15. Juni 2020 Studie · Mobile Sicherheit · Web & Cloud · 13 Min. Lesezeit

Messung Timing-Unterschiede in der Android API

Archiviert. Dieser Inhalt wird nicht mehr aktualisiert. Er bleibt zur Information verfügbar.

Stand 2020-06-08
Quelle https://technology2.a-sit.at/veroeffentlichungen/messung-timing-unterschiede-in-der-android-api/

Der Bericht stellt ein Framework vor, welches Timing-Lecks auf Android-Geräten automatisch erkennen kann. Dazu ruft es automatisiert Methoden der API mit unterschiedlichen Parametern auf und misst die Ausführungszeit der verschiedenen Aufrufe. Unterscheiden sich diese gemittelt über mehrere Aufrufe signifikant, so kann möglicherweise auf für den Aufrufer eigentlich nicht zugreifbare Informationen geschlossen werden. Der Bericht zeigt anhand zweier Beispiele wie durch solche Timing-Unterschiede das Berechtigungssystem vom Android teilweise umgangen werden kann.

1. Einführung

Auf Smartphones ist eine Vielzahl an zu schützenden Daten gespeichert, sowohl privater als auch geschäftlicher Natur. Gleichzeitig werden jedoch auch Applikationen von vielen verschiedenen Entwicklern verwendet, die in der Regel keinen Zugriff auf vertrauliche Daten haben sollten. Um diese Daten vor unberechtigtem Zugriff zu schützen, verwendet das Smartphone-Betriebssystem Android ein Sandboxing-Konzept. Dadurch verhindert Android, dass Applikationen auf die Daten anderer Applikationen bzw. des Benutzers/der Benutzerin zugreifen können. Zur Funktion mancher Applikationen ist ein solcher Zugriff jedoch erforderlich. Android regelt den Zugriff auf Benutzerdaten deshalb mit einem Berechtigungssystem. Dieses regelt auch den Zugriff auf sensible Ressourcen wie Kamera oder Mikrophon. Für den Schutz der Privatsphäre der Nutzerin bzw. des Nutzers ist deshalb ein Funktionieren dieser Abgrenzungsmechanismen essenziell. In diesem Bericht wird © A-SIT, Partner für Sicherheitsfragen untersucht, ob durch Timing-Unterschiede beim Aufruf von Methoden der Android API das Berechtigungssystem von Android partiell umgangen werden kann. Dazu wurde ein Framework erstellt, welches diese Timing-Unterschiede erkennen kann. Auf Basis einer Liste von Klassen und Methoden erstellt es die benötigten Objekte und ruft die Methoden automatisiert auf. Dabei wird gezielt auf Methoden gesetzt, welche auf geschützte Informationen zugreifen. Die Methoden werden jeweils mit gültigen vordefinierten Parametern aufgerufen. Anhand dieser Parameter erstellt das Framework adaptierte Parameter mit denen die Methode zusätzlich zu den vordefinierten Parametern aufgerufen wird. Dabei wird die Ausführungszeit der beiden Aufrufe gemessen. Unterscheidet sich diese merkbar, so liegt eine potentielle Umgehung einer Berechtigung vor.

2. Hintergrund

In den folgenden Abschnitten werden verwandte Arbeiten vorgestellt und ein Einblick in Androids Berechtigungssystem gegeben.

