Breaking into the KeyStore: A Practical Forgery Attack Against Android KeyStore

Mohamed Sabt[1,2(✉)] and Jacques Traorè[1]

[1] Orange Labs, 42 Rue des Coutures, 14066 Caen, France
{mohamed.sabt,jacques.traore}@orange.com
[2] Sorbonne Universités, Université de technologie de Compiègne Heudiasyc,
Centre de recherche Royallieu, 60203 Compiègne, France

Abstract. We analyze the security of Android KeyStore, a system service whose purpose is to shield users credentials and cryptographic keys. The KeyStore protects the integrity and the confidentiality of keys by using a particular encryption scheme. Our main results are twofold. First, we formally prove that the used encryption scheme does not provide integrity, which means that an attacker is able to undetectably modify the stored keys. Second, we exploit this flaw to define a forgery attack breaching the security guaranteed by the KeyStore. In particular, our attack allows a malicious application to make mobile apps to unwittingly perform secure protocols using weak keys. The threat is concrete: the attacker goes undetected while compromising the security of users. Our findings highlight an important fact: intuition often goes wrong when security is concerned. Unfortunately, system designers still tend to choose cryptographic schemes not for their proved security but for their apparent simplicity. We show, once again, that this is not a good choice, since it usually results in severe consequences for the whole underlying system.

Keywords: Android KeyStore · Authenticated encryption · Integrity

1 Introduction

Smartphones are used in an ever-growing variety of use-cases, including highly-sensitive tasks. Third party applications often need to generate and use some sensitive data, such as authentication credentials and cryptographic keys. Unfortunately, no strong protection is guaranteed for these highly valuable data, which might attract powerful attackers motivated by economic gain. This lack has hindered the adoption of smartphones in certain areas in which the use of cryptographic keys is crucial. The development of smartphone market spurs mobile system designers to reinvent their security features. Starting from Android 4.3, aka Jelly Bean, official support for app-specific secrets storage has been provided by a newly introduced component, called *Android KeyStore*.

The Android KeyStore is an Android system service that allows applications to generate, use and store their cryptographic keys. Once inside the KeyStore,

© Springer International Publishing Switzerland 2016
I. Askoxylakis et al. (Eds.): ESORICS 2016, Part II, LNCS 9879, pp. 531–548, 2016.
DOI: 10.1007/978-3-319-45741-3_27

keys can no longer be extracted. They can be used for cryptographic operations without ever leaving the KeyStore.

Multiple implementations exist for the KeyStore. The default one, provided by Google, does all key-related operations using the OpenSSL library. It protects the integrity and the confidentiality of its keys by storing them in encrypted form using authenticated encryption (AE). For some reason, the scheme in use is particular and does not follow any standardized or provably secure construction. Its idea is simple: the message (representing the stored key) is appended to its MD5 hash value before encrypting it with CBC (cipher block chaining) mode. Henceforth, we call this AE scheme *Hash-then-CBC-Encrypt*.

At the first look, Hash-then-CBC-Encrypt is a lightweight mode that has many advantages over other popular AE schemes. It is more efficient than those based on the generic composition approach [6], since the message is needed not to be processed twice. In addition, it is much simpler to implement compared to others. Therefore, it might seem to be the most fitting scheme to implement inside mobile devices for the protection of users keys.

1.1 Our Contribution

In this paper, we show that the use of non-provably secure cryptographic schemes in complex architectures could cause severe consequences. We start by proving that the AE scheme **Hash-then-CBC-Encrypt** does not provide authenticity **regardless of the used hash function**. To this end, we show that it does not satisfy two notions of integrity: integrity of ciphertext (INT-CTXT) and ciphertext unforgeability (CUF-CPA). Then, we present a selective forgery attack where an adversary exploits this weakness to substantially reduce the length of the symmetric keys protected by the KeyStore.

We illustrate this security flaw by defining an attack scenario in which an application entrusts the KeyStore with its symmetric key. Our attack lulls users into a false sense of security by silently transforming, for instance, 256-bit HMAC keys into 32-bit ones. This allows a malicious third party that controls the network to break any secure protocol based on these weak keys. Such an attack might constitute a real threat, since it could happen undetected. At the writing of this paper, our attack affects the latest Android build (android-6.0.1_r22).

Our work brings to light an interesting fact: security in modern systems still does not withstand a simple cryptanalysis. Astonishingly, recently, the KeyStore has been significantly enhanced by new features without reviewing its security correctness. We show that, once again, security by feature-enhancing is disappointedly misleading. Moreover, it is really tempting for system designers to use ad hoc cryptographic schemes due to their straightforwardness and flexibility to meet special needs. The particularity of our work is that we use advanced security notions, such as indistinguishability, in order to compromise a system like Android. Our attack demonstrates that any theoretical weakness concerning the security of a cryptographic scheme could be utilized to break the whole system. We thus show that the scope of these notions extends beyond theory. We advocate the shift onto *provably secure cryptography* in order to prevent potential vulnerabilities that will be hard to find inside a complex system.

1.2 Related Work

KeyStore Security. Encryption in mobile devices is increasingly becoming a topic of utmost importance. Teufl et al. have thoroughly analyzed the encryption components of Android in [24]. This concerns both full disk encryption and credential storage. Authors provide a descriptive study of the two systems. However, no cryptanalysis of the presented cryptographic schemes is given. Works in [9,17] highlight the severity of physical attacks, such as cold boot, against Android's disk encryption. The primary limitation of these attacks is that they require a physical access to the targeted mobile devices.

As for secure credential storage, authors in [25] show that app developers tend to implement their own mechanisms to store credentials. They underline the prevalence of flawed solutions by designing a tool capable of automatically identifying and retrieving app credentials. Developers are thus urged to use the security services proposed by the Android system itself, that is KeyStore. The different flavors, software-based and hardware-based, of the KeyStore are subjected to close scrutiny in [8]. The investigation involves how an adversary is able to compromise the different access controls to the stored keys. However, the study assumes that all the cryptographic algorithms were properly defined and implemented, which is proved not to be true in [10]. Hay et al. exploit a buffer overflow vulnerability that permits the execution of an arbitrary code inside the keystore process. To the best of our knowledge, we present the first cryptanalysis-based attack against the KeyStore. In addition, our attack has the advantages to be software-only and remotely executable.

