COMPUTERS & SECURITY 88 (2020) 101516

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Exploiting non-uniform program execution time to evade record/replay forensic analysis

Yang Hu[a], Mingshen Sun[b], John C.S. Lui[b,*]

[a] School of Computing, National University of Singapore, 13 Computing Drive, Singapore
[b] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong, China

ARTICLE INFO

Article history:
Received 7 January 2019
Revised 19 March 2019
Accepted 11 April 2019
Available online 3 May 2019

Keywords:
Forensic analysis
Record/replay system
Anti-forensics
Vulnerability exploitation
Malware analysis

ABSTRACT

Record/replay system is an essential and widely used module in forensic analysis, as it can help forensic analysts to reconstruct programs' behaviors. However, the security implication of record/replay systems (i.e., whether record/replay systems can faithfully reproduce all behaviors of a program) has not been thoroughly studied. This paper is the first work which investigates and explores the security limitations of record/replay systems from the perspective of software forensics. In particular, we reveal a type of vulnerability in record/replay systems caused by non-uniform program execution time. A program can exploit this vulnerability to prevent its malicious behavior from being replayed. We conduct a series of experiments on three platforms (i.e., web browser, mobile operating system and virtualized sandbox) to illustrate the wide footprints of the vulnerability. Finally, we discuss possible methods to mitigate the vulnerability. The goal of this work is to study the inherent security limitations of record/replay systems, discover the vulnerability and explore potential mitigation methods, from which forensic analysts can be informed and cautious when applying record/replay systems to software forensics.

1. Introduction

In software testing and fault tolerance analysis, *record/replay systems* are essential components, since software analysts can use them to reconstruct programs' behavior and to check whether these programs satisfy their software specifications. They record *events that may affect the program flow* (e.g., external inputs and random decisions) and then reproduces the bugs by replaying the same events. Chen et al. (2014). An example of a record/replay system is *RERAN* (Gomez et al., 2013), an Android user-interface testing tool, which records screen touch and key press events of an application when a human tester interacts with it, and then RERAN can automatically test the application by replaying those recorded events.

Some record/replay systems such as *RnR-Safe* (Shalabi et al., 2018) and WebCapsule (Neasbitt et al., 2015) are designed for software forensics. However, a program may exploit *vulnerabilities* in such record/replay systems to prevent its malicious behavior from being discovered or replayed, which significantly diminishes the values of the underlying forensic analysis. The primary goal of this paper is to explore these limitations, state their vulnerabilities and explore potential mitigation methods. This way, forensic analysts will be more informed and cautious when handling various forensic tasks.

To the best of our knowledge, this paper is the first work which systematically studies security issues of *record/replay systems* from the perspective of software forensics. We discover a kind of security vulnerability caused by *non-uniform*

* Corresponding author.
E-mail addresses: dcshuy@nus.edu.sg (Y. Hu), mssun@cse.cuhk.edu.hk (M. Sun), cslui@cse.cuhk.edu.hk (J.C.S. Lui).

https://doi.org/10.1016/j.cose.2019.04.012
0167-4048/© 2019 Elsevier Ltd. All rights reserved.

program execution time, which indicates that program execution time cannot be exactly the same as the program re-execution time. By exploiting this vulnerability, a program can identify whether it is replayed in a forensic context, thereby hiding its malicious behavior during the replay stage.

To illustrate the type of vulnerability in real world, we conduct attack experiments on record/replay systems for three different platforms: web browser, mobile operating system and virtualized sandbox. Our experimental results show that all our attack programs can successfully hide malicious behavior on the corresponding record/replay systems.

Lastly, we discuss how to mitigate the vulnerability through *hardware-assisted record/replay analysis*, *ensemble record/replay analysis* (i.e., combine multiple record/replay methods into one), *symbolic execution* and *exploit detection*.

The rest of this paper is organized as follows. In Section 2, we briefly introduce record/replay systems for software forensics. In Section 3, we thoroughly introduce details on the vulnerability. Section 4 demonstrates our attack experiments. Finally, we give a brief discussion on vulnerability mitigation in Section 5.

2. Background

In this section, we introduce the background of record/replay systems. We use a motivating example to explain the basic ideas of widely-used record/replay methodologies.

2.1. A motivating example: gplay forensic analysis on a QR code scanner

We introduce a simple forensic case, which will be used to explain six widely-used record/replay methodologies later. Let us consider that Alice, a government clerk, uses a QR code scanner application to scan a malicious QR code which links to a phishing website. Alice enters her email account password to the phishing website. As a result, many secret government documents are stolen and disclosed to the public. To investigate whether Alice deliberately leaked these secret documents or suffered from a phishing attack, forensic analysts need to obtain more trustworthy details such as how Alice visited the phishing website and how her email account password was leaked, which a record/replay system could help to reconstruct.

Now let us demonstrate the technical details of record/replay forensic analysis on the QR code scanner. QR code scanner consists of two threads: *camera thread* and *decoder thread*. The camera thread captures an image and sends it to the decoder thread, which extracts the URL in the QR code and informs the camera thread of the decoding result. Fig. 1a illustrates the workflow of the QR code scanner. The camera thread first captures the image 1, and then the decoder thread analyzes the image 1 to extract the needed information. During the analysis, the image 2 and the image 3 are ignored because the camera thread has not received the decoding result of image 1 yet. Because the QR code in the image 1 is incomplete, the decoding result of the image 1 is "`decoding failure`". Now the camera thread sends the image 4 to the decoder thread, for it is the first image after receiving the decoding result of the image 1. The image 5 and the image 6 are ignored just like the image 2 and the image 3. As the image 4 contains the complete QR code, the decoding result of the image 4 is "`decoding succeeded`".

Let us assume that the QR code scanner behaves as illustrated in Fig. 1a during the record stage, and the image 3, 4, 6 contain a complete QR code which links to the phishing website. A record/replay system should store these images during the record stage and exactly replay them to the camera thread during the replay stage. If the behavior in replay stage is similar to Fig. 1b, which is different from Fig. 1a, the replay is an *unsuccessful replay*.

