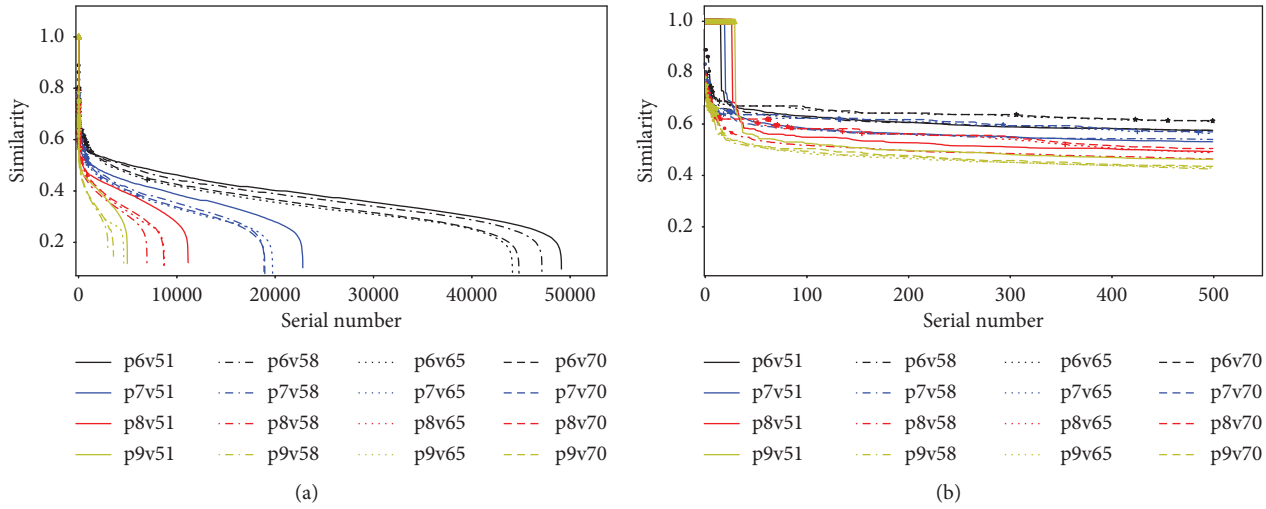

Figure 19: Result 2 of Chrome based on ISMCP.

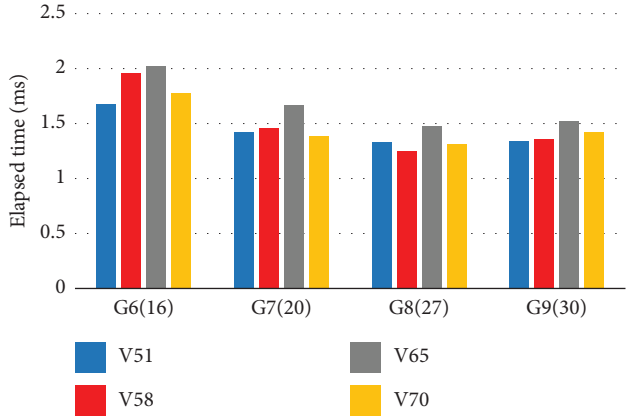

Figure 20: Execution time of matching per graph in different conditions. G6 (16) denotes that the path length is 6 and the number of paths is 16, others are similar.

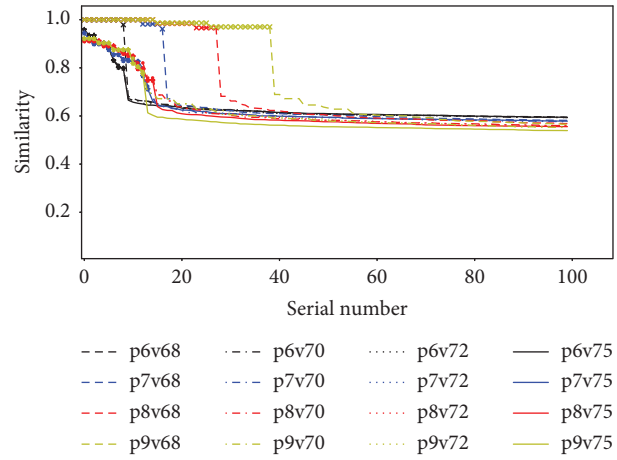

Figure 21: Result 3 of *Chrome* based on ISMCP.