Authenticated Encryption. Authenticated encryption is a symmetric encryption scheme that protects data confidentiality and integrity. Integrity (authenticity) means that no adversary is able to produce new valid ciphertexts. This entails that encrypted data cannot be undetectably modified. Recently, the design of AE primitives has renewed interest, not least because of the currently running CAESAR competition [7]. The security notions of AE were formalized in the early 2000s in [5,11]. Generic composition [6] is the most popular approach for numerous security protocols, such as SSH,TLS and IPsec. This approach is about combining a confidentiality-providing encryption scheme together with a message authentication code (MAC). Nevertheless, the pursuit of more efficiency than that offered by these two-pass schemes has motivated the construction of dedicated AE designs, such as the Galois Counter Mode (GCM) [14].

It turns out that designers do not only strive for efficiency, but also for implementation simplicity. Therefore, authenticity obtained from *Encryption-with-Redundancy* (EwR) has long been attractive. In such a paradigm, encryption consists of computing some public function h over the message M to get a checksum $\sigma = h(M)$. Then, $M||\sigma$ is encrypted and returned. As for decryption, the ciphertext is decrypted to get $M||\sigma$ and then the equality $\sigma = h(M)$ is verified. Several of these schemes have been partially or fully broken [15,16]. A generic attack attributed to Wagner on a large class of CBC-Encryption-with-Redundancy is described in [20]. An and Bellare in [1] formally prove that this AE scheme does not guarantee security regardless of how the checksum is computed.

Some might argue that Hash-then-Encrypt (HtE) is just a special case of EwR, where the checksum function h is a hash function. However, we argue that this is not true. Indeed, the checksum is appended at the end of the message $(M||\sigma)$ in EwR, while the hash value is appended at the beginning of the message $(\sigma||M)$ in HtE. Thus, generic attacks against EwR, Wagner's for instance, are easier to apply than those against HtE. This is due to the fact that the former typically requires to remove the last block of ciphertexts, and for many schemes, the decryption of the first blocks does not depend on the last ones (e.g. CBC and CTR). Moreover, the proof of Bellare only shows that EwR does not offer a sufficient condition for security even if the underlying base encryption is secure. A similar result related to MAC-then-Encrypt (MtE) is given in [6]. In order to avoid misinterpretation, we emphasize that these results only imply that such constructions are not *generically* secure: the soundness of the underlying primitives does not constitute a sufficient condition to guarantee security. Indeed, the proof consists of providing a counterexample, i.e. a particular MtE scheme that it is not IND-CCA although its encryption and MAC algorithms are secure. The proof is applicable only for special IND-CPA encryption schemes whose ciphertexts can be modified without changing their corresponding plaintexts, which is clearly not the case for CBC and CTR modes. We note that the results of [6] do not mean that all MtE schemes are inherently broken. A body of results (e.g. [18]) has proved the security of several schemes following this construction.

In this paper, we give the first proof that HtE for both CBC and CTR modes, indeed, does not guarantee integrity. In addition, the proof that we provide is not a mere existential forgery or a theoretical distinguishing attack. Unlike related work, we provide a practical attack that could be exploited to compromise the Android KeyStore. The threat is concrete: the broken HtE in CBC mode (Hash-then-CBC-Encrypt) is the cryptographic scheme that is used to safeguard the stored keys in Android mobile devices.

1.3 Responsible Disclosure

We communicated our findings to Google in January 2016. The Android security team has acknowledged the attack presented in this paper and confirmed that the broken encryption scheme is planned for removal.

1.4 Paper Outline

The rest of the paper is structured as follows: Sect. 2 reviews some classical definitions and notations. In Sect. 3, we provide two proofs that Hash-then-CBC-Encrypt does not provide integrity. Section 4 describes some technical details about the Android KeyStore. We present our attack scenario in Sect. 5. Section 6 provides some discussion and specific recommendations related to the identified vulnerability.

2 Definitions

A message is a string. A string is a member of $\{0,1\}^*$. The concatenation of strings X and Y is denoted $X||Y$ or simply XY. For a string X, its length is represented by $|X|$. A *block cipher* is a function $E : \texttt{Key} \times \{0,1\}^n \longrightarrow \{0,1\}^n$, where $\texttt{Key}$ is a finite nonempty set and $E_{\mathsf{k}}(.) = E(\mathsf{k}, .)$ is a permutation, hence invertible, on $\{0,1\}^n$. The number n is called the *block length*. We use the notation $\mathbf{A}^{\mathcal{O}}$ to denote the fact that the algorithm $\mathbf{A}$ can make queries to the function $\mathcal{O}$. Hereafter, we say that the *adversary* $\mathbf{A}$ has access to the *oracle* $\mathcal{O}$. If f is a randomized (resp., deterministic) algorithm, then $y \stackrel{R}{\leftarrow} f(x)$ (resp., $y \leftarrow f(x)$) denotes the process of running f on input x and assigning the result to y.

Symmetric Encryption Schemes. Following Bellare et al. in [4], a *symmetric encryption scheme* $\mathcal{SE}$ is given by three algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$, where (1) the *key generation algorithm*, $\mathcal{K}$, takes a security parameter $k \in \mathbb{N}$ and returns a key K. We write $\mathsf{K} \stackrel{R}{\longleftarrow} \mathcal{K}(k)$; (2) the *encryption algorithm*, $\mathcal{E}$, takes a key K and a plaintext M to produce a ciphertext C. We write $C \stackrel{R}{\longleftarrow} \mathcal{E}_{\mathsf{k}}(M)$; and (3) the *decryption algorithm*, $\mathcal{D}$, takes a key K and a ciphertext C to return either the corresponding plaintext M or a special symbol $\perp$ to indicate that the ciphertext is invalid. We require that $\mathcal{D}_{\mathsf{k}}(\mathcal{E}_{\mathsf{k}}(M)) = M$ for all M and K.

Secrecy of a Symmetric Encryption Scheme. The security of symmetric encryption schemes is usually classified from the point of view of their goals and attack models. The classical goal of secure encryption is to protect the confidentiality of messages, which could be defined by various concepts [23].

The most used one is *indistinguishability* (IND) that is formalized as follows [4]: given a symmetric encryption $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and a ciphertext of one of two plaintexts, no adversary can distinguish which one was encrypted. IND can be expressed as an experiment. Let $\mathcal{E}_{\mathsf{k}}(\mathcal{LR}(.,.,b))$ be a *left-or-right* oracle where $b \in \{0,1\}$: the oracle takes two messages of equal length as input, m_0 and m_1, and returns $C \leftarrow \mathcal{E}_{\mathsf{k}}(m_b)$. The adversary submits queries of the form (m_0, m_1), where $|m_0| = |m_1|$, to the oracle, and must guess which message was encrypted. If all adversaries cannot succeed with probability better than a random guess, then $\mathcal{SE}$ is called *IND-ATK secure*, where *ATK* represents the attack model.