2.2. Record/Replay methods

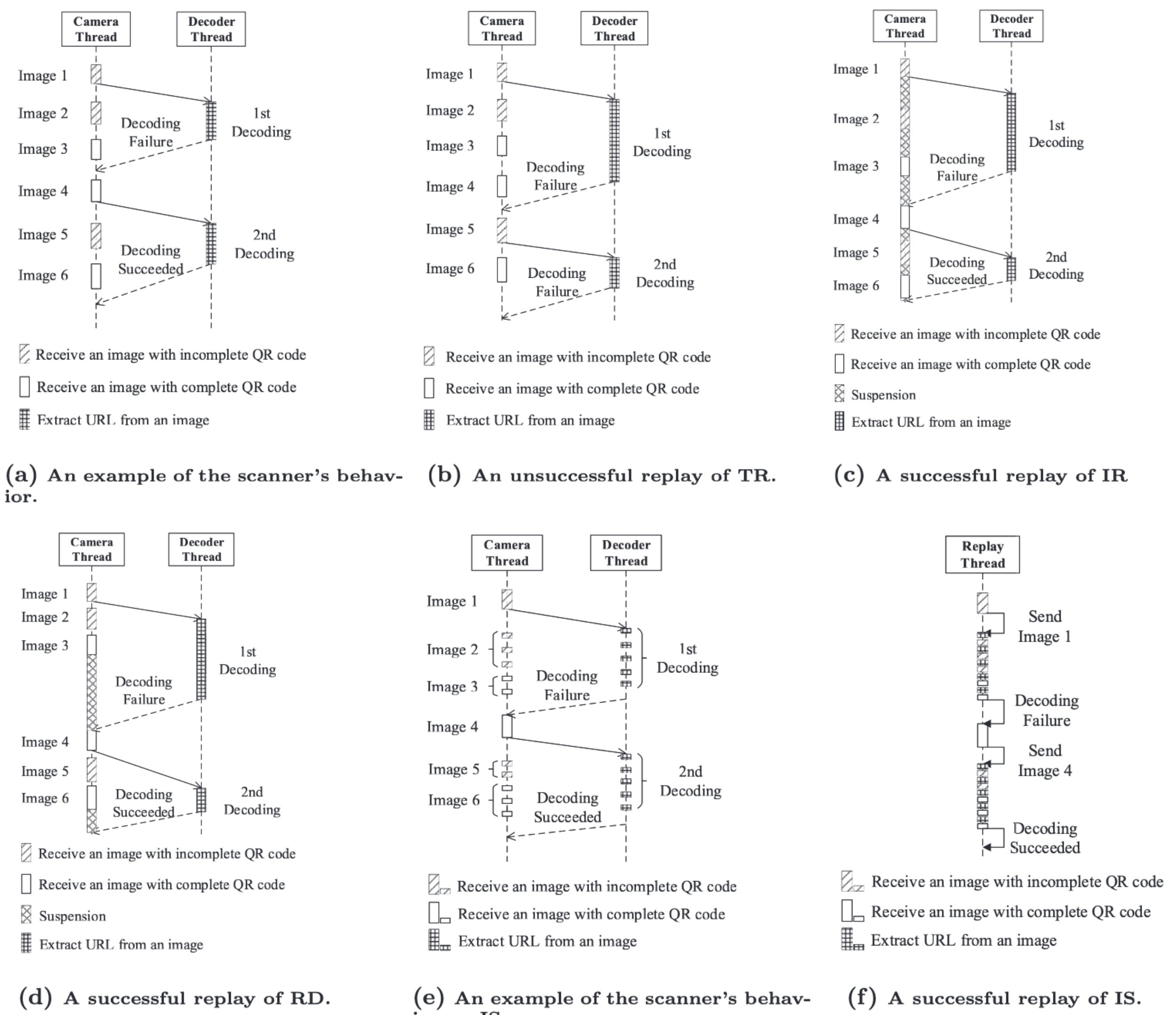
According to a recent study (Chen et al., 2015), there are six typical methodologies for record/replay analysis.

(1) Time-based record/replay (TR) method: The first record/replay method is the *time-based record/replay* (TR). The basic idea of this method is to replay an external input in the replay stage according to its arrival time in the record stage in order to reproduce some events regarding the arrival time of external inputs (Halpern et al., 2015). Consider that we use the TR method to conduct record/replay analysis on the QR code scanner. It needs to record these images and their arrival times during the record stage. During the replay stage, it replays an image only when the current time is the same as its arrival time in the record stage.

We focus on one limitation of this method that the execution time of a program during the record stage may differ from that during the replay stage due to other events such as thread scheduling and interrupt requests. This difference, if sufficiently big, may cause an unsuccessful replay. Fig. 1b shows an unsuccessful replay. In this example, the decoding time of the image 1 in Fig. 1b is longer than that in Fig. 1a due to different thread scheduling. Therefore, after decoding the image 1, the decoder thread ignores the image 4 but decodes the image 5 in Fig. 1b. Because the QR code in the image 5 is *incomplete* (i.e., the image only contains a part of QR code or does not contain any QR code), the decoding result of the image 5 will result a "`decoding failure`" event. As a result, the program behavior in Fig. 1b is different from that in Fig. 1a due to the different execution time of decoding the image 1.

(2) Check-and-Restart (CR) method: To overcome the problem stated above, researchers proposed to make the TR method tolerate faults introduced by program execution time. One such proposal is the *check-and-restart* methodology (CR) (Lee et al., 2011; Montesinos et al., 2008; Voskuilen et al., 2010; Xu et al., 2018), which separates the record stage into several sub-stages, records the execution environment at the beginning of each sub-stage and creates *checkpoints* at the end of each sub-stage during the record stage. For each sub-stage, if its replay result does not match its checkpoint, it needs to restore the current execution environment to the beginning of this sub-stage and then redoes the replay on this sub-stage.

To reconstruct the scanner's behavior in Fig. 1, the improved CR needs to record the decoding result and execution environment at the beginning of each sub-stage in the record stage. If the second decoding result in the replay stage is



(a) An example of the scanner's behavior. (b) An unsuccessful replay of TR. (c) A successful replay of IR

(d) A successful replay of RD. (e) An example of the scanner's behavior on IS. (f) A successful replay of IS.

Fig. 1 – Example of record/replay analysis on a QR code scanner.

different from that in the record stage, we only need to restore current execution environment to the latest recorded execution environment (i.e., the execution environment when the camera thread receives the first decoding result), rather than going back to the execution environment at the beginning of the record stage. In summary, the CR method reduces the timing overhead by storing multiple system checkpoints. But it may cause *severe storage overhead* because it needs to record the execution environment several times.

(3) Instruction-Based Replay (IR) Method: Different from the TR and CR methods, the IR method replays external inputs according to instruction execution progresses rather than time (Dunlap et al., 2002; King et al., 2005). Specifically, when an external input happens during the record stage, the IR method records values in the program counter and instruction counter (i.e., a counter containing the number of executed instructions) for each thread as a *milestone*. During the replay stage, the IR method suspends a thread when the thread reaches a milestone. After all threads of the program have been suspended, the IR method replays the external inputs and then wakes up all these threads.

Fig. 1c demonstrates how the IR method conducts a successful replay on the QR code scanner. Here, we assume that some events make the program execution time in the replay stage different from that in the record stage. For example, the decoder thread in the replay stage runs slower than that in the record stage during the first decoding phase due to different thread schedule. Then, the IR method suspends the camera thread when it arrives at certain milestone until the decoder thread reaches it. As a result, the IR method can ensure that the image 3 will be replayed before the camera thread receives the first decoding result, and the image 4 will be replayed after the camera thread receives the first decoding result. Therefore, the image 4 can still be sent to the decoder thread, which ensures the second decoding result in the replay stage is the same as that in the record stage.

