The Not-So-Silent Type: Vulnerabilities in Chinese IME Keyboards' Network Security Protocols

Jeffrey Knockel
jeff@citizenlab.ca
Citizen Lab, University of Toronto
Toronto, Ontario, Canada

Mona Wang
monaw@princeton.edu
Princeton University
Princeton, New Jersey, USA

Zoë Reichert
zoe.reichert@citizenlab.ca
Citizen Lab, University of Toronto
Toronto, Ontario, Canada

Abstract

Popular Chinese Input Method Editor (IME) keyboards almost universally feature "cloud-based" features that improve character prediction when typing. Handling such sensitive data (i.e., keystrokes) in transit demands security in transit. In this work, we perform a comprehensive security measurement of the Chinese IME keyboard ecosystem, investigating the network security of keystrokes sent in transit by popular Chinese IME keyboards from nine vendors. We studied the three most popular third-party keyboards, comprising 95.9% of the third-party keyboard market share in China, as well as the default Chinese IME keyboards pre-installed on six popular Android mobile device manufacturers in China. We found that the vast majority of IME keyboards utilize proprietary, non-TLS network encryption protocols. Our measurement revealed critical vulnerabilities in these encryption protocols from eight out of the nine vendors in which network attackers could completely reveal the contents of users' keystrokes in transit. We estimate that up to one billion users were affected by these vulnerabilities. Finally, we provide recommendations to various stakeholders to limit the harm from this existing set of vulnerabilities, as well as to prevent future vulnerabilities of this kind.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

cryptography; IMEs; reverse engineering; TLS; privacy

ACM Reference Format:
Jeffrey Knockel, Mona Wang, and Zoë Reichert. 2024. The Not-So-Silent Type: Vulnerabilities in Chinese IME Keyboards' Network Security Protocols. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690302

1 Introduction

Input Method Editors (IMEs) are a class of software keyboards that make it possible to type languages with large character sets on a smaller keyboard. As a logographic language, Chinese is especially difficult to input. In order to enable users to type Chinese, companies like Sogou, Baidu, and iFlytek developed complex IME software with predictive capabilities, which was often better than the default software available on many operating systems. As a result, third-party IME keyboards have become almost universal for Chinese users. Chinese marketing firms estimate there were over 800 million users of Chinese IMEs in 2022 [20].

IMEs also handle a particularly sensitive class of data; keystrokes encompass everything that we input or enter into our devices, including passwords, credit cards and other financial data, and messages that are otherwise end-to-end encrypted. Simultaneously, IMEs have become more technically sophisticated, many purporting to incorporate "cloud-based" functionalities that can predict the correct character from context more accurately. For Chinese, enhanced predictive capabilities can significantly improve typing speed. However, as many security researchers have previously pointed out, "cloud-based" IMEs can essentially act as keyloggers [5, 40, 47, 50], so it is clear that there is still a dire need for Chinese IMEs that are both performant and privacy-preserving [18].

In contrast to previous work which concerns the first-party collection of keystroke data [5], our work is the first to thoroughly study the security of this traffic from third-party network eavesdroppers. In this paper, we measure the transport security of "cloud-based" IMEs from nine vendors: Baidu, Honor, Huawei, iFlytek, OPPO, Samsung, Tencent, Vivo, and Xiaomi. Sogou, Baidu, and iFlytek IMEs alone comprise almost 95.9% of the market share for third-party IMEs in China [20, 31].

We conduct a systematic security measurement of 24 popular IME keyboard applications across nine vendors, including fully reverse-engineering their network security protocols and mapping vulnerabilities to known classes of cryptographic flaws. To do this, we designed and applied an analysis framework to study these apps' transmission of users' keystrokes. Our framework to systematically measure the security of IME network traffic is as follows: (1) procure the vendors' apps, (2) capture their network traffic when typing, (3) analyze their binaries to reverse engineer the traffic's cryptography, and (4) analyze the cryptography for weaknesses.

The most popular third-party Chinese IMEs functioned as keyloggers that were broadly exploitable by any network eavesdropper. Our analysis revealed critical vulnerabilities in IMEs from eight out of the nine vendors — all but Huawei — in which third-party network attackers could exploit those vulnerabilities to completely reveal the contents of users' keystrokes in transit. The vulnerabilities we discovered are known classes of cryptographic flaws, such as containing padding oracles, static key use, or key and IV re-use.

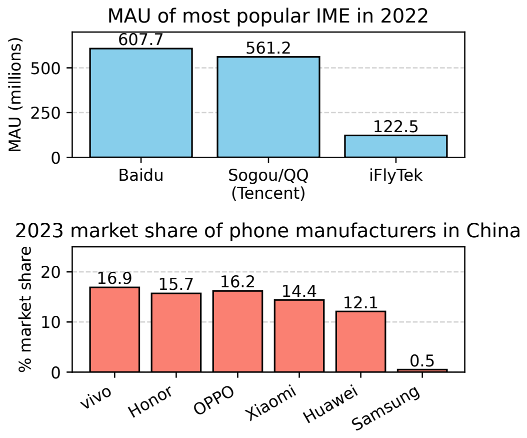


Figure 1: Monthly active users of the most popular IME software we analyzed, and Chinese market share of the devices whose pre-installed keyboards and IMEs we analyzed [20, 52].

	Android	iOS	Windows
Sogou	● → ○	○ → ○	● → ○
QQ	● → ●		● → ●
Baidu	◐ → ◐	◐ → ◐	● → ◐
iFlytek	● → ○	○ → ○	○ → ○

	Pre-installed keyboard partner			
Manufacturer	Own	Sogou	Baidu	iFlytek
Honor			● → ●*	
Huawei	○ → ○*	○ → ○†		
OPPO		● → ○	● → ◐*	
Samsung	● → ○	○ → ○*†	● → ◐	
Vivo	○ → ○*	● → ○		
Xiaomi		● → ○*	● → ◐	● → ○

●	working exploit to decrypt keystrokes in transit
◐	weaknesses present in cryptography implementation
○	no known issues
*	default keyboard/IME on our test device
†	we tested these forks of Sogou after our initial disclosure to Tencent, but they may have been previously vulnerable

Table 1: Summary of vulnerabilities discovered in popular keyboards and in keyboards pre-installed on popular Android-based mobile devices in China, before → after our vulnerability disclosure.

The scope of these issues is massive. Baidu, Sogou, and iFlytek IME keyboards comprise 95.9% of almost 800 million third-party IME users, and each have hundreds of millions of monthly active users, as demonstrated in Figure 1 [20]. In addition to the vulnerabilities we found on the Baidu, Sogou, and iFlytek IMEs, we found that the default keyboards on devices from three manufacturers (Honor, OPPO, and Xiaomi) were also vulnerable to our attacks, with devices from Samsung and Vivo also bundled with a vulnerable keyboard that was not used by default. In 2023, Honor, OPPO, and Xiaomi alone comprised nearly 50% of the smartphone market in China, also demonstrated in Figure 1 [52]. Across users of third-party IMEs and users of the default keyboards on their mobile devices, we estimate that up to one billion users could be affected by these vulnerabilities.

We responsibly disclosed these vulnerabilities to all IME companies and mobile device manufacturers. Tencent, iFlytek, OPPO, and Samsung responded positively and worked to update their IMEs' protocols to use TLS, except for QQ Pinyin. Although Vivo, Xiaomi, and Baidu were unresponsive, based on our testing, they have also updated their keyboards after our initial disclosures. Unfortunately Baidu has not switched to TLS and instead switched to a different proprietary protocol. Honor asked us to disclose to Baidu instead, which we had already done. To summarize, we no longer have working exploits against any products except Honor's default keyboard and Tencent's QQ Pinyin. Baidu's IME on other devices continues to contain weaknesses in its cryptography. The full status of vulnerabilities before and after our disclosures is summarized in Table 1.

Our work joins others in presenting important evidence that there still exists a larger, understudied class of apps sharing a problematic security property: the use of home-rolled cryptography. We find that, despite the ecosystem's overall trend toward TLS, massively popular cloud-based keyboard apps which handle extremely sensitive data are almost universally members of this class and that their encryption can be completely defeated by network eavesdroppers. By analyzing apps in this class, our work helps to further characterize the shape of this class by dispelling misconceptions such as that its members are not widely popular or not written by large technology companies. Such an understanding is crucial so that researchers can adequately prioritize and address this class of apps' unique security challenges.

2 Background

Compared to typing alphabetic languages whose small number of letters can be represented uniquely by keys, typing logographic languages such as Chinese is more difficult. Chinese has tens of thousands of characters used in varying frequencies, many of which are homophones and, in speech, are only distinguishable by context. The necessity for predictive capabilities therefore arose much earlier in Chinese input than for alphabetic language input, even as early as typewriter usage [37]. Chinese IMEs with better predictive capabilities can dramatically improve one's typing speed. Nowadays, there are different competing standards for predictive digital Chinese character input, including phonetic inputs like Pinyin (the most popular in mainland China [20]) and Zhuyin, and radical or stroke-based inputs like Cangjie or Wubi. Modern input methods also support handwriting or voice recognition.

As IME apps' predictive input capabilities have become more sophisticated, they have also become universally popular. Market reports estimate there are up to 800 million users of third-party Chinese IME keyboards, with the major players being Sogou, Baidu, and iFlytek comprising 95.9% of the total market share [20]. In addition, all the major mobile device manufacturers have developed their own IMEs or have partnered with some of the above three companies to provide predictive input capabilities by default on their devices. In this work, we study the top three third-party Chinese IME apps across Android, iOS, and Windows, as well as the pre-installed IMEs on the most popular Android phones in China, as demonstrated in Figure 1.

2.1 Privacy risks of Chinese IMEs

Awareness of the potential security and privacy risks associated with these apps has grown. Some researchers have expressed concern over companies handling sensitive keystroke data and have made attempts to ameliorate the risk of the cloud server being able to record what users type. In 2013, the Japanese government published concerns it had with privacy regarding the Baidu IME, particularly the cloud input function [40]. Others have also been concerned with surveillance via other "cloud-based" IMEs, like iFlytek's voice input [19]. While there has been a push to develop privacy-aware, cloud-based IMEs that would keep user data secret, they are not widely used [21].

In 2015, Chen et al. proposed and evaluated a system to identify and prevent keystroke leakages in IME traffic, revealing that at least one IME was transmitting sensitive data without encrypting it at all [5]. Another security report in the same year showed that the most popular IME at the time, Sogou, was sending users' device identifiers in the clear [28].