The standard attack models are as follows: (1) The *chosen plaintext attack* (CPA) in which an adversary has access to the encryption oracle $\mathcal{E}_{\mathsf{k}}(\mathcal{LR}(.,.,b))$, so that she can choose a set of plaintexts and obtain the corresponding ciphertexts; (2) the *chosen ciphertext attack* (CCA) in which an adversary has access, besides the encryption oracle, to the decryption oracle $\mathcal{D}_{\mathsf{k}}(.)$, so that she can choose a set of ciphertexts and obtain their plaintexts.

Definition 1 *(Indistinguishability of a Symmetric Encryption Scheme).* Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme. Let A be a polynomial-time adversary. For $b \in \{0,1\}$ and $k \in \mathbb{N}$, consider the following experiments:

Experiment $Exp^{ind-cpa-b}_{\mathcal{SE},A_{\rm cpa}}(k)$	**Experiment** $Exp^{ind-cca-b}_{\mathcal{SE},A_{\rm cca}}(k)$
1: $\mathsf{K} \xleftarrow{R} \mathcal{K}(k)$	1: $\mathsf{K} \xleftarrow{R} \mathcal{K}(k)$
2: $x \longleftarrow A_{cpa}^{\mathcal{E}_{\mathsf{k}}(\mathcal{LR}(.,.,b))}$	2: $x \longleftarrow A_{cca}^{\mathcal{E}_{\mathsf{k}}(\mathcal{LR}(.,.,b)),\,\mathcal{D}_{\mathsf{k}}}$
3: **return** x	3: **return** x

The adversary A is prohibited from querying $\mathcal{D}_{\mathsf{k}}(.)$ on a ciphertext C output by the encryption oracle. For $atk \in \{cpa, cca\}$, the advantage of the adversary is defined as follows:

$$Adv^{ind-atk}_{\mathcal{SE}}(k) = Pr[Exp^{ind-atk-1}_{\mathcal{SE},\,A} = 1] - Pr[Exp^{ind-atk-0}_{\mathcal{SE},\,A} = 1]$$

The scheme $\mathcal{SE}$ is *secure* if the advantage of any adversary is negligible.

The Cipher Block Chaining (CBC) Mode. Encryption with a raw block cipher is not used in practice. Instead, several modes of operation exist. Here, we only consider the CBC mode.

Definition 2 *(The CBC Encryption Scheme).* Let $E_{\mathsf{k}} : \mathtt{Key} \times \{0,1\}^l \longrightarrow \{0,1\}^l$ be a block cipher and let E_{k}^{-1} be its inverse. Let $\text{CBC}[E_{\mathsf{k}}] = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be its associated CBC encryption scheme. Given a message $M = m_1||...||m_n \in \{0,1\}^{ln}$, the encryption and the decryption algorithms are defined as follows:

CBC Encryption $\mathcal{E}_{\mathsf{k}}^{\text{CBC}}(M)$	**CBC Decryption** $\mathcal{D}_{\mathsf{k}}^{\text{CBC}}(C)$
1: Parse M as $m_1\|\|...\|\|m_n$	1: Parse C as $c_0\|\|c_1\|\|...\|\|c_n$
2: $c_0 \xleftarrow{R} \{0,1\}^l$	2: **for** $i = 1...n$ **do**
3: **for** $i = 1...n$ **do**	3: $\quad m_i \longleftarrow E_{\mathsf{k}}^{-1}(c_i) \oplus c_{i-1}$
4: $\quad c_i \longleftarrow E_{\mathsf{k}}(c_{i-1} \oplus m_i)$	4: **end for**
5: **end for**	
6: **return** $c_0\|\|c_1\|\|...\|\|c_n$	5: **return** $m_1\|\|...\|\|m_n$

Two points should be noted in the definition. First, the random IV is denoted $\mathbf{c_0}$ in order to highlight that the IV is included along with the ciphertext. Second, we make the simplifying assumption that $\mathcal{D}_{\mathsf{k}}^{\text{CBC}}(.)$ never returns the error message $\perp$. It takes any ciphertext as input, and always returns some string.

3 Hash-Then-CBC-Encrypt Does Not Provide Integrity

In this section, we start by reviewing the different concepts of integrity which our proof relies on. We then provide a formal definition of Hash-then-CBC-Encrypt. We end by proving that this scheme is not secure.

3.1 Integrity of a Symmetric Encryption Scheme

In the context of symmetric encryption, integrity (or authenticity) means that only valid parties possessing the secret key K are able to produce a valid ciphertext; i.e. whose decryption does not give $\perp$. Symmetric encryption schemes in

general do not protect the integrity of messages. For example, the CBC mode does not provide integrity, since it never returns $\perp$. The IND-CPA secure schemes that also provide integrity are called authenticated encryption schemes.

Throughout this paper, we consider two notions of integrity: integrity of ciphertext (INT-CTXT) [6] and ciphertext unforgeability (CUF-CPA) [12]. Both notions require that no adversary be able to produce a valid ciphertext which the encryption oracle had never produced before. However, contrary to INT-CTXT, the adversary in CUF-CPA has no access to the decryption oracle and outputs only one attempted forgery. Despite of their similarity, these two notions are defined to accomplish different goals. Indeed, INT-CTXT is a strong measure for security, while CUF-CPA is a strong one for the effectiveness of the potential attacks. Thus, proving that a symmetric scheme does not achieve neither INT-CTXT nor CUF-CPA entails two consequences: (1) the scheme does not provide high security and therefore it should not be used by scheme designers; and (2) the found attack is very damaging due to its readily implementation in practice.