(4) Race detection (RD) method: The IR method can replay some events regarding external inputs, but cannot replay the events regarding data race. To reproduce events related to data race, researchers propose a record/replay method based on *race detection* (RD). In essence, it attempts to detect all *data races* in a program and ensure all data race results in the replay stage are the same as these in the record stage (Hsiao et al., 2014; Xue et al., 2009). For instance, in Fig. 1a the decoder thread sends the first decoding result to the camera thread, and then the camera thread uses it to decide whether the image 4 should be sent to the decoder thread. In fact, this is a typical reader-writer data race, and the RD method can detect it. Therefore, it is necessary to ensure that the first decoding result is received after the camera thread captures the image 3 and before the camera thread captures the image 4. To this end, during the replay stage, RD suspends the camera thread after capturing the image 3. When the decoder thread sends the first decoding result to the camera thread, RD replays the image 4 and then wakes up the camera thread. The other data race situation regarding the second decoding result can also be handled in this manner.

(5) Instruction serialization (IS) method: State-of-the-art RD methods can detect common data races such as data races in the shared memory, but fail to detect some special data races such as races in the *covert channel* (Wendzel et al., 2015). If a RD method ignores certain data races in a program, the replay on the program may be an unsuccessful replay because the results of these ignored data races in the replay stage may be different from these in the record stage.

To address this problem, researchers propose a record/replay method called *instruction serialization* (IS) (Devietti et al., 2009; 2011). In essence, the IS method regulates the execution order of all instructions in a program, and it intercepts routines to ensure that program executions in the record and replay stage obey the same order. Note that when the instruction execution orders remain the same, the data race results become the same. One possible implementation of the IS method is illustrated in Fig. 1e and Fig. 1f. In Fig. 1e, the scanner runs on a single CPU with a time-sharing policy during the record stage. For each time slice, only one thread is running, therefore, all instructions of the scanner are executed sequentially. The IS method records the instruction execution order during the record stage and then re-executes these instructions on one thread in similar order during the replay stage, which is shown in Fig. 1f.

(6) Load logging (LL) method: Note that one major performance overhead of the IS method is that it is difficult, if not impossible, to extend it to the parallel execution framework, and given that many CPUs are multi-cores, the IS method will significantly underutilize the CPU resources. Researchers propose a novel record/replay method called *load logging (LL)* (Lee et al., 2011; Narayanasamy et al., 2005; Patil et al., 2010), which can both reproduce the aforementioned events and utilize CPU resources as much as possible. During the record stage, the LL method records the initial values of registers, instruction sequence, the execution results of all *load instructions* (i.e., read the value in certain memory address and store it in specific register). During the replay stage, the LL method restores registers with their initial values and then re-executes the recorded instruction sequence. In particular, when facing a load instruction, the LL method does not re-execute it but restores its execution results to its corresponding results.

LL method may cause severe performance overhead, especially for its software-only implementations. The major reason is that LL method relies on *system call side effect analysis* and *shared memory access order analysis*, which usually involve a large amount of computation[1].

3. A type of vulnerability

In essense, malware utilizes non-uniform program execution time to generate a new random number, which is then used to conduct challenge-response authentication. The malware will hide its malicious behavior once the authentication fails. In theory, the vulnerability exploitation works on record/replay systems which use TR, CR and IR methods, because malware can observe the non-uniform program execution time, which can be used to conduct authentication. The vulnerability exploitation may not work on record/replay systems which use the RD method, when the data race used to observe the non-uniform program execution time can be recognized and hindered. Record/replay systems which use IS or LL methods do not have such vulnerability, since all execution states of malware are stored or reconstructed so that malware cannot observe the non-uniform program execution time in the replay stage. Then we will give the threat model first and introduce details on vulnerability exploitation.

3.1. *Threat model*

We assume that the malware can access the Internet or send/receive SMS (if the malware is designed for mobile platform). Malware can also interact with users so that it can receive users' inputs. The malware does not have the root privilege. The malware has a payload which contains its core malicious code. We say the malware evades the record/replay system successfully if the malicious behavior caused by running the payload in the record stage cannot be reconstructed in the replay stage. Besides, we assume that TR, CR and IR methods are used to conduct record/replay forensic analysis. We do not consider RD, IS and LL methods, since they are too heavy-weight and no existing record/replay forensic systems use those methods.

3.2. *Vulnerability exploitation*

3.2.1. *Producing a new random number based on non-uniform program execution time*

Generating a new random number should have been easy to accomplish, since many operating systems have random number generation components. For example, in Linux, a random number can be obtained from the `/dev/random` (Barak and Halevi, 2005). However, to avoid the replay divergence caused by a random number, record/replay systems may record the random number generated by the component during the record stage, which is then replayed to the program

[1] https://software.intel.com/en-us/articles/pinplayfaq.

sample when it asks for a new random number during the replay stage. In other words, the random number obtained by the program sample during the replay stage is the same as that during the record stage, while our trick requires that the program sample uses *two different* random numbers during the record stage and the replay stage respectively.

To address this issue, we generate a new random number using *non-uniform program execution time*, which basically means the time cost of a program execution in the replay stage is usually different from that in the record stage, and this difference cannot be precisely predicted. For example, consider a program that can solve the sum of an integer array. Ideally, given the same integer array, all the executions of the program would take the same amount of time. But, in fact, some may take longer time than others. A recent study (Chen et al., 2014) points out two sources/causes of non-uniform program execution time. First, program execution time is highly associated with a series of factors like:

- Physical Factors: temperature and voltage affect the performance of the electron components.
- Factors in CPU: branch prediction results decide which instruction will be executed in advance. Therefore, they affect program execution time.
- Factors in IRQs: the position where an interrupt blocks a program affects the program execution time.
- Factors in Thread Scheduling: the priority of the thread determines the time cost of its execution.
- Factors in Storage: the cache layout, memory layout, disk layout, magnetic needle status (for mechanical disk) and wear leveling status affect the time cost of storage operations.

Second, the operations that a record/replay system conducts during the record stage have different time consumption from operations during the replay stage. For example, a record/replay system may *write* key events to file system during the record stage, while it may *read* those events from storage during the replay stage.

A program sample can generate a new random number in two phases. First, it uses a *software counter* to measure the execution time of a program sample. In Fig. 2, the program sample first creates a counter and initialize it to 0. Then the program sample repeats the increment operations of the counter asynchronously. After sleeping for several seconds, the program sample stops the counter and gets the current value in the counter. Second, we use state-of-the-art random number generators such as (Barak and Halevi, 2005; Corrigan-Gibbs and Jana, 2015) to transform the value in the counter to a new random number.