Although there is an abundance of work covering the privacy or surveillance risk posed by IME providers' collection of keystrokes, ours is the first work to thoroughly examine the security of this data in transit from third-party network eavesdroppers.

2.2 Transport security and Chinese apps

Third-party network adversaries can leverage vulnerabilities in transport security to compromise user privacy at very large scales. Most work studying the transport security of network traffic has historically studied vulnerabilities in SSL or TLS [26, 35]. In 2018, Böck et al. describes and measures the practical mass-exploitability of Bleichenbacher oracles, a known vulnerability, in particular ciphersuites used by SSL or TLS [4].

More related to our work is the 2013 study of Egele et al. [11] who systematically analyzed Google Play Store apps for common cryptographic flaws, finding that hard-coded symmetric keys to be the second most common flaw encountered. While the ecosystem has improved in the intervening 11 years, our study shows that such flaws affect even apps developed by large technology companies with hundreds of millions of users. Acar et al.[1] investigated whether low-level cryptographic APIs are too complicated for developers to use, finding that experienced python developers commonly could not reconstruct secure cryptographic systems even using official documentation. Even if developers use TLS, researchers have measured multiple pitfalls [14, 38, 39] in how developers interact with its higher level APIs in practice.

The broad security impact of these works inspired us to study the traffic encryption employed by Chinese IMEs, which were especially interesting since they were largely not using SSL/TLS. Early work from 2015 noted the existence of proprietary network protocols used by Chinese apps, but did not study them in detail [5], or has studied apps individually, such as with Chinese-developed Zoom [33]. Notably, a 2015–2016 series of reports found extensive flaws in the network cryptography applied by Chinese browsers to data which they "phone home" [6, 22–25]. Our work shows that such flaws exist in popular apps in 2023.

While our work shows how ineffective, proprietary cryptography exposes users' keystrokes, other work has demonstrated how users' typing can be revealed via side-channel attacks. Song et al. [45] demonstrated how, with training data collected from a user, the same user's ssh keystrokes, including password contents, can be inferred by the keystrokes' timing and via other network side channels. Bhanu [3] has shown that certain features of network keystroke data can be inferred, such as the language a user is typing, without any training data specific to that user. While research has worked toward ameliorating timing side channels in user typing by faking keystrokes [43], such methods are generally not deployed and would require, for network-based apps, increased network utilization.

While keystroke exfiltration via side-channels is a concerning security threat, our primary threat model is a network adversary interested in performing precise and scalable mass surveillance of individuals' keystrokes. We are especially interested in this threat model because similar vulnerabilities from Chinese applications have been targets for mass exploitation, expressly for the purpose of mass surveillance by government agencies. While many governments may possess sophisticated mass surveillance capabilities, the Snowden revelations gave us unique insight into the capabilities of the United States National Security Agency (NSA) and more broadly the Five Eyes. The revelations disclosed, among other programs, an NSA program called XKEYSCORE for collecting and searching Internet data in realtime across the globe. Leaked slides describing the program specifically reveal only a few examples of XKEYSCORE plugins. However, one was a plugin that was written by a Five Eyes team to take advantage of vulnerabilities in the network cryptography of Chinese-developed UC Browser to enable the Five Eyes to collect device identifiers, SIM card identifiers, and account information pertaining to UC Browser users [24, 48]. Given the enormous intelligence value of knowing what users are typing, we can conclude that not only do the NSA and more broadly the Five Eyes have the capabilities to mass exploit the vulnerabilities we found but also the strong motivation to exploit them. This is generally true for other governments as well.

In this work, we design and apply a framework for systematically analyzing the security of the network encryption used to protect keystroke traffic between Chinese IMEs and their cloud servers. Ours is the first to measure the security of the proprietary encryption protocols deployed by IME vendors to protect keystroke traffic.

2.3 Threat model

As we are studying the security of data in transit, our threat model concerns third-party network attackers and not the first-party collection of keystroke data by the IME provider. The primary adversary we consider is a network adversary interested in performing precise and scalable mass surveillance of individuals' keystrokes, as with the Five Eyes' XKEYSCORE UCWeb plugin, which exploited weak network cryptography in a Chinese application for the express purpose of mass surveillance [24, 48].

More specifically, our threat model includes a passive network adversary that can monitor network packets and an active network adversary that attacks messages sent between an IME client and server. Either adversary also has access to a copy of the client software, but the server is a black box. In the case of a passive network adversary, they can monitor and collect any number of messages that are sent between a user's IME and the IME's cloud server. Additionally, the passive network adversary can send messages to the server but does not necessarily have to be an active MITM or spoof messages from the user in a layer 3 sense. In the case of an active network adversary, the adversary sits on the network path between the client and server and can block, alter, resend, or otherwise manipulate any messages. The goal of all adversaries is firstly to decrypt any of the ongoing messages between the client and server or altering messages between the client and server to change what the client sees. In the absence of complete decryption, the adversary is still interested in learning information about the client and server's communication or successfully altering their communications.

As we mentioned in the previous section, prior work has explored keystroke exfiltration via side-channels and sophisticated traffic analysis in other contexts. While this form of keystroke exfiltration is an increasingly concerning security threat, we consider these capabilities beyond the scope of our primary adversary. These attacks do not require the adversary to compile training data on a targeted user or in any other capacity, and do not require the adversary to accept probabilistic errors, both in failing to recover keystrokes or recovering incorrect keystrokes. It is important to evaluate both the security of these keystrokes in transit to defend against mass surveillance, as well as the security of keystrokes against more complex side-channel and fingerprinting attacks to defend against more targeted surveillance. We focus on the former. As an analogy to web browsing, it is critical to evaluate security transports like Tor and TLS/HTTPS for vulnerabilities despite the existence and efficacy of powerful website fingerprinting attacks [34].

3 Methods

We describe our methods for conducting a systematic network security measurement of 24 IME keyboard applications across nine vendors. In this process, we fully reverse-engineered their protocols and mapped vulnerabilities to known classes of cryptographic flaws.

We analyzed the Android, and, if present, the iOS and Windows versions of IMEs from the following vendors: Tencent, Baidu, iFlytek, Samsung, Huawei, Xiami, OPPO, Vivo, and Honor. The first three — Tencent, Baidu, and iFlytek — are software developers of IMEs whereas the remaining six — Samsung, Huawei, Xiami, OPPO, Vivo, and Honor — are mobile device manufacturers who either developed their own IMEs or include one or more of the former three's IMEs preinstalled on their mobile devices. We chose these nine vendors as the former three develop the most popular third-party IMEs, and the latter six companies comprise the bulk of the mainland Chinese smartphone market [31, 52].

3.1 Analysis framework

For each of these vendors, we applied the following analysis framework: we (1) procure the vendors' apps, (2) capture their network traffic when typing, (3) analyze their binaries to reverse engineer the traffic's cryptography, and (4) analyze their cryptography for weaknesses.

3.1.1 Procuring the vendor's IME applications. To procure the apps that we analyzed, between August and November 2023, we downloaded the latest versions of them from their product websites, the Apple App Store, or, in the case of the apps developed or bundled by mobile device manufacturers, by procuring a mobile device that had the app preinstalled. The devices we obtained were intended for the mainland Chinese market, and, when device manufacturers had two editions of their device, a Chinese edition and a global edition, we analyzed the Chinese edition. When analyzing apps that we obtained from a mobile device, we ensured that the device's apps and operating system were fully updated before beginning analysis of its apps.

3.1.2 Capturing network traffic during character input. After procuring each app, to capture its network traffic, we first installed it and enabled the pinyin input if it was not already enabled by default. The keyboards we analyzed generally prompted users to enable cloud functionality after installation or on first use. In such cases, we answered such prompts in the affirmative or otherwise enabled cloud functionality through the mobile device's or app's settings. We then typed with the apps, using Wireshark and mitmproxy to capture network traffic and to look for any traffic that was consistently sent upon every keystroke.

3.1.3 Analyzing the binary and dynamic instrumentation. After capturing an app's network traffic, to reverse engineer the cryptography applied to the network traffic, we applied standard static and dynamic reverse-engineering methods [2, 7, 8, 32]. We used jadx to statically analyze and decompile Dalvik bytecode and both IDA Pro and Ghidra to statically analyze and decompile native machine code. We used frida to dynamically analyze the Android and iOS versions and IDA Pro to dynamically analyze the Windows version.

3.1.4 Cryptography analysis. Finally, after reverse engineering the network traffic's cryptography, we analyzed it for weaknesses. We looked for violations of cryptographic fundamentals, such as a lack of asymmetric cryptography, secrets generated with predictable seeds, and the use of ciphers with known weaknesses. When applicable, we also evaluated the cryptography for oracles and oracle-related attacks. Whenever possible, we authored proof-of-concept attacks that fully decrypt captured network traffic. Otherwise, we reported any flaws as weaknesses in the cryptography that we were unable to completely exploit.

In this work we are motivated by compelling app developers to switch from home-rolled cryptography to TLS. Thus, for purposes of this report we consider any use of a standard TLS implementation to be sufficient cryptography. While vulnerabilities may yet exist in TLS implementations and in how they are used, our work is primarily concerned with discovering flaws in apps which do not use TLS and which implement their own cryptography.

We note that, as neither Apple's nor Google's keyboards have a feature to transmit keystrokes to cloud servers for cloud-based recommendations, we did (and could) not analyze these keyboards for the security of this feature. However, we observed that none of the mobile devices that we analyzed included Google's keyboard, Gboard, preinstalled, either. This finding likely results from Google's exit from China due to failing to comply with China's censorship requirements.

3.2 Ethical considerations

We did not use any actively used devices for this project, using only devices which were purchased for the express purpose of research and emulators. We only collected and analyzed apps and network data on these test devices while they were operated by researchers.

3.2.1 Coordinated disclosures and timeline. We undertook an extensive coordinated vulnerability disclosure with eight different vendors: Tencent, Baidu, iFlytek, Samsung, Honor, OPPO, Vivo, and Xiaomi[1]. These vulnerabilities are severe and, if not fully fixed, could cause privacy harm to billions of users.

In our follow-up analyses, we considered the cryptography sufficiently resolved if it used a version of TLS provided by the operating system or by a well-known provider. If the vendor continued using proprietary cryptography but made improvements to it, we evaluated the improved cryptography for weaknesses.

iFlytek, OPPO, and Samsung responded positively and worked to update their IMEs' protocols to use TLS. Tencent did the same for Sogou IME, but not for QQ Pinyin. Honor responded, but asked us to disclose to and coordinate with Baidu instead, which we had already attempted. Vivo, Xiaomi, and Baidu were unresponsive, but, based on our testing, they have all updated their keyboards after our initial disclosures. Unfortunately Baidu has not switched to TLS and instead switched to an upgraded proprietary protocol, *BAIDUv4*. Although this protocol is not as trivially broken as the previous Baidu protocols, it still contains many weaknesses compared to TLS, which we detail in Section 4.2.2.

