MysteryChecker: Unpredictable Attestation to Detect Repackaged Malicious Applications in Android

Jihwan Jeong, Dongwon Seo, Chanyoung Lee, Jonghoon Kwon, Heejo Lee§
Dept. of Computer Science and Engineering, Korea University
Seoul, Republic of Korea
Email: {askjjh, aerosmiz, parfait777, signalnine, heejo}@korea.ac.kr

John Milburn
FDCC
Seoul, Republic of Korea
Email: jem@fastddc.com

***Abstract*—The number of malicious applications, sometimes known as malapps, in Android smartphones has increased significantly in recent years. Malapp writers abuse repackaging techniques to rebuild applications with code changes. Existing anti-malware applications do not successfully defeat or defend against the repackaged malapps due to numerous variants. Software-based attestation approaches widely used in a resource-constrained environment have been developed to detect code changes of software with low resource consumption. In this paper, we propose a novel software-based attestation approach, called MysteryChecker, leveraging an unpredictable attestation algorithm. For its unpredictable attestation, MysteryChecker applies the concept of code obfuscation, which changes the syntax in order to avoid code analysis by adversaries. More precisely, unpredictable attestation is achieved by chaining randomly selected crypto functions. A verifier sends a randomly generated attestation module, and the target application must reply with a correct response using the attestation module. Also, the target application periodically receives a new module that contains a different attestation algorithm. Thus, even if the attacker analyzes the attestation module, the target application replaces the existing attestation module with a new one and the analysis done by the attacker becomes invalid. Experimental results show that MysteryChecker is completely able to detect known and unknown variants of repackaged malapps, while existing anti-malware applications only partially detect the variants.**

***Index Terms*—Smartphone Security, Repackaged Application Detection, Software-based Attestation.**

I. Introduction

The popularity and demand for smartphones are increasing rapidly. However, the number of mobile malicious applications, called *malapps* [2], is also increasing because of the popularity of smartphones and the attractive characteristic that they contain users' private information. Malapps pose a threat in that they are able to steal users' private information from compromised smartphones. A security report from NQ mobile Inc. [10] in 2013 stated that around 32.8 million Android smartphones were infected with malapps.

Numerous methods for detecting malapps have been proposed, and they can be categorized into two approaches: the signature-based and the behavior-based. The signature-based detection identifies malapps by characteristic vestiges such as permissions and certificates [8], [11]. However, polymorphic and metamorphic techniques give a chance to malapps to evade the detection by generating the large number of variants The behavior-based detection identifies malapps according to predefined malicious activities [4], [5]. Since this approach monitors all the activities generated by many different applications, it requires more computational power than the signature-based approach.

Most significant problem of the phenomenon of malapps is the repackaged malapps. Since, there are many application source analysis tools available on the Internet. Users can extract the source code of many Android applications. Consequently, attackers are easily able to rebuild benign applications into malapps, called repackaged malapps, based on legitimate applications. Many malapps have been built using repackaging techniques [10]. The repackaged apps create security threats such as billing attacks and unauthorized access to sensitive personal data (banking certificates, PIN numbers, etc). Moreover, in the case of applications such as banking applications that deal with sensitive personal data, malapps can retrieve account information, authentication certificates, account numbers, and even PIN numbers from users [18]. For securing smartphone applications, detecting the modification of an application by repackaging is a compelling issue.

A software-based attestation scheme for detecting malware has been used in resource-constrained environments such as embedded devices [14], [15], sensor networks [13], and mobile devices [6] by using lightweight operations. Attestation is an effective technique to detect widely deployed and variously manipulated codes when a verifier knows the valid code. By detecting the manipulated apps without heavy monitoring and tons of signatures, the software-based attestation effectively overcomes the limitations of previous approaches on responding huge number of repackaged malapps. But legacy attestation schemes are not fully suitable in terms of secrecy. In challenge-response attestation, which is widely used in software-based attestation, when the verifier sends a challenge to the target device, the target runs a built-in attestation algorithm (usually, a crypto functions) and returns the hashed output as a response [14]. However, an attacker can manipulate communication pairs through a man-in-the-middle attack. If

§ Corresponding author: heejo@korea.ac.kr

978-1-4799-7329-3/14/$31.00

attackers intercept a challenge value at challenge step, they can generate correct response for deceiving the verifier by analysing the built-in attestation algorithm. One-way attestation, which can prevent man-in-the-middle attacks, periodically sends a response without receiving a challenge [17]. However, it also utilizes a built-in attestation algorithm, which attackers can analyze and exploit through reverse engineering.

Considering all these factors, a viable repackaged malapp detection scheme should meet three requirements: 1) malapp variants are detected 2) attestation algorithm secrecy and 3) low resource consumption. Since malapp variants can be easily produced by repackaging, a proposed approach should detect known/unknown malapp variants. The secrecy of an attestation algorithm prevents attackers from analyzing the algorithm; attackers cannot infer correct responses. In addition, low resource consumption in terms of storage usage and computation time is necessary considering the resource-constrained environment.