3.2.2. *Challenge-response authentication*

The program sample can use a random number to conduct *challenge-response authentication* to determine whether it is in a replay context or not. For instance, the program sample first sends the challenge with a random number to the attacker. After receiving the challenge, the attacker sends its response with the received number back to the program sample. Finally, the program sample executes the malicious code only

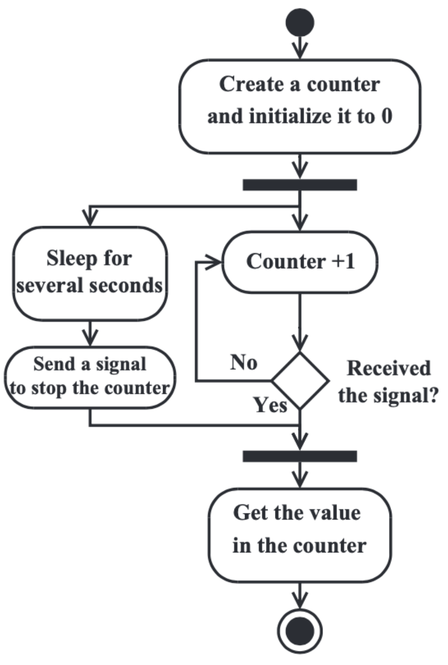

Create a counter and initialize it to 0
Sleep for several seconds
Counter +1
Send a signal to stop the counter
No
Received the signal?
Yes
Get the value in the counter

Fig. 2 – Use non-uniform program execution time to generate a new random number.

when the received response and the challenge have the same number.

Fig. 3 illustrates the authentication protocol. The program sample and the attacker share one encryption key denoted by K. The record/replay system is a communication interceptor between the program sample and the attacker. Therefore, the record/replay system can intercept all challenges and responses before they are received. During the record stage, the program sample generates a random number S_1 and sends the challenge $C(K, S_1)$. The record/replay system intercepts $C(K, S_1)$ and then sends $C(K, S_1)$ to the attacker. Then the attacker sends a response $R(K, S_1)$, which is then intercepted by the record/replay system. Since $R(K, S_1)$ is a type of external input to the program sample, the record/replay system records $R(K, S_1)$ and then sends it to the program sample. Since $C(K, S_1)$ and $R(K, S_1)$ contain S_1, the program then sample executes its malicious code. In the replay stage, the program sample produces a new random number S_2 and sends the challenge $C(K, S_2)$. Since the record/replay system only replays what has been recorded, it sends $R(K, S_1)$) back to the program sample. Since $R(K, S_1)$ contains S_1 which is not what $C(K, S_2)$ contains, the program sample does not execute its malicious code.

In the example mentioned above, we assume that the program sample can communicate with the attacker via network. If the program sample does not have the network permission, the authentication cannot be done between the program sample and the attacker. In that case, the program sample may conduct the authentication via the user interface. For example, the program sample can produce the verification code based on random numbers and ask the user to input the verification code. The program sample runs malicious code only

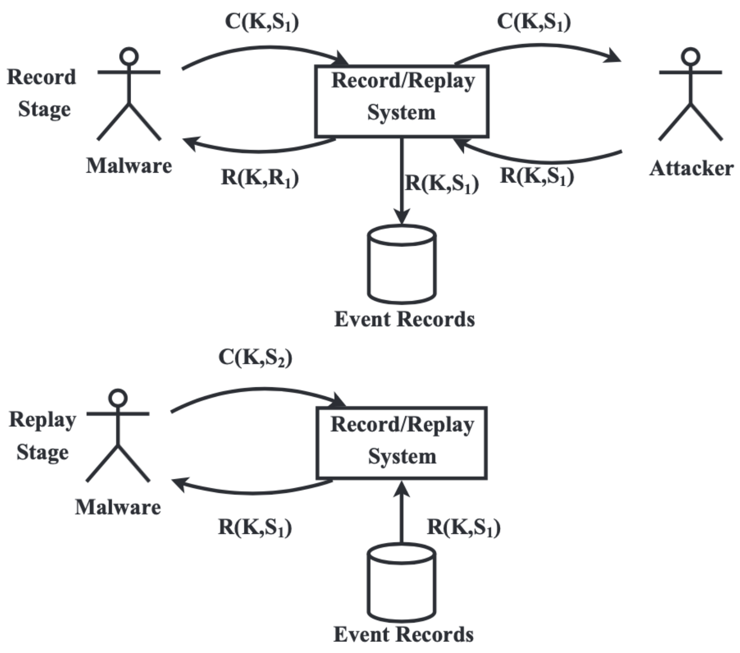


Fig. 3 – Use challenge-response authentication to hide malicious behavior.

when the user correctly inputs the verification code. Since the verification code in the replay stage is different from that in the record stage, the program sample does not execute its malicious code.

4. Evaluation

In this section, we conduct attack experiments on three different platforms: web browser, mobile operating system and virtualized sandbox.

4.1. *Web browser*

Conducting forensic analysis to investigate web-based security incidents is necessary, because it helps security researchers to better understand security incidents and propose defense mechanism. To illustrate the security issues of web browser forensic systems in practice, we conduct a proof-of-concept attack on *WebCapsule* (Neasbitt et al., 2015), which is a record/replay based forensic system for web browsers.

WebCapsule works in two phases. During the record stage, a user launches a browser to visit web pages, while WebCapsule records the user's inputs to the browser (e.g., key press, mouse click and network flow) by monitoring the *Web rendering engine* and *V8 JavaScript engine* in the browser. During the replay stage, WebCapsule replays those external inputs, and it follows the time order of those external inputs in the record stage.

Besides, WebCapsule implements a *self-healing mechanism* to mitigate potential divergence caused by obstinate events. During the record stage, it records *DOM trees* of the web pages visited by the user. During the replay stage, WebCapsule restores those web pages one by one and replays external inputs on their corresponding web pages.

To set up our experimental environment, we downloaded the source code of WebCapsule from its repository[2] and deployed it on *Ubuntu 15.10* and the *Google Chrome* browser. In addition, we build a website using *HTML* and *JavaScript* hosted on *Apache Tomcat 7.0*. Our website only has one button. If the user clicks the button, a piece of JavaScript code to conduct the attack will be executed in the browser. The attack process is shown in Algorithm 1. The function ATTACK() describes key procedures to hide malicious behavior in the replay stage. First, it invokes the function OBSERVE() n times to observe the non-uniformity of program execution time. Then it produces a random number R based on those observed results (i.e., $A_1, \ldots, A_n$) and sends the ciphertext of R to the attacker. Next, it receives the response and decrypts it as R'. If $R = R'$, it submits the user's private information (e.g., email address and password) to the attacker. The function OBSERVE() first applies a variable s and initializes it to 0, and then it sends an HTTP request to the server and then repeats executing incremental operations on the variable s until the program receives the corresponding response from the server. Finally, it returns s as an observed result. The experimental results show that our attack are successfully conducted. We present the whole attack process for WebCapsule in a video which can be accessed via a YouTube link[3].