The default keyboard on our Honor device remains vulnerable to these issues. We also note that our Honor device does not have a straightforward update mechanism for their default keyboard, as it is not possible to receive updates for it via the Honor App Market.

Regarding QQ Pinyin, Tencent indicated that "with the exception of end-of-life products, we aim to finalize the upgrade for all active products to transmit EncryptWall requests via HTTPS by the conclusion of Q1 [2024]", but, as of April 1, 2024, though Sogou IME has been updated, we have not seen fixes to QQ Pinyin. Tencent may consider QQ Pinyin end-of-life as it has not received updates since 2020, although it is still available for download.

In summary, we no longer have working exploits against any products except Honor's default IME and Tencent's QQ Pinyin. Baidu's IMEs on other devices continue to contain weaknesses in their cryptography. The full status of vulnerabilities after our disclosures, as of April 1, 2024, is summarized in Table 1. As it is long past the disclosure windows from our initial reports in October and November of 2023, we hope widely publicizing these vulnerabilities will prevent further privacy harm from befalling existing users of these applications.

[1]For the full disclosure timeline for each vendor, see https://citizenlab.ca/2024/04/vulnerabilities-across-keyboard-apps-reveal-keystrokes-to-network-eavesdroppers/#disclosure-timelines.

4 Results

Among the nine vendors whose apps we analyzed, there was only one vendor, Huawei, in whose apps we could not find security issues regarding the transmission of users' keystrokes. For each of the remaining eight vendors, in at least one of their apps we discovered a vulnerability in which keystrokes could be completely revealed by a network eavesdropper (see Tables 2 and 3 for details).

The ease with which the keystrokes in these apps could be revealed varied. In one app, Samsung Keyboard, we found that the app performed no encryption whatsoever. Some apps appeared to internally use Sogou's cloud functionality. Most vulnerable apps failed to even use asymmetric cryptography, relying solely on home-rolled symmetric encryption to protect users' keystrokes.

In procuring the apps, we found that each mobile device manufacturer bundled a variant of at least one of Sogou's, Baidu's, or iFlytek's keyboards, sometimes alongside the manufacturer's own IME. Due to code and API licensing, we found that the 24 apps in Table 2 that do not use TLS, used one of six unique network encryption schemes, which we refer to as *EncryptWall-And*, *EncryptWall-Win*, *BAIDUv3-1*, *BAIDUv3-2*, *BAIDUv4*, *iFlytek-And*. The first two are subvariants of the `EncryptWall` scheme we identified in Sogou-related keyboards, and then we identified the three `BAIDU` schemes in different Baidu-related keyboards. *iFlytek-And* was used by all the Android-based iFlytek variants we studied. Table 3 summarizes each encryption protocol as well as the key vulnerabilities we identified in each protocol.

In the following sections, we describe each of the protocols and the core vulnerabilities we identified.

4.1 EncryptWall

Sogou IME's encryption system is internally referred to as `EncryptWall`. We found that the Windows, iOS, and Android implementations of `EncryptWall` each differ slightly, so we name these protocols *EncryptWall-And*, *EncryptWall-iOS*, and *EncryptWall-Win*, respectively. In addition to Sogou-branded keyboards, QQ Pinyin also uses *EncryptWall-And* and *EncryptWall-Win* to encrypt to Sogou endpoints.

EncryptWall-And and *EncryptWall-Win* were both vulnerable to variants of a CBC padding oracle attack [49] that allowed us to completely recover encrypted keystrokes. In the case of the Android version, we were also able to recover the second halves of the symmetric encryption keys used to encrypt traffic. We also found vulnerabilities affecting *EncryptWall-iOS*, but we are not presently aware of methods to exploit them since the requests from the iOS version were wrapped in TLS.

Since there were differences in the EncryptWall implementation across the three operating system platforms that we analyzed, we first overview the common system before detailing the specific implementations of it in the following sections.

4.1.1 Overview. When analyzing Sogou IME's network traffic, we found that it communicates users' keystrokes via plain HTTP POST requests to Sogou API endpoints which are referred to internally as "EncryptWall" endpoints. These plain HTTP requests do however contain an encrypted payload. We call the outer, plain HTTP request the *EncryptWall* request and the single tunneled HTTP request each EncryptWall request encapsulates the *tunneled* request. We also

Vendor	Program name	File/package name	Version analyzed	Platform	Protocol used	As of 2024-04-01
Tencent	Sogou IME	SogouInput_11.20_android_sweb.apk	11.20	Android	*EncryptWall-And*	TLS
Tencent	Sogou IME	com.sogou.sogouinput	11.21	iOS	TLS	TLS
Tencent	Sogou IME	sogou_pinyin_guanwang_13.4e_1111.exe	13.4	Windows	*EncryptWall-Win*	TLS
Tencent	QQ IME	com.tencent.qqpinyin	8.6.3	Android	*EncryptWall-And*	*EncryptWall-And*
Tencent	QQ IME	QQPinyin_Setup_6.6.6304.400.exe	6.6.6304.400	Windows	*EncryptWall-Win*	*EncryptWall-Win*
Baidu	Baidu IME	com.baidu.input	11.7.19.9	Android	*BAIDUv4*	*BAIDUv4*
Baidu	Baidu IME	com.baidu.inputMethod	11.7.20	iOS	*BAIDUv4*	*BAIDUv4*
Baidu	Baidu IME	BaiduPinyinSetup_6.0.3.44.exe	6.0.3.44	Windows	*BAIDUv3-2*	*BAIDUv4*
iFlytek	iFlytek IME	com.iflytek.inputmethod	12.1.10	Android	*iFlytek-And*	TLS
iFlytek	iFlytek IME	com.iflytek.inputime	12.1.3338	iOS	TLS	TLS
iFlytek	iFlytek IME	iFlyIME_Setup_3.0.1734.exe	3.0.1734	Windows	TLS	TLS
Honor	Baidu IME Honor Version	com.baidu.input_hihonor	8.2.501.1	Android	*BAIDUv3-2*	*BAIDUv3-2*
Huawei	Celia IME	com.huawei.ohos.inputmethod	1.0.19.333	Android	TLS	TLS
Huawei	Sogou IME	com.sohu.inputmethod.sogou	11.31	Android	TLS*	TLS
OPPO	Sogou IME Custom Version	com.sohu.inputmethod.sogouoem	8.32.0322.2305171502	Android	*EncryptWall-And*	TLS
OPPO	Baidu IME Custom Version	com.baidu.input_oppo	8.5.30.503	Android	*BAIDUv3-2*	*BAIDUv4*
Samsung	Samsung Keyboard	com.samsung.android.honeyboard	5.6.10.26	Android	No encryption	TLS
Samsung	Sogou IME Samsung Version	com.sohu.inputmethod.sogou.samsung	10.32.38.202307281642	Android	TLS*	TLS
Samsung	Baidu IME	com.baidu.input	8.5.20.4	Android	*BAIDUv3-1*	*BAIDUv4*
Vivo	Jovi IME	com.vivo.ai.ime	2.6.1.2305231	Android	TLS	TLS
Vivo	Sogou IME Custom Version	com.sohu.inputmethod.sogou.vivo	10.32.13023.2305191843	Android	*EncryptWall-And*	TLS
Xiaomi	Sogou IME Xiaomi Version	com.sohu.inputmethod.sogou.xiaomi	10.6.120.480	Android	*EncryptWall-And*	TLS
Xiaomi	Baidu IME Xiaomi Version	com.baidu.input_mi	10.6.120.480	Android	*BAIDUv3-2*	*BAIDUv4*
Xiaomi	iFlytek IME Xiaomi Version	com.iflytek.inputmethod.miui	8.1.8014	Android	*iFlytek-And*	TLS

* Tested after our initial disclosure with Tencent Sogou, but may have been previously using *EncryptWall-And*.

Table 2: The 24 IMEs we tested and the network protocols they used to transmit keystrokes. We tested the most popular third-party IMEs, as well as the IMEs that were pre-installed on 6 devices from popular Chinese device manufacturers.

Protocol	Status	Core vulnerability	Mode	Variation
EncryptWall-And	Decryptable	CBC padding oracle	AES-CBC	Splits key into two, uses fixed IV
EncryptWall-Win	Decryptable	CBC padding oracle	AES-CBC	
BAIDUv3-1	Passively decryptable	Fixed key	AES-ECB	Additional permutations each AES round
BAIDUv3-2	Passively decryptable	Fixed key	AES-ECB	Missing AES round
BAIDUv4	Not CPA-secure	IV and key re-use	AES-BCTR	Uses home-rolled CTR mode
iFlytek-And	Passively decryptable	Key derived from timestamp	DES-ECB	

Table 3: Each of the proprietary cryptography protocols we identified in the 24 IMEs we studied, a summary of their encryption mode, and the key vulnerabilities we found in each. In addition, *EncryptWall-And* and *EncryptWall-Win* use RSA to protect key material in transit. *BAIDUv4* uses static Diffie-Hellman.

observed that in addition to a tunneled payload each EncryptWall request contains at least five HTTP form fields.

By analyzing Sogou IME's binaries, we found that these five HTTP form fields specify cryptographic parameters used to encrypt the tunneled request in addition to the encrypted tunneled data. These form fields and the data they encode or encrypt are summarized in Table 4.

Two form fields relate to specifying the key and initialization vector (IV) used to encrypt other fields in the EncryptWall request. pk is an RSA public key pinned to the application, and both k and IV are randomly generated for each request.

- "K": the base64 encoding of the encryption of a 256-bit AES key k with a hard-coded 1024-bit public RSA pk using PKCS#v1.5 padding.
- "V": the base64 encoding of a 128-bit IV used as an IV.

Three of the form fields "U", "G", and "P", are individually zlib compressed, encrypted using k and IV, and base64-encoded according to the following pseudo-code:

$$E^{wall}(k, IV, data) = \text{base64\_encode}(\text{AES\_cbc\_encrypt}(\text{zlib\_compress}(data, \text{wbits}{=}{-15}), k, IV))$$

We found that the EncryptWall system is vulnerable to a CBC padding oracle attack, a type of chosen ciphertext attack originally published in 2002 [49] impacting block ciphers using cipher block chaining (CBC) block cipher mode and PKCS#7 padding. Specifically, we found that a ciphertext sent in the "U" form field returns an HTTP 400 status code when it contains incorrect padding, whereas, when correctly padded, it returns either a 200 status or 500 status code depending on whether the decrypted URL is a valid URL or not, respectively. By performing a CBC padding oracle attack, this padding oracle allows us to not only reveal the entire plaintexts p_u, p_g, and p_p, since they are encrypted using the same k and IV.