2.1. Verwandte Arbeiten

Timing-Unterschiede stellten bereits in verschiedenen Bereichen ein ernstzunehmendes Problem dar. Vor allem im kryptographischen Bereich konnten dadurch oft Schutzkonzepte umgangen werden. So wurde beispielsweise gezeigt, dass durch Timing-Angriffe sogar private Schlüssel verschiedener Systeme herausgefunden werden konnten [1] [2]. Auch auf Hardware-Ebene können Timing-Unterschiede, die beispielsweise durch mikroarchitekturelle Unterschiede hervorgerufen werden, den Rückschluss auf den internen Zustand und so auf die verarbeiteten Daten ermöglichen [3]. Auch in Software-Systemen können Schutzmechanismen durch Timing-Messungen umgangen werden. So wurde beispielsweise in [4] gezeigt, dass Schutzfunktionen des Betriebssystems wie Address Space Layout Randomisation (ASLR) umgangen werden können. Weiters wurde in [5] gezeigt, dass durch Messung von Timing-Unterschieden installierte Betriebssysteme und Programme in benachbarten virtuellen Maschinen erkannt werden können, sofern das verwendete Dateisystem Copy-on-Write verwendet.

Auch im Smartphone-Bereich konnten durch Zeitmessungen Schutzkonzepte umgangen werden. Zhang et al. [6] stellten fest, dass die Dateisystem-API unter iOS anfällig für Timing-Angriffe ist. Es wurde gezeigt, dass durch Timing-Unterschiede bei der Ausführung von Dateisystemoperationen auf installierte Programme geschlossen werden kann. So erstellen alle Applikationen, welche Push-Benachrichtigungen oder Schnellzugriffe zur Verfügung stellen, Dateien auf dem Smartphone, die für Drittprogramme nicht direkt zugreifbar sind. Versucht man jedoch über die Dateisystem-API darauf zuzugreifen, so unterscheidet sich die Ausführungszeit, je nachdem ob eine Applikation installiert ist oder nicht (sofern die Applikation eine der genannten Funktionen nutzt). Weiters zeigten Diao et al. [7], dass durch Messung von Interrupt-Timings auf aktuell ausgeführte Applikationen sowie das Entsperrungsmuster gefolgert werden kann.

2.2. Androids Berechtigungssystem

Um Sandboxing umzusetzen, verwendet Android verschiedene Funktionen des eingesetzten Linux-Kernels. Dazu wird jeder Applikation eine Benutzer-ID (UID) zugeordnet. Seit Android 4.3 verwendet Android zusätzlich SELinux (security-enhanced Linux), um die Sandbox zu verbessern. Bei SELinux handelt es sich um ein Kernel-Modul, welches Mandatory Access Control (MAC) im Linux Kernel umsetzt. Dabei setzt der Kernel definierte Regeln für die einzelnen Applikationen durch, um Zugriff auf sensible Informationen und Funktionen zu verhindern.

Damit Applikationen auf Ressourcen außerhalb der Sandbox zugreifen können, verwendet Android ein Berechtigungssystem, welches die Ressourcen in verschiedene Berechtigungskategorien einteilt. Hierbei wird grob zwischen drei verschiedenen Berechtigungstypen unterschieden. Diese unterscheiden sich nach ihrem Schutzlevel und den Applikationen die sie anfordern dürfen. Die drei Schutzlevel sind laut Entwicklungsdokumentation [8] wie folgt:

- **Normale Berechtigungen** : Hierbei handelt es sich um das niedrigste Schutzlevel. Fordert eine Applikation eine solche Berechtigung an, so wird sie vom System automatisch zum Zeitpunkt der Installation gewährt. Die Berechtigung kann der Applikation durch die Nutzerin/den Nutzer nicht

entzogen werden.

- **Signatur Berechtigungen** : Diese Berechtigungen werden ebenso wie normale Berechtigungen durch das System zum Installationszeitpunkt gewährt. Dies jedoch nur, wenn die Applikation mit demselben Zertifikat signiert wurde wie die Applikation, welche die Berechtigung anbietet.

- **Gefährliche Berechtigungen** : Berechtigungen dieses Typs schützen Bereiche, welche besonders zu schützende, private Informationen der Nutzerin oder des Nutzers beinhalten oder welche anderen Applikationen beeinflussen können. Dies beinhaltet beispielsweise den Zugriff auf Kontakte oder die Kamera. Diese Berechtigungen müssen explizit durch die Nutzerin oder den Nutzer freigegeben werden. Ebenso können sie auch im Nachhinein der Applikation wieder entzogen werden.

