Die Messung des Energieverbrauchs - Welche Möglichkeiten und Risiken birgt es?

Jean-Frédéric Vogelbacher
Friedrich-Alexander Universität Erlangen/Nürnberg

ZUSAMMENFASSUNG

Heute sind mobile Endgeräte selbstverständlich und an vielen Stellen gar nicht mehr wegzudenken. Doch einer der größten limitierenden Faktoren ist die Stromversorgung mittels Akku/Batterie. Dieses Papier soll einen Überblick über die Hürden und Techniken geben, wie der Energieverbrauch eines Prozesses und verschiedener anderer Komponenten gemessen werden kann. Dabei wird auf einen möglichen theoretischen Aufbau eines Frameworks eingegangen, welches für den alltäglichen Gebrauch konzipiert ist. Auch wird *Power Sandbox* und dessen Funktionsweise vorgestellt, welches bekannte Probleme wie die Leistungs-Verschränkung einzelner Applikationen löst, und versucht, den Stromverbrauch eines Prozesses möglichst genau zu bestimmen. Doch je genauer der Energieverbrauch gemessen werden kann, umso mehr kann auf das Verhalten von Prozessen und auf private Daten geschlossen werden.

1 EINLEITUNG

In der modernen Welt sind mobile Endgeräte ein fester Bestandteil vieler Menschen geworden. Sie ersetzen mittlerweile die eigene Kamera, Taschenlampen, Telefone, dicke Notizbücher mit Kalender, Wecker, Navigationsgeräte, Kreditkarten, und vernetzen die Menschen durch soziale Netzwerke und mittels Messenger mehr, als je zuvor. Dabei verwalten Smartphones zum Teil private Daten, die es zu schützen gilt. Aufgrund der begrenzten Stromversorgung von Smartphones mittels Batterien spielt der Stromverbrauch eine wichtige Rolle. Welche Möglichkeiten bestehen nun, den Stromverbrauch einzelner Komponenten und Applikationen zu messen, und auf welche Art und Weise funktionieren diese? Was kann alleine durch Analysieren des Stromverbrauchs herausgefunden werden? Das ist die Fragestellung, um die es in dieser Arbeit gehen soll.

Zu Beginn wird der Stromverbrauch verschiedener Komponenten in verschiedenen Szenarien eines Smartphones gemessen, und auf Unterschiede analysiert. In Kapitel 3 wird ein Konzept vorgestellt, welches die in Kapitel 2 beschriebenen Daten auch im Alltagsbetrieb messen kann. In Kapitel 4 wird eine Methode vorgestellt, den Stromverbrauch einer einzigen App möglichst genau auf Software-Basis zu messen. Zum Ende wird gezeigt, auf welche verschiedenen Daten und Nutzungsverhalten durch das Analysieren des Stromverbrauchs zurückgeschlossen werden kann *(=Leistungs-Seitenkanalangriff)* [10], und es wird die Frage geklärt, wie leicht eine solche Analyse gelingen kann.

2 VERBRAUCH EINES SMARTPHONES IN VERSCHIEDENEN SZENARIEN

Ein modernes Smartphone kann man mit einem mobilen Computer vergleichen, der zusätzliche Komponenten, wie zum Beispiel verschiedene Sensoren *(Gyro, Luftdruck, GPS, etc.)* beinhaltet. Das Prinzip bleibt dabei das gleiche. Das Ziel des Abschnitts ist, den unterschiedlichen Energieverbrauch einzelner Komponenten eines Smartphones in unterschiedlichen Szenarien zu untersuchen. Dies wurde mit Benchmark-Tests bewerkstelligt. Da aber jedes Smartphone-Modell aus unterschiedliche Komponenten besteht, kann man die hier gezeigten Ergebnisse nur als Anhaltspunkt annehmen. Das Verhältnis des Stromverbrauchs der Komponenten kann von Modell zu Modell schwanken. Getestet wurde ein *Samsung S3C2442* mit *Android 1.5*. Verifiziert wurden die Ergebnisse mit einem *HTC Dream* und einem *Google Nexus One*. Diese Smartphones sind zwar knapp 10 Jahre alt, für den Zweck dieser Arbeit reichen die Testergebnisse dennoch aus, da sich an der Art und Weise der Funktion des Smartphones nichts grundsätzlich geändert hat. Bei den Benchmarks wurde auf ein realistisches Nutzerverhalten geachtet. Es wurde der Stromverbrauch des GSM *(kommuniziert mit Mobilfunkmasten)*, der CPU, des RAM, des WLAN-Moduls, der Grafikeinheit, des LCD-Bildschirms, und des Audio-Moduls gemessen. Der Stromverbrauch des Hintergrundlichts des Smartphones wurde nicht mit berücksichtigt [7].