Field	*EncryptWall-And*	*EncryptWall-Win*	Underlying data purpose
K	$E^{RSA}(pk, k)$	$E^{RSA}(pk, k)$	k: 32-byte symmetric key
V	IV	$E^{RSA}(pk, IV)$	IV: 16-byte initalization vector
U	$E(k, IV, p_u)$	$E^{wall}(k, IV, p_u)$	p_u: Tunneled URL
G	$E(k, IV, p_g)$	$E^{wall}(k, IV, p_g)$	p_g: GET params to p_u
P	$E(k, IV, p_p)$	$E^{wall}(k, IV, p_p)$	p_p: POST params to p_u
R	$k \oplus r$	-	r: Second 32-byte key
S	$k \oplus E(r, IV_f, p_s)$	-	p_s: Sogou and device identifiers
E	$k \oplus E(r, IV_f, p_e)$	-	p_e: Various internal parameters
F	$k \oplus E(r, IV_f, p_f)$	-	p_f: Not set on version analyzed

Table 4: Overview of the HTTP POST form fields sent by *EncryptWall-And* and *EncryptWall-Win* to http://v2.get.sogou.com/q or http://get.sogou.com/q. These fields were all base64-encoded. pk and IV_f are pinned to the application. k and IV are generated per-request. r is generated per-"session" as it is cached in application memory.

Thus, by using this padding oracle we can decrypt the contents of the entire EncryptWall request.

In the following sections, we adapt this attack for all variations in the implementation of *EncryptWall-And* and *EncryptWall-Win*. Although they do not presently appear exploitable, we also detail defects in the EncryptWall system on iOS.

4.1.2 EncryptWall on Windows: EncryptWall-Win. When analyzing the Windows binaries, we found that the EncryptWall system deviated from the basic implementation described above in one detail, namely that IV, instead of being public, was encrypted in the same manner as k, as demonstrated in Table 4. Due to this discrepancy, IV is not immediately known, which is potentially problematic for our attack for two reasons: first, in the CBC padding oracle attack, the IV must be known in order to decrypt the first block of plaintext. Second, since the data p_u, p_g, and p_p are compressed before being encrypted, the first block of plaintext is important for decompressing the remaining blocks.

However, we developed a method to recover IV that exploits the fact that IV is reused to encrypt multiple plaintexts. Specifically, since the URL p_u is easily predictable and is ever only one of a small number of possible endpoints, we are able to recover IV by performing a CBC padding oracle attack on the first ciphertext block of "U", assuming an all zero IV. The result of this attack will be $p_{u_1} \oplus IV$. Since p_{u_1}, the first block of p_u, is predictable, we recover IV. Then, we can perform the CBC padding oracle attack on "G" and "P" as usual.

As one example of the kind of transmitted data vulnerable to this attack, we found that for EncryptWall requests sent to "http://get.sogou.com/q", when p_u was "http://master-proxy.shouji.sogou.com/swc.php", p_g contained version information pertaining to Sogou IME's software, and p_p was a protobuf buffer containing the keystrokes that had been recently typed in, as shown in Figure 2.

Since these transmissions are vulnerable to our attack, the keystrokes of Sogou Input Method users can be decrypted by a network eavesdropper, informing the eavesdropper of what users are typing as they type.

4.1.3 EncryptWall on Android: EncryptWall-And. When analyzing the Android version's network traffic, we found that the Android version adopts the basic implementation of EncryptWall but with the inclusion of four additional form fields: "R", "S", "E", and "F", which are also depicted in Table 4. By analyzing its binaries, we found that the field "R" transmits another 32-byte key r. Notably, however, each byte of r is randomly chosen from the 36-character set of ASCII uppercase letters and numbers. Therefore, instead of $\log_2 256^{32} = 256$ bits of entropy, r only has $\log_2 36^{32} < 166$ bits of entropy. Furthermore, unlike k, r is not generated randomly for each request and is only generated once per application lifetime as it is cached in C static memory. The field "R" is then transmitted as the base64 encoding of $k \oplus r$. Note that due to this transmission, k's entropy is also reduced to $\log_2 36^{32} < 166$ bits of entropy. The parameters k and r are used to encode "S", "E", and "F" according to the following pseudo-code:

$$E^{SEF}(data) = \text{base64\_encode}(k \oplus \text{AES\_cbc\_encrypt}(data, r, \texttt{"EscowDorisCarlos"}))$$

Note that unlike the typical $E^{wall}()$ function, $E^{SEF}()$ features a hard-coded fixed IV IV_f = "EscowDorisCarlos" and no zlib compression. Additionally, although $E^{SEF}()$ uses r instead of k as an AES key, k is additionally XORed with the result of the AES encryption. Each of the plaintexts p_s, p_e, and p_f are individually encrypted and encoded according to the $E^{SEF}()$ function.

We were able to apply the CBC padding oracle attack, using Sogou's processing of the "E" form field, with the following two accommodations:

First, since the key k is 32 bytes but AES blocks are 16 bytes, when the output of the AES block cipher is XORed with k, we can think of the output being XORed with two keys k_1 and k_2, where k_1

```
1 {
    1: "com.android.messaging"
    2: "11.20"
    4: 1
    6: "android_sweb"
    8: "Google"
    10: "android_sweb"
    11: "11.20"
    14: "30"
    18: "-1"
    22: "5682b3aa4fa7bd40d776c93a35a77c6d"
}
2 {
    1: 0xbff0000000000000
    2: 0xbff0000000000000
    3: "-1"
}
3: 1
4: "canyoureadthis"
11 {
    1: "onekeyimageenable"
    2: "1"
}
```

Figure 2: Example recovered protobuf data from Sogou IME on Android; line 19 contains the typed text and line 2 contains the package name of the app in which the text is being typed.

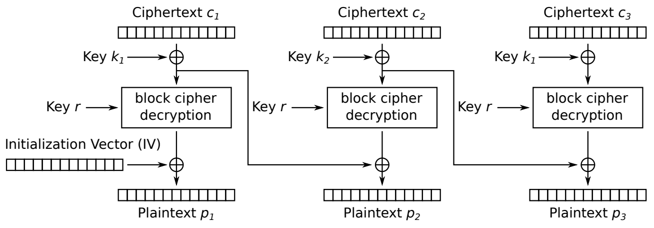


Figure 3: A modified version of CBC used by *EncryptWall-And* in which a 32-byte key $k = k_1 \parallel k_2$ composed of two 16-byte keys k_1 and k_2 is XORed with ciphertext blocks before being decrypted by the block cipher such that k_1 is XORed with odd-numbered blocks (1, 3, . . .) and k_2 is XORed with even-numbered blocks (2, 4, . . .).

is XORed with odd-numbered blocks (1, 3, . . .) and k_2 is XORed with even-numbered blocks (2, 4, . . .), as demonstrated in Figure 3. Thus, when performing the CBC padding oracle attack, we had to ensure that the block that we were attacking was in an even-numbered position if it was originally even-numbered or in an odd-numbered position if it was originally odd-numbered. In other words, we had to preserve the parity of the block's position.

Second, since IV_f is hard-coded, we cannot modify it and thus, similar to the Windows version, the CBC padding oracle attack cannot recover the first block of plaintext p_1 without an adaptation. Namely, we found that p_{s_1}, p_{e_1}, and p_{f_1} were still recoverable via the following procedure:

(1) We treat the fixed IV_f as a ciphertext block $c_0 = IV_f$ preceding the first ciphertext block c_1 and send it to the oracle. Since c_1 must be in an odd-numbered position, we ensure that c_0 is in an even-numbered position. Thus, during the attack, the oracle uses $c_0 \oplus k_2$ when decrypting c_1.
(2) Decryption of c_1 produces $p'_1 = p1 \oplus IV_f \oplus c_0 \oplus k_2$.
(3) Since (per step 1) $c_0 = IV_f$, p'_1 is merely $p_1 \oplus k_2$. Therefore, we can recover $p_{s_1} \oplus k_2$, $p_{e_1} \oplus k_2$, and $p_{f_1} \oplus k_2$.
(4) Moreover, the value of p_{s_1} was highly predictable. Namely, it encoded the version of Sogou being used, which was already transmitted in the clear as an HTTP header of the EncryptWall request. Thus, since we have recovered $p_{s_1} \oplus k_2$ and p_{s_1} is known, we have therefore recovered k_2.
(5) Once we know k_2, we have also recovered p_{e_1} and p_{f_1}.

Additionally, we can now also recover the second half of r, r_2. Since r is reused for multiple EncryptWall requests, this can be used to more easily recover k_2 in subsequent requests. Since the form field "R" encodes $k \oplus r$ we can recover r_2 by XORing the second half of the "R" field's encoded contents with k_2. Once r_2 is recovered, since r, unlike k, is generated once per application lifetime, we can more easily recover k_2 in future requests by simply XORing the second half of "R" with r_2, making the attack even easier to perform in the future. Furthermore, this reduces the entropy of r, and thus, also k, to $\log_2 36^{16} < 83$ bits.

As one example of the kind of transmitted data vulnerable to this attack, we observed that for EncryptWall requests sent to "http://v2.get.sogou.com/q", when p_u was "http://swc.pinyin.sogou.com/swc.php", p_p was a protobuf buffer containing all of the text currently present in the input field in which the user is currently typing as well as the package name of the app in which the text was being typed. These transmissions occurred when pressing the magnifying glass icon, and we believe that these transmissions are related to an image search feature in which typed text is searched against a database of animations and memes which can be inserted into the typed message.

As one other example of the kind of transmitted data vulnerable to this attack, we observed that for EncryptWall requests sent to "http://v2.get.sogou.com/q", when p_u was "http://update.ping.android.shouji.sogou.com/update.gif", p_p was a query string containing a list of every app installed on the Android device. We are unaware of what feature this data transmission is intended to implement. While one can imagine knowing which app a user is presently using may be useful for providing better typing suggestions in that app, it is difficult to imagine how knowing every app that a user has installed can provide better typing suggestions, even apps which users do not intend to use with Sogou IME.

