A Probabilistic Test based Detection Scheme against Automated Attacks on Android In-app Billing Service

Heeyoul Kim[1]
[1] Department of Computer Science, Kyonggi University
Republic of Korea
[e-mail: heeyoul.kim@kgu.ac.kr]
*Corresponding author: Heeyoul Kim

Received September 28, 2018; accepted February 4, 2019; published March 31 2019

Abstract

Android platform provides In-app Billing service for purchasing valuable items inside mobile applications. However, it has become a major target for attackers to achieve valuable items without actual payment. Especially, application developers suffer from automated attacks targeting all the applications in the device, not a specific application. In this paper, we propose a novel scheme detecting automated attacks with probabilistic tests. The scheme tests the signature verification method in a non-deterministic way, and if the method was replaced by the automated attack, the scheme detects it with very high probability. Both the analysis and the experiment result show that the developers can prevent their applications from automated attacks securely and efficiently by using of the proposed scheme.

Keywords: In-app Billing, automated attack, probabilistic test, Android, security

A preliminary version of this paper was presented at APIC-IST 2018, and was selected by the conference review process.

http://doi.org/10.3837/tiis.2019.03.031 **ISSN : 1976-7277**

1. Introduction

With the widespread use of smartphone, the markets for mobile applications have changed the way people purchase interested applications [1]. They can easily find, install and update various useful applications via Apple's App Store or Google's Play Store. Moreover, various mobile payment methods using smartphone have been popular in recent years. One of important business models related to these markets is to encourage users to purchase valuable contents such as game items or premium features within mobile applications. A user first tries out an application with limited feature and if he wants, he can purchase additional virtual goods such as premium items or additional features such as extra contents. He can also subscribe to regular content delivery service inside the application. This approach allows users to experience the functionality of valuable services before purchasing. Apple introduced In-app Purchase service for this business model in 2011 [2], and Google also has provided similar In-app Billing service since 2011 [3]. The application developers can improve profitability through these in-app payment services.

The importance of security has been emphasized in mobile ecosystems, and lots of research have been presented to prevent mobile applications from various attacks [4-5]. However, the attacks targeting the In-app Billing service should be considered from a different viewpoint than other attacks. These attacks mainly focus on bypassing legitimate billing process to purchase valuable items, and the user himself rather than an external attacker attacks his own device with full access right, which makes it more vulnerable than other traditional systems. The recompile attack is a simple billing crack attack. The attacker decompiles the apk file of the application installed on the device, modifies the source code to make an invalid payment or to bypass the payment check routine, then re-installs the cracked code. However, this attack has an inconvenience of decompiling and source modification for each target application, and some countermeasures such as code obfuscation technique [6] have been applied.

Automated attacks resolving this inconvenience have been emerged in a more advanced manner. These kinds of attacks do not target a specific application, but they target all the installed application using the In-app Billing service. VirtualSwindle [7] was presented in the literature, and Freedom [8] is another representative automated attack tool working well up to now. The attack obtains the list of applications using the In-app Billing service in the target device. Whenever a purchase is requested within an application, it bypasses the normal payment process and thus it succeeds in purchasing without actual payment.

In this paper, we propose a novel scheme detecting automated attacks on In-app Billing service with probabilistic tests. The key part of the automated attack is bypassing the signature verification with a fake signature generated by the attack. For this purpose, the attack replaces the original signature verification method in the application side to the attack's malicious function returning true always. The proposed scheme detects this replacement by performing multi-round probabilistic tests. Each test is done in a non-deterministic way to check whether the verification works correctly with a valid public key of the application or a fake key generated for the test. We also provide both the analysis and the experiment result of the proposed scheme to show that it efficiently succeeds in detecting both current automated attacks and more sophisticated attacks with very high probability.

The paper is organized as follows. In Section 2, we explain the Android In-app Billing service and then we analyze the mechanism of automated attacks. In Section 3, we propose our detection scheme against automated attacks with the analysis of the scheme. Section 4 describes the implementation of the scheme with a technique for performance improvement. Also, both the experiment result and performance analysis are provided. Finally, Section 5 concludes the paper.

2. Analysis of Android In-app Billing and Automated Attacks

2.1 Android In-app Billing Service

The In-app Billing service in Android platform enables a developer sell digital content inside his mobile application. This service consists of four components as shown in **Fig. 1**. The Google Play server is responsible for performing the actual financial transactions remotely requested, and it uses the same checkout backend service as is used for application purchases. The Google Play application installed on the user's smartphone conveys billing requests and responses between the application and the Google Play server. The application on the device accesses the In-app Billing service using the IInAppBillingService interface exposed by the Google Play application. It is worth noting the application does not communicate with the Google Play server directly. Instead, it communicates with the Google Play application over interprocess communication (IPC) for purchase. The fourth component is the application server, which is optional depending on the developer's choice. The application may communicate with its own server to enhance payment verification and to provide additional digital content to the application.