4.2. *Mobile operating system*

Mobile malware analysts or forensic analysts usually adopt record/replay analysis to investigate and understand security

[2] https://github.com/perdisci/WebCapsule.
[3] https://youtu.be/FvopAIlQOCM.

Algorithm 1 Hide malicious behavior in WebCapsule.

```
1:  function ATTACK()
2:      $A \leftarrow Array(n)$;
3:      for $i = 1 \rightarrow n$ do
4:          $A_i \leftarrow$ OBSERVE_HTTP;
5:      end for
6:      produce a random number $R$ based on $A_1, \ldots, A_n$;
7:      encrypt $R$ and send an HTTP request with its ciphertext
        to the attacker;
8:      receive an HTTP response and decrypt it as $R'$;
9:      if ($R = R'$) then
10:         send user's private information to the attacker;
11:     end if
12: end function
13:
14: function OBSERVE_HTTP()
15:     $s \leftarrow 0$;
16:     send an HTTP request to server;
17:     repeat
18:         $s \leftarrow s + 1$;
19:     until (receive its response);
20:     return $s$;
21: end function
```

events on mobile platform. In particular, they usually focus on recording/replaying the events regarding the interaction between users and the mobile devices such as finger operations, because those events are usually highly related to the attack or malicious behavior. For example, in the motivating example introduced in Section 2.1, the clerk needs to operate the QR code scanner to scan malicious QR code and input the email password to the phishing website. If the finger operations are not precisely recorded and replayed, the phishing website may not appear or the malicious code that steals clerk's data may not be executed.

To record/replay finger operations, one of the most common methods used by Android record/replay systems such as (Gomez et al., 2013; Halpern et al., 2015; Qin et al., 2016) is to capture/inject relevant events via `/dev/input/event*`, which is a set of device files in Android OS that serves as the interface to the input device module in kernel. When a user operates a mobile device, the finger operations are encoded into a sequence of events in a standard five-field format Gomez et al. (2013) and stored in kernel space. Those events can be extracted by reading those device files. Artificial events can also be made based on the event format and injected to kernel by writing the events to those device files.

We find malware can use our approach to evade record/replay systems based on `/dev/input/event*`. For example, malware can generate a random number based on the non-uniform program execution time and ask the user to input the number. Although events regarding finger operations can be collected via `/dev/input/event*` in the record stage, the random number generated in the replay stage is different so that replaying those events inputs an unmatched number that can not help to pass the authentication held by the malware.

To verify the opinion we state above in practice, we perform an attack experiment on *FRep*[4], which is an Android record/replay tool based on `/dev/input/event*`. We conduct the experiment on *Google Nexus 6P* smartphone with Android 6.0 OS. We implemented an Android application with four activities as shown in Fig. 4. When we start our application, the default activity with `START` button is displayed. After a user clicks the `START` button, the application will produce random numbers based on non-uniform program execution time, generate verification code with the random numbers and show the verification code in the authentication activity. To finish the authentication task, a user needs to input the verification code and click the `SUBMIT` button. If the code entered by the user is the same as the verification code, our application will run malicious payload. Otherwise, our application will behave normally.

In our attack experiment, we click the `START` button,input the verification code and click the `SUBMIT` button. Since we correctly input the verification code, the program sample carries out the malicious activity after finishing those operations. Then we let FRep replay our operations and check whether the program sample still conducts the malicious activity. The experimental results show that the payload is not executed during the replay stage, which indicates our attack succeeds. We show the whole attack process in a video which is accessible via a YouTube link[5].

4.3. *Virtualized sandbox*

Hardware virtualization has been widely used to build a secure production or analysis environment, and record/replay analysis is also used to facilitate forensic or security analysis in a virtualized sandbox such as ROP attack defense (Shalabi et al., 2018), memory forensic analysis (Dolan-Gavitt et al., 2013) and honeypot forensic analysis (Srinivasan and Jiang, 2011). For example, RnR-Safe (Shalabi et al., 2018) is a security framework to enhance the security hardware features such as return address stack that may not be always precise in detecting ROP attacks. In essense, it conducts record/replay analysis in two steps. First, RnR-Safe records the events such as program inputs and attack alarms to a log file on a recording hypervisor. Second, it uses checkpointing and alarm replayers to reconstruct the attack on the replaying VMs and analyze whether the recorded alarms are false positives or not.

Despite the merits of RnR-Safe, we find RnR-Safe can be evaded through our approach. Intuitively, RnR-Safe works when checkpointing and alarm replayers can precisely reconstruct program behavior in the record stage. But malware may exploit the vulnerability caused by non-uniform program execution time to hinder reconstructing attacks on the replaying VMs. As a result, RnR-Safe recognizes the alarm events reported in the record stage as a false positive rather than a real attack.

Since we fail to obtain RnR-Safe's source code or binary, we implement a prototype tool based on QEMU to imitate the

[4] https://play.google.com/store/apps/details?id=com.x0.strai.frep.

[5] https://youtu.be/i6hMoHXmodE.

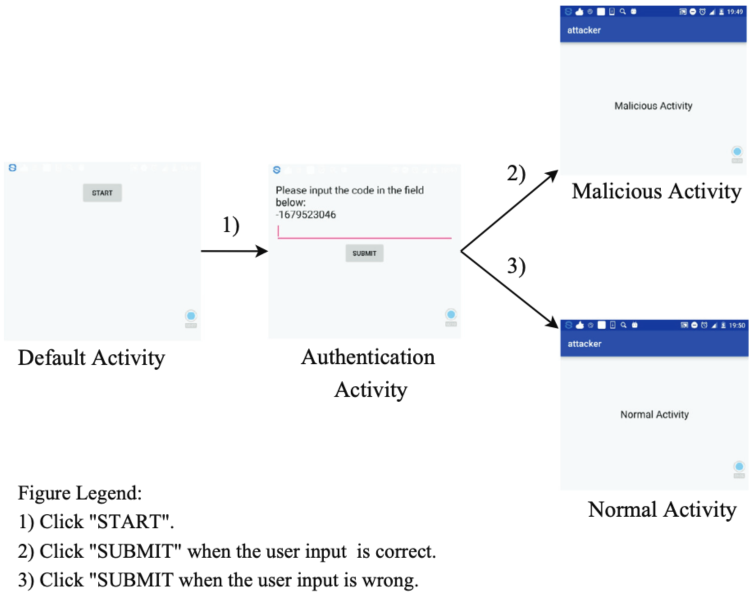


Fig. 4 – The application for the attack on FRep.

record/replay part of RnR-Safe. In the record stage, our prototype builds a snapshot/checkpoint to back up the data in virtual memory space. During the record stage, it intercepts part of the inputs to the virtual machine including mouse and keyboard inputs, network packets, audio controller inputs and hardware clocks. During the replay stage, it imports the snapshot to the virtual machine and replays all recorded inputs back to the virtual machine with the *icount* feature of QEMU or its extension.