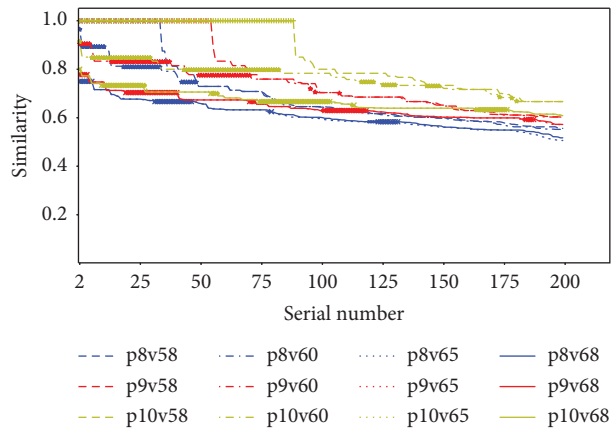

Figure 22: Result 1 of *Firefox* based on ISMCP.

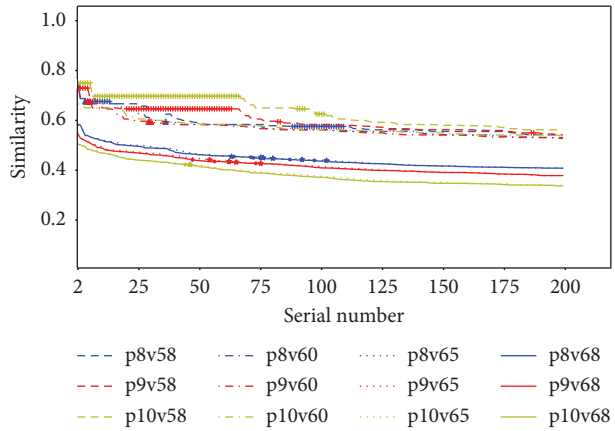

Figure 23: Result 2 of Firefox based on ISMCP.

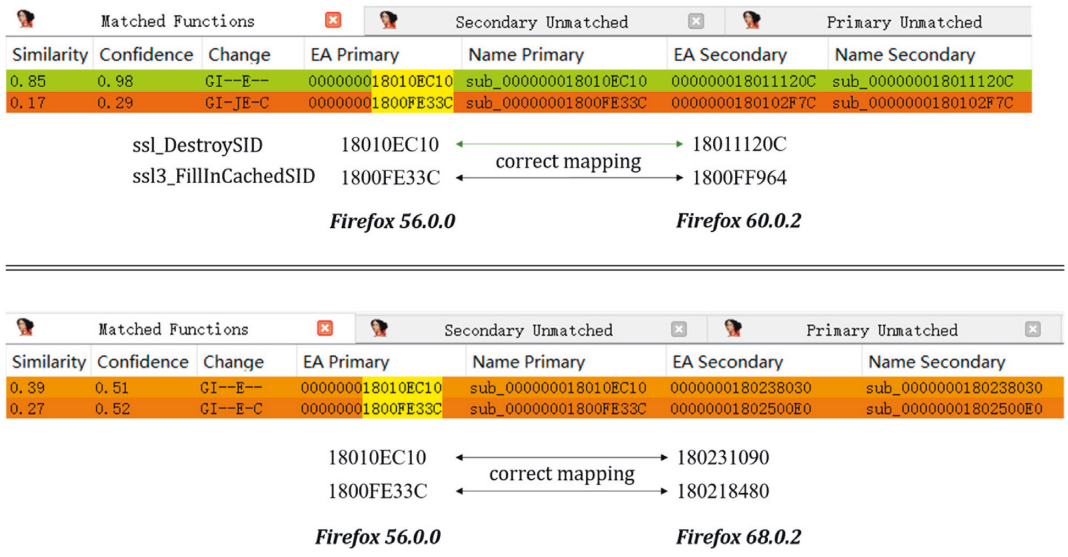

Figure 24: Comparisons of different *Firefox* versions by bindiff.

Table 3: Variation of the parameter offset in different versions.

Program	Structure	Member	Version	Offset value
Firefox	sslSessionID	keys	56.0	0xAC
			60.0.2	0xB5
			65.0.1	0xB5
Chrome	ssl_session_st	master_key	65.0.3325.181	0xC8
			70.0.3538.77	0xC8
			75.0.3770.80	0xC8
Edge	CSslUserContext	ReadWriteKey	10.0.14393.1613	0x18
			10.0.16299.15	0x18
			10.0.17134.1	0x18

Table 4: Variation of target objects of different versions.