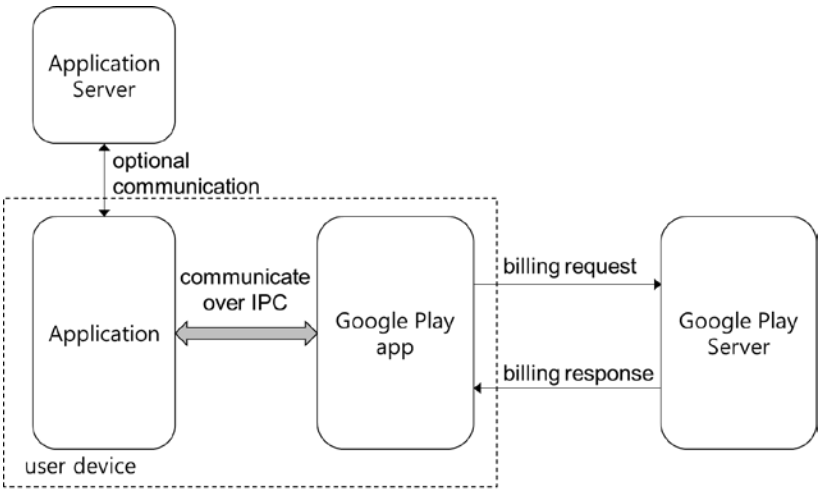

Fig. 1. Components of Android In-app Billing Service

The application developer has to perform registration process via the Google Play Developer Console [9] before publishing an application using the In-app Billing service. The application and related digital contents that are available for purchase from the application are registered in the Developer Console. Then a cryptographic public/private key pair is generated for the application. The private key is registered in the Google Play server for signing and confirming purchase transactions. The corresponding public key is located inside the

application to verify the purchase confirmation later during the In-app Billing service. Google recommends that the public key is not hard-coded in the application. Instead, some obfuscation techniques can be utilized.

A simplified process between the application and the Google Play application when a user purchases something inside the application is shown in **Fig. 2**. The application sends *isBillingSupported* request for supported version check. Then the application makes *getBuyIndent* request including the product ID of the item to be purchased. This request is sent to the Google Play application, and then a corresponding response Bundle containing a *PendingIntent* which will be used to start the checkout UI is returned. The application launches this Intent by calling the *startIntentSenderForResult* method, and the user inputs his billing information such as credit card number. After verifying the billing information with the Google Play server, the Google Play application sends a response Intent containing the detailed information about both the purchased item and the purchase transaction. The Intent also contains the signature of the purchase data which is signed by the Google Play server with the application's private key registered in advance. The application must to verify this signature with the corresponding public key inside the application to check whether a valid payment has been made. This verification is the key process to ensure a legitimate purchase has been completed and to avoid malicious purchase attempts on the application. In addition, the In-app Billing service provides another APIs for querying product details or consuming purchased items. Recently Google released the Google Play Billing library [10] which simplifies the development process for In-app Billing, and it still utilizes the In-app Billing service explained above to manage in-app billing transactions.

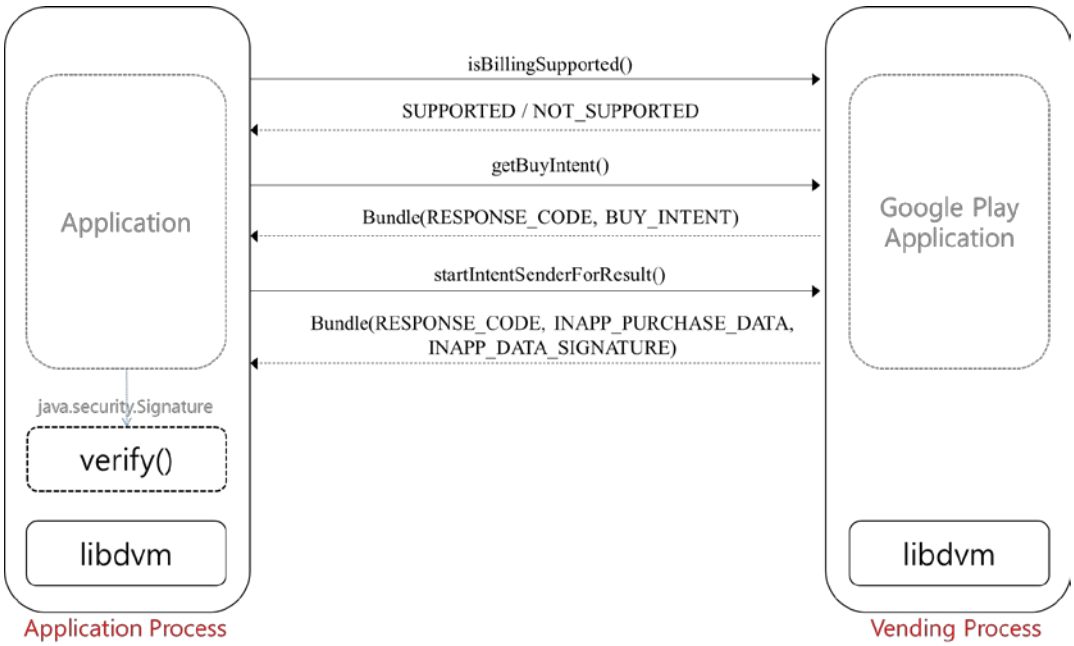

Fig. 2. A Simplified process between application and Google Play in the In-app Billing Service

2.2 Automated Attacks on In-app Billing Service

As the use of In-app Billing service is becoming widespread, various billing cracks have appeared [7, 8, 11]. The goal of these attacks is to find the way to bypass legitimate payment process and for obtaining valuable items without payment. It is very attractive to the attacker because the success of the attack will give huge financial benefit to him. For example, some expensive items obtained by the attack can be traded in the black market to make money.