In this paper, we propose a viable repackaged malapp detection scheme, called MysteryChecker[1], which uses an unpredictable attestation algorithm. A verifier provided by an application provider (it can be a any organization which manages applications which use users' private information), generates an attestation module that includes randomly selected crypto functions, such as crypto hash functions. Then, the target application, an application containing modules/functions that checks the integrity of the application itself, receives the attestation module instead of a challenge. The verifier builds a new attestation module which use a different attestation algorithm and different hash function each time, whereas existing approaches have fixed, built-in attestation algorithms. Although attackers can analyze the attestation module in the target application, the target application replaces the existing attestation module each time with a new one; the attacker's analysis thus becomes invalid.

MysteryChecker has three phases: 1) attestation module generation 2) bootstrapping attestation and 3) on-demand attestation. A verifier randomly generates an attestation algorithm in the first phase. When a user installs and launches the target application, the second phase, the user can check the integrity of the application. In the third phase, the target application exchanges attestation modules and responses whenever a critical process is in progress. MysteryChecker is focused on protecting games and banking applications, checking critical processes such as money transfer.

In our experiments, MysteryChecker successfully detects all the known and unknown variants of repackaged malapps, whereas 7 out of 10 off-the-shelf anti-malware applications were not able to detect the known variants, and none of the anti-malware applications detects unknown malapp variants. Furthermore, the attestation processes of MysteryChecker takes 0.383 sec, therefore MysteryChecker does not interfere useability, in a Samsung Galaxy S4.

Our main contribution is the design of an unpredictable software-based attestation approach that detects repackaged malapps in smartphones and satisfies the three stated requirements (malapp variants detection, attestation algorithm secrecy, and low resource consumption).

The rest of this paper is organized as follows. In Section II, we examine an application repackaging and software-based attestation scheme and discuss the problem definition. In Section III, we describe MysteryChecker's framework. In Section IV and V, we evaluate and analyze MysteryChecker. We introduce more related papers in Section VI. We present our conclusions in Section VII.

II. Background

Most malapps in Android smartphones are generated by the repackaging technique [10]. The existing signature-based malware detecting methods can not detect variant malapps, since attackers can easily change the signature of apps by the repackaging technique. The behavior-based detecting methods can recognize repackaged malapps, but existing behavior-based methods are not suitable for the Android smartphones because the smartphones are usually a resource-constrained environment and the methods requires high computational power. To secure Android environmnet, detecting repackaged malapps is a compelling issue, so we suggest to detect known and unknown variants of repackaged malapps by a software-based attestation method. However, existing software-based attestations such as Challenge-response and One-way attestation have significant weaknesses. Consequently, we propose a new software-based attestation method.

- **Application Repackaging:** The code of Android applications can be easily extracted using repackaging tools such as Apktool. Application writers can insert code into existing applications, rebuild them as new or updated applications, and distribute repackaged applications through the Google Play store or third-party markets. Arise from this, attacker used to apply repackaging technique to insert malicious code to popular applications. While repackaged malapps operate like legitimate applications in the foreground, they attempt malicious behaviors, such as accessing contacts and sending spam messages in the background. The signature-based methods must have an application of the malware to detect the malware, so it is very hard to discover unknown malapps. Furthermore, an attacker can easily change the signature of malapps by repackaging technique. As a result, the signature-based methods are hard to detect variant malapps.
- **Software-based Attestation:** An attestation approach serves to confirm the integrity of a target device. It checks whether the target's memory has changed. We introduce two types of the software-based attestation approach, which are cheap and easily applicable [15], that motivated our work. The first is ***challenge-response attestation***. The verifier transmits a challenge to a target and requests a response. Then, the target produces a unique output and returns it as a response to the verifier. The verifier compares the response with the expected result. The response is changed according to the challenge. Although challenge-response attestation prevents replay attacks, it is still susceptible to network attacks such as Man-In-The-Middle attack. Another approach is ***one-way attestation***. This was designed to prevent network attacks.

[1]A mystery checker, also known as a mystery shopper, measures quality of service in a store. While gathering information, mystery checkers usually blend in as regular shoppers at the store being evaluated. Our proposed attestation scheme performs similarly to the mystery checker.

The target sends a response without receiving a challenge from the verifier, using its built-in algorithm which is shared with the verifier. Nevertheless, since the attestation module is located in the target device, attackers can analyze the module by reverse engineering and infer proper responses.

- **Resource-constrained environment:** A mobile smartphone is a resource-constrained environment. The smartphone has a limited battery and its computation power is lower than a traditional desktop-computing environment. Therefore, light-weight execution is very important in terms of practicality. But, existing malware detection methods, especially the dynamic detection method, are too heavy to operate on a mobile environment.

The main weaknesses of existing anti-malware and attestation solutions are the lack of ability to detect variants which generated by applying repackaging technique, and high storage and compute resource consumption. A viable approach to successfully detecting repackaged malapps should meet next three requirements:

- **Malapp variants detection:**
  An attestation approach should detect not only known, but also unknown variants of malapps. The approach requires an ability to identify changes in application code.
- **Attestation algorithm secrecy:**
  Attackers should not be able to infer correct responses by analyzing the attestation algorithm. If it is possible to infer the correct response, malapps can easily evade detection even though the method can detect code changes.
- **Low resource consumption:**
  An attestation approach should use low resources because smartphones are resource-constrained. In particular, storage usage and computational time which can reduce the usability of an application, are sensitive resources for users.