We install QEMU 2.8.0 on Ubuntu 15.10 OS and create a virtual machine with two virtual CPUs of the x86 architecture, 1024-MB virtual memory and 20-GB virtual hard disk. We also set up the Ubuntu 15.10 OS on this virtual machine. In addition, we implement an attack program based on our approach, whihc first produces authentication code and then executes ROP attack only when someone correctly inputs the code.

We run the program in the virtual machine and let QEMU record the behavior of the program. During the execution of the program, we correctly input the authentication code so that the program finally conducts ROP attack. Then we let QEMU replay the whole process according to what it has been recorded. As a result, during the replay stage, the program correctly recognizes that current execution is in the replay context so that the ROP attack code is not executed. The video capturing the whole attack process is accessible via a YouTube link[6].

5. Defense discussion

In this section, we discuss potential mitigation methods for the vulnerability we presented in Section 3.

[6] https://youtu.be/khHyQ4WUGZA.

Ideally, *precisely controlling the timing of program execution in the replay stage* can eliminate the vulnerability. However, as non-uniform program execution time originates from inherent properties of computer systems, precisely controlling timing is impractical in current architecture of computer systems. Given this, we do not try to control timing precisely but to *hide* timing, which means to prevent a program sample from observing program execution time. Based on this idea, we discuss four mitigation methods: hardware-assisted record/replay analysis, ensemble record/replay analysis, symbolic execution and exploit detection.

5.1. *Hardware-assisted record/replay analysis*

Our vulnerability exploitation method uses a software counter to observe non-uniform program execution time. If we can reproduce all *instruction execution events* (i.e., record/restore opcode, operands and results for each instruction execution), the ending value of the counter in the replay stage must be the same as that in the record stage. As a result, a program sample cannot observe non-uniform program execution time via the counter.

It is crucial for us to point out that this method can cause significant performance overhead, especially for software-only record/replay systems, because it has to execute extra instructions to record/replay each instruction execution of the program. Since a forensic system should be *always-on* to collect all data for forensic analysis, it should be *light-weight* to avoid affecting the daily usage of user's device (Neasbitt et al., 2015).

Different from software-only solutions, hardware-assisted solutions, such as (Lee et al., 2011; Qian et al., 2013), usually use dedicated hardware to record/replay execution information to reduce the performance overhead of record/replay analysis. But, hardware-assisted solutions also have some limitations such as weak compatibility, replay slowdown, high implementation cost, etc. (Chen et al., 2015). So far, hardly do any commercial processors adopt state-of-the-art hardware-assisted record/replay solutions. In brief, hardware-assisted solutions can mitigate the vulnerability, but more efforts need to be paid to remove side effects to make them applicable to real forensic tasks.

5.2. *Ensemble record/replay analysis*

We have introduced six record/replay methods, and two light-weight methods (i.e., TR, CR and IR) are currently used in software forensics. Now let us switch back to the remaining four methods. We discuss whether we can mitigate the vulnerability by combining these two light-weight methods with the remaining four methods. We call this mitigation method as *ensemble replay*.

We give two ensemble replay schemes and discuss their strengths and weaknesses.

(1) RD+IR Let us reconsider the vulnerability exploitation algorithm in Fig. 2. One thread sends a signal, and another thread receives the signal to stop the counter, which is a typical race condition. If we can force the race result (i.e., the sequence that all threads access one data item) in the replay stage to be the same as that in the record stage, a program sample cannot observe the non-uniform program execution time via the counter. Given this, we can recognize all data races in the program sample during the program execution at first, also known as race detection (RD). Then we can make data race results in the replay stage similar to these in the record stage via lock mechanism. More details of the RD method can be found in Section 2. Except for the RD method, we also use the IR method to record/replay inputs.

The mitigation method works when all data races in the program can be detected without missing and error. However, state-of-the-art race detection methods can detect common data races (e.g., shared-memory data race), but they may miss special data races (e.g., data race via a covert channel). In conclusion, ensemble replay based on the RD method can mitigate the vulnerability only when the race detection method which forensic analysts adopt never makes mistakes.

(2) LL+IR: The LL method hides non-uniform program execution time by recording/restoring memory operations. However, it usually incurs significant performance overhead. To mitigate its performance overhead, we can combine the LL method with other record/replay methods. In particular, we can use the LL method to record/replay the behavior of vulnerability exploitation, while using other light-weight methods (e.g., the IR method) to record/replay other behaviors. This way, we only need to use the LL method *occasionally* so that performance overhead is reduced. To illustrate, let us consider our attack program for WebCapsule in Algorithm 1. It uses an `if` statement to decide whether to conduct malicious behavior or not. In this case, we only need to use the LL method to record/replay the execution of the last `if` statement in Algorithm 1.

The main challenges of this mitigation method are how to recognize the behavior of vulnerability exploitation and how to switch record/replay methods automatically. One may roughly regard all `if` statements as suspicious vulnerability exploitation behavior and switch current record/replay method to the LL method whenever it meets an `if` statement. However, when a program has to execute a group of `if` statements, the performance may remain low. This will be studied in our future work.

5.3. *Symbolic execution*

In essence, malware exploits the vulnerability to make the execution path in the record stage different from that in the replay stage. To avoid the divergence caused by non-uniform program execution time, one intuition is to use symbolic execution to force the execution path to be exactly the same with that in the replay stage and solve constraints in the execution path to get the inputs that can trigger such execution path.

Symbolic execution should be useful in the security analysis scenarios which mainly aims at triggering malicious behavior. For example, in malware analysis or detection, security analysts can use symbolic execution to execute suspicious code in a program. But, directly applying symbolic execution to forensic cases may cause some problems. In fact, although symbolic execution can force to execute every possible execution paths in the replay stage, it cannot prove the same execution path happened in the record stage, which is usually meaningless for forensic scientists who aim at extracting evidences for a criminal investigation. Recall the motivating example in Section 2.1. Symbolic execution has the capability to execute the path to trigger a phishing attack. But after solving the constraints in the execution path, we find the solved inputs may be different from the recorded inputs, which cannot prove that the clerk inputs its email password to the phishing website. Given such scenario, further research should be conducted to apply symbolic execution to record/replay forensic analysis.

5.4. *Exploit detection*

Malware which exploits such vulnerability may leave certain footprints (i.e., static or dynamic features) which could be used to detect such malware. Based on this intuition, a record/replay system may scan the malware before doing replay on it to know whether it exploits such vulnerability or not. If so, forensic or security analysts will do manual analysis on the malware instead of replay analysis. Although this approach cannot ensure a faithful replay, it can notice forensic or security analysts to take care when facing such anti-replay malware.

The major challenge of this approach is to extract effective features to reduce false positives on detecting such malware. For example, although malware which exploit such vulnerability may observe data race results to generate random numbers, data race cannot be directly used to detect such malware. Because benign programs or malware which do not exploit such vulnerability may also have data races. The feature extraction to detect such malware is also included in our future work.