Zusätzlich zu den drei beschriebenen Typen gibt es auch noch weitere sogenannte Sonderberechtigungen. Hierbei handelt es sich um Berechtigungen für besonders sensitive Informationen, welche über einen eigenen Einstellungsdialog freigegeben werden müssen. Diese Gruppe beinhaltet beispielsweise die Möglichkeit Systemeinstellungen zu ändern oder detaillierte Nutzungsstatistiken abzurufen.

3. Methodik

Um Timing-Unterschiede in der Android API zu erkennen, wurde eine Applikation erstellt, welche die Android API systematisch nach Timing-Unterschieden untersucht. Der Vorgang lässt sich in zwei Phasen einteilen. Zuerst werden in der Vorbereitungsphase verschiedene Informationen gesammelt, welche von der Applikation benötigt werden, jedoch zum Ausführungszeitpunkt der Applikation nicht direkt abrufbar sind. Diese werden vorab in Listen aufbereitet und mit der Applikation mitgeliefert. In der zweiten Phase, der Erhebungsphase, findet dann die eigentliche Suche nach für Timing-Attacken anfälligen Methoden statt. Hierbei werden zuerst die mitgelieferten Informationen geparst und aufbereitet und dann die Methoden aufgerufen und die Ausführungszeit gemessen. Auffällige Methoden werden für die weitere Analyse markiert.

3.1. Vorbereitungsphase

In der Vorbereitungsphase werden Daten aus der Android API Dokumentation gesammelt, welche während der Ausführung der Applikation nicht abrufbar sind. Der Abruf dieser Informationen läuft analog zu [9]. Für das Parsing der jeweiligen Werte verweisen wir auf die Lektüre dieses Werkes. Im Folgenden wird auf die Verwendung der Informationen näher eingegangen. Alle vorab gesammelten Informationen werden mit der Erkennungsapplikation mitgeliefert.

Um die Methoden mit gültigen und adaptierten Parametern aufrufen zu können, müssen gültige Parameter vordefiniert werden können. Dazu ist eine Zuordnung des vordefinierten Wertes mit dem Parameter einer Methode notwendig. Diese Zuordnung erfolgt hierbei über den Parameternamen. Der Parametername ist jedoch nicht zur Laufzeit mittels Reflection eruierbar. Die jeweilige API `Parameter.getName()` gibt unter Android nur generische nummerierte Parameternamen (`arg0`, `arg1`, ..., `argN`) zurück. Dies macht es unmöglich, Parameter anhand des Parameternamen direkt auf dem Gerät zuzuordnen. Es könnte zwar eine Zuordnung anhand der Position des Parameters durchgeführt werden, dieses Vorgehen ist jedoch bei der Erstellung der vordefinierten Werte aufwändiger und fehleranfälliger. Ebenso wäre es nicht möglich, denselben vordefinierten Wert gleichzeitig mehreren Methoden, welche Parameter mit demselben Namen aufweisen, zuzuweisen. So gibt es beispielsweise in der Klasse `android.accounts.AccountManager`

sechs verschiedene Methoden, welche einen Parameter namens `accountType` aufweisen, die dieselbe Semantik vorweisen. Um diese Zuordnungen zu ermöglichen, werden Informationen vorab aus der Dokumentation geparst. So wird eine Liste erstellt, welche alle Klassen der API, deren Methoden sowie den jeweiligen Parameternamen enthält. Dadurch kann das Framework die Verbindung zwischen vordefinierten Werten und den jeweiligen Parametern herstellen. Ebenso macht es das Framework möglich, einen vordefinierten Wert nur einer Methode, allen Methoden einer Klasse oder auch allen Methoden der API, die den jeweiligen Parameternamen haben, zuzuordnen.