4.1.4 *EncryptWall on iOS.*

The iOS version which we analyzed had no major deviations from the basic EncryptWall implementation. However, all EncryptWall requests that we observed transmitted by the iOS version which we analyzed were transmitted over HTTPS, and thus were additionally encrypted using TLS. However, we note that without the additional protection of HTTPS, the iOS version would have been the most vulnerable due to the existence of an additional defect in the implementation of EncryptWall.

Before randomly generating k and again before randomly generating the IV the random number generator is seeded with the current time as seconds since the Unix epoch, rounded down to a whole second. There are two consequences to this behavior: first, the only information needed to derive k is the time which the request was sent, which any network eavesdropper would be able to easily record. Second, since the random number generator is re-seeded before generating IV with what will almost always be the same time in seconds after rounding, IV is almost always the first 128 bits of k. Since IV is public, all EncryptWall messages reveal the first half of k in IV, despite the fact that k is encrypted with the public RSA key pk.

4.2 *BAIDUv3* and *BAIDUv4*

In our network traffic analysis, we discovered that all versions of Baidu's protocols transmit keystrokes via UDP packets. We found that Baidu IMEs are generally split between two major protocol variations, which differ in both structure and cryptography. Most notably, in structure they can be differentiated by the first two bytes. One variation's UDP payload always begins with 0x03 0x01, and the other's UDP payload begins with the bytes 0x04 0x00. We henceforth refer to these protocols as the *BAIDUv3* and *BAIDUv4* protocols, respectively.

Additionally, we identified two minor variations of *BAIDUv3*, which we refer to as *BAIDUv3-1* and *BAIDUv3-2*. Both *BAIDUv3-1* and *BAIDUv3-2* contain a vulnerability that allows network eavesdroppers to decrypt network transmissions. We also found that *BAIDUv4* is not CPA-secure, and thus has severe privacy weaknesses. However, we were unable to exploit *BAIDUv4* to reveal users' keystrokes completely.

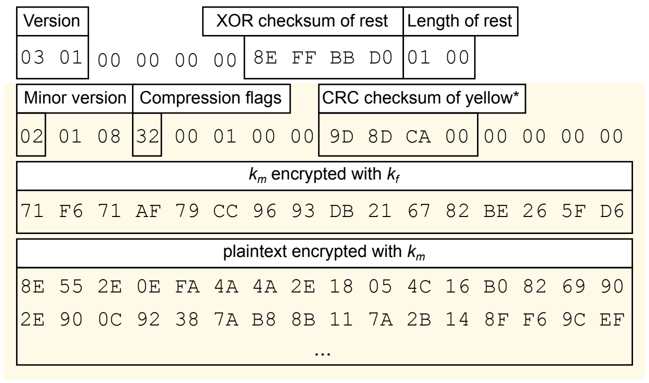

Version
XOR checksum of rest
Length of rest
03 01 00 00 00 00 8E FF BB D0 01 00
Minor version
Compression flags
CRC checksum of yellow*
02 01 08 32 00 01 00 00 9D 8D CA 00 00 00 00 00
k_m encrypted with k_f
71 F6 71 AF 79 CC 96 93 DB 21 67 82 BE 26 5F D6
plaintext encrypted with k_m
8E 55 2E 0E FA 4A 4A 2E 18 05 4C 16 B0 82 69 90
2E 90 0C 92 38 7A B8 8B 11 7A 2B 14 8F F6 9C EF
...

Figure 4: Wire format and sample payload for *BAIDUv3-2*. *Note that the CRC checksum is calculated on the yellow portion with the checksum itself taken to be all `00` bytes.

```
[...]
2 {
  1: "nihaonihao"
}
3 {
  1: 28
  2: 10
  3: 1240
  4: 2662
  5: 5
}
4 {
  1: "47148455BDAEBA8A253ACBCC1CA40B1B%7CV7JTLNPID"
  2: "p-a1-5-105|PHK110|720"
  3: "8.5.30.503"
  4: "com.android.mms"
  5: "1021078a"
}
[...]
```

Figure 5: Example excerpt of recovered protobuf data from Baidu IME Custom Version pre-installed on an OPPO device, encrypted using *BAIDUv3-2*, including what we had typed ("nihaonihao") and the app into which it was typed ("com.android.mms").

4.2.1 BAIDUv3 *Encryption.*

BAIDUv3 encrypts keystrokes using a modified version of AES. However, we found in our work two different variations of AES used; to differentiate between the protocols which use different versions of AES, we refer to them as *BAIDUv3-1* and *BAIDUv3-2*. These two variations also have slightly different wire formats.

When encrypting, *BAIDUv3-1*'s key expansion is like that of standard AES, except, on each but the first subkey, the order of the subkey's bytes are additionally permuted. Furthermore, on the encryption of each block, the bytes of the block are additionally permuted in two locations, once near the beginning of the block's encryption immediately after the block has been XOR'd by the first subkey and again near the end of the block's encryption immediately before S-box substitution. Aside from complicating our analysis, we are not aware of these modifications altering the security properties of AES, and we have developed an implementation of this algorithm to both encrypt and decrypt messages given a plaintext or ciphertext and a key.

BAIDUv3-2 does not contain the same modifications to AES as *BAIDUv3-1*. Normally, AES when used with a 128-bit key performs 10 rounds of encryption on each block. However, we found that *BAIDUv3-2* uses only 9 rounds but is otherwise equivalent to AES encryption with a 128-bit key.

Otherwise, both *BAIDUv3-1* and *BAIDUv3-2* generally encrypt in the same way. In both, the following key k_f is derived according to a fixed function:

```
def derive_key():
  key = []
  x = 0
  for i in range(16):
    b = ~i ^ ((i + 11) * (x >> (i & 3)))
    key.append(b & 0xff)
    x += 1937
  return bytes(key)
```

Note that the function takes no input nor references any external state and thus always generates the same fixed, 128-bit key k_f = ⟨ff 9e d5 48 07 5a 10 e4 ef 06 c7 2e a7 a2 f2 36⟩.

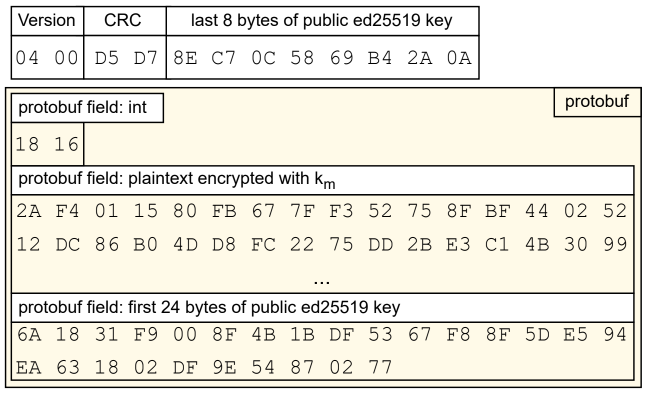
To encrypt a protobuf-serialized message, the *BAIDUv3* protocol first compresses it, forming a compressed buffer. The 32-bit, little-endian length of this compressed message is then prepended to the compressed buffer, forming the plaintext. A randomly generated 16-byte key k_m is used to encrypt the plaintext using AES in ECB mode. The resulting ciphertext is stored in bytes 44 until the end of the final UDP payload. Key k_f is used to encrypt k_m using AES in ECB mode. The resulting encrypted key is stored in bytes 28 until 44 of the final UDP payload (see Figure 4 for an illustration).

Even with modifications, both *BAIDUv3-1* and *BAIDUv3-2* are ultimately symmetric encryption schemes with no asymmetric features. Therefore, the same key used to encrypt a message can also be used to decrypt it. Since k_f is fixed, any network eavesdropper with knowledge of k_f can decrypt k_m and thus can decrypt the plaintext contents of each message encrypted in the manner described above. These encrypted messages included our typed keystrokes as well as the name of the application into which we were typing them (see Figure 5 for an example payload).

4.2.2 BAIDUv4.

The upgraded *BAIDUv4* protocol uses elliptic-curve Diffie-Hellman with a pinned server public key (pk_s) to establish a shared secret key for use in yet another modified version of AES.

Upon opening the keyboard, before the first outgoing *BAIDUv4* protocol message is sent, the application randomly generates a client Curve25519 public-private key pair (pk_c, sk_c). Then, a Diffie-Hellman shared secret k is generated using sk_c and a pinned public key pk_s. To send a message with plaintext P, the application reuses the first 16 bytes of pk_c as the initialization vector (IV) for symmetric encryption, and k is used as the symmetric encryption key. The resulting symmetric encryption of P is then sent along with pk_c



Version	CRC	last 8 bytes of public ed25519 key
04 00	D5 D7	8E C7 0C 58 69 B4 2A 0A

protobuf

protobuf field: int

18 16

protobuf field: plaintext encrypted with k_m

```
2A F4 01 15 80 FB 67 7F F3 52 75 8F BF 44 02 52
12 DC 86 B0 4D D8 FC 22 75 DD 2B E3 C1 4B 30 99
                       ...
```

protobuf field: first 24 bytes of public ed25519 key

```
6A 18 31 F9 00 8F 4B 1B DF 53 67 F8 8F 5D E5 94
EA 63 18 02 DF 9E 54 87 02 77
```

Figure 6: Wire format and sample payload for *BAIDUv4*.

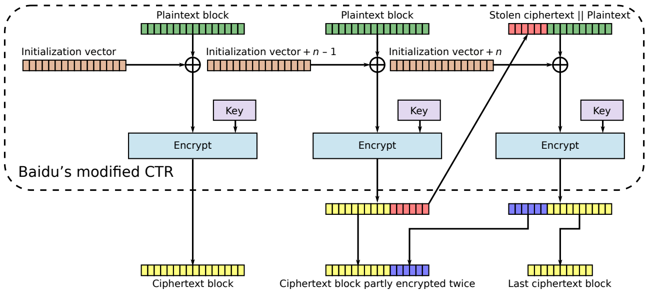


Figure 7: Illustration of BCTR mode encryption scheme used by *BAIDUv4* on Android and iOS.

to the server (see Figure 6 for an illustration). The server can then obtain the same Diffie-Hellman shared secret k from pk_c and sk_s, the private key corresponding to pk_s, to decrypt the ciphertext.

The *BAIDUv4* protocol symmetrically encrypts data using a modified version of AES from *BAIDUv3-1* and *BAIDUv3-2*. Compared to ordinary AES, *BAIDUv4* uses a built-in cipher mode and padding. This cipher mode mixes bytes differently and uses a modified counter (CTR) mode which we call Baidu CTR (BCTR) mode, illustrated in Figure 7.