Definition 3 *(Integrity of an Authenticated Encryption Scheme).* Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme. Let A be a polynomial-time adversary. Let S be the list of all ciphertexts generated by the adversary queries to $\mathcal{E}_{\mathsf{k}}(.)$. For $k \in \mathbb{N}$, the following experiments are defined:

Experiment $Exp^{\text{int-ctxt}}_{\mathcal{SE},A_{\text{ctxt}}}(k)$

1: $\mathsf{K} \xleftarrow{R} \mathcal{K}(k)$
2: **if** $C \leftarrow A^{\mathcal{E}_{\mathsf{k}}(.),\,\mathcal{D}_{\mathsf{k}}(.)}_{ctxt}$ such that $\mathcal{D}_{\mathsf{k}}(C) \neq \perp$ and $C \notin \mathsf{S}$ **then**
3: **return** 1
4: **else**
5: **return** 0
6: **end if**

Experiment $Exp^{\text{cuf-cpa}}_{\mathcal{SE},A_{\text{-cpa}}}(k)$

1: $\mathsf{K} \xleftarrow{R} \mathcal{K}(k)$
2: $C \longleftarrow A^{\mathcal{E}_{\mathsf{k}}(.)}_{\text{cuf-cpa}}$
3: **if** $\mathcal{D}_{\mathsf{k}}(C) \neq \perp$ and $C \notin \mathsf{S}$ **then**
4: **return** 1
5: **else**
6: **return** 0
7: **end if**

For both experiments, the adversary's advantage is defined to be:

$$Adv^{int}_{\mathcal{SE},\,A}(k) = Pr[Exp^{int}_{\mathcal{SE},\,A} = 1]$$

The scheme $\mathcal{SE}$ is *INT-CTXT secure* (or *CUF-CPA secure*) if the corresponding advantage is negligible for any adversary.

3.2 Hash-then-CBC-Encrypt

Conceptually, Hash-then-CBC-Encrypt in its general setting is an authenticated encryption scheme obtained from the association of any given hash function with any given CBC encryption algorithm.

Construction 1 *(Hash-then-CBC-Encrypt (hCBC)).* Let $\text{CBC}[E_{\mathsf{k}}] = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an IND-CPA CBC encryption scheme, where E_{k} is a block cipher of block length l. Let h be a hash function. Without loss of generality, we suppose that the

output length of h is l bits (otherwise, padding is needed). For $M \in \{0,1\}^{ln}$, we define the composite Hash-then-CBC-Encrypt $h\text{CBC} = (h, \mathcal{K}, \mathcal{E}', \mathcal{D}')$ as follows:

Encryption $\mathcal{E}'_{\mathsf{k}}(M)$

1: $\sigma \longleftarrow h(M)$
2: $C \longleftarrow \mathcal{E}^{\text{CBC}}_{\mathsf{k}}(\sigma || M)$
3: **return** C

Decryption $\mathcal{D}'_{\mathsf{k}}(C)$

1: Parse $\mathcal{D}^{\text{CBC}}_{\mathsf{k}}(C)$ as $\sigma' || M$
2: **if** $\sigma' \neq h(M)$ **then**
3: **return** $\perp$
4: **end if**
5: **return** M

3.3 Hash-then-CBC-Encrypt is not INT-CTXT

Here, we provide an indirect proof that hCBC is not secure against INT-CTXT. For this, we use the relations among notions that are defined in [6]. In particular, we use a derived one: if an AE scheme is IND-CPA and not IND-CCA, then it is not INT-CTXT (**IND-CPA** $\wedge$ **¬IND-CCA** $\Rightarrow$ **¬INT-CTXT**), which is easily obtained from **IND-CPA** $\wedge$ **INT-CTXT** $\Rightarrow$ **IND-CCA**. Therefore, our proof is composed of two parts: firstly we prove that hCBC is IND-CPA and secondly we prove that it is not IND-CCA.

Proposition 1 *Hash-then-CBC-Encrypt is IND-CPA secure.*

The proof is based on a standard reduction argument, and the understanding of the rest of the paper does not depend on it. We leave it for [22].

Proposition 2 *Hash-then-CBC-Encrypt is not IND-CCA secure.*

Proof Let A be an IND-CCA adversary for $h\text{CBC} = (h, \mathcal{K}, \mathcal{E}, \mathcal{D})$. Its algorithm is shown below.

Algorithm. $A_{cca}^{\mathcal{E}_{\mathsf{k}}(\mathcal{LR}(.,.,b)),\, \mathcal{D}_{\mathsf{k}}}$

1: Let m_0 and m_1 be two messages
2: $m'_0 \longleftarrow h(m_0) || m_0$
3: $m'_1 \longleftarrow h(m_0) || m_1$
4: $C \longleftarrow \mathcal{E}_{\mathsf{k}}(\mathcal{LR}(m'_0, m'_1, b))$
5: Parse C as $c_0 || c_1 || c_2 || c_3$
6: $C' \longleftarrow c_1 || c_2 || c_3$
7: $x \longleftarrow \mathcal{D}_{\mathsf{k}}(C')$
8: **if** x $\neq \perp$ **then**
9: **return** 0
10: **else**
11: **return** 1
12: **end if**

We claim that the previous adversary succeeds whether $b = 0$ or $b = 1$. Therefore, $Adv^{ind-cca}_{h\text{CBC}}(A) = 1$, and as a result, hCBC is not CCA-secure. Recall that the oracle $\mathcal{E}_{\mathsf{k}}(\mathcal{LR}(.,.,b))$ returns the ciphertext of one of the two submitted messages. Thus, we have $C = \mathcal{E}_{\mathsf{k}}(m'_b = h(m_0)||m_b)$. Applying hCBC, C can be written as $\mathcal{E}^{\text{CBC}}_{\mathsf{k}}(h(h(m_0)||m_b) \,||\, h(m_0) \,||\, m_b)$, which is composed as follows:

$$C = c_0 \;||\; \overbrace{E_{\mathsf{k}}(c_0 \oplus h(h(m_0)||m_b))}^{c_1} \;||\; \overbrace{E_{\mathsf{k}}(c_1 \oplus h(m_0))}^{c_2} \;||\; \overbrace{E_{\mathsf{k}}(c_2 \oplus m_b)}^{c_3}$$

We see that for C', c_0 is removed and c_1 becomes the new initial value. Considering the new IV, the CBC decryption algorithm performed over C' returns the rest of the plaintext $h(m_0)||m_b$. Therefore, $\mathcal{D}_{\mathsf{k}}(C')$ outputs m_0 when $b = 0$, $\perp$ otherwise (unless $h(m_0) = h(m_1)$), which concludes our proof.

3.4 Hash-then-CBC-Encrypt is not CUF-CPA

As a matter of fact, we have already proved that hCBC is not CUF-CPA. Indeed, following [19], if a scheme is not INT-CTXT, then consequently, it is not CUF-CPA. Nevertheless, our goal here is to explicitly provide a selective forgery upon which our attack scenario against the KeyStore is built. We note that the presented attack is quite powerful: the adversary succeeds in forging a valid ciphertext for any message M after only one query to the encryption oracle.