This kind of attacks on the In-app Billing service should be considered from a different viewpoint than other security threats on the mobile device. In general, the user owning a mobile device is assumed in a defensive position to prevent his sensitive data from malicious outside attacks. However, in this case, the user himself becomes an attacker and tries various attempts on his device to find vulnerability of the In-app Billing service. Moreover, he can easily attack his device with full access right by performing rooting on the device.

A simple billing crack attack is the recompile attack. The attacker decompiles the apk file of the application installed on the device to get the source code, then he modifies the code to make an invalid purchase transaction or to bypass the payment routine. The cracked code is re-installed on the device and the attack succeeds in cracking.

The dangerousness of this attack has been already recognized. In the security and design guidelines for In-app Billing service [12], Google recommends that the developer should obfuscate the codes in order that it is difficult for the attackers to reverse engineer security protocols and other application components. Using obfuscation tools such as Proguard and using method inlining are also recommended. Obfuscation is effective as it imposes time-consuming burdens on the attacker. Moreover, this recompile attack has the inconvenience of decompile and code modification for each target application, which makes obfuscation more effective.

However, automated attacks resolving this inconvenience have emerged in a more advanced manner. Instead of taking the manual approach of trying to reverse engineer individual application, these attacks automatically try to bypass the purchase verification process of all the applications on the device using In-app Billing service. VirtualSwindle is the first and excellent automated attack in the literature against the In-app Billing service on Android. The attack code runs in the background, and when invoked, attacks every application that performs signature verification on the device itself (not on the remote server). It allows the attacker to access digital content and services without paying for them by subverting the signature verification process. According to the authors, among the 85 popular applications using the In-app Billing, 60% of them employed on-device signature verification, and therefore they were easily attacked successfully. Here we analyze the detailed mechanism of the VirtualSwindle and we also discuss why this kind of attack successes in most applications.

The attack flow of VirtualSwindle is shown in **Fig. 3**. The attack emulates and subverts the part responsible for the IInAppBillingService interface in the Google Play app on the device. It works like a proxy between the application using In-app Billing and the Google Play app, and it generates a fake response Intent that makes the calling application believe the payment process was performed normally. In the victim application side, to bypass the signature verification of the purchase data, the attack replaces the standard Dalvik library method *java.securty.Signature.verify()* with its own function. The replaced function returns true indicating success if the input is the fake signature, and otherwise it redirects this call to the original method in order to avoid detection of this attack.

This attack utilizes a dynamic Dalvik instrumentation approach [13], which is implemented as a libddi library, enabling to replace any Dalvik method to an alternative native

function by abusing the Java Native Interfaces (JNI) layer. In the first step, the *com.android.vending* process responsible for the Google Play application is hijacked by injecting the libddi library. Then the prepared Dalvik classes for acting as a proxy are loaded into the vending process. The attack also injects a native library into the zygote process to bypass the signature verification in the application side. This library then manipulates the method struct of *java.security.Signature.verify* so that the attack's malicious function is called through JNI instead of the original method. It is worth noting that once the library is injected into the Zygote it is automatically propagated to all processes running on the device.

When the victim application sends a purchase request through the *getBuyIntent* call, the proxy intercepts and redirects it to the original Google Play app to get a valid *PendingIntent*. The response generated by the Google Play app is then passed back to the victim application. The victim application launches this *PendingIntent* for checkout, and the proxy also intercepts this request. Because the actual payment is not made at this time, the proxy itself generates a JSON object containing purchase information such as the order ID, instead of receiving it from the remote Google Play server. The proxy then returns the JSON object including a fake signature. After receiving it, the victim application performs signature verification generally by calling *java.security.Signature.verify* method for on-device verification. But, the replaced attack function is actually called as mentioned above. The function returns true as the fake signature is provided, and finally the attack succeeds.

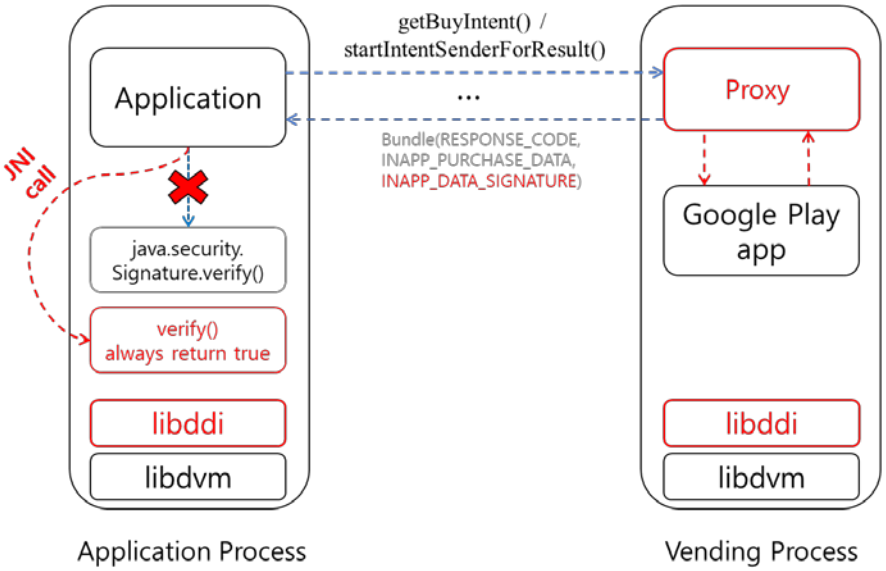

Fig. 3. Attack flow of VirtualSwindle

Freedom is another automated attack tool which is very popular among the black market users. Though the detailed mechanism of has not been discovered, it looks very similar to VirtualSwindle above at least in the way to bypass the signature verification. This tool obtains the list of all applications using the In-app Billing service in the target device, then whenever a purchase request happens it bypasses the normal payment process and consequently succeeds in purchasing without actual payment. A lot of game applications have been cracked by Freedom up to now.