Zusätzlich zu den manuell definierten Parameterwerten werden auch Konstanten aus der Dokumentation übernommen. Da diese nicht immer eindeutig erkennbar sind, werden alle potentiellen Konstanten geparst. Etwaige Nicht-Konstanten werden durch die Applikation nach fehlgeschlagener Interpretation verworfen.

3.2. Erkennungsphase

In der Erkennungsphase wird nach möglichen Timing-Schwachstellen in der Android API gesucht. Dazu werden zuerst die vorab geparsten Informationen geladen und Konstanten interpretiert. Sind alle Informationen geladen, so werden die Methoden aufgerufen. Um eine Methode aufzurufen, werden zuerst die vordefinierten Parameter abgerufen und adaptiert. Diese Adaptierung soll sicherstellen, dass die Methode einmal mit gültigen und einmal mit zufälligen Werten aufgerufen wird.

3.2.1. Initialisierung der Applikation

Bei der Initialisierung werden die vordefinierten Parameter sowie geparste Konstanten interpretiert und alle benötigten Objekte erstellt.

Interpretation vordefinierter Parameter

Vordefinierte Werte und Konstanten werden mit dem BeanShell Script Interpreter [10] interpretiert. Dieser gibt das Auswertungsergebnis nach Interpretation des vordefinierten Wertes bzw. der Konstante zurück. Fehlerhafte Konstanten können in diesem Schritt verworfen werden.

Erstellung von (Parameter-)Objekten

Bei der Evaluierung werden Methoden betrachtet, welche durch Änderung von Parameterwerten die Ausführungszeit beeinflussen. Um diese Methoden aufzurufen müssen die entsprechenden Objekte erstellt werden. Für die vorab definierten Werte wird dabei auf die im vorherigen Schritt interpretierten Werte zurückgegriffen. Da es mehrere Methoden mit dem gleichen Namen aber unterschiedlicher Parameteranzahl geben kann, werden diese Werte werden anhand des Typs und der Position des Parameters den vordefinierten Parameterobjekten zugeordnet. Parameter eines primitiven Typs, die nicht spezifisch vordefiniert wurden, werden durch generische Werte für den jeweiligen Datentyp ersetzt. Um nicht-primitive Objekte zu erstellen werden die Konstruktoren der Klasse aufgerufen. Analog zu den Methoden werden auch für die Konstruktoren die Parameter nach dem genannten Schema erstellt.

3.2.2. Methodenaufruf

Manche Klassenobjekte werden in der Android API nicht durch Aufruf des jeweiligen Konstruktors erstellt, sondern durch Aufruf von speziellen Methoden abgerufen. Deswegen werden zusätzlich zu Methoden mit Parametern auch jene Methoden aufgerufen, die mit Informationsbeschaffung verbunden werden und die ein komplexes Objekt zurückgeben. Die Auswahl nach dem ersten Kriterium erfolgt hierbei durch das Präfix des Methodennamens. Dies bedeutet, dass Methoden aufgerufen werden, deren Name mit `get` oder `query` beginnt. Das zurückgegebene Objekt wird dann nach weiteren Methoden durchsucht. Dieser Vorgang wird fortgesetzt, bis die definierte Aufruftiefe erreicht wurde. In den Experimenten wurde die Aufruftiefe auf maximal zwei Ebenen beschränkt.

Die dabei erhaltenen Objekte müssen ebenso wie die per Konstruktor erstellten Objekte eine der geparsten Klassen zugeordnet werden, um die Zuordnung der vordefinierten Parameter auch hier zu ermöglichen. Dieser Abgleich erfolgt über den Klassennamen des erhaltenen Objektes. Sollte die Klasse nicht in den geparsten Klassen enthalten sein, wird zusätzlich überprüft, ob es sich beim erhaltenen Objekt um ein Objekt handelt, das von einer geparsten Klasse abgeleitet wurde. In diesem Fall werden die Informationen der Basisklasse verwendet, um zumindest für die Methoden der Basisklasse eine Zuordnung zu ermöglichen. Sind alle benötigten Objekte vorhanden und Informationen zugeordnet, so kann mit dem Aufruf der Methoden begonnen werden.