The goal of this paper is to demonstrate the design of a viable attestation approach that satisfies the above three requirements.

III. MysteryChecker description

In this section, we first describe the design principles of MysteryChecker. Then, we give a brief overview of the MysteryChecker and describe each phase of MysteryChecker in detail.

A. Design Principles

We established three design principles to satisfy the three requirements as mentioned in Section II.

Application digest. The attestation module in MysteryChecker computes the hashed output, called the app digest, of the target application binary by a crypto functions (e.g., SHA-1). Therefore, it detects any changes of the application code.

Non built-in and random attestation algorithm. The target application *does not contain* the attestation module. While running, target application receives a new random attestation module from the verifier each time. Certainly, attackers can intercept and analyze the attestation algorithm after the target application downloads the attestation module. However, the module has already sent the app digest as a response before the attackers complete their analysis, and the analysis becomes immediately invalid since the attackers cannot predict the next algorithm. For this reason, attackers can not replay a response packet from a different device or a previous response.

Detection with simple operations. MysteryChecker utilizes a crypto functions only; it does not require storage for signatures and operations for scanning signatures. Thus, MysteryChecker can reduce the storage usage and operation.

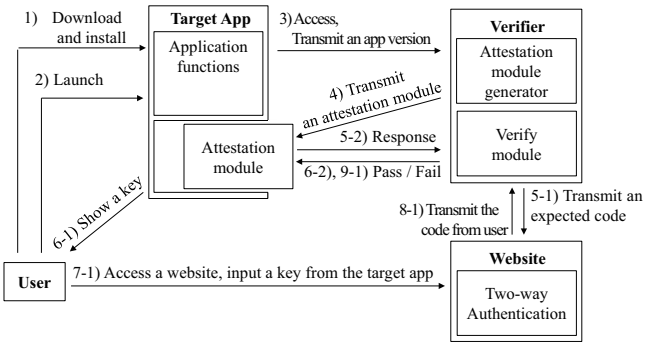

Fig. 1. Overview of MysteryChecker: A verifier includes an attestation module generator, which builds a random attestation algorithm. The target application receives the attestation module from the verifier.

B. Overview

Here, we briefly describe the operation of MysteryChecker. MysteryChecker consists of three phases: 1) attestation module generation 2) bootstrapping attestation and 3) on-demand attestation. In the following, we explain the core operation of each phase, as shown in Figure 1.

1) **Attestation module generation:** An attestation module verifies the target application. The attestation module generator, which is a component of the verifier, randomly generates an attestation module by using unique information of the target application. The APK file and hash value of the target application, or the Android keystore can become unique information. The generator randomly selects an information and build a transform function chain for the attestation module. The transform function chain is a stack or sequence of transform functions such as string reordering, random string attaching and hashing by several possible hash algorithms. The order of the functions in chain and the size of execution round are also random. The module generator convert transform function chain to attestation module. After generating the attestation module, the verifier generates a key by using the generated attestation module with an original, clean room version of the target application and keeps the key for attestation.
2) **Bootstrapping attestation:** When a user installs or launches an application which implements MysteryChecker, it detects repackaged applications. (1)First, user downloads and installs a target application. (2)

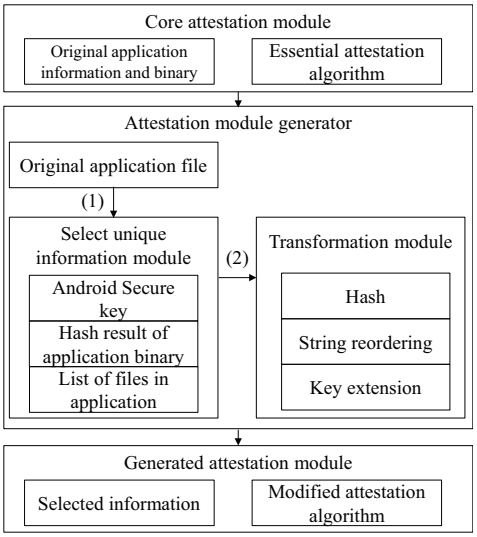

Fig. 2. Phase 1: Attestation module generator in a verifier, which builds attestation modules that produce varying results.

When user launch the target application, (3) the target application attempts to access the verifier for the first time. At this point, the target application sends application information to the verifier such as the version of the application. (4) The verifier transmits the attestation module to the target application. (6-1) The target application executes the attestation module and shows a key to the user. (7-1) After the user inputs the response value to the website for two way authentication, (9-1) the user can obtain its integrity according to the result received from the verifier.

3) **On-demand attestation:** MysteryChecker checks the integrity of an application whenever the user tries to access a server that requires private information. Whenever the target application accesses the verifier (3), the verifier transmits a different attestation module to the target application (4). Then, (5-2) the target application executes the attestation module and returns corresponding response to the verifier. (6-2) If the response is correct, the verifier replies Pass. If not, the verifier replies Fail.