Bevor man sich mit bestimmten Benchmarks für das Nutzerverhalten beschäftigt, sollte man zuerst das Smartphone im Leerlauf betrachten *(Abbildung 1)*. Hierbei ist das Smartphone komplett aktiv, es werdenmöglichst keine dedizierten Applikationen ausgeführt [7]. (Das Hintergrundlicht des Smartphones wurde ausgeschaltet.) Wie schon oben erwähnt, wird wohl bei jedem Modell der Leerlauf anders ausfallen. Misst man auf den unterschiedlichen Geräten den Gesamtverbrauch bei Nutzung der gleichen Applikationen, und zieht dann jeweils den Verbrauch bei Leerlauf ab, so erhält man den relativen Verbrauch eines bestimmten Nutzungsverhaltens nahezu Modell unabhängig.

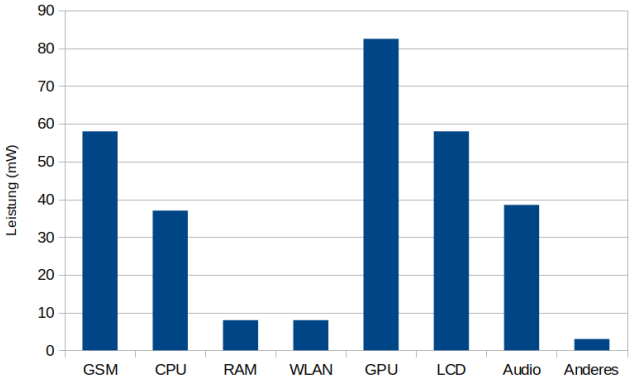

Abbildung 1: Der Verbrauch der Komponenten eines Smartphones im Leerlauf [7]

Betrachten wir nun reelle Szenarios *(Abbildung 2)*. Alle Tests wurden mit 100% Helligkeit des Bildschirms ausgeführt *(jedoch*

nicht in der Grafik erfasst), um die verschiedenen Szenarien besser zu vergleichen. Alle Szenarien haben unterschiedliche Verteilungen des prozentualen Anteils des Stromverbrauchs [7]. Daraus kann man folgendes schließen: Jede Anwendung zeigt eine signifikante Stromverbrauchs-Verteilung, womit man nur durch Betrachten des Stromverbrauchs auf verschiedene Nutzungsarten rückschließen kann.

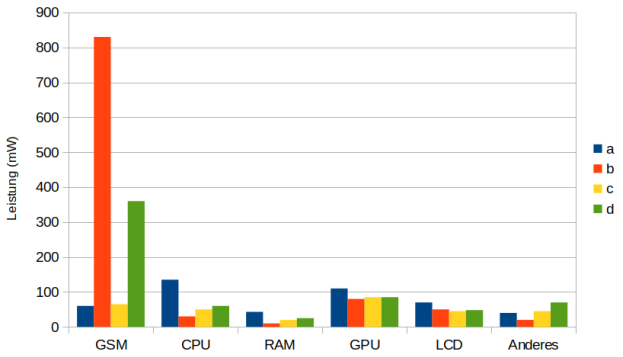

Abbildung 2: Der Verbrauch der Komponenten eines Smartphones während dem *(a)* Video Abspielen, *(b)* Telefonieren, *(c)* SMS senden, *(d)* E-Mail schreiben [7]

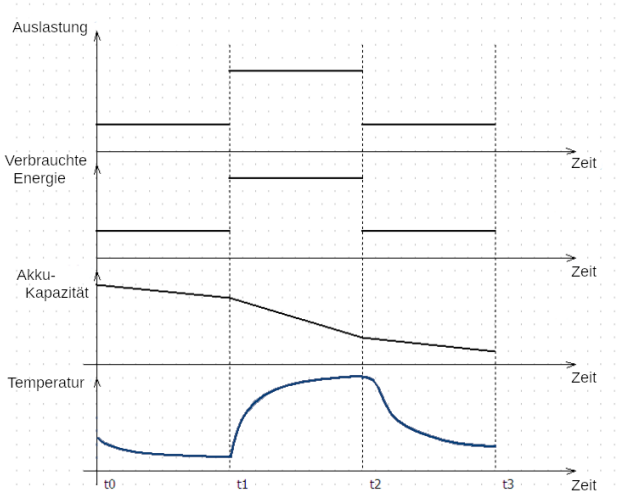

Abbildung 3: Beispielhaftes Verlaufsdiagramm eines Smartphones vor *(t0-t1)*, während *(t1-t2)*, und nach *(t2-t3)* einer spezifischen Last [8]