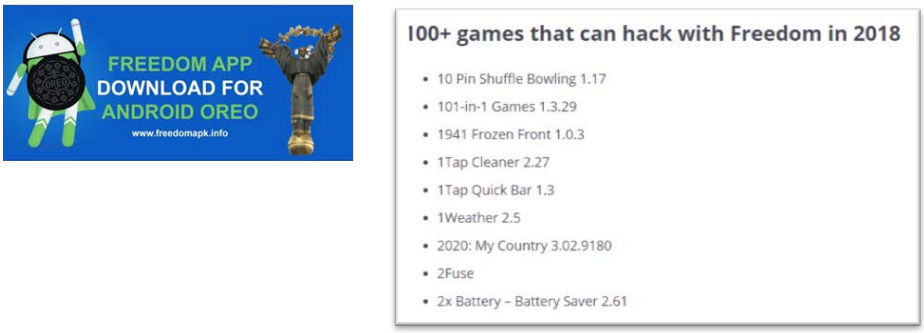

Fig. 4. A screenshot of the list of applications cracked by Freedom [8]

As the damage caused by the automated attacks increases, there has been research to prevent or detect this kind of attacks. One way is to block applications running with root privilege because both VirtualSwindle and Freedom need to get root privilege to access other application processes. However, it has a bad influence on normal applications requiring root privilege and the users hesitate to apply it to their device. Another practical way discussed among application developers is to check whether these attacks exist or not in the device based on the list of installed application names. However, it is not a sufficient and fundamental solution as it is very hard to detect various and unknown attacks.

3. Proposed Detection Scheme against Automated Attacks

Here we present a novel scheme detecting automated attacks on In-app Billing service with probabilistic tests. In section 3.1, we first explain the basic idea to check whether the signature verification method is replaced to the attack's malicious function or not. Then, in section 3.2 we consider the possible countermeasure that improved attacks may take to avoid this checking routine. Finally, in section 3.3 we present a robust scheme to detect even these attacks with very high probability.

3.1 Checking signature verification method

The key part of automated attacks is bypassing the signature verification process in the application side. When the attack is performed via the In-app Billing service, a valid signature for the purchase cannot be generated because a legitimate payment is not provided and the attack cannot discover the victim application's private key in any way. Instead, by the attack, the call for *java.security.Signature.verify()* method is redirected to the attack's malicious function returning true always with the fake signature.

If we can detect the replacement of verification method, we can also detect and thus prevent the attack. One possible way is to monitor the integrity of the application process at runtime. However, runtime monitoring imposes a heavy burden on the system and it requires modification of Android kernel codes. Here we present a simple but effective idea to detect the replacement, which is deduced from inspecting the verification method. We proposed a preliminary version of this idea in [14].

Let us assume that the call for *java.security.Signature.verify()* method was replaced to the attack function *verify'()*. The signature verification utilizes three input parameters: the purchase data to be verified(*data*), the victim application's public key(*pub*) and the signature(*sig*). Among them, the signature is in fact a fake signature generated by the proxy. Thus, the function can easily check whether the input signature is equal to the fake signature.

The purchase data is known to the proxy and then it also can be known to the function through a covert channel between the proxy and the function. However, differently from these parameters, the application's public key is not known to the function unless reverse engineering the target application. Because we are dealing with automated attacks not performing reversing manually, this argument is reasonable.

The check routine for the verification method is as follows. A new fake public key(*pub'*) is generated, and instead of the original public key it is inputted with other parameters to the verification method. If the attack is not in progress, the call returns false as the provided public key is invalid. On the other hands, if the attack is in progress, the call is redirected to *verify'()* and this function returns true because the fake signature is provided as planned. Therefore, by checking the return value, we succeed in detecting automated attacks. If true is returned, the application stops without providing items. Otherwise, the application calls the verification method again with the valid public key to verify a legitimate payment was processed before providing corresponding items. The pseudocode of the check routine is provided in **Fig. 5**.

Algorithm 1 Checking signature verification function

```
1:  function CheckAndVerify(data, pub, sig)  ▷ data: purchase data, pub: application's public
    key, sig: signature
2:      pub' ← GenerateFakeKey()                          ▷ generate a fake public key
3:      r ← Verify(data, pub', sig)
4:      if r = true then                                  ▷ verification was replaced by attack
5:          return false
6:      end if
7:      r ← Verify(data, pub, sig)
8:      if r = false then                                 ▷ Signature is invalid
9:          return false
10:     end if
11:     return true
12: end function
```

Fig. 5. A pseudocode of check routine for signature verification

3.2 Improving automated attack against checking routine

The above checking routine is useful to detect current automated attacks. We experimented on Freedom with the routine, and we successfully detected it. However, if this routine is widely applied, the attackers will seek to find some ways to avoid the routine. Here we discuss possible countermeasures that more sophisticated attacks can take.