Program	Object	Version	Size (bytes)
Firefox	sslSessionID	56.0	0x190
		60.0.2	0x188
		62.0.3	0x188
		65.0.1	0x188
Chrome	ssl_session_st	61.0.3163.100	0x1B0
		65.0.3325.146	0x198
		70.0.3538.77	0x198
		72.0.3626.121	0x198
IE/Edge	CSecureSocket	11.0.14393.1770	0x170
		11.0.14393.2273	0x170
		11.0.16299.15	0x180
		11.0.17134.1	0x198

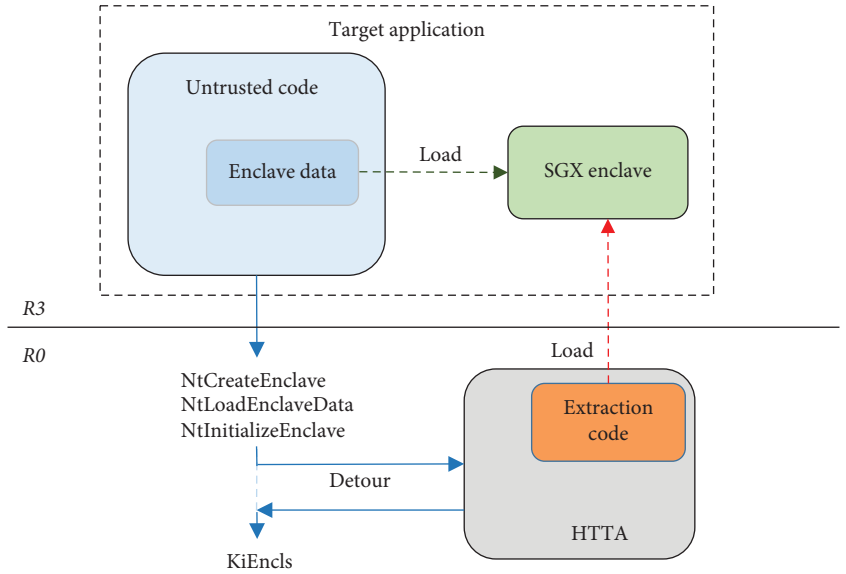

Figure 25: A solution to the case of SGX.

changes. In practice, we can properly increase the search range to improve the adaptability.

6. Discussion

The proposed method is practical and universal, as it does not depend on vulnerabilities of the program and protocol. It is mainly applicable for browsers and analogous programs since we discover the uniform pattern of information extraction, and it is applicable for the programs which adopt the similar crypto infrastructure. But it is difficult to directly extract the session information for the program that has the personalized implementation of TLS protocol. In the situation, the individual reverse engineering work is needed to make the analysis cover the program.

The version of browser program may change frequently; in most cases, we only need to check the variation with low cost, due to the semiautomated method. Little manual work

is needed to ensure the accuracy in practice. But when the target function in the new version has major changes, we would spend much time to update the new function template to locate the target parameters.

The noninvasive method of this paper means that it does not intervene the execution flow of target program. As mentioned above, the solution has the interception mode and audit mode. In the interception mode, few related handshake packets will be intercepted at the system kernel level, but there is no reassembly and decryption. Meanwhile, in the audit mode, the decryption rate may not reach the one hundred percent, but it can minimize the impact on the target system and application.

Besides, as a supplement, we also propose the extraction method based on the low fragmentation heap. When the amount of traffic of the program is small, the corresponding heap bucket may not be activated. If the bucket is needed to be forcedly activated, there would be the thread injection which slightly interferes with the target system.

For the standalone deployment, other security products may exist. Because the solution does not involve the function hooking methods, they can coexist. Moreover, as a defense solution, we can also add the related modules to the while list of security products.

While the Intel SGX is also a barrier to the solution, it has not been used by target programs. In that case, it needs to load the extraction code into the same enclave of the process when the target program initializes, which also causes the intervention. Because as a defense solution the module of HTTA can start in advance of other applications. One response solution on the Windows platform is shown in Figure 25. When the target program calls the *NtInitializeEnclave* function that corresponds to the *EINIT* privileged instruction, we load the enclave part of extraction module to the same enclave. In this case, the kernel function hooking is needed, and there are many stable methods.