Generally speaking, any CTR cipher mode involves combining an IV IV with the value i of some counter, whose combination we shall notate as $IV + i$. Most commonly, the counter value used for block i is simply i, i.e., it begins at zero and increments for each subsequent block, and AESv3's implementation follows this convention. There is no standard way to compute $IV + i$ in CTR mode, but the way that BCTR combines the IV and i is by adding i to the left-most 32-bits of IV, interpreting this portion of IV and i in little-endian byte order. If the sum overflows, then no carrying is performed on bytes to the right of this 32-bit value. The implementation details we have thus far described do not significantly deviate from a typical CTR implementation. However, where BCTR mode differs from ordinary CTR mode is in how the value $IV + i$ is used during encryption. In ordinary CTR mode, to encrypt block i with key k, you would compute

$$plain_i \oplus \text{encrypt}(IV + i, k).$$

Block	Plaintext																Ciphertext															
0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	e2	d4	00	1c	c6	5d	80	33	0c	b9	48	7d	d5	27	72	7a
1	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	e2	d4	00	1c	c6	5d	80	33	0c	b9	48	7d	d5	27	72	7a

Figure 8: When encrypted with the randomly generated 128-bit key ⟨`96 66 08 d1 6f 80 82 86 a7 b7 da 43 96 ee d1 a2`⟩ and IV ⟨`48 5b 54 92 0c 80 a6 20 29 6f 95 e5 c5 6a 3d e2`⟩ using *BAIDUv4*'s modified CTR mode, the above plaintext blocks in positions 0 and 1 encrypt to the same ciphertext.

In BCTR mode, to encrypt block i, you compute

$$\text{encrypt}(plain_i \oplus (IV + i), k).$$

As we will see later, this deviation will have implications to the security of the algorithm.

While ordinarily CTR mode does not require the final block length to be a multiple of the cipher's block size (in the case of AES, 16 bytes), due to Baidu's modifications, BCTR mode no longer automatically possesses this property but rather achieves it by employing ciphertext stealing [36]. If the final block length n is less than 16, *BAIDUv4*'s implementation encrypts the final 16 byte block by taking the last $(16 - n)$ bytes of the penultimate ciphertext block and prepending them to the n bytes of the ultimate plaintext block. The encryption of the resultant block fills the last $(16 - n)$ bytes of the penultimate ciphertext block and the n bytes of the final ciphertext block. Note, however, that this practice only works when the plaintext consists of at least two blocks. Therefore, if there exists only one plaintext block, then *BAIDUv4* right-zero-pads that block to be 16 bytes.

Key and IV re-use. Since the IV and key are both directly derived from the client key pair, the IV and key are reused until the application generates a new key pair. This only happens when the application restarts, such as when the user restarts the mobile device, the user switches to a different keyboard and back, or the IME is evicted from memory. From our testing, we have observed the same key and IV in use for over 24 hours. There are various issues that arise from key and IV reuse.

Re-using the same IV and key means that the same inputs will encrypt to the same encrypted ciphertext. Then, by definition, this construction is not CPA-secure. Additionally, due to the block cipher's construction, if blocks in the same positions of the plaintexts are the same, they will encrypt to the same ciphertext blocks. As an example, if the second block of two plaintexts are the same, the second block of the corresponding ciphertexts will be the same.

Weakness in cipher mode. While BCTR mode used by Baidu does not as flagrantly reveal patterns to the same extent as ECB mode, there do exist circumstances in which patterns in the plaintext can still be revealed in the ciphertext. Specifically, there exist circumstances in which there exists a counter-like pattern in the plaintext which can be revealed by the ciphertext. These circumstances are possible due to the fact that $(IV + i)$ is XORed with each plaintext block i and then encrypted, unlike ordinary CTR mode which encrypts $(IV + i)$ and XORs it with the plaintext. Thus, when using BCTR mode, if the plaintext exhibits similar counting patterns as $(IV + i)$, then for multiple blocks the value $((IV + i)$ XOR plaintext block $i)$ may be equivalent and thus encrypt to an equivalent ciphertext.

```
1: 0
2: 0
3: 49
4: "xxxxx"
5: 0
7 {
  1: "app_id"
  2: "100IME"
}
7 {
  1: "uid"
  2: "230817031752396418"
}
[...]
```

Figure 9: Example excerpt of recovered protobuf data encrypted by iFlytek on Android, revealing what we had typed on line 4 ("xxxxx")

More generally, BCTR mode fails to provide the cryptographic property of diffusion [44]. Specifically, if an algorithm provides diffusion, then, when we change a single bit of the plaintext, we expect half of the bits of the ciphertext to change. However, the example in Figure 8 illustrates a case where changing a single bit of the plaintext caused zero bits of the ciphertext to change, a violation of the property's expectations. Diffusion is vital so that patterns in the plaintext are not visible as patterns in the ciphertext.

Forward secrecy issues with static Diffie-Hellman. The use of a pinned server key means that the cipher is not forward secret, a property of other modern network encryption ciphers like TLS. If the server key is ever revealed, any past message where the shared secret was generated with that key can be successfully decrypted.

Lack of message integrity. There are no cryptographically secure message integrity checks, which means that a network attacker may freely modify the ciphertext. There is a CRC32 checksum calculated and included with the plaintext data, but a CRC32 checksum does not provide cryptographic integrity, as it is easy to generate CRC32 checksum collisions. Therefore, modifying the ciphertext would be possible.

4.3 *iFlytek-And*

We found that some iFlytek-related IMEs use a proprietary encryption protocol to encrypt keystrokes. We refer to this protocol as *iFlytek-And*. This protocol contains a vulnerability that allows network eavesdroppers to fully decrypt encrypted keystrokes. *iFlytek-And* transmits keystrokes via encrypted payloads of plain, unencrypted HTTP. We also observed that it transmits the current time in milliseconds since the Unix epoch as the HTTP GET parameter `"time"`, among others.

Let s be the current time in milliseconds since the Unix epoch at the time of the request. For each request, an 8-byte encryption key is then derived by first performing the following computation:

$$x = (s \bmod \texttt{0x5F5E100}) \oplus \texttt{0x1001111}.$$

The 8-byte key k is then derived from x as the lowest 8 ASCII-encoded digits of x, left-padded with leading zeroes if necessary, in big-endian order. The payload of the request is then padded with PKCS#7 and then encrypted with DES using key k in ECB mode.

Since DES is a symmetric encryption algorithm, the same key used to encrypt a message can also be used to decrypt it. k can be easily derived from s and since s is transmitted in the clear as the `"time"` parameter of every HTTP request whose payload is encrypted by k, any network eavesdropper can easily decrypt the contents of each HTTP request encrypted in the manner described above. We found users' keystrokes and the app into which they were typing were encrypted using this algorithm (see Figure 9 for an example payload).

```
1 {
  1: "8f2bc112-bbec-3f96-86ca-652e98316ad8"
  2: "android_oem_samsung_open"
  3: "8.13.10038.413173"
  4: "999"
  5: 1
  7: 2
}
2 {
  1: "\351\000"
  2: "\372\213"
}
4: "com.tencent.mobileqq"
7: "nihaocanyoureadthis"
16: 10
17 {
  3 {
    1: 1
    2: 5
  }
  5: 1
  9: 1
}
18: ""
19 {
  1: "0"
  4: "339"
}
```

Figure 10: Example protobuf message transmitted by Samsung Keyboard after typing "nihaocanyoureadthis" using the Pinyin IME.

4.4 No encryption

All but one of the 24 IMEs we studied used TLS or one of the above protocols to encrypt keystrokes. The full list of IMEs and which protocol they were using is in Table 3. The single exception was that we found that Samsung Keyboard transmits keystroke data to the following URL in the clear via HTTP POST:

http://shouji.sogou.com/web_ime/mobile_pb.php?durtot=339&h=8f2bc112-bbec-3f96-86ca-652e98316ad8&r=android_oem_samsung_open&v=8.13.10038.413173&s=&e=&i=&fc=0&base=dW5rbm93biswLjArMC4w&ext_ver=0

The keystroke data is contained in the request's HTTP payload in a protobuf serialization (see Figure 10 for an example payload).

5 Discussion

In this section we discuss the feasibility of our attacks, survey how the security community has responded to other large-scale vulnerabilities, and make recommendations to multiple stakeholders to holistically address apps' insecure transmissions.

5.1 Feasibility of attacks

Even though we disclosed the vulnerabilities to vendors, some vendors failed to fix the issues that we reported. Moreover, users of devices which are out of support or that otherwise no longer receive updates may continue to be vulnerable. As such, many users of these apps may continue to be under mass surveillance for the foreseeable future.

In Table 3, all passively decryptable protocols are particularly at risk. They require only a single AES or DES computation per block to attack. Furthermore, these protocols' traffic can be decrypted in real-time, and any historically recorded data can be decrypted at any future time without sending network transmissions that could reveal the presence of the attack.

For *EncryptWall-And* and *EncryptWall-Win*, decrypting a block requires 16 network round-trips, attacking each bit of a 16-bit block one at a time. Since blocks can be attacked in parallel, only the number of network requests, not the latency of the attack, is affected by the number of blocks, where each block requires at most 4,352 network requests to decrypt. As this attack requires sending network requests to Sogou's servers, Sogou and other network operators may be able to detect the presence of such attacks using realtime or historical data.

5.2 Related widespread risks and responses

We analyzed a broad sample of the most popular Chinese IMEs, finding that they are almost universally vulnerable to having their users' keystrokes being decrypted by network eavesdroppers. These findings suggest that insufficient, proprietary cryptography continues to be used to protect sensitive data transmitted by immensely popular apps developed by large technology companies. Our work joins others in suggesting that this issue affects at least the Chinese ecosystem but perhaps also others.

The security community has addressed widespread, prolific security issues before. For example, in light of the Heartbleed OpenSSL vulnerability, Durumeric et al. [10] performed a massive information gathering and outreach study monitoring patching behavior over time, exposing real attacks attempting to exploit the bug, and conducting a large-scale vulnerability notification experiment. Similarly, the need for monitoring proprietary cryptography use in app ecosystems is critical. However, the apps that we analyzed use no common library nor was there a single implementation flaw responsible for these vulnerabilities. While some of the IMEs did license their code from other companies, our overall findings can only be explained by a large number of developers independently making the same kind of mistake. As such, we conclude that this is an ecosystem issue that arose from the lack of oversight by a number of critical stakeholders.