One possible approach is to cope with each step of the checking routine after careful analyzing the sequence of the routine. Specifically, the above routine calls *verify()* method twice: the former is with a fake public key, and the latter is with a valid public key. Let us assume that the attack function *verify'()* knows recent fake signatures through a covert channel with the proxy. The function is modified so that it returns false if the call is the first call with the fake signature and otherwise it returns true if the call is the second call. This improvement makes the attack pass the checking routine.

The checking routine can be enhanced to defend against this approach. For example, it can be modified to call *verify()* method three times where the former two calls are with a fake public key. However, this kind of variation will also become the target of attacker's analysis and it is not difficult to make sophisticated attacks against even this enhanced routine. It is worth pointing out that such a checking routine having a fixed and deterministic sequence is vulnerable to the attacks specialized to the routine.

Another approach is to randomly choose the return value of the attack function instead of coping with the checking routine. Current attack function always returns true if a fake signature is provided. This approach modifies it so that the return value is randomly chosen between true and false if a fake signature is provided. Obviously, it does not guarantee that the attack always succeeds, and in the case where the checking routine is not applied, the success probability decreases to 50%. However, in the case where the above checking routine is applied, this approach provides a reasonable success probability. To pass the checking routine, the attack has to return correct answers for the two function calls. Since the probability that it returns a correct answer for each call is 50%, the success probability of the attack becomes 25%. From the viewpoint of the attacker, this probability satisfies him as he can gain sufficient financial benefit by attempting the attack several times. It is worth pointing out that this approach does not require deep analysis of the routine and it operates successfully against various checking routines.

3.3 Detecting automated attacks with probabilistic tests

Here we present a novel detection scheme against even sophisticated automated attacks. This scheme is located in the application side as in **Fig. 6**, and it is responsible for both detecting automated attacks and verifying the signature for the purchase. It is based on the idea in Section 3.1 using a fake public key to check the signature verification method, and it performs multiple probabilistic tests to have a non-deterministic property which defends against sophisticated attacks explained in Section 3.2.

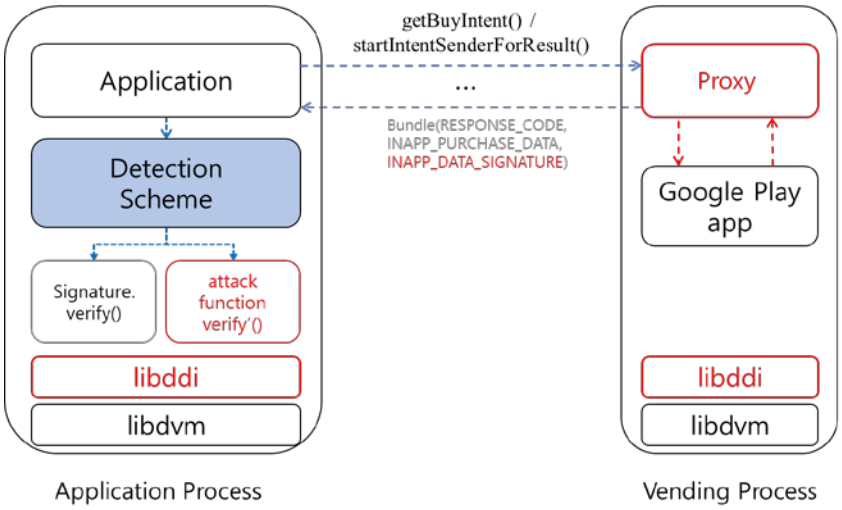

Fig. 6. The relation of proposed scheme with the In-app Billing service

The scheme consists of *n* rounds, where *n* is adjustable in consideration of both required security level and performance. For each round, the scheme decides whether to test it with a fake public key or with the application's valid public key. If the former is chosen, the scheme randomly generates a fake public key and then it calls the verification method with the key. If the result is true, the scheme finishes with returning false meaning that some attack was detected. Otherwise, it is regarded as passing the test of this round and it goes to the next round. If the latter is chosen, the scheme calls the verification method with the application's valid public key. If the result is false, the scheme finishes with returning false meaning that the signature is invalid or some attack is undergoing. Otherwise, it is regarded as passing the test of this round and it goes to the next round. The scheme performs this test *n* times, and if all the

tests are passed it finally returns true meaning that no attack was detected. The pseudocode of the proposed scheme is provided in **Fig. 7**.

Algorithm 2 Automated attack detection scheme

```
1: function DetectionTest(data, pub, sig, n)  ▷ data: purchase data, pub: application's public key, sig: signature, n: # of rounds
2:     for k ← 1 to n do
3:         b ← RandomBoolean()                 ▷ choose between true and false
4:         if b = true then                    ▷ test with a fake public key
5:             pub' ← GenerateFakeKey()        ▷ generate a fake public key
6:             r ← Verify(data, pub', sig)
7:             if r = true then                ▷ attack was detected
8:                 return false
9:             end if
10:        else                                ▷ test with a valid public key
11:            r ← Verify(data, pub, sig)
12:            if r = false then               ▷ invalid signature or attack detected
13:                return false
14:            end if
15:        end if
16:    end for
17:    return true                             ▷ signature is valid
18: end function
```

Fig. 7. A pseudocode of the proposed detection scheme

Now let us analyze the proposed scheme and calculate the probability that an automated attack passes the scheme. Suppose an automated attack is being performed. Whenever the signature verification method is called by the scheme, the attack function perceives that the fake signature is inputted as expected. Then, it has to make an appropriate result instead of redirecting it to the original verification method. However, as explained earlier, it cannot determine whether the inputted public key is valid or not. Thus, the attack function has to choose between true and false as return value, and the choice would at best be not much better than flipping a coin.