Vor dem Aufruf der Methode wird mittels `System.nanoTime()` der aktuelle Wert des präzisesten auf dem System vorhandenen System-Timers abgerufen. Dieser Wert wird zwischengespeichert. Danach wird direkt die Methode aufgerufen. Nach dem Aufruf der Methode wird erneut `System.nanoTime()` aufgerufen. Aus der Differenz der beiden Werte ergibt sich die Ausführungszeit der Methode. Dieser Vorgang wird jeweils zehnmal mit den gültigen Parametern und zehnmal mit den adaptierten Parametern wiederholt. Dies deshalb, um mögliche Ausreißer zu kompensieren. Aus den gesammelten Ausführungszeiten wird der Mittelwert gebildet. Unterscheidet sich dieser um einen Schwellenwert (dieser wurde nach empirischen Versuchen auf 10% festgelegt) so wird die Methode als potentiell anfällig für Timing-Attacken markiert. Um das Ergebnis zu verifizieren wird daraufhin der oben beschriebene Prozess mit einer höheren Wiederholungsanzahl wiederholt. Um die Ausführungszeit des Frameworks zu beschränken, wird diese hohe Wiederholungsanzahl nur für potentiell anfällige Methoden angewandt. Bestätigt sich das Ergebnis, so wird die Methode zur manuellen Begutachtung markiert.

Zusätzlich zu normalen Methoden, die etwaige Resultate als Rückgabewerte zurückliefern, werden auch Methoden unterstützt, welche Callbacks zur Datenübergabe verwenden. Callbacks werden in Java in der Regel als Interface ausgestaltet, welches durch den Aufrufer implementiert wird. Dieses Interface enthält in der Regel eine oder mehrere Methoden, welche vom System aufgerufen werden sobald ein Resultat vorhanden ist. In der Regel werden solche Callbacks von Methoden verwendet, welche länger laufende Aufgaben starten. Dies kann beispielsweise Dateisystem- oder Netzwerkoperationen umfassen. Dadurch kann der Aufrufer zwischenzeitlich andere Aufgaben erledigen, bis ein Ergebnis vorhanden ist. Das heißt aber auch, dass ein Aufruf von `System.nanoTime()` nach dem Aufruf der Methode nicht die Ausführungszeit des angestoßenen Vorgangs messen würde. Zur Messung der tatsächlichen Ausführungszeit muss die Endzeit bei Aufruf des Callbacks eruiert werden. Dazu muss dieses mittels Reflection so implementiert werden, dass diese Messung möglich ist. Um ein (Callback-)Interface dynamisch zur Laufzeit zu implementieren, wird mit Hilfe von Reflection ein Proxy Objekt erstellt, welches

vom Typ des Callback-Interfaces ist. Wird eine Methode des Proxy Objekts aufgerufen, so wird dieser Aufruf an einen Listener weitergeleitet. Dieser empfängt alle Aufrufe der Methoden des Interfaces. Dadurch kann die Endzeit direkt beim Aufruf des Listeners gemessen werden.

4. Ergebnisse

In den folgenden Abschnitten wird auf den Testaufbau und auf die Ergebnisse näher eingegangen.

4.1. Testaufbau

Das folgende Experiment wurden auf einem Samsung Galaxy S9-Smartphone mit 64 GB Speicher durgeführt. Das Smartphone wurde auf die neueste zum Zeitpunkt des Projekts verfügbare Softwareversion aktualisiert. Dadurch läuft auf dem Gerät Android 10 mit Sicherheitspatches vom