More relevant is the security community response to the practical need for strong encryption of web traffic revealed by the 2013 Snowden revelations. The TLS ecosystem has largely stabilized, with CA root lists of many major browsers and OSes controlled by voting bodies such as the Certification Authority Browser Forum. Browsers warn when submitting web forms over HTTP. Furthermore, vendors such as Google have deployed certificate transparency in their browsers [29]. Researchers performed Internet-wide surveys looking for HTTPS weaknesses [9, 16, 17, 46]. Search engines prioritize HTTPS results.

Given the community's success in promoting HTTPS adoption through action from a variety of stakeholders, we believe that a similar strategy is necessary for ensuring that popular desktop and mobile applications, which handle equally sensitive information, are using sufficient network encryption. Therefore, we discuss interventions to address these ecosystem failures that could be implemented by a number of stakeholders similar to as was done with addressing insecure HTTP web traffic.

5.3 Recommendations to relevant stakeholders

Individually analyzing apps for this class of vulnerabilities and individually reporting issues discovered is limited in the scale of apps that it can fix. First, while we can attempt to manually analyze some of the most popular IMEs, we will never be able to analyze every app at large. Second, we might not be able to predict which apps to look at in the first place. For instance, before we analyzed the IMEs featured in this report, we never would have expected that their network transmissions would be so easily vulnerable to interception. In light of the limitations of the methods that we employed in this report, in the remainder of this section we discuss possibilities for how we might systematically or wholesale address apps which transmit sensitive data over networks without sufficient encryption.

Users. Users of any Sogou, Baidu, or iFlytek keyboard, including the versions that are bundled or pre-installed on operating systems, should ensure their keyboards and operating systems are up-to-date. Users with privacy concerns should not enable "cloud-based" features on their keyboards or IMEs. iOS users with privacy concerns should not enable "full Access" for their keyboards or IMEs.

IME developers. Use well-tested and standard encryption protocols, like TLS or QUIC. Make every attempt to provide features on-device without requiring transmitting sensitive data to cloud servers.

Mobile device manufacturers. Conduct security audits of third-party keyboards that you intend to pre-install by default on your operating systems.

Operating systems developers. On Android devices, installing any keyboard, regardless of whether or how it communicates with servers over the Internet, brings up a pop-up with the following:

> This input method may be able to collect all the text you type, including personal data like passwords and credit card numbers.

The wording of these warning messages is overbroad and does not necessarily help users distinguish between keyboards that transmit keystrokes over the network, keyboards that transmit keystrokes insecurely (using something other than standard TLS) over the network, and keyboards that do not transmit any data at all.

iOS devices, on the other hand, sandbox their keyboards by default. There is a "Full Access" or "open access" permission that must be explicitly granted to keyboards before they have network access, among other privileges. Without this permission, third-party keyboards cannot transmit network data. We recommend Android also adopt a more fine-grained permission model for keyboards.

Furthermore, the vulnerable apps that we studied transmit data using low level socket APIs versus higher level APIs that require the usage of TLS or HTTPS. One might desire that separate system calls be designed for TLS or HTTPS traffic in addition to the lower level socket system calls so that devices could implement an UNSAFE_-INTERNET permission that would be required for apps to use the lower level system calls while still allowing TLS-encrypted traffic for apps that do not have this permission.

While this approach may have some merit, it also has certain drawbacks. It makes sense for situations where apps are untrustworthy and the operating system is completely trustworthy, but there are common situations where the operating system could be not as or even less trustworthy than apps that it is running. One common case would be a user who is running an up-to-date app on an out of date operating system, possibly because the user's device is no longer receiving operating system updates. In such a case, the app's implementation of TLS is more likely to be secure than that of the operating system. Furthermore, a user's operating system may be compromised by malware or otherwise be untrustworthy in itself. Introducing a TLS system call would centralize the encryption of all sensitive data and grant the operating system easy visibility into all unencrypted data. In any case, innovating in areas of encryption is an important right of application developers, and it may not make sense to stifle apps like Signal because of their use of end-to-end or other novel encryption by requiring them to obtain an UNSAFE_INTERNET permission.

One might alternatively desire for apps at large to not be able to access the Internet at all. Instead of an UNSAFE_INTERNET permission, what about introducing an INTERNET permission to govern all Internet socket access, similar to the "Full Access" permission which iOS already applies to keyboard apps? Android devices in fact already have such a permission that apps must request to use Internet (AF_INET) sockets, but it is not a permission that is exposed to ordinary users either in the Google Play Store or through any stock Android user interface, and it is automatically granted when installing an app. Unfortunately, given all of the interprocess communication (IPC) vehicles on modern smart devices, restricting Internet socket access may not guarantee that the app could not communicate over the Internet (e.g., through Google Play services). GrapheneOS, an open source Android-based operating system, implements a NETWORK permission. However, denying this permission can lead to surprising results where apps can still communicate with the Internet via IPC with other apps [41]. As such, we recommend that both the developers of Android and iOS work toward a meaningful INTERNET permission that would adequately inform users of whether an app communicates over the Internet.

Application store operators. One might call on app stores to enforce the use of sufficient encryption to protect sensitive data in transit. App stores already have a number of rules that they enforce through a combination of automated and manual review. Calling on app stores to enforce sufficient encryption of in-transit sensitive data is tempting given the resources of the companies operating the app stores. However, failing any other innovation, the same scaling issues that apply to other researchers studying these apps will apply to those working for these companies.

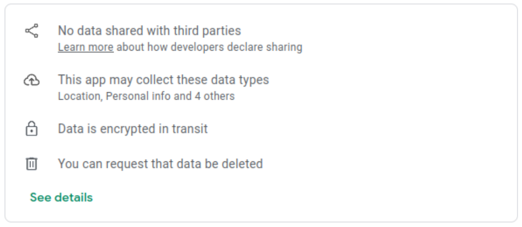

No data shared with third parties
Learn more about how developers declare sharing
This app may collect these data types
Location, Personal info and 4 others
Data is encrypted in transit
You can request that data be deleted
See details

Figure 11: An example of an attestation for Microsoft SwiftKey in Google's Play Store.

Another way for users to gain visibility into the security and privacy properties of their apps is through the use of developer attestations, such as the ones that appear in data safety sections in many popular app stores. Both the Apple App Store and the Google Play Store collect and display such attestations to varying extents, including attestations as to what data an app collects (if any) and with whom it is shared (if anyone). Additionally, the Play Store allows developers the opportunity to attest to performing "encryption in transit", as shown in Figure 11. These attestations allow users to clearly see what security and privacy properties an app's developer claims it to have and, like privacy policies, they provide means of redress if violated. We wanted to evaluate whether the apps that we analyzed lived up to their attestations concerning their encryption in the app stores in which they are available. Among the apps that we analyzed, only Baidu IME was available in the Play Store. At the time of this writing, it does not attest to its data being encrypted in transit. Although other apps that we analyzed were available in Apple's App Store, to our knowledge, this store does not display an attestation for whether the app encrypts data in transit. As such, across both the Google Play and the Apple App stores, attestations were insufficient for compelling the IMEs' developers to implement proper encryption or in providing users any opportunity for redress.

In light of the above findings, we believe that users would benefit from the following recommendations: (1) that app store operators require developers to attest to whether or not an app encrypts data in transit, (2) that app store operators display not only when developers attest to all data being encrypted in transit but also display a warning when they fail to, and (3) that app store operators require apps in certain sensitive categories, such as keyboard apps, to either positively attest to encrypting all data in transit or to attest to not transmitting any data at all.

Since most of the apps that we found perform some type of encryption, even if it were wholly inadequate, one might wonder if attesting that data is merely "encrypted" is enough, since the data arguably did have some manner of encryption applied to it during transit. The Play Store provides some guidance on this topic. Under the question — "How should I encrypt data in transit?" — the documentation notes: "You should follow best industry standards to safely encrypt your app's data in transit. Common encryption protocols include TLS (Transport Layer Security) and HTTPS."

Another issue with attestations is that they provide no guarantee that an app behaves as its developers attest, as developers can, after all, make false attestations. While we wish that attestations could guarantee that an app sufficiently implements proper cryptography to the same extent that a permission system can guarantee an app does not use a microphone, false attestations provide an opportunity for redress. For instance, apps which are found to violate attestations could be subject to removal from app stores. Furthermore, apps which violate attestations could be subject to fines by regulatory bodies such as the FTC. Finally, apps which violate the attestation could be liable to civil suits.

While the apps we analyzed were predominantly available from Chinese app stores, we equally recommend that Chinese app stores adopt these recommendations in addition to the Apple App Store and the Google Play Store. Moreover, while this report focuses on the problem of poor encryption practices in Chinese apps, the problem to varying extents applies to apps of all other provenances.

6 Conclusion

This work is the first comprehensive network security analysis of the Chinese IME keyboard ecosystem. Our results demonstrate that the vast majority utilized independently-developed, non-TLS transport encryption protocols to protect keystrokes in transit. In the majority of cases, these encryption protocols were vulnerable to decryption by a network eavesdropper, potentially revealing hundreds of millions of users' keystrokes to third parties.

This work contributes to the growing body of evidence highlighting a significant, yet understudied, class of apps that rely on insecure, custom-built cryptography. Despite the general shift toward TLS, we discovered that many highly popular cloud-based keyboard apps, which process extremely sensitive data, fall into this class and can have their encryption easily compromised by network eavesdroppers. By examining these apps, our study helps to better define this class, dispelling misconceptions that these apps are neither widely popular nor developed by major technology companies, allowing for researchers to effectively prioritize and address the unique security challenges posed by these apps.

6.1 Future work

Our results generally demonstrate a larger need to analyze Chinese apps and the Chinese Internet ecosystem at large. The Google Play Store and Apple App Store ecosystems, for instance, are commonly studied by privacy and security researchers [12, 30, 51, 53], but many Chinese app stores are overlooked, despite that many popular Chinese apps have more users than their counterparts on the Google Play Store. While the vulnerabilities we discovered were mostly nontrivial to find and took substantial analysis to attack, most could have been discovered by any skilled security researcher analyzing these apps. A researcher studying network traffic from users of Chinese devices could also have identified non-standard traffic.

While our work focused on apps from the Chinese ecosystem, there is preliminary evidence that apps from other ecosystems may be equally understudied and employ similarly problematic cryptography. As examples, a 2017 analysis of the popular Japanese-developed LINE Messenger resulted in attacks on the app's proprietary end-to-end cryptography [13], and a 2023 survey of prominent Latin American apps found multiple issues relating to the transmission of sensitive data without encryption [27].