Let $P_1(pass)$ denote the probability that an automated attack passes the one round test. Let T_f denote the event that the scheme tests with a fake public key, and let T_o denote the event that the scheme tests with the original public key. Then,

$$P_1(pass \mid T_f) = 1/2,\ P_1(pass \mid T_o) = 1/2, \tag{1}$$

and $P_1(pass)$ is

$$P_1(pass) = P_1(pass \mid T_f) \cdot P_1(T_f) + P_1(pass \mid T_o) \cdot P_1(T_o) = 1/4 + 1/4 = 1/2. \tag{2}$$

Let $P_n(pass)$ denote the probability that the attack successfully passes the scheme. Since it must pass all the *n* round tests,

$$P_n(pass) = P_i(pass)^n = 1/2^n. \tag{3}$$

The success probability is very low for sufficiently large n. For example, if n=20 the probability is

$$P_{20}(pass) = 1/2^{20} \simeq 0.00000095. \tag{4}$$

Therefore, the proposed scheme succeeds in detecting automated attacks with very high probability.

4. Implementation and Analysis

4.1 Implementation

The most time-consuming part of the proposed scheme is to generate fake public keys. For *n* rounds test, the scheme generates *n/2* public keys on average, and such generation is slow especially in mobile devices having limited computing power. We applied a novel trick to overcome this problem. Let us assume a 1024-bits RSA [15] key pair is utilized in the In-app Billing. An RSA public key consists of two numbers (N, e) where N is the multiplication of two 512-bits large prime numbers. Such large prime numbers can be efficiently found using a primality test such as Miller-Rabin test [16], but it is still not sufficient for our scheme requiring a lot of keys. So, we generate N by multiplying eight 128-bits prime numbers instead of two 512-bits prime numbers. Obviously, the value N is not suitable for RSA algorithm and the RSA operations do not work correctly with it. However, from the viewpoint of the attacker, he must factorize N to discover this trick and the factorization of such N requires several hours and days. Since the attacker must factorize *n* times to pass our scheme with *n* rounds, it is impractical in nature. We could reduce a large amount of consumed time by using of this trick.

By using of the proposed scheme, the mobile applications using the In-app Billing service can ensure safe and trustworthy purchase process against automated attacks. One possible way to apply the scheme easily and widely is to put the implementation of the scheme into the In-app Billing service codes, for example with the method name *detectionTest()*. However, the method itself can be the target of more sophisticated attacks. Specifically, the attacker can try to replace the *detectionTest()* method instead of *verify()* method to the attack function in order to bypass the probabilistic test. Therefore, we suggest the detection scheme is located in the application codes by the application developer with different method name respectively.

We implemented and applied our detection scheme to the sample application TrivialDrive provided by Google to show the correctness of the scheme. We tried to purchase an item after launching the Freedom, and consequently the scheme successfully detected the automated attack. Then the application stopped the purchase process with a warning message as can be seen in **Fig. 8**.

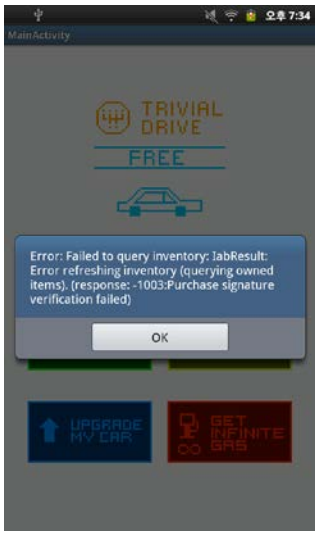

Fig. 8. A screenshot of detection success of Freedom attack by the proposed scheme

4.2 Performance Analysis

We evaluated the performance of the proposed scheme by performing experiments with smart devices of two categories: low-performance smartphone and mid-range smartphone. The most important factor of performance in the In-app Billing service is the waiting time during the purchase process. If the user has to wait long time to purchase an item, he feels uncomfortable and withdraws the purchase. Moreover, Android limits the ANR(Application Not Responding) time to 5 seconds. So, we measured the elapsed time for testing with the proposed scheme according to the number of rounds to deduce appropriate value in the aspect of both security level and performance.

We performed an experiment with a mid-range smartphone having 1.7GHz quad core CPU and 2GB RAM, and the result is shown in **Table 1** and **Fig. 9**. The number of rounds varies from 10 times to 30 times, and per each number of rounds the elapsed time was measures six times to obtain the average elapsed time. In consideration of the ANR time, the reasonable threshold of the elapsed time is 2 seconds. In this experiment, even the elapsed time of 30 rounds does not exceed 2 seconds. And the probability of attack success is extremely low ($P_{30}(pass) \simeq 10^{-9}$). Therefore, the proposed detection scheme with 30 rounds is recommended for such a mid-range smartphone environment.