Proof Let A be a CUF-CPA adversary for $h\text{CBC} = (h, \mathcal{K}, \mathcal{E}, \mathcal{D})$. We will show that A can forge a valid ciphertext for any $M \in \{0,1\}^{ln}$.

Algorithm. $A^{\mathcal{E}_{\mathsf{k}}}_{\text{cuf-cpa}}(M)$

1: $M' \longleftarrow h(M)||M$
2: $C \longleftarrow \mathcal{E}_{\mathsf{k}}(M')$
3: Parse C as $c_0||c_1||c_2||...||c_{n+2}$
4: $C' \longleftarrow c_1||c_2||...||c_{n+2}$
5: **return** C'

As mentioned in Definition 3, the adversary A wins if the output ciphertext C' is both new and valid. Trivially, C' has never been produced by the encryption oracle $\mathcal{E}_{\mathsf{k}}(.)$ before, and thus it is new. In addition, we argue that the oracle $\mathcal{D}_{\mathsf{k}}(.)$ on C' will not return $\perp$. Indeed, using the same arguments given in Proposition 2, C' could be written as $\mathcal{E}^{\text{CBC}}_{\mathsf{k}}(h(M)||M)$. Thus, $\mathcal{D}_{\mathsf{k}}(C') = M (\neq \perp)$.

4 The Android KeyStore

The Android KeyStore is a high-level service that enables applications to store their credentials. The original credential store was created in Android 1.6 and was limited to store VPN and Wi-Fi EAP credentials. Back then, only the operating system, and not user applications, could access the stored keys and certificates. It is worth mentioning that hereafter all the implementation details that we provide concern the KeyStore of the build *android-6.0.1_r22*, which is the latest version of Android at the writing of this paper.

As illustrated in Fig. 1, the KeyStore is comprised of three layers: *Public APIs*, *Keystore service*, and *Keymaster*. The security of keys is primarily ensured by the *Keymaster* which is designed to protect keys from extraction. This implies that it is the only component that has a direct access to keys material, and therefore keys are represented differently outside Keymaster: alias (name) in *Public APIs* and key handlers in *Keystore service*.

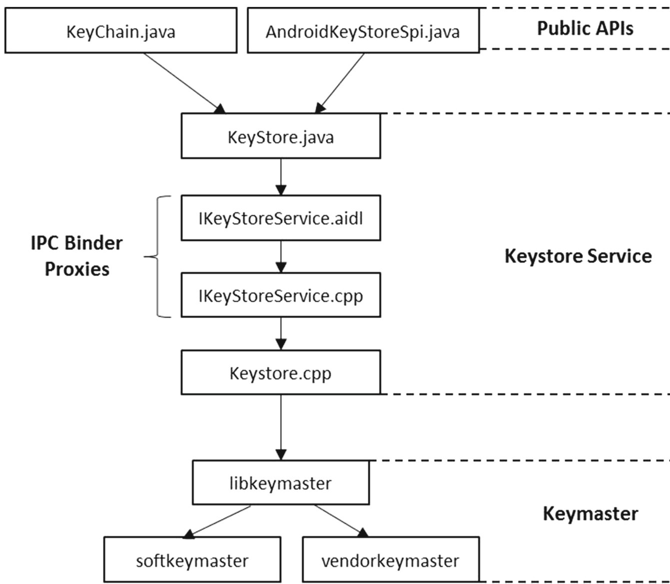

Fig. 1. Android KeyStore Architecture

Generally speaking, the key handler is an opaque object that identifies a keymaster-protected key. Key handlers are implementation-dependent. We only consider the default software-only keymaster provided by Google. By inspecting its implementation that is found in `keymaster_openssl.cpp`, we see that the key handler is just an encoded version of the corresponding key. Encoding is achieved by concatenating a header of describing meta data to the key. The header includes: a 4-byte constant value for software keys, a 4-byte key type, and a 4-byte big endian integer for key length. Thus, the default key handler is written as follows: `Soft_Key_Magic` || `Key_Type` || `Key_Length` || `Key`.

Our target in this paper is the stored keys on mobile device. Therefore, in what follows, we focus solely on the secure mechanism performed by the *Keystore service* for storing keys (or more precisely key handlers).

4.1 Keystore Service

Similar to other services, the Keystore service spans two layers in the Android architecture: the Java world (application framework) and the native world (system service). Based on the Binder, its different components, *KeyStore.java* and *Keystore.cpp*, communicate via the Binder proxy *IKeyStoreService*.

The implementation [2] of the Keystore reveals how the blobs of key handlers are stored on mobile device. A key handler blob (binary large object) contains a serialized version of the key handler. The keystore saves its files in */data/misc/keystore*, where there is one directory for each user. Each directory includes files that have the following content:

- A single master key. The Keystore service is initialized by generating a 128-bit master key using the internal entropy source */dev/urandom*. The master key is then encrypted by a 128-bit AES key derived from the screen passcode. The encrypted keymaster is stored in the *.masterkey* file.
- Key handler blobs related to user's applications. Each file contains a header of meta data as well as the encryption of the key handler using Hash-then-CBC-Encrypt. The content of the file is written as follows:

$$\texttt{meta data} \,||\, \mathcal{E}^{\text{CBC[AES]}}_{master_key}(\texttt{MD5(key handler)} \,||\, \texttt{key handler})$$

We note that the KeyStore applies $h\text{CBC} = (\texttt{MD5}, \mathcal{K}, \mathcal{E}^{\text{CBC}}_{\text{AES}}, \mathcal{D}^{\text{CBC}}_{\text{AES}})$ to protect key handlers. Therefore, the adversary defined in Sect. 3.4 is able to maliciously forge new key handlers given valid ones. However, this attack fails in practice when performed against the Keystore service because the produced key handlers would yield errors while being decoded. We recall that key handlers have a special encoding format that is specified by the keymaster. In the next Section, we adapt our forgery attack so that an adversary could fabricate a valid key handler which the keymaster successfully parses to its related key.

5 Attacking the Android KeyStore

5.1 Technical Background

As mentioned previously, our target is the secure storage of keys. As a result, among all other operations provided by the KeyStore, only those involving the encryption of the stored keys will be relevant to us. This includes two operations: *key generation* and *key import*.