To address the need to study understudied ecosystems more broadly, automated tools could also work to detect insecure traffic at large. Longitudinal TLS telemetry has largely been focused on web-based perspectives, and the mobile perspective is often overlooked, despite the increasing dominance of mobile traffic globally. Although there are some research projects that survey TLS usage in Android mobile apps at scale, there is no public longitudinal data from these projects (i.e., they are run as one-off studies), and many focus on the Google Play's Android ecosystem, thereby excluding the Chinese mobile Internet [15, 42]. Another direction for future work could involve public longitudinal TLS telemetry for popular mobile applications globally, via automated static or dynamic analysis at scale.

Acknowledgments

The authors thank the anonymous reviewers for their helpful feedback. We are grateful to Jedidiah Crandall, Jakub Dałek, Ron Deibert, Jonathan Mayer, and Prateek Mittal for their review and support. This work was supported by an Open Technology Fund (OTF) Information Controls Fellowship (ICFP).

References

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*. 154–171.

[2] Ashwag Albakri, Huda Fatima, Maram Mohammed, Aisha Ahmed, Aisha Ali, Asala Ali, and Nahla Mohammed Elzein. 2022. Survey on reverse-engineering tools for Android mobile devices. *Mathematical Problems in Engineering* 2022 (2022), 1–7.

[3] Harikrishnan Bhanu. 2010. *Timing side-channel attacks on SSH*. Ph. D. Dissertation. Clemson University. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2024-07-18.

[4] Hanno Böck, Juraj Somorovsky, and Craig Young. 2018. Return Of Bleichenbacher's Oracle Threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 817–849.

[5] Jin Chen, Haibo Chen, Erick Bauman, Zhiqiang Lin, Binyu Zang, and Haibing Guan. 2015. You Shouldn't Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 657–690.

[6] Jakub Dalek, Katie Kleemola, Adam Senft, Christopher Parsons, Andrew Hilts, Sarah McKune, Jason Q Ng, Masashi Crete-Nishihata, John Scott-Railton, and Ron Deibert. 2015. *A chatty squirrel: Privacy and security issues with uc browser*. Technical Report. The Citizen Lab.

[7] Anthony Desnos and Geoffroy Gueguen. 2011. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi* 1 (2011), 1–24.

[8] Alexander Druffel and Kris Heid. 2020. Davinci: Android app analysis beyond frida via dynamic system call instrumentation. In *Applied Cryptography and Network Security Workshops: ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, Proceedings 18*. Springer, Springer, Cham, Switzerland, 473–489.

[9] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. 2013. Analysis of the HTTPS certificate ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference* (Barcelona, Spain) *(IMC '13)*. Association for Computing Machinery, New York, NY, USA, 291–304.

[10] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488.

[11] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 73–84.

[12] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*.

[13] Antonio M. Espinoza, William J. Tolley, Jedidiah R. Crandall, Masashi Crete-Nishihata, and Andrew Hilts. 2017. Alice and Bob, who the FOCI are they?: Analysis of end-to-end encryption in the LINE messaging application. In *7th USENIX Workshop on Free and Open Communications on the Internet (FOCI 17)*. USENIX Association, Vancouver, BC.

[14] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why eve and mallory love android: an analysis of android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 50–61.

[15] David Sounthiraraj Justin Sahs Garret Greenwood and Zhiqiang Lin Latifur Khan. 2014. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Network and Distributed System Security Symposium (NDSS). Internet Society, San Diego, CA.* 1–14.

[16] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kaafar. 2016. TLS in the Wild: An Internet-wide Analysis of TLS-based Protocols for Electronic Communication. In *Proceedings 2016 Network and Distributed System Security Symposium (NDSS 2016)*. Internet Society.

[17] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. 2011. The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (Berlin, Germany) *(IMC '11)*. Association for Computing Machinery, New York, NY, USA, 427–444.

[18] Lanlan Huang and Xiaoyi Lin. 2021. Chinese keyboard apps on the spot over suspicion of privacy violation. https://www.globaltimes.cn/page/202101/1214202.shtml

[19] Mara Hvistendahl. 2020. How a Chinese AI Giant made chatting — and surveillance — easy. https://www.wired.com/story/iflytek-china-ai-giant-voice-chatting-surveillance/

[20] iiMedia. 2023. https://www.iimedia.cn/c400/92144.html

[21] Junpei Kawamoto and Kouichi Sakurai. 2014. Privacy-aware cloud-based input method editor. In *2014 IEEE/CIC International Conference on Communications in China (ICCC)*. IEEE, Shanghai, China, 209–213.

[22] Jeffrey Knockel, Sarah McKune, and Adam Senft. 2016. *Baidu's and don'ts: privacy and security issues in Baidu browser.* Technical Report. The Citizen Lab.

[23] Jeffrey Knockel, Thomas Ristenpart, and Jedidiah R. Crandall. 2018. When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users. *CoRR* abs/1802.03367 (2018). arXiv:1802.03367 http://arxiv.org/abs/1802.03367

[24] Jeffrey Knockel, Adam Senft, and Ron Deibert. 2016. A Tough Nut to Crack: A Further Look at Privacy and Security Issues in UC Browser.

[25] Jeffrey Knockel, Adam Senft, and Ronald Deibert. 2016. *WUP! There It Is: Privacy and Security Issues in QQ Browser. Citizen Lab report.* Technical Report. The Citizen Lab.

[26] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. 2013. On the Security of the TLS Protocol: A Systematic Analysis. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 429–448.

[27] Beau Kujath, Jeffrey Knockel, Paul Aguilar, Diego Morabito, Masashi Crete-Nishihata, and Jedidiah R. Crandall. 2024. Analyzing Prominent Mobile Apps in Latin America. In *Workshop on Free and Open Communications on the Internet (FOCI 2024)*. Privacy Enhancing Technologies Symposium, Bristol, UK, 83–92.

[28] McAfee Labs. 2015. Apps Sending Plain HTTP Put Personal Data at Risk. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/apps-sending-plain-http-put-personal-data-risk/

[29] Ben Laurie. 2014. Certificate Transparency: Public, verifiable, append-only logs. *Queue* 12, 8 (aug 2014), 10–19. https://doi.org/10.1145/2668152.2668154 (see also https://certificate.transparency.dev/).

[30] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, Florence, Italy, 280–291.

[31] Shumin Liao. 2021. iFlytek Sinks After Chinese AI Firm's Virtual Keyboard Is Pulled From App Stores. https://www.yicaiglobal.com/news/iflytek-sinks-after-chinese-ai-firm-virtual-keyboard-is-pulled-from-app-stores

[32] Kaizheng Liu, Ming Yang, Zhen Ling, Huaiyu Yan, Yue Zhang, Xinwen Fu, and Wei Zhao. 2021. On Manually Reverse Engineering Communication Protocols of Linux-Based IoT Systems. *IEEE Internet of Things Journal* 8, 8 (2021), 6815–6827.

[33] Bill Marczak and John Scott-Railton. 2020. *Move Fast and Roll Your Own Crypto: A Quick Look at the Confidentiality.* Technical Report. The Citizen Lab.

[34] Nate Mathews, James K Holland, Se Eun Oh, Mohammad Saidur Rahman, Nicholas Hopper, and Matthew Wright. 2023. SoK: A Critical Evaluation of Efficient Website Fingerprinting Defenses. In *2023 IEEE Symposium on Security and Privacy (SP)*. 969–986.

[35] Christopher Meyer and Jörg Schwenk. 2014. SoK: Lessons Learned from SSL/TLS Attacks. In *Information Security Applications*, Yongdae Kim, Heejo Lee, and Adrian Perrig (Eds.). Springer International Publishing, Cham, 189–209.

[36] Carl H. Meyer and Stephen M. Matyas. 1982. *Cryptography: a new dimension in computer data security.* John Wiley & Sons, New York, NY, USA.

[37] Thomas S. Mullaney. 2012. The Moveable Typewriter: How Chinese Typists Developed Predictive Text during the Height of Maoism. *Technology and Culture* 53, 4 (2012), 777–814.

[38] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. 2015. To Pin or Not to Pin—Helping App Developers Bullet Proof Their TLS Connections. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 239–254.

[39] Lucky Onwuzurike and Emiliano De Cristofaro. 2015. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (New York, New York) *(WiSec '15)*. Association for Computing Machinery, New York, NY, USA, Article 15, 6 pages.

[40] Palo Alto Networks. 2013. Is Baidu Secretly Collecting Japanese User Data? https://www.paloaltonetworks.com/blog/2013/12/baidu-secretly-collecting-japanese-user-data/

[41] David Prock. 2023. GrapheneOS Network Permissions not being honored. https://github.com/GrapheneOS/os-issue-tracker/issues/2810

[42] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Philippa Gill. 2018. Studying TLS Usage in Android Apps. In *Proceedings of the Applied Networking Research Workshop* (Montreal, QC, Canada) *(ANRW '18)*. Association for Computing Machinery, New York, NY, USA, 5.

[43] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clementine Lucie Noemie Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *Network and Distributed System Security Symposium 2018*. 15. Network and Distributed System Security Symposium 2018, NDSS'18 ; Conference date: 18-02-2018 Through 21-02-2018.

[44] Claude E Shannon. 1945. *A Mathematical Theory of Cryptography.* Technical Report. Bell Telephone Labs.

[45] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. 2001. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *10th USENIX Security Symposium (USENIX Security 01)*. USENIX Association, Washington, D.C.

[46] Drew Springall, Zakir Durumeric, and J. Alex Halderman. 2016. Measuring the Security Harm of TLS Crypto Shortcuts. In *Proceedings of the 2016 Internet Measurement Conference* (Santa Monica, California, USA) *(IMC '16)*. Association for Computing Machinery, New York, NY, USA, 33–47.

[47] Masaki Suenaga. 2005. IME as a possible keylogger. https://www.virusbulletin.com/virusbulletin/2005/11/ime-possible-keylogger/

[48] The Five Eyes. 2012. Synergising Network Analysis Tradecraft. https://www.eff.org/document/20150521-cbc-synergising-network-analysis-tradecraft

[49] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS.... In *Advances in Cryptology — EUROCRYPT 2002*, Lars R. Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 534–545.

[50] Websense Security Labs ThreatSeeker Network. 2010. Fake Input Method Editor (IME) Trojan. https://blog.pages.kr/739

[51] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1043–1054.

[52] Archie Zhang. 2024. China Smartphone Market Analysis 2023. https://www.counterpointresearch.com/research_portal/china-smartphone-market-analysis-2023-q4/

[53] Hengshu Zhu, Hui Xiong, Yong Ge, and Enhong Chen. 2014. Mobile app recommendations with security and privacy awareness. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 951–960.