Table 1. Estimated elapsed time of the proposed scheme with a mid-range smartphone

round #	Exp. #1	Exp. #2	Exp. #3	Exp. #4	Exp. #5	Exp. #6	Average
10	0.4321	0.5023	0.4884	0.5460	0.4498	0.5892	0.5013
12	0.6879	0.5379	0.5880	0.7056	0.3058	0.6219	0.5745
14	0.8214	0.7892	0.7622	0.6088	0.8457	0.7316	0.7598
16	0.8925	0.8933	0.8149	0.7615	1.1659	0.7978	0.8876
18	1.1679	0.9487	1.0297	0.8972	1.3460	1.0490	1.0731
20	1.3252	0.8103	1.4827	1.5338	1.0949	1.1917	1.2398
22	0.9313	1.0655	1.2647	1.4894	1.3547	1.3657	1.2452
24	1.3616	1.5650	1.5499	1.0232	1.2199	1.4452	1.3608
26	1.3225	0.8734	1.6949	1.4266	1.8019	1.2783	1.3996
28	1.2670	1.5593	1.3763	1.5932	1.7046	1.4702	1.4951
30	1.4777	1.5649	1.4775	1.4134	1.3364	2.3067	1.5961

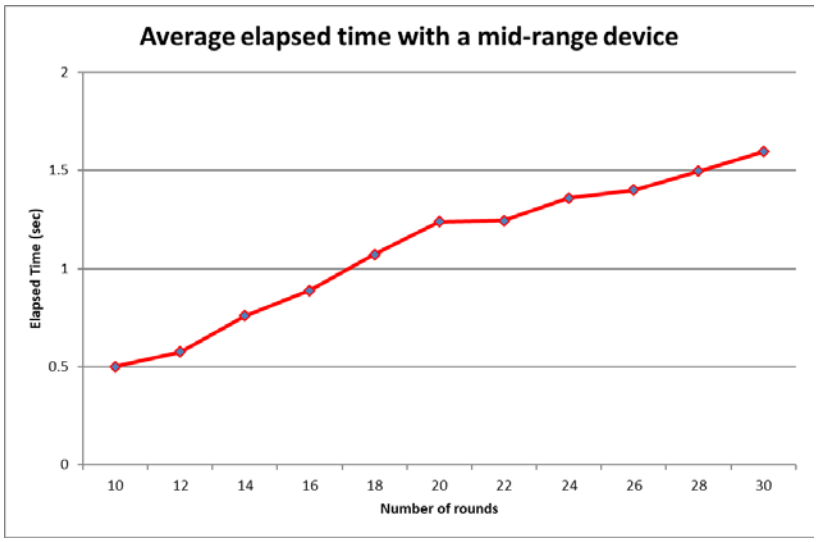

Fig. 9. Average elapsed time of the proposed scheme with a mid-range smartphone

We also performed another experiment with a low-performance smartphone having 1.4GHz dual core CPU and 1GB RAM, and the result is shown in **Table 2** and **Fig. 10**. The number of rounds also varies from 10 times to 30 times, and per each number of rounds the elapsed time was measures six times to obtain the average elapsed time. In this experiment, the elapsed time of 30 rounds exceeds 2 seconds, however, the elapsed time of 22 rounds does not exceed 2 seconds. And the probability of attack success is very low ($P_{22}(pass) \simeq 2 \times 10^{-7}$) which is very acceptable in comparison with the benefit of attack success. Therefore, the proposed detection scheme with 22 rounds is recommended for such a low-performance smartphone environment.

Table 2. Estimated elapsed time of the proposed scheme with a low-performance smartphone

round #	Exp. #1	Exp. #2	Exp. #3	Exp. #4	Exp. #5	Exp. #6	Average
10	0.6520	0.7217	1.4322	1.3206	1.4236	0.9846	1.0891
12	0.8977	1.5179	0.9220	1.2165	1.3135	1.4565	1.2207
14	1.4239	1.4085	0.8325	2.1790	1.8166	1.6577	1.5530
16	1.3550	1.2365	1.9213	1.8805	1.7622	2.0217	1.6962
18	1.2672	1.5892	1.7981	2.0160	1.7981	1.8850	1.7256
20	1.4776	1.5665	2.2865	2.0149	1.7613	1.7217	1.8048
22	1.1101	1.7704	2.4368	1.6650	2.0249	2.3532	1.8934
24	1.5314	1.7218	3.8597	1.8283	2.3109	1.7261	2.1630
26	1.7954	3.0345	3.2519	3.0335	2.4427	2.6262	2.6974
28	2.3216	2.9253	2.3392	3.8683	2.4329	3.0301	2.8196
30	2.8479	2.8162	1.4265	4.4032	2.4231	3.4340	2.8918

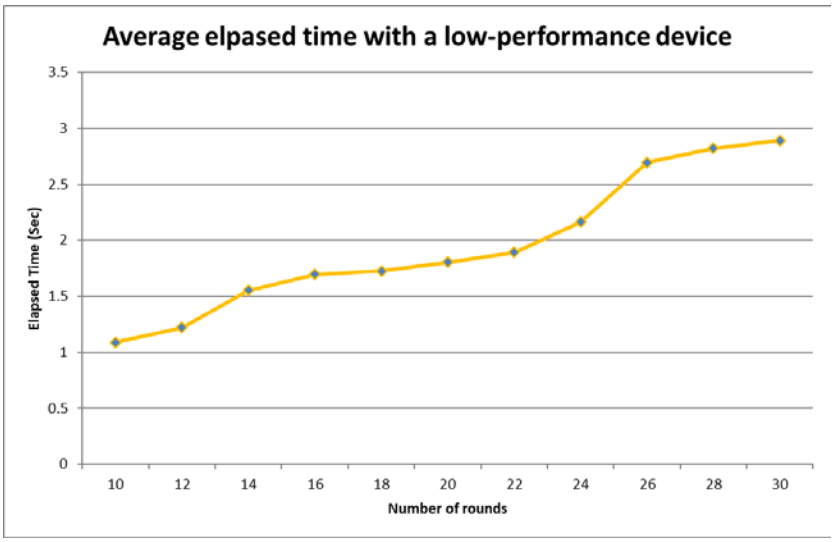

Fig. 10. Average elapsed time of the proposed scheme with a low-performance smartphone