MysteryChecker provides secure attestation through unpredictable algorithms. MysteryChecker builds a sufficiently large number of attestation algorithms to prevent predicting the attestation algorithm. Moreover, when an attacker analyzes an attestation module, MysteryChecker keeps its secret by replacing the existing attestation module with a new one that contains a different algorithm.

C. Phase 1: Attestation Module Generation

In phase 1, The verifier generates an attestation module for target application which wants to get attestation and also generates the key K_v which is used for attestation in Phase 2. When a user installs a target application which is implemented with built-in MysteryChecker, built-in MysteryChecker sends information of taget application's name and version to a verifier. After the verifier receives the information from the target application, the verifier starts to build attestation module. First, the verifier randomly select one information of the target application such as a hash result of application binary, secure key and list of files in application. This information selection step leads increment of variety of attestation module and it makes hard to attackers to guess a right input parameter of the attestation module. Therefore, the verifier must have original APK file and the version information of target application. After selecting information, the verifier build a transform function chain T which converts the selected information to K_v. T is a chain of transform functions which consist of string reordering functions R, random string attaching functions A and hash functions H_a using several different crpyto hash algorithms. R convert the string into a random reorder. For example, string 'ABCD' converts to 'DCBA' after reordering. A attach a random string such as '012X' to the input string 'DCBA', so then the output string is 'DCBA012X'. H_a make hash value of the input string with a random hash function a. MD5, SHA-1 and whirlpool can become used as a. Figure 3 shows an example of how to build an attestation module. E_n denotes an execution round and the box represents a

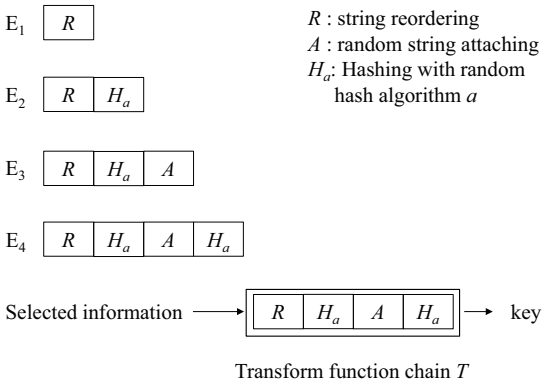

Fig. 3. Transforming process selects functions to generate attestation module.

Algorithm 1 Attestation module generation

Input: Unique information of target application
Output: $attestation\_module$;

$I \leftarrow RandomInfoSelecter(AppVersion, AppName)$;
// Select information from an application
$function\_chain \leftarrow \{\}$
$execution\_round \leftarrow random()$;
while $execution\_round$ **do**
  $execution \leftarrow randomFuncSelecter()$;
  if $execution$ is stringReorder **then**
    $function\_chain.add(R)$;
  else
    if $execution$ is randomStringAttach **then**
      $function\_chain.add(A)$;
    else
      if $execution$ is hash **then**
        $a = randomHashAlgorithmSelecter()$;
        $function\_chain.add(H_a)$;
      end if
    end if
  end if
end while
$attestation\_module \leftarrow convert(function\_chain)$;

transforming function. As E increases, the module generator randomly selects one of the transform functions and stacks the function into T. At the end of E, the module generator converts the transform function chain to the new attestation module. Algorithm 1 shows the whole process to generate an attestation module. Before transmitting the attestation module to the target application, the module is used for generating K_v. The verifier executes the module with the original version of target application. The executed module generate K_v and the verifier keeps K_v for attestation at Phase 2. Finally, the verifier sends the attestation module to the target application.

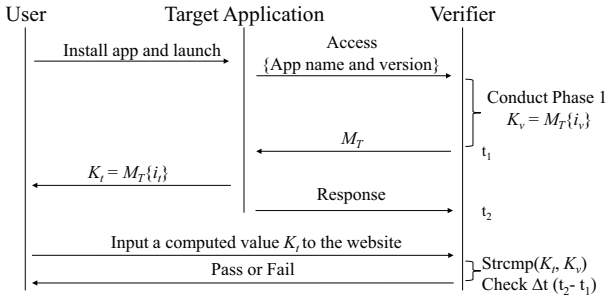

Fig. 4. MysteryChecker bootstraps in Android smartphones. It operates when the application is launched for the first time.

D. Phase 2: Bootstrapping Attestation

Phase 2 protects sensitive personal data from being leaked through the repackaged malapps at the first use. MysteryChecker uses a two-way authentication method in Phase 2 like some banking apps, because two-way authentication prevents tampering attacks. Figure 4 illustrates information exchanges between the user, the target application, and the verifier. When the user installs an application which use MysteryChecker, the application sends the name and version information of the application to the verifier. Upon receipt of the information, verifier generates an attestation module by Phase 1 based on received information and transmits the attestation module M_T to the target application. The target application execute M_T by using built-in MysteryChecker module. The built-in module is implemented as package form. Therefore, original applications should be added built-in MysteryChecker package. When M_T is executed, M_T collects an information which is selected from the verifier in Phase 1 and inputs the information to transform function chain in M_T to generate the key K_t. After generating K_t, built-in MysteryChecker display K_t to the user. Then, the user manually inputs K_t to a verifier web server form. Finally, the verifier checks the result by comparing the user response to the generated key result to determines whether the installed application has been tampered with or not. Phase 2 can be attacked by a Man-In-The-Middle attack. Attackers can steal the attestation module before the target application receives it, then reverse engineer the module to obtain the key. To prevent a Man-In-The-Middle attack, MysteryChecker checks Δt, which is the time difference between time t_1 when the attestation module is sent and t_2 when the response arrives. If Δt=(t_2-t_1) is not too large, MysteryChecker determines that the attestation operated normally. Detailed analysis on Δt will be discussed in Section 6.