1. April 2020.

4.2. Resultate

4.2.1. Erkennen von Konten

Während vor Android 8 die Berechtigung GET_ACCOUNTS benötigt wurde, um eingerichtete Konten abzurufen, geht dies seit Android 8 nur mit Einwilligung der Nutzerin bzw. des Nutzers oder wenn die Signatur des Abrufers mit der des Erstellers übereinstimmt. Die jeweiligen Konten können dann mit dem `android.accounts.AccountManager` abgerufen werden. Um das Konto abzurufen wird der Methode `getAccountsByType` der Typ des Kontos (z.B. `bitfire.at.davdroid`) als String übergeben. Ist ein Konto des angegebenen Typs vorhanden, so dauert die Ausführung der Methode länger, als wenn diese nicht vorhanden ist. Die Ausführungszeit über mehrere Aufrufe der Methode ist in Abbildung 1 dargestellt.

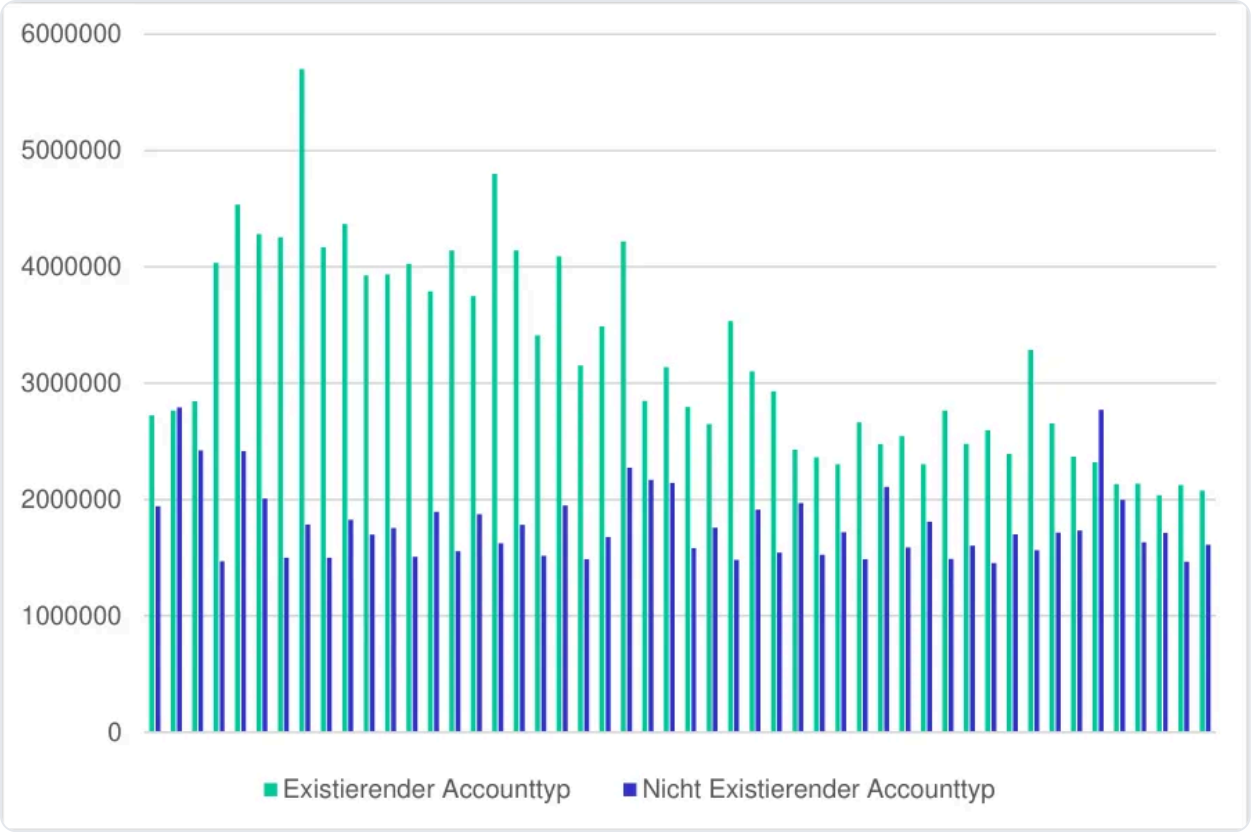


Abbildung 1: Ausführungszeit der Methode `getAccountsByType` mit existierendem und nicht-existierendem Accounttyp als Parameter.