The KeyStore is designed to work not only with its own keys, but with those generated by a third party system. This implies that all keys, generated or imported, must follow a special format when being serialized. For instance, the keymaster requires formatting keys before wrapping them inside key handlers. The file `keymaster_defs.h` shows that there are three categories of formats:

```
typedef enum {
     KM_KEY_FORMAT_X509 = 0,  /* for public key export */
     KM_KEY_FORMAT_PKCS8 = 1, /* for asymmetric key pair import */
     KM_KEY_FORMAT_RAW  = 3,  /* for symmetric key import */
} keymaster_key_format_t;
```

We notice that standard formats (i.e. X.509 and PKCS#8) are used for key-pairs, while no format is provided for symmetric keys. Thus, the exact bytes comprising a symmetric key are encapsulated inside the stored key handler. This is due to the fact that their support is quite recent. Indeed, until lately, the KeyStore was limited to asymmetric key-pairs (e.g. RSA, DSA and EC).

This lack of formatting makes the adversary task easier. Indeed, it is hard to fabricate a ciphertext that is both valid and properly formatted. Consequently, the current version of our attack is limited to applications using symmetric keys.

5.2 Threat Model

The adversary's goal is to undetectably undermine the security of the applications relying on symmetric keys for their security. For this purpose, we assume that the adversary installs some malware on the mobile device. This malware is capable of importing keys inside the KeyStore, since any installed application does have this capability. In addition, the malware is supposed to be granted the read-write permission on the KeyStore directory (i.e. */data/misc/keystore*).

Furthermore, the malware is executed inside a mobile device with protective tools. First, the mobile system detects any malware trying to connect to a remote server. Second, the mobile system imposes the use of a strong screen passcode. This helps to avoid exhaustive attacks, since the master key of the KeyStore is derived from this passcode. Third, the system prohibits the KeyStore from storing short or obviously non-random keys. Thus, the adversary cannot perform the trivial attack consisting of generating the same key for all applications or generating a different key for each application and communicating it to a server. In both cases, the attack would be detected. We insist that these assumptions are highly plausible in corporate environments where companies enforce the security of their employees mobile devices.

Finally, the adversary controls all communications with the mobile, and thus can intercept and tamper with any exchanged message. Besides, it is assumed that any proved cryptographic mechanism is secure unless weak keys are used.

To sum up, in order to succeed her attack, the adversary should silently "break into" the KeyStore to shorten, and hence weaken, the stored keys which the targeted applications would blindly continue using.

5.3 The Forgery Attack

The purpose of the forgery attack is that given a ciphertext of a symmetric key, the adversary can fabricate another ciphertext that decrypts to a shorter key. As already stated, the KeyStore protects keys by encrypting their key handlers with hCBC. Thus, keys protection, involving their confidentiality and integrity, is done using a variant of hCBC which we call *encode-then-hCBC* (ehCBC).

Informally, ehCBC is an AE scheme where messages are encoded before hCBC-encrypting them. To be more precise, let $eh\text{CBC} = (\mathcal{K}', \mathcal{E}', \mathcal{D}')$ be an encoded version of $h\text{CBC} = (h, \mathcal{K}, \mathcal{E}, \mathcal{D})$. Then, for all message M, the next relation holds: $\mathcal{E}'(M) = \mathcal{E}(\texttt{Length}(M)||M)$. In what follows, we adapt the CUF-CPA adversary of hCBC in order to compromise ehCBC.

Let M be an arbitrary weak symmetric key, and let $\mathbf{A}$ be an attacker that can import keys of its choice to the KeyStore. For the sake of clarity, we omit the constant values in the header of the key handler, and so only the `key_length` is kept. Therefore, the `import` function corresponds to the ehCBC-encryption operation ($\mathcal{E}'_{\mathsf{k}}$). It is worth mentioning that this simplifying assumption does not alter the logic of the attack. $\mathbf{A}$ wins if it can produce a valid ehCBC-ciphertext of M. However, conforming to our threat model (Sect. 5.2), the attacker cannot import M directly. To this end, $\mathbf{A}$ executes the algorithm below:

Algorithm. $A^{\texttt{import}}_{\text{malicious}}(M)$

1: $M' \longleftarrow \texttt{Len}(M) \,||\, M$
2: $M'' \longleftarrow \texttt{MD5}(M') \,||\, M'$
3: $C \longleftarrow \mathcal{E}'_{\mathsf{k}}(\texttt{padding} \,||\, M'')$
so that $\texttt{Len}(.)||\texttt{padding}$ is l-block
4: Parse C as $c_0||c_1||c_2||C'$
5: $C'' \longleftarrow c_2 \,||\, C'$
6: **return** C''

Following the same arguments provided in Sects. 3.3 and 3.4, we can see that $\mathcal{D}^{eh\text{CBC}}_{\mathsf{k}}(C'')$ outputs M, which means that $\mathbf{A}$ achieves its goal. Though, it is important to notice that the attacker owes part of its success to the absence of verification of sound key lengths. Indeed, considering all the technical details that we provided, the length in bytes of the imported key ($\texttt{padding}||M''$) is always greater than 32, since it is constructed of at least two AES blocks (i.e. MD5(.) and $\texttt{Len}(.)||\texttt{padding}$). For instance, if $\mathbf{A}$ selects 4-byte M (or key), it calls the `import` function on a key of length 36 bytes. We recall that AES keys cannot be longer than 32 bytes. Fortunately (for the attacker), no checking is done by `import`, and consequently the attack ends successfully.

We underline that the interest of the above attack is twofold. First, it can be abused by some malware to breach the KeyStore security even in a well-protected mobile system. Second, we prove that encoding does not improve the security of hCBC unlike for many other AE schemes. We believe that this result is of independent importance regardless of the introduced attack scenario.

5.4 The Undetected Malware

We illustrate the fallout of our forgery against the KeyStore by a complete attack scenario. We emphasize that the severity of protecting highly sensitive data, like keys, by a broken cryptographic scheme is not limited to the suggested scenario.

In our scenario, the intent of the attacker is to maliciously modify all the exchanged messages between an app and a remote server even if they are protected by proved cryptography. This is possible thanks to some malware installed on the mobile and which soundlessly weakens the keys of the KeyStore. This attacker represents a new kind of threat, since she can go undetected while compromising the security of users including those hiding behind secure protocols.

Actors. We define five actors to describe the plot of the attack: (1) a *security manager* who enforces the security of the mobile system. In particular, the KeyStore refuses to store weak (i.e. short) keys. Additionally, the system would detect any malware trying to communicate with its accomplice server; (2) a *victim* who uses the said mobile to perform some services requiring to protect their critical transactions. The corresponding cryptographic keys are managed by the KeyStore; (3) a *remote server* related to the running services and to which the critical transactions are sent; (4) a *malicious application* viciously shortening the keys of other applications; and (5) a *colluding party* that is able to intercept and alter any exchanged message on the network.