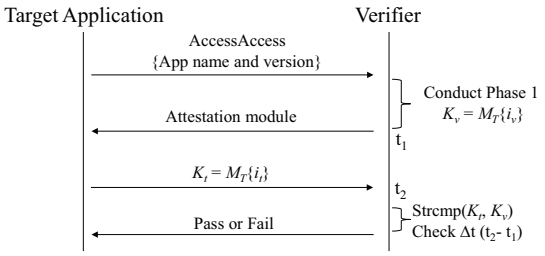

Fig. 5. After Phase 2, MysteryChecker exchanges messages whenever a critical process is in a progress.

E. Phase 3: On-demand Attestation

Figure 5 shows the attestation operation after bootstrapping. There are two attestation methods for on-demand attestation. The first is to attempt the attestation from the verifier periodically. Another is an attempt from the target application when the target application or user does any critical process, such as fund transfer in a banking app. In our work, whenever the target application tries to access the server, the attestation operation is run. The verifier transmits the attestation module to the target application. The target application executes the attestation module and replies to the verifier. The verifier decides Pass or Fail, depending on whether the expected value coincides with the result from the target application. At Phase 3, the verifier also checks Δt, and if it is too large, the attestation will fail.

IV. Evaluation

We evaluated the effectiveness of MysteryChecker in terms of repackaged malapp detection and resource consumption. We also estimated diversity of generable modules. For an experiment environment, we used Samsung GALAXY S2 and S4 for android device and use WIFI. We added built-in MysteryChecker to the sample application which is distributed by Google for experiment. We also repackaged the sample application according to scenarios of repackaged malapp detection below to make malicious samples.

A. Repackaged Malapp Detection

We compared MysteryChecker with off-the-shelf anti-malware applications to confirm that MysteryChecker detects repackaged malapps correctly. In our experiments, we used three scenarios, which can cover sufficient types of transformation schemes based on [12], as follows: 1) false-positive diagnosis 2) variants detection and 3) unknown malicious detection. Tested off-the-shelf anti-malware applications are selected from "Detailed Test Reports" published in AV-TEST [1].

False positive diagnosis. We examined whether off-the-shelf anti-malware applications considered the package path and name as malicious signatures. We built fake malapps containing the same path or name as the malapp called "Suiconfu". We found that 50% of anti-malware applications detect the fake malapps as malicious. This implies that many anti-malware applications determine the package path and name as signatures.

Variants detection. We manipulated applications by repackaging. We change the package path and name to modify

TABLE I
WHILE NO EXISTING ANTI-MALWARE APPLICATIONS CAN DETECT CRAFTED MALAPP AND SOME OTHER VARIANTS, MYSTERYCHECKER DETECTS ALL TYPES OF REPACKAGED MALAPPS.

Repackaging methods		MysteryChecker	Bitdefender	Trend Micro	Avast	ESET	Comodo	F-secure	Symantec	MicroWorld	NQ Mobile	Antiy
False positive diagnosis	Fake path of malapp	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Fake path, name of malapp	Y	N	Y	N	Y	N	Y	Y	N	Y	N
Variants detection	Repack	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Disassemble & assemble	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Rename package name	Y	Y	Y	Y	N	Y	Y	N	Y	Y	N
	Rename package path	Y	N	Y	Y	N	Y	Y	N	Y	Y	N
	Rename identifiers	Y	N	Y	Y	N	Y	Y	N	Y	Y	N
	Reorder code	Y	N	Y	Y	N	Y	Y	N	Y	Y	N
	Encrypt string and array	Y	N	Y	Y	N	Y	Y	N	Y	Y	N
	Insert junk code	Y	N	Y	Y	N	Y	Y	N	Y	Y	N
Unknown malicious	Crafted malicious APK	Y	N	N	N	N	N	N	N	N	N	N

the application's code through the smali language. We also fixed the applications's identifiers for this experiment. Rastogi *et al.* [12] proposed these techniques. As a result, 40% of anti-malware applications could not detect the variants, even when only the package name was modified. However, MysteryChecker detected all variants in the experiment.

Unknown malicious detection. We generated an unknown malicious application which has four malicious activities (SMS scanning, steal Google account information, GPS tracking, malicious URL propagation) defined by the "Android Trojan Horse" from Jeremy et al. [7]. None of the off-the-shelf anti-malware applications detected the crafted malapps whereas MysteryChecker did.