4.2.2. Erkennen von gespeicherten Dateien

Um Dateien zu erkennen, die auf dem internen Speicher des Smartphones abgelegt sind, wird die Berechtigung READ_EXTERNAL_STORAGE benötigt. Hierbei handelt es sich um eine gefährliche Berechtigung, das heißt die Nutzerin bzw. der Nutzer muss explizit zustimmen, bevor eine Applikation diese Berechtigung nutzen kann.

Über die `android.media.MediaScannerConnection` kann auf den in Android integrierten Media Scanner Service zugegriffen werden. Dieser indiziert Medien die auf einem Android-Smartphone abgespeichert sind. Über eine `MediaScannerConnection` kann der Media Scanner dazu aufgefordert werden, eine neu hinzugefügte oder heruntergeladene Mediendatei zu indizieren. Die hinzugefügten Pfade können als Parameter der Methode `scanFile` angegeben werden. Sobald die Datei erfolgreich vom Media Scanner eingescannt wurde wird das ebenso als Parameter angegebene Callback aufgerufen. Dieser Methode kann jedoch nicht nur wie in der Dokumentation angegeben durch die Applikation erstellte oder heruntergeladene Dateien übergeben werden, sondern jegliche durch den Media Scanner zugängliche Datei. Je nachdem, ob die Datei existiert oder nicht, dauert der Aufruf unterschiedlich lange. Existiert die Datei, so dauert der Aufruf in der Regel länger, als wenn diese nicht existiert, wie in Abbildung 2 ersichtlich ist.