6. Related work

Although, to our best knowledge, there is no existing work focusing on the security issues of record/replay systems for software forensics, we drew inspiration from a variety of sources, which can be roughly classified into four categories.

6.1. *Record/Replay analysis*

Record/replay analysis has been extensively studied in software testing and fault tolerance. Existing work such as (Jiang et al., 2014; Neasbitt et al., 2015; Qian et al., 2013) mainly focuses on *precisely conducting record/replay analysis with small log size, high record/replay speed and little probe effect* (Chen et al., 2015). Different from existing work, our work explores security issues of record/replay systems for software forensics.

6.2. *Software forensics*

Software forensics has about forty years of history. In the early days, most of forensic tasks were data recovery in essence (Garfinkel, 2010). Now, software forensics are facing great challenges from different kinds of cybercrimes, and there are a large variety of new forensic tasks much more complicated than data recovery such as forensic imaging (Guido et al., 2016), memory forensics (Socała and Cohen, 2016), network forensics (Joshi and Pilli, 2016), etc. In particular, record/replay analysis is an important technique which forensic analysts adopt in order to accomplish these forensic tasks. For example, Panda conducts record/replay as a way to facilitate memory forensic analysis (Dolan-Gavitt et al., 2013). WebCapsule (Neasbitt et al., 2015) records/replays users' operations on the Web browser to collect evidence in a web-based cybercrime. Timescope (Srinivasan and Jiang, 2011) applies record/replay to honeypot forensics. This paper tries to remind forensic analysts of the weakness or limitations of record/replay in order to help them deal with complex forensic tasks.

6.3. *Anti-forensics*

Anti-forensics refers to the countermeasures to software forensic analysis. Existing anti-forensic work is mainly about the definitions of anti-forensics (Harris, 2006; Stamm et al., 2012), anti-forensic techniques (Afshin et al., 2016; Lee et al., 2016), detection or mitigation of anti-forensic attacks (Rekhis and Boudriga, 2012), etc. We proposed a new anti-forensic technique for the forensic systems based on record/replay analysis. We also discussed how to mitigate our anti-forensic technique.

6.4. *Malware analysis evasion*

Malware may utilize a series of techniques to evade malware analysis. For example, code obfuscation techniques such as Banescu et al. (2016); Kuang et al. (2018) are used to deter static analysis. Anti-debugging techniques such as Wan et al. (2018), Chen et al. (2016) and sandbox evasion techniques such as (Costamagna et al., 2018; Diao et al., 2016) are used to evade dynamic analysis. Despite the merits of state-of-the-art malware analysis evasion techniques, they show limitations to evade record/replay forensic analysis, because the record/replay forensic analysis is a special dynamic analysis that does not rely on debugging or sandbox techniques. To overcome such issues, we propose a novel approach to evade record/replay forensic analysis.

7. Conclusion and future work

Record/replay analysis is initially utilized in software debugging. It records the events that may affect program flow and reproduces the bug by replaying the same events. Record/replay analysis is now used in software forensics. However, the program under forensic analysis may use tricks to prevent malicious behavior from being replayed. In this paper, we explored the vulnerabilities which malware may exploit to hinder a faithful replay.

We discovered a type of vulnerability associated with the phenomenon that program execution time cannot be exactly the same as the program re-execution time due to the non-uniform noise from hardware. By exploiting this vulnerability, malware can prevent its malicious behavior from being replayed. We conducted attack experiments on web browser, mobile operating system and virtualized sandbox, which verified the vulnerability in practice. Finally, we discussed the effectiveness and challenges to mitigate the vulnerability via hardware-assisted record/replay analysis, ensemble record/replay analysis, symbolic execution and exploit detection.

As for future work, we plan to look for more vulnerabilities which malware can explore to hinder record/replay forensic analysis. This way, we could understand the security issues of record/replay analysis more comprehensively. Besides, we plan to propose a novel record/replay forensic analysis approach which is immune to existing evasion techniques or vulnerabilities, and conduct a series of experiments on the prototype tool based on this approach to evaluate its effectiveness to defend against evasion and performance.

Acknowledgment

The work of John C.S. Lui is supported in part by the RGC R4032-18.

REFERENCES

Afshin N, Razzazi F, Moin MS. A dictionary based approach to jpeg anti-forensics. In: Proceedings of the 2016 IEEE 8th international conference on intelligent systems (IS). IEEE; 2016. p. 778–83.

Banescu S, Collberg C, Ganesh V, Newsham Z, Pretschner A. Code obfuscation against symbolic execution attacks. In: Proceedings of the 32nd annual conference on computer security applications, ACSAC '16. New York, NY, USA: ACM; 2016. p. 189–200. doi:10.1145/2991079.2991114.

Barak B, Halevi S. A model and architecture for pseudo-random generation with applications to /dev/random. In: Proceedings of the 12th ACM conference on computer and communications security. ACM; 2005. p. 203–12.

Chen A, Moore WB, Xiao H, Haeberlen A, Phan LTX, Sherr M, Zhou W. Detecting covert timing channels with time-deterministic replay. In: Proceedings of the 11th USENIX symposium on operating systems design and implementation (OSDI 14); 2014. p. 541–54.

Chen P, Huygens C, Desmet L, Joosen W. Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. In: Proceedings of the IFIP international information security and privacy conference. Springer; 2016. p. 323–36.

Chen Y, Zhang S, Guo Q, Li L, Wu R, Chen T. Deterministic replay: a survey. ACM Comput Surv (CSUR) 2015;48(2):17.

Corrigan-Gibbs H, Jana S. Recommendations for randomness in the operating system, or how to keep evil children out of your pool and other random facts. Proceedings of the 15th workshop on hot topics in operating systems (HotOS XV), 2015.

Costamagna V, Zheng C, Huang H. Identifying and evading android sandbox through usage-profile based fingerprints. In: Proceedings of the first workshop on radical and experiential security. ACM; 2018. p. 17–23.

Devietti J, Lucia B, Ceze L, Oskin M. Dmp: deterministic shared memory multiprocessing, 37. ACM; 2009. p. 85–96.

Devietti J, Nelson J, Bergan T, Ceze L, Grossman D. RCDC: a relaxed consistency deterministic computer, 46. ACM; 2011. p. 67–78.

Diao W, Liu X, Li Z, Zhang K. Evading android runtime analysis through detecting programmed interactions. In: Proceedings of the 9th ACM conference on security & privacy in wireless and mobile networks. ACM; 2016. p. 159–64.

Dolan-Gavitt B, Leek T, Hodosh J, Lee W. Tappan zee (north) bridge: mining memory accesses for introspection. In: Proceedings of the ACM Sigsac conference on computer and communications security; 2013. p. 839–50.

Dunlap GW, King ST, Cinar S, Basrai MA, Chen PM. Revirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Oper Syst Rev 2002;36(SI):211–24.

Garfinkel SL. Digital forensics research: the next 10 years. Digit Investig 2010;7:S64–73.