In conclusion, table I shows a comparison of the detection performance of MysteryChecker and some anti-malware applications. It can be seen that 7 out of 10 off-the-shelf anti-malware applications do not detect the variant malapps and none of off-the-shelf anti-malware applications detects unknown malapps. MysteryChecker, on the other hand, can completely detect all the variants.

B. Diversity Estimation

TABLE II
COMPARISON OF ATTESTATION APPROACHES: MYSTERYCHECKER PROVIDES SECURE ATTESTATION BECAUSE IT HAS HIGH DIVERSITY AND PROVIDES SECRECY OF THE ATTESTATION ALGORITHM.

Approach	Protocol	Diversity	Secrecy
Challenge-response	R = F(A, c)	Limited	Low
One-way	R = F($A, \cdot$)	Limited	Intermediate
MysteryChecker	R = F($A_k, \cdot$)	Not limited	High

Table II compares MysteryChecker with two existing attestation approaches. R and $F(\cdot)$ denote a produced response and a attestation function, respectively. $F(\cdot)$ consists of attestation algorithm A of attestation function and the challenge c from the verifier. k denotes the changes of A. The attestation algorithm of challenge-response and one-way attestation has limited diversity, since the attestation algorithm can not changed after distributing an application which apply the attestation algorithm. However, The diversity of attestation algorithm of MysteryChecker is not limited. Because, MysteryChecker build a new attestation module and send it to a target application when the target application needs to be verified. Consequently, secrecy of MysteryChecker is higher then other attestation method.

To get the secrecy of attestation algorithm in our proposed method, high diversity is very important. We can estimate the diversity of MysteryChecker based on transform functions.

$$\begin{aligned} N &= H^{E_1} \cdot R^{E_2} \cdot A^{E_3} (E = E_1 + E_2 + E_3) \\ &= H^{E_1} \cdot \left(\frac{l_1!}{2!(l_1-2)!}\right)^{E_3} \cdot \left(2^{8 \times l_2}\right)^{E_2} \end{aligned} \quad (1)$$

N denotes the number of possible attestation algorithm. H,R and A denote transform functions which are the number of cases of hash function, string reordering functions and random string attaching function, respectively. E denotes entire round time and E_1, E_2 and E_3 denote round time of each transform function. H is related with hash function. The number of hash functions which are used to attestation algorithm and type of hash function effect to H. R is related with string length l_1. R increases according to l_1. A increased by string length l_2 which is random string to attach to an original string. To estimate the number of possible attestation algorithm, we fixed the hash function to SHA-1 and length of random string to attaching to 16 byte which same is as our test environment. In the selected environment, R is 2^{160} because SHA-1 hash value has 160 bits and A is 2^{128}. As a result, estimated N is at least $2^{288 \cdot \frac{E}{3}}$. It is enough large number to prevent the attack by rainbow table or reply attack. If the verifier use more secure functions, it makes attackers more difficult to exploit MysteryChecker.

C. Measuring Resource Consumption

We checked the time consumption of the attestation process and the size of the attestation module, since low time consumption and small size of attestation modules are important in the resource-constrained mobile device environment. First, we measured the average size of the attestation module for execution rounds from 1 to 2,048. To obtain more precise results, we conducted the test several 10s of times and calculated the average value. Figure 6 shows the measured attestation module size. The average size of the module is 6,084 bytes at execution round 1 and 7,210 bytes at execution round 2,048. The result shows that the increase rate of the attestation module size according to execution round is low enough to use. We estimate the transfer time of the module using the result. 3G networks currently provide data transfer speeds of up to 384kbps, so the transfer time of 7kb module is under 0.2 second. This means that the size of the attestation module is small enough to transmit in android, 3G environment.

Second, we measured the attestation execution time on the android device. we also measured the time according to execution round 1 to 2048 and fixed the hash algorithm to obtain more precise results. We use MD5 and SHA-1 for hash algorithms because these algorithms are popular. The results of this experiment are presented in Figure 7. The attestation