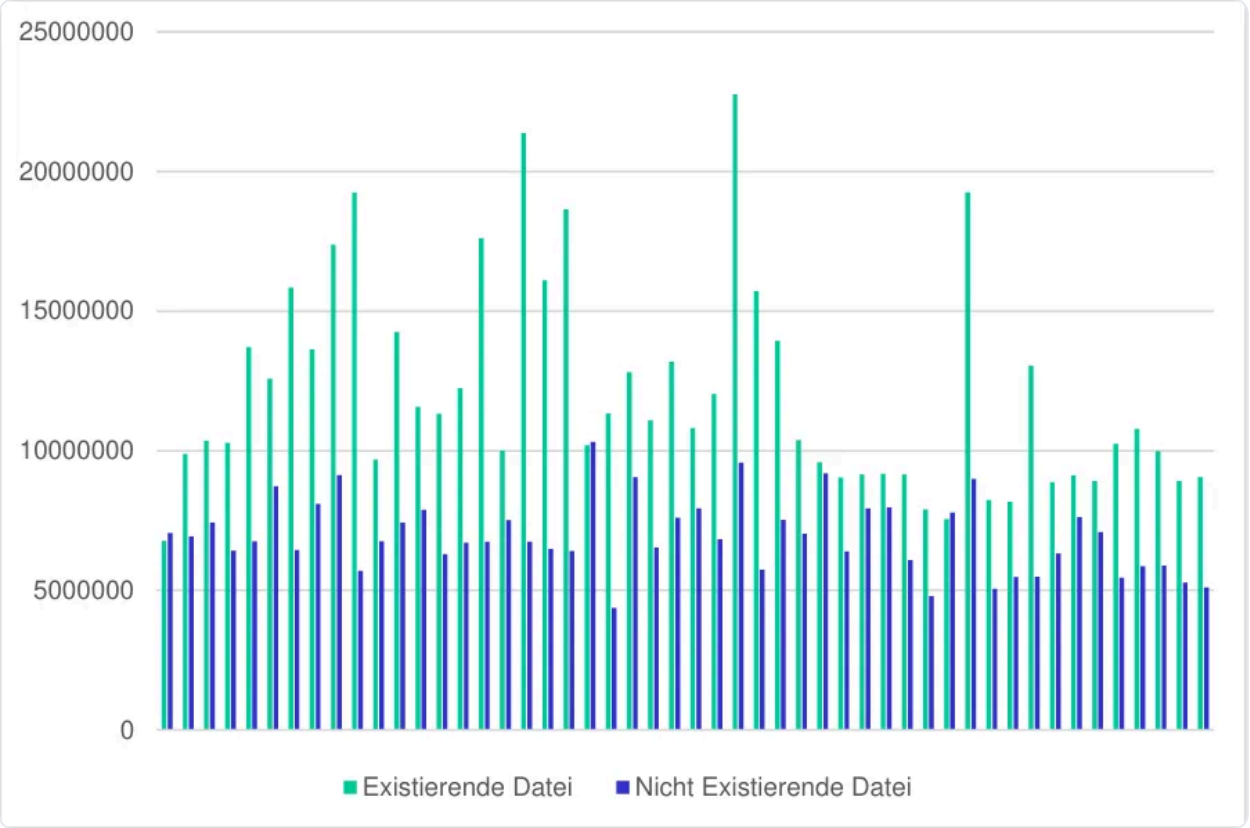


Abbildung 2: Ausführungszeit des Media Scanner mit existierendem und nichtexistierendem Pfad als Parameter.

5. Fazit

In diesem Bericht wurde ein Framework vorgestellt, welches Timing-Unterschiede in der Android API aufdecken kann. Das Framework automatisiert dabei viele Schritte, die zum Auffinden solcher Schwachstellen notwendig sind. Eine Herausforderung stellte hierbei jedoch das automatische Vordefinieren von Parametern dar. Durch das Framework konnten zwei Timing-Unterschiede in Methoden der Android API erkannt werden, welche es erlauben ansonsten geschützte Informationen herauszufinden. So kann ein Angreifer ohne Berechtigung herausfinden, ob spezifische Daten auf dem Smartphone abgespeichert sind oder ob bestimmte Applikationen benutzt werden und Konten damit eingerichtet wurden. Damit kann der Angreifer Informationen über die Nutzung des Smartphones herausfinden, welche ohne Berechtigung nicht zugänglich sein sollten.

Referenzen

[1] D. Brumley und D. Boneh, „Remote Timing Attacks Are Practical," in *USENIX Association*, 2003.

[2] B. B. Brumley und N. Tuveri, „Remote Timing Attacks Are Still Practical," in *ESORICS - 16th European Symposium on Research in Computer Security*, 2011.

[3] Q. Ge, Y. Yarom, D. Cock und G. Heiser, „A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," in *J. Cryptographic Engineering*, 2018.

[4] R. Hund, C. Willems und T. Holz, „Practical Timing Side Channel Attacks against Kernel Space ASLR," in *IEEE Symposium on Security and Privacy*, 2013.

[5] G. Palfinger, B. Prünster und D. Ziegler, „Prying CoW: Inferring Secrets across Virtual Machine Boundaries," in *SECRYPT*, Prag, 2019.

[6] X. Zhang, X. Wang, X. Bai, Y. Zhang und X. Wang, „OS-level Side Channels without Procfs:

Exploring Cross-App Information Leakage on iOS," in *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[7] W. Diao, X. Liu, Z. Li und K. Zhang, „No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[8] Google, „Permissions Overview - Normal permissions - Android Developers," [Online].

Available: https://developer.android.com/training/permissions/requesting#normal-dangerous.

[Zugriff am 30 03 2020].

[9] G. Palfinger, „Systematische Analyse von Android auf Fingerprintbarkeit," 2020.

[10] BeanShell, „BeanShell scripting language," [Online]. Available:

https://github.com/beanshell/beanshell. [Zugriff am 19 02 2020].