7. Conclusions

In this paper, we propose a new TLS traffic large-scale automated analysis method for browsers and analogous programs, which is efficient and transparent to the programs. It extracts multiple types of data by several modes and correlates them together to accomplish the overall analysis of the target in real time. The experimental results have shown that the method can effectively capture and analyze all the packets, with the high decryption rate and low runtime overhead. Moreover, we propose the aided location method of targets to solve the program diversity problem, which can reduce the workload of binary analysis and support the framework.

In this paper, we mainly focus on the researching and performing experiments on the browser programs and would pay more attentions to other programs in the near future. Moreover, the automated method of binary reversing and parameter extraction should be continuously improved.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (General Program) under grant no. 61572253 and the Innovation Program for Graduate Students of Jiangsu Province, China (grant no. KYLX16_0384).

References

[1] G. K. Jayasinghe, J. S. Culpepper, and P. Bertok, "Efficient and effective realtime prediction of drive-by download attacks," *Journal of Network and Computer Applications*, vol. 38, no. 1, pp. 135–149, 2014.

[2] "RFC6101," August 2018, https://tools.ietf.org/html/rfc6101.

[3] "RFC5246," August 2018, https://tools.ietf.org/html/rfc5246.

[4] "RFC2818," August 2018, https://tools.ietf.org/html/rfc2818.

[5] "CVE-2014-0160," August 2018, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[6] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: results from the 2008 Debian OpenSSL vulnerability," in *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, pp. 15–27, ACM, Chicago, Illinois, USA, November 2009.

[7] "CVE-2012-4929," August 2018, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929.

[8] T. Duong and J. Rizzo, "Here come the ⊕ ninjas," August 2018, http://www.hpcc.ecs.soton.ac.uk/~dan/talks/bullrun/Beast.pdf.

[9] N. J. A Fardan and K. G. Paterson, "Lucky thirteen: breaking the TLS and DTLS record protocols," in *Proceedings of the IEEE Symposium on Security and Privacy 2013*, pp. 526–540, IEEE, San Francisco, CA, USA, May 2013.

[10] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, and D. A. Osvik, "Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate," in *Advances in Cryptology*, pp. 55–69, Springer, Berlin, Germany, 2009.

[11] K. Dan, M. L. Patterson, and L. Sassaman, "PKI layer cake: new collision attacks against the Global X.509 infrastructure," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, pp. 289–303, Springer-Verlag, Tenerife, Spain, January 2010.

[12] D. Akhawe and A. P. Felt, "Alice in warning land: a large-scale field study of browser security warning effectiveness," in *Proceedings of the USENIX Conference on Security 2013*, pp. 257–272, USENIX Association, Washington, DC, USA, August 2013.

[13] A. Parsovs, "Practical issues with TLS client certificate authentication," in *Proceedings of the Network and Distributed System Security Symposium 2014*, San Diego, CA, USA, February 2014.

[14] M. Kranch and J. Bonneau, "Upgrading HTTPS in mid-air: an empirical study of strict transport security and key pinning," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA., USA, February 2015.

[15] "TLS 1.3 draft," August 2018, https://tools.ietf.org/html/draft-ietf-tls-tls13-28.

[16] T. Nichols, A. Bates, J. Pletcher et al., "Securing SSL certificate verification through dynamic linking," in *Proceedings of the ACM Sigsac Conference on Computer and Communications*

Security, pp. 394–405, ACM, Scottsdale, AZ, USA, November 2014.

[17] "HTTP strict transport security (HSTS)," August 2018, https://tools.ietf.org/html/rfc6797.

[18] "Public key pinning extension for HTTP," August 2018, https://tools.ietf.org/html/rfc7469.

[19] "Certificate transparency," August 2018, https://tools.ietf.org/html/rfc6962.

[20] M. Husák, M. Čermák, T. Jirsík, and P. Čeleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting," *EURASIP Journal on Information Security*, vol. 2016, no. 1, 2016.

[21] "Deciphering malware's use of TLS (without Decryption)," August 2018, https://arxiv.org/pdf/1607.01639.pdf.

[22] X. C. Carnavalet and M. Mannan, "Killed by proxy: analyzing client-end TLS interception software," in *Proceedings of the the Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2016.