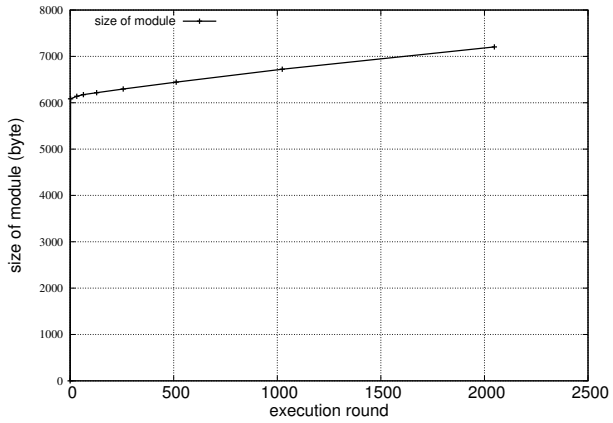

Fig. 6. The size of a module according to execution round

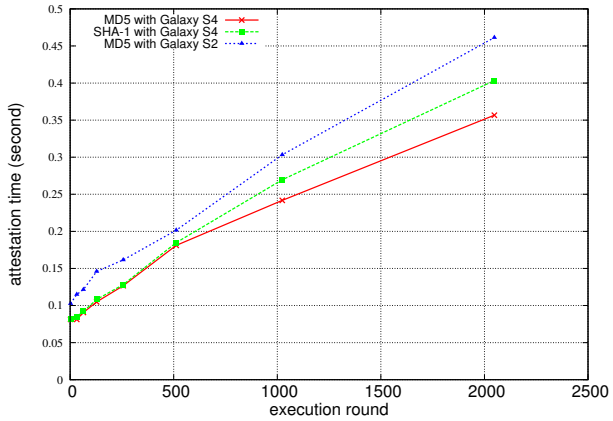

Fig. 7. Attestation time consumption graph according to execution round

time increases 0.08 second to 0.35 second in case of MD5 and 0.08 second to 0.4 second in case of SHA1. We also estimate the attestation time with a Galaxy S2 which has lower computation power than the Galaxy S4. The attestation time used 0.1 second to 0.46 second on the Galaxy S2 and the time is not much higher than the attestation time on the Galaxy S4. The result shows that the attestation of MysteryChecker is light enough to use on low computation power machine.

The low time consumption when running a certain operation on a smartphone is important in actual use. The average time of whole attestation process of MysteryChecker takes 1.0 second in our experiment. It is not much slower than other attestation methods. Therefore, MysteryChecker is an efficient and high secrecy method for attestation.

V. Security Analysis

If an attacker wants to evade attestation, he may attempt to modulate the attestation process, or to tamper with a network process between the target application and the verifier. In this section, we discuss potential attacks as describe above, and explain how MysteryChecker can defeat them.

A. Systematic Attacks

Brute Force Attack. An attacker tries to collect and analyze attestation modules, so as to predict the next response. With MysteryChecker, however, the verifier can produce an infinite number of attestation modules using combinations of crypto functions, string reordering, and key extension. Therefore, it is impossible for attackers to predict the next response.

Pause-and-Resume Attack. An attacker utilizes reverse engineering tools on the target application and attempts to manipulate its attestation module in order to replace incorrect with correct responses. As shown in Figure 7, the verifier receives a response within 0.383 sec; therefore, we can set an expiration time. If the verifier does not receive a response within the expiration time, the verifier can assume that the target application has been manipulated.

Replication Repackaged Malapp. A replication malapp is a kind of strengthened repackage scheme. This type of attack stores the original application to a dummy folder. When a verifier sends a challenge and tries to attest, the replication malapp uses the original application to authenticate instead of the repackaged malapp. Then the repackaged malapp can reply with a correct response from the original application. We use unique information provided in Android smartphones to detect a replication attack. More specifically, the attestation module requires the absolute path of identifiers, paths, and values in a target application. The command that obtains the absolute path of an application can not be easily modified by the attacker. In addition, we use the "Secure.ANDROID_ID" value in a smartphone device, which represent a device unique ID for identifying application users.

B. Network Attacks

Attestation Process Forgery Attack. We assume that two types of tampering attacks can happen. First, an attacker can tamper with a link address for verifier, causing the application to download the attestation module from a fake verifier. The fake verifier prepares a bogus attestation module in advance; this leads to false attestation. However, MysteryChecker checks the integrity of the target application using two-way authentication, as in Phase 2, and thus, the user can identify whether he has download a genuine application. Second, an attacker can skip Phase 2 by using repackaging schemes. However, the attacker can not steal user personal data because the verifier will close a connection that is not approved.

Man-In-The-Middle Attack. An attacker can try to intercept the attestation module sent to the target application from verifier and reverse the attestation module to get the proper key to deceive the verifier. If the attacker knows that which information are used and which transforming functions are included in the attestation module, the attacker can generate a proper key for deceiving the verifier. To prevent Man-In-The-Middle attack, MysteryChecker checks the time difference of the response arrival time from the attestation module sent time. In our work, we determined that 2 seconds was a proper time. We can divide attestation process into sending the module, attestation and receiving the response. Sending the module and receiving the response times are affected by network delay. However, the size of an attestation module is less than than 7kb. Therefore, the transfer time of the attestation module should not exceed 0.2 seconds, and the response time should not be more than 0.5 seconds, particularly since the response

can be transmitted in a single packet. For overall attestation time, we compare with other software-based attestation. The SBAP [9], which is one of the software-based attestation approaches, take about 1.7 seconds for attestation. In our experiment, MysteryChecker's average attestation time is 0.4 seconds. As a result, we can decide to set the maximum proper time to 2 seconds. If an attacker wants to deceive the MysteryChecker, the attacker should intercept and reverse the module and send the response in 2 seconds. It is a difficult problem, since the attacker does not have the benefit of knowledge about the module, and the number of possible modules is large enough to be unpredictable.

VI. Related work

In terms of mobile anti-malware, Cha *et al.* [3] proposed SplitScreen, which has two-phase scanning. Its FFBF (feed-forward Bloom filter) and a small set of files and signatures enables fast, memory efficient malware detection, and reduces the amount of signatures (storage). However, since this mechanism is signature-based, it struggles to detect unknown malicious applications, especially repackaged apps. Yan *et al.* [19] presented three ways to defend against mobile malware. They suggest monitoring power consumption, increasing platform diversity, and enforcing hardware sandboxes are methods of detecting mobile malware. However, our evaluation is that those three proposed methods lead to ambiguous results and high false-positive rates.