Attack Workflow. We suppose that the attacker has already convinced the victim in some way to install the malicious application on her device. The attack scenario is structured into three phases: provisioning, lulling and attacking.

Provisioning phase. The malicious application runs in background and executes the algorithm described in Sect. 5.3. Thus, it craftily generates several symmetric keys of length $32 + x$ bytes. Then, it imports these keys into the KeyStore which accepts them for two reasons: they are seemingly strong and no verification is done concerning their abnormal length. Afterward, it cuts them down into keys of length x bytes. Here, we take x to be 4, so that keys are small enough to allow a swift brute-force attack. For the sake of completeness, we precise that once the keys are trimmed, their meta data are required to be padded with some dummy data. This is because the files containing the keys must remain of constant size. For brevity, we omit the technical details related to this balancing operation.

Lulling phase. In this phase, an application on the victim's device asks the KeyStore to generate a key with *alias* as its name. The malicious application, snooping on the KeyStore, notes this alias as well as the UID of the caller application. As soon as the key is generated and its associated file is created, the malicious application modifies the name of one of its keys in such a way that the renamed key is believed to belong to the targeted application. Some might argue that this operation is delicate, since the malicious application is assumed to continuously supervise the KeyStore activities. Nevertheless, we argue that no special privilege is required. Indeed, it can be done with quite ease by monitoring the content of the KeyStore folder. This is due to the fact that the key's alias and the creating application's UID could be guessed from the key file name.

Attacking phase. Now, the user is carrying out some operations that involve transmitting sensitive messages to a server. The application handling such operations needs to protect the integrity of these messages. Therefore, it asks the KeyStore to generate an HMAC tag over each message. The KeyStore returns a tag unwittingly generated with the weak key. Concatenated to their tag, the messages are then intercepted by the colluding party while being sent to the server. The latter performs an exhaustive search to find the secret key used to generate the HMAC tag. Since the search space being explored is shrunk, the brute-force search ends quite fast. The colluding party then modifies the content of some messages (e.g. the total amount of a payment transaction), and recomputes a valid tag for them before forwarding the new messages to the server. In this way, the attacker effortlessly breaks into victims who think that they are safe with primitives, HMAC for example, which are believed to be secure.

5.5 The Hidden Assumption

The malicious application is supposed to have read/write permissions to the folder `/data/misc/keystore`. Nevertheless, in practice, the Android system restricts access to this folder: only the *keystore* user is allowed to see or modify its contents. Thus, the success of our attack depends on how likely the

malicious application is to bypass the access control mechanisms of Android. This requires one of these two extra abilities: (1) executing an arbitrary code inside the keystore process by either code injection or code reuse; and (2) obtaining root or kernel-level privileges. Some might argue that once such abilities have been gained the presented attack in Sect. 5.4 could be realized otherwise. Here, we present three possible scenarios and we discuss how our attack is more effective.

The Trivial Scenario. With a root privilege, we need not bother mutating key blobs. Instead, we can simply recover the master key from the keystore memory in order to decrypt/re-encrypt any keystore file. This scenario is not as straightforward as it seems to be. Indeed, it involves a program to parse the memory. The problem is that the keystore has been regularly updated recently, so its memory layout has been continuously changing. Therefore, this program may require to be different depending on the installed Android version. In addition, it should be constantly maintained to keep on with any further update. We note that the format of the keystore files has not changed since Android 4.3. Involving only basic I/O file operations, our attack is much simpler and more portable.

The Big-Brother Function. The malicious application and her colluding party agree on a function $\mathcal{B}$ to generate keys that could be quickly guessed. Unable to communicate, otherwise the subversion will be detected, the function $\mathcal{B}$ is embedded into the malicious application. It is easy to see that this attack ends successfully following our threat model. However, we claim that our attack is more practical because it satisfies two additional properties: (1) *stateless*: the adversary (i.e. colluding party) needs not to store data related to the victim (i.e. the mobile device) so as to win; and (2) *size-oblivious*: the complexity of the attack does not increase with the number of the targeted users. In contrast, the other attack cannot be both stateless and size-oblivious. Indeed, the function $\mathcal{B}$ outputs a new key for each device. Keys shall seem to be strong, otherwise they will be rejected. The more the attacker targets new devices, the bigger the keys search space becomes. Avoiding this increase in time of execution involves the parameterization of the function $\mathcal{B}$ for each user. For instance, $\mathcal{B}$ might be seeded with the device IMEI (International Mobile station Equipment Identity). Hence, the attack becomes size-oblivious, but stateful. Statelessness is important in our context due to its relevance to stronger undetectability.

Man in the KeyStore. The scenario supposes that all calls to the KeyStore are intercepted at runtime by the malicious application. Subverted values are returned for any intercepted call, including all cryptographic operations. Surely, this attack is powerful, but we argue that it is more limited than ours. Firstly, actively proxying all calls might be resource-consuming, i.e. slowing down the mobile or shortening its battery life, which makes the attack quite detectable. Secondly, the Keystore service is based on the Binder architecture, and thus

intercepting calls requires an attack of type *Man in the Binder* (MitB). However, a success MitB [3] necessitates deep insight on how Binder works, consequently it is version-dependent and more complicated than just reading/writing files.

6 Discussion and Recommendations

An important aspect of any forgery is what it implies in practice. Here, we have demonstrated how a theoretical weakness could be exploited to undermine the security of a real-world system, namely Android. In addition, the defined attack is attractive to implement, since it is simple and not demanding in term of resources. We insist that this scenario is just an example: a wholly new class of threat could be built from our forgery attack.

Furthermore, it is worth noting that the attack of Sect. 5 is conceived to be applicable only against software-only implementations of KeyStore. We admit that it does not directly impact hardware-based implementations which exist on some mobile devices. Indeed, our scenario involves forging keys by forging key handlers. Hardware-backed implementations, such as those based on Trusted Execution Environment (TEE) [21], encrypt their keys with AE schemes to produce their key handlers. Therefore, the integrity of keys is protected by two means: the Keystore service and the TEE. In our scenario, an attacker can still forge a valid key handler that is sent to the Keymaster (i.e., TEE). The TEE in its turn will detect the forgery when it decodes the forged key handler, which means that the attack does not succeed. However, we can imagine other possible vectors of attack. For example, an attacker might perform a fuzzy attack by generating valid key handlers and send them to the TEE. A malformed key handler might allow the attacker to carry out, for instance, a stack overflow attack.