5. Conclusion

Recent automated attacks on Android In-app Billing service enable to get valuable contents and items without paying for them legitimately. In this paper, we presented a novel scheme to prevent applications from these attacks by detecting the attempts to replace the signature verification process. We firstly analyzed the mechanism of automated attacks that bypass the signature verification process. Then we presented a basic idea to check whether the signature verification method is replaced to the attack function or not. We then considered more sophisticated attacks that may avoid the idea although it is effective currently. Finally, we proposed a scheme succeeding in detecting even sophisticated attacks with very high probability.

This scheme consists of multiple rounds where a probabilistic test is performed per each round. The test is done in a non-deterministic way to check whether the verification works correctly with a fake public key of the application. For a sufficiently large number of rounds, the scheme efficiently succeeds in detecting automated attacks with very high probability. The experiment results show that current smartphones can perform 22~30 round tests within 2 seconds, and the probability that an attack passes the detection scheme is very low (from 10^{-7} to 10^{-9}). The experiment also shows it successfully detects the representative Freedom attack.

The proposed scheme is targeted on detecting automated attacks against Android In-app Billing service, but it can be further applied to other areas where the signature verification routine is suspected. For example, it can be extended to detect malware installation disguising itself as a normal firmware update code. As a future work, we will study on efficient and secure firmware verification scheme in IoT environment based on the proposed scheme.

Acknowledgment

This work was supported by Kyonggi University Research Grant 2016.

References

[1] Li Ma, Lei GU and Jin Wang, “Research and Development of Mobile Application for Android Platform,” *International Journal of Multimedia and Ubiquitous Engineering*, vol. 9, no. 4, pp. 187-198, 2014. Article (CrossRef Link)

[2] “In-App Purchase for Developers – Apple Developer,” https://developer.apple.com/in-app-purchase/

[3] “Use In-app Billing with AIDL,” https://developer.android.com/google/play/billing/api

[4] Aditya Kurniawan, Doni Nathaniel Pranama, Junius, and Martina Megasari, “Droidglance: Network Topology Generator and Device Security Assessment Application on Android Mobile Device,” *International Journal of Software Engineering and Its Applications*, vol. 8, no. 5, pp. 189-204, 2014. Article (CrossRef Link)

[5] Sujit Biswas, Wang Haipeng and Javed Rashid, “Android Permissions Management at App Installing,” *International Journal of Security and Its Applications*, vol. 10, no. 3, pp. 223-232, 2016. Article (CrossRef Link)

[6] Wang, Pei, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei and Dinghao Wu, "Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation," in *Proc. of the 40th International Conference on Software Engineering (ICSE 2018)*. 2018. Article (CrossRef Link)

[7] Mulliner, Collin, William Robertson, and Engin Kirda, "Virtualswindle: An automated attack against in-app billing on android," in *Proc. of the 9th ACM symposium on Information, computer and communications security*, pp. 459-470, 2014. Article (CrossRef Link)

[8] “Freedom APK v3.0.1+Officially 2018,” https://freedomapk.info/

[9] “Google Play Console,” https://developer.android.com/distribute/console/

[10] “Use the Google Play Billing Library,” https://developer.android.com/google/play/billing/billing_library_overview

[11] Reynaud, Daniel, Dawn Xiaodong Song, Thomas R. Magrino, Edward XueJun Wu, and Eui Chul Richard Shin, "FreeMarket: Shopping for free in Android applications," in *NDSS*, 2012.

[12] Google, “In-app Billing Security and Design,” http://developer.android.com/google/play/billing/billing_best_practices.html, 2016.

[13] R. Xu, H. Saidi and R. Anderson, “Aurasium: Practical Policy Enforcement for Android Applications,” in *USENIX Security Symposium*, August 2012.

[14] H. Kim and S. Kim, "Securing Android In-app Billing Service against Automated Attacks," *International Journal of Security and Its Applications*, vol. 10, no. 7, pp. 259-268, 2016. Article (CrossRef Link)

[15] R.L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM 21*, vol. 21, no. 2, pp. 120-126, 1978. Article (CrossRef Link)

[16] M. Rabin, “Probabilistic algorithm for testing primality,” *Journal of Number Theory*, vol. 12, no. 1, pp. 128-138, 1980. Article (CrossRef Link)

Heeyoul Kim received the B.E. degree in Computer Science from KAIST, Korea, in 2000, the M.S. degree in Computer Science from KAIST in 2002, and the Ph.D. degree in computer science from KAIST in 2007. From 2007 to 2008, with the Samsung Electronics as a senior engineer. Since 2009 he has been a faculty member of Department of Computer Science at Kyonggi University. His major research interests include cryptography, security and blockchain.