[23] Z. Durumeric, Z. Ma, D. Springall et al., "The security impact of HTTPS interception," in *Proceedings of the the Network and Distributed System Security Symposium*, San Diego, California, USA, February-March 2017.

[24] "Fiddler," August 2018, http://www.telerik.com/fiddler.

[25] "Mitmproxy," August 2018, https://github.com/mitmproxy/mitmproxy.

[26] "NSS key log format," August 2018, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format.

[27] "Jkambic," August 2018, http://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Jkambic-Cunning-With-Cng-Soliciting-Secrets-From-Schannel-WP.pdf.

[28] J. Rizzo and T. Duong, "Practical padding oracle attacks," in *Proceedings of the 4th USENIX Workshop on Offensive Technologies*, pp. 1–8, USENIX Association, Washington, DC, USA, August 2010.

[29] "BREACH," August 2018, https://github.com/nealharris/BREACH.

[30] M. Marlinspike, "More tricks for defeating SSL in practice," in *Proceedings of the DEFCON'17*, Winchester, NV, USA, July-August 2009.

[31] J. Liang, J. Jiang, and H. Duan, "When HTTPS meets CDN: a case of authentication in delegated service," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 67–82, IEEE, San Jose, CA, USA, May 2014.

[32] Y. Jia, Y. Chen, X. Dong, P. Saxena, J. Mao, and Z. Liang, "Man-in-the-browser-cache: persisting HTTPS attacks via browser cache poisoning," *Computers and Security*, vol. 55, pp. 62–80, 2015.

[33] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: Deep packet inspection over encrypted traffic," *Acm Sigcomm Computer Communication Review*, vol. 45, no. 4, pp. 213–226, 2015.

[34] L. Wu, B. Chen, S. Zeadally, and D. He, "An efficient and secure searchable public key encryption scheme with privacy protection for cloud storage," *Soft Computing*, vol. 22, no. 23, pp. 7685–7696, 2018.

[35] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan Zee (north) bridge: mining memory accesses for introspection," in *Proceedings of the ACM Sigsac Conference on Computer & Communications Security*, pp. 839–850, ACM, Berlin, Germany, November 2013.

[36] B. Taubmann, C. Frädrich, D. Dusold, and H. P. Reiser, "TLSkex: harnessing virtual machine introspection for decrypting TLS communication," *Digital Investigation*, vol. 16, pp. S114–S123, 2016.

[37] Q. Feng, A. Prakash, M. Wang, C. Carmony, and H. Yin, "ORIGEN: automatic extraction of offset-revealing instructions for cross-version memory analysis," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, pp. 11–22, ACM, Xi'an, China, May-June 2016.

[38] P. Vadrevu, J. Liu, B. Li, B. Rahbarinia, K. H. Lee, and R. Perdisci, "Enabling reconstruction of attacks on users via efficient browsing snapshots," in *Proceedings of the Network and Distributed System Security Symposium 2017*, San Diego, CA, USA, February-March 2017.

[39] "Desktop browser market share worldwide," August 2018, http://gs.statcounter.com/browser-market-share/desktop/worldwide.

[40] "Chromium source code," August 2018, https://chromium.googlesource.com/chromium/src.git/+/62.0.3188.1/net/ssl/.

[41] "OpenSSL," August 2018, https://www.openssl.org.

[42] "LibreSSL," August 2018, http://www.libressl.org.

[43] "Firefox source code," August 2018, http://releases.mozilla.org/pub/firefox/releases/.

[44] "Segment heap internals," August 2018, https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals.pdf.

[45] F. E. Allen, "Control flow analysis," *Acm Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[46] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the ACM Sigsoft International Symposium on Foundations of Software Engineering*, pp. 389–400, ACM, Hong Kong, China, November 2014.

[47] S. Alam, R. N. Horspool, and I. Traore, "A framework for metamorphic malware analysis and real-time detection," *Computers and Security*, vol. 48, pp. 212–233, 2015.

[48] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[49] G. Dan, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, London, UK, 1997.

[50] "IDA," August 2018, https://www.hex-rays.com/products/ida/index.shtml.

[51] "Windows filtering platform," August 2018, https://docs.microsoft.com/zh-cn/windows/desktop/FWP/windows-filtering-platform-start-page.