Finally, we believe that even if some may argue that our attack is difficult to mount, there is value in identifying these types of design flaws. Corporate-issued devices or state-level malware could easily execute the described attack in order to gain undetectable long-term access to device communications.

Recommendations. Having thus presented our main results, we are now on a position to make specific recommendations. We recall that any countermeasure intended to fix a deployed system must not cause intrusive changes that affect the entire architecture of this system. Fortunately, the KeyStore design is modular enough to allow modifying the scheme hCBC without involving the rest.

The quickest solution would be to keep the *hash-then-encrypt* paradigm and use it with another encryption mode. The Counter (CTR) mode is often perceived as being advantageous to other modes. However, we prove that the scheme *Hash-then-CTR-Encrypt* does not provide integrity either. The full proof is given in [22]. We could have proposed other encryption modes, however the lack of obvious attacks cannot be taken as evidence of the soundness of a scheme. Instead, it would be better to switch to proved AE encryption schemes. At first glance, the simplest solution to make would seem to be Encrypt-then-MAC (EtM). Unfortunately, the 'generic composition' approach does not suit systems like Android. In fact, efficiency is important for mobile devices. EtM might incur

some overhead while computing ciphertexts. Moreover, it might be hard to implement because of manually managing two different cryptographic primitives.

Thus, we might believe that mobile designers should just go and pick up one of the AE one-pass dedicated schemes. It turns out that choosing a proper scheme is a great hassle for system designers. Let us discuss two popular ones:OCB (Offset Codebook Mode) [13] and GCM (Galois Counter Mode) [14]. OCB is a fast, secure and easy to implement AE encryption scheme. However, Rogaway, its inventer, holds a patent on it, and therefore it is not free to use. As for GCM, it is also fast and secure, but it involves hard mathematical concepts. As a result, most system designers feel unable to go through and implement GCM. We suspect that the absence of trusted implementations while defining the first KeyStore architecture might have been the reason of using hCBC. Today, GCM is being increasingly supported by free libraries, such as OpenSSL. Hence, we recommend to replace hCBC by GCM in the Android KeyStore.

It is worth reiterating that proved cryptography is the way to go. A key lesson from this paper is that cryptographers and system designers must work closely together. Bridging the gap that separates these communities will be essential for keeping future systems secure.

Acknowledgments. We would like to thank Mohammed Achemlal, Marc Girault and Olivier Sanders for valuable discussions.

References

1. An, J.H., Bellare, M.: Does encryption with redundancy provide authenticity? In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 512–528. Springer, Heidelberg (2001)
2. Android: Keystore implementation. https://android.googlesource.com/platform/system/security/+/master/keystore/keystore.cpp
3. Artenstein, N., Revivo, I.: Man in the Binder: He who controls IPC, controls the droid (2014). www.blackhat.com/docs/eu-14/materials/eu-14-Artenstein-Man-In-The-Binder-He-Who-Controls-IPC-Controls-The-Droid-wp.pdf
4. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A concrete security treatment of symmetric encryption. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, pp. 394–405. IEEE (1997)
5. Bellare, M., Namprempre, C.: Authenticated encryption: relations among notions and analysis of the generic composition paradigm. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, p. 531. Springer, Heidelberg (2000)
6. Bellare, M., Namprempre, C.: Authenticated encryption: relations among notions and analysis of the generic composition paradigm. J. Cryptology **21**(4), 469–491 (2008)
7. Bernstein, D.J.: CAESAR: Competition for Authenticated Encryption, December 2015. http://competitions.cr.yp.to/caesar.html
8. Cooijmans, T., de Ruiter, J., Poll, E.: Analysis of secure key storage solutions on android. In: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM 2014, pp. 11–20. ACM (2014)

9. Götzfried, J., Müller, T.: Analysing android's full disk encryption feature. J. Wireless Mobile Netw. Ubiquitous Comput. Dependable Appl. (JoWUA) **5**(1), 84–100 (2014)
10. Hay, R., Dayan, A.: Android keystore stack buffer overflow - CVE-2014-3100 (2014)
11. Katz, J., Yung, M.: Unforgeable encryption and chosen ciphertext secure modes of operation. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, p. 284. Springer, Heidelberg (2001)
12. Krawczyk, H.: The order of encryption and authentication for protecting communications (or: how secure is SSL?). In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, p. 310. Springer, Heidelberg (2001)
13. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 306–327. Springer, Heidelberg (2011)
14. McGrew, D.A., Viega, J.: Flexible and efficient message authentication in hardware and software. Manuscript (2003)
15. Mitchell, C.J.: Analysing the IOBC authenticated encryption mode. In: Boyd, C., Simpson, L. (eds.) ACISP. LNCS, vol. 7959, pp. 1–12. Springer, Heidelberg (2013)
16. Mitchell, C.J.: Cryptanalysis of two variants of PCBC mode when used for message integrity. In: Boyd, C., González Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 560–571. Springer, Heidelberg (2005)
17. Müller, T., Spreitzenbarth, M.: FROST: forensic recovery of scrambled telephones. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 373–388. Springer, Heidelberg (2013)
18. Namprempre, C., Rogaway, P., Shrimpton, T.: Reconsidering generic composition. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 257–274. Springer, Heidelberg (2014)
19. Paterson, K.G., Watson, G.J.: Authenticated-encryption with padding: a formal security treatment. In: Naccache, D. (ed.) Cryphtography and Security: From Theory to Applications. LNCS, vol. 6805, pp. 83–107. Springer, Heidelberg (2012)
20. Preneel, B.: Cryptographic primitives for information authentication - state of the art. In: Preneel, B., Rijmen, V. (eds.) State of the Art in Applied Cryptography. LNCS, vol. 1528, p. 49. Springer, Heidelberg (1998)
21. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In: Trustcom/BigDataSE/ISPA, vol. 1, pp. 57–64 (2015)
22. Sabt, M., Traoré, J.: Breaking into the keystore: a practical forgery attack against android keystore. Cryptology ePrint Archive, Report 2016/677 (2016)
23. Shafi, G., Micali, S.: Probabilistic encryption. J. Comput. Syst. Sci. **28**(2), 270–299 (1984)
24. Teufl, P., Fitzek, A.G., Hein, D., Marsalek, A., Oprisnik, A., Zefferer, T.: Android encryption systems. In: Privacy & Security in Mobile Systems (2014)
25. Zhou, Y., Wu, L., Wang, Z., Jiang, X.: Harvesting developer credentials in android apps. In: Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec 2015, pp. 23:1–23:12. ACM, New York (2015)