For the Attestation method, Shi *et al.* [16] proposed BIND (Binding Instructions aNd Data), a fine-grained attestation method that ties the proof of what code has executed to the data the code has produced. By attesting to the critical code immediately before it executes, they try to narrow the gap between time-of use and time-of-attestation discrepancy. Seshadri *et al.* [14] proposed SWATT (SoftWare-based ATTestation technique), a challenge-response method of attestation. In embedded devices or sensor devices, they constructed verification procedures that computes a checksum over memory. Then attacker can't tamper with the content of that memory easily. Moreover, they use a randomized access pattern to make alteration more difficult.

To defend against repackaging, DroidChameleon from Rastogi *et al.* [12] is a systematic framework with various transformation techniques. They tested the effectiveness of ten anti-malware products on Android and all the studied anti-malware products were vulnerable to transformations. This paper briefly suggests two defense methods, one is semantics-based malware detection and the other is support from the platform. Both defense mechanisms are hard to implement and still quite challenging (the author of this paper readily admitted these difficulties).

VII. conclusion

We propose a novel software-based attestation approach to detect repackaged malapps, called MysteryChecker, which leverages unpredictable attestation algorithms. It satisfies the necessary requirements to be a viable attestation approach. MysteryChecker detects code changes to a target application using an application digest. Since the attestation module is randomly generated in the verifier, attackers cannot predict correct responses. Experimental results show that MysteryChecker detects all the known and unknown variants of repackaged malapps promptly, whereas existing anti-malware applications detect only few variants.

VIII. acknowledgment

This research was supported by the Public Welfare & Safety Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2012M3A2A1051118).

References

[1] AV-TEST, "Detailed Test Reports - Android," Jan. 2013, http://www.av-test.org/en/tests/mobile-devices/android/may-2013/.

[2] I. Bente, J. von Helden, B. Hellmann, J. Vieweg, and K.-O. Detken, "Esukom: Smartphone security for enter-prise networks," *ISSE 2011 Securing Electronic Business Processes*, 2012.

[3] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, "Splitscreen: Enabling efficient, distributed malware detection," *Communications and Networks, Journal of*, vol. 13, no. 2, pp. 187–200, 2011.

[4] A. Desnos and P. Lantz, "Droidbox: An Android application sandbox for dynamic analysis," 2011.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones." in *OSDI*, vol. 10, 2010, pp. 255–270.

[6] M. Jakobsson and K.-A. Johansson, "Retroactive detection of malware with applications to mobile platforms," in *Proceedings of the 5th USENIX conference on Hot topics in security*. USENIX Association, 2010, pp. 1–13.

[7] Jeremy Klein, Parker Spielman, "Android Trojan Horse," Dec. 2011, http://www.cse.wustl.edu/~jain/cse571-11/ftp/trojan/index.html.

[8] I. Jeun, K. Lee, and D. Won, "Enhanced code-signing scheme for smartphone applications," in *Future Generation Information Technology*. Springer, 2011, pp. 353–360.

[9] Y. Li, J. M. McCune, and A. Perrig, "Sbap: Software-based attestation for peripherals," in *Trust and Trustworthy Computing*. Springer, 2010, pp. 16–29.

[10] NQ mobile, "NQ mobile security report 2012," 2013, http://www.nq.com/2012_NQ_Mobile_Security_Report.pdf.

[11] M.-W. Park, Y.-H. Choi, J.-H. Eom, and T.-M. Chung, "The Permission-Based Malicious Behaviors Monitoring Model for the Android OS," in *Computational Science and Its Applications–ICCSA 2013*. Springer, 2013, pp. 382–395.

[12] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating Android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 329–334.

[13] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Proceedings of the 5th ACM workshop on Wireless security*. ACM, 2006, pp. 85–94.

[14] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*. IEEE, 2004, pp. 272–282.

[15] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, "Remote software-based attestation for wireless sensors," in *Security and Privacy in Ad-hoc and Sensor Networks*. Springer, 2005, pp. 27–41.

[16] E. Shi, A. Perrig, and L. Van Doorn, "Bind: A fine-grained attestation service for secure distributed systems," in *Security and Privacy, 2005 IEEE Symposium on*. IEEE, 2005, pp. 154–168.

[17] K. Song, D. Seo, H. Park, H. Lee, and A. Perrig, "OMAP: One-way memory attestation protocol for smart meters," in *Parallel and Distributed Processing with Applications Workshops (ISPAW), 2011 Ninth IEEE International Symposium on*. IEEE, 2011, pp. 111–118.

[18] Threat post, "Zeus Variant Targets Mobile Online Banking Apps," Sep. 2010.

[19] Q. Yan, R. H. Deng, Y. Li, and T. Li, "On the potential of limitation-oriented malware detection and prevention techniques on mobile phones," *International Journal of Security and Its Applications*, vol. 4, no. 1, pp. 21–30, 2010.