Gomez L, Neamtiu I, Azim T, Millstein T. Reran: timing-and touch-sensitive record and replay for android. In: Proceedings of the 2013 international conference on software engineering. IEEE Press; 2013. p. 72–81.

Guido M, Buttner J, Grover J. Rapid differential forensic imaging of mobile devices. Digit Investig 2016;18:S46–54.

Halpern M, Zhu Y, Peri R, Reddi VJ. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In: Proceedings of the 2015 IEEE international symposium on performance analysis of systems and software (ISPASS),. IEEE; 2015. p. 215–24.

Harris R. Arriving at an anti-forensics consensus: examining how to define and control the anti-forensics problem. Digit Investig 2006;3:44–9.

Hsiao CH, Yu J, Narayanasamy S, Kong Z, Pereira CL, Pokam GA, Chen PM, Flinn J. Race detection for event-driven mobile applications, 49. ACM; 2014. p. 326–36.

Jiang Y, Gu T, Xu C, Ma X, Lu J. Care: cache guided deterministic replay for concurrent java programs. In: Proceedings of the 36th international conference on software engineering. ACM; 2014. p. 457–67.

Joshi R, Pilli ES. Network forensics. Fundamentals of network forensics. Springer, 2016. 3–16.

King ST, Dunlap GW, Chen PM. Debugging operating systems with time-traveling virtual machines. Proceedings of the annual conference on USENIX annual technical conference, 2005. 1–1.

Kuang K, Tang Z, Gong X, Fang D, Chen X, Wang Z. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. Comput Secur 2018;74:202–20.

Lee D, Said M, Narayanasamy S, Yang Z. Offline symbolic analysis to infer total store order. In: Proceedings of the 2011 IEEE 17th International Symposium on high performance computer architecture (HPCA). IEEE; 2011. p. 357–8.

Lee K, Hwang H, Kim K, Noh B. Robust bootstrapping memory analysis against anti-forensics. Digit. Investig. 2016;18: S23–S32.

Montesinos P, Ceze L, Torrellas J. Delorean: recording and deterministically replaying shared-memory multiprocessor execution effficiently. In: Proceedings of the 35th international symposium on computer architecture, 2008. ISCA'08.. IEEE; 2008. p. 289–300.

Narayanasamy S, Pokam G, Calder B. Bugnet: continuously recording program execution for deterministic replay debugging, 33. IEEE Computer Society; 2005. p. 284–95.

Neasbitt C, Li B, Perdisci R, Lu L, Singh K, Li K. Webcapsule: towards a lightweight forensic engine for web browsers. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM; 2015. p. 133–45.

Patil H, Pereira C, Stallcup M, Lueck G, Cownie J. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings of the 8th annual IEEE/ACM international symposium on code generation and optimization. ACM; 2010. p. 2–11.

Qian X, Huang H, Sahelices B, Qian D. Rainbow: efficient memory dependence recording with high replay parallelism for relaxed memory model. In: Proceedings of the 2013 IEEE 19th international symposium on high performance computer architecture (HPCA2013),. IEEE; 2013. p. 554–65.

Qin Z, Tang Y, Novak E, Li Q. Mobiplay: a remote execution based record-and-replay tool for mobile applications. In: Proceedings of the 38th international conference on software engineering. ACM; 2016. p. 571–82.

Rekhis S, Boudriga N. A system for formal digital forensic investigation aware of anti-forensic attacks. IEEE Trans Inf Forensics Secur 2012;7(2):635–50.

Shalabi Y., Yan M., Honarmand N., Lee R.B., Torrellas J. Record-replay architecture as a general security framework. In: Proceedings of the 2018 IEEE international symposium on high performance computer architecture (HPCA). IEEE; 2018. p. 180–193.

Socała A, Cohen M. Automatic profile generation for live linux memory analysis. Digit Investig 2016;16:S11–24.

Srinivasan D, Jiang X. Time-traveling forensic analysis of VM-based high-interaction honeypots. In: Proceedings of the international conference on security and privacy in communication systems. Springer; 2011. p. 209–26.

Stamm MC, Lin WS, Liu KR. Forensics vs. anti-forensics: a decision and game theoretic framework. In: Proceedings of the 2012 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE; 2012. p. 1749–52.

Voskuilen G, Ahmad F, Vijaykumar T. Timetraveler: exploiting acyclic races for optimizing memory race recording, 38. ACM; 2010. p. 198–209.

Wan J, Zulkernine M, Liem C. A dynamic app anti-debugging approach on android art runtime. In: Proceedings of the 2018 IEEE 16th international conference on dependable, autonomic and secure computing, 16th international conference on pervasive intelligence and computing, 4th international conference on big data intelligence and computing and cyber science and technology congress (DASC/PiCom/DataCom/CyberSciTech). IEEE; 2018. p. 560–7.

Wendzel S, Zander S, Fechner B, Herdin C. Pattern-based survey and categorization of network covert channel techniques. ACM Comput Surv (CSUR) 2015;47(3):50.

Xu C, Lemaitre RP, Soto J, Markl V. Fault-tolerance for distributed iterative dataflows in action. Proc VLDB Endow 2018;11(12):1990–3.
Xue R, Liu X, Wu M, Guo Z, Chen W, Zheng W, Zhang Z, Voelker G. MPIWIZ: subgroup reproducible replay of MPI applications. ACM Sigplan Not 2009;44(4):251–60.

Yang Hu received the B.Eng. degree in Software Engineering in Xi'an Jiaotong University, Xi'an, China in 2014 and received the M.Eng. degree in Software Engineering in Xi'an Jiaotong University in 2017. He was a research assistant in the Department of Computer Science and Engineering, The Chinese University of Hong Kong, and the Department of Computing, The Hong Kong Polytechnic University. He is currently a research assistant in School of Computing, National University of Singapore. His research interests include system security and software engineering.

Mingshen Sun is a senior security researcher of Baidu X-Lab at Baidu USA. He received his Ph.D. degree in Computer Science and Engineering from The Chinese University of Hong Kong, under the supervision of Prof. John C.S. Lui. He was a member of Advanced Networking and System Research Laboratory (ANSRLab) in CUHK. During the Ph.D. studies, he worked as a research intern in Qihoo 360. Mingshen also worked in National University of Singapore as a research assistant. He maintains and actively contributes to several open source projects. His research interests include system security, mobile/IoT security, car hacking, and memory-safe programming language.

John C.S. Lui is currently the Choh-Ming Li Chair Professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His current research interests are in Internet, network sciences with large data implications, machine learning on large data analytics, network/system security, network economics, large scale distributed systems and performance evaluation theory. He received various departmental teaching awards and the CUHK Vice-Chancellor's Exemplary Teaching Award. John also received the CUHK Faculty of Engineering Research Excellence Award (20112012). John is a co-recipient of the best paper award in the IFIP WG 7.3 Performance 2005, IEEE/IFIP NOMS. He is a Fellow of ACM and IEEE.