3 SOFTWARE-ANSATZ ZUM MESSEN EINZELNER KOMPONENTEN

Wie schon im vorherigen Kapitel gezeigt, kann man nicht nur die Akku-Daten selber zum Messen verwenden, sondern die verschiedensten Komponenten eines Systems. Für jeden Energieverbraucher in einem System kann man verschiedene Energieprofile festmachen, welche dann einem bestimmten Verhalten der Komponente zugeordnet werden können. Dies könnte beim Akku zum Beispiel die Entladungsrate, die aktuelle Kapazität, die Temperatur, und der Ladungs/Endladungszustand sein. Daraus lassen sich unterschiedliche Szenarien erkennen [8]. Ein einfaches Beispiel sieht man beim Betrachten des Akkkus während des Betriebs. Zwar kann man hier nur erkennen, wie das System an verschiedenen Zeitpunkten ausgelastet war, es gibt aber einen Einblick, wie das Framework später funktionieren kann. Betrachten wir folgendes modellhaftes Protokoll *(Abbildung 3)*:

Man kann drei verschiedene Energieprofile betrachten: Von *t0* bis *t1* vor einer spezifischen Last, von *t1* bis *t2* die spezifische Last selber, nach *t2* die Last danach. Betrachtet man nur die Kurve der Temperatur, erkennt man an einer fallenden Kurve, dass wohl eine Arbeitsauslastung in der nahen Vergangenheit liegt. Steigt die Kurve erheblich, war das System zuvor im Ruhezustand. Dies zeigt, dass alleine durch die Veränderungsrate von bestimmten Werten Rückschlüsse auf das Nutzungsverhalten gezogen werden können. Um die Verteilung des verbrauchten Stroms genauer zu bestimmen, sollten zu Beginn *(zum Beispiel nach dem Installieren der Applikation)* ein paar Schritte unternommen werden. Zuerst müssen die einzelnen oben genannten Komponenten vom Rest des Systems möglichst gut abgeschirmt, also alle andere Komponenten am besten deaktiviert werden. In der Praxis bei einem Endnutzer lässt sich dies aber nur teilweise umsetzen. Das WLAN-Modul lässt sich zum Beispiel auf einem Alltags-Gerät wohl kaum ohne CPU testen. Zwar kann man dann bei den Testergebnissen des WLAN-Moduls den geschätzten Stromverbrauch der CPU abziehen, die man zuvor alleine getestet hat, dennoch können die Ergebnisse so ungenau werden, weil beispielsweise die CPU durch unterschiedliches Caching oder andere Vorgänge einen veränderten Stromverbrauch haben kann [8]. Nach dem Isolieren der einzelnen Komponente, wird die jeweilige Komponente in simulierten Szenarien gemessen, und diese verschiedene Testergebnissen gewissen Arbeitszuständen zugeordnet. Das Framework soll somit später verschiedene Arbeitszustände einer Komponente erkennen, und dann den jeweiligen Stromverbrauch der Komponente zuordnen. Auch wird es schwierig, bei einem Linux Kernel sicherzustellen, dass wirklich während der Mess-Zeit nur der gewünschte Prozess ausgeführt wird [8]. Dieser Ansatz setzt sich auch nicht mit dem Multicore-Betrieb eines modernen Prozessors auseinander, bei dem mehrere Prozesse nebenläufig ausgeführt werden können. Ansatzmöglichkeiten, dieses Problem zu lösen, werden in Kapitel 4 aufgezeigt.

Allerdings könnte ein derartiges Framework zusätzlich *(siehe Kapitel 2)* das grobe Nutzungsverhalten des Benutzers erkennen *(Telefonieren, SMS schreiben, Video ansehen, etc.)*. Auf vielen gängigen Linux-Systemen zum Beispiel können noch neben den wenigen WLAN-Daten die das Framework selbst braucht, leicht viele weitere Informationen, wie auch die WLAN-SSID gesammelt werden [1], sogar vom einfachen Nutzer. Auf Android besteht beispielsweise beim WLAN das gleiche Problem [2]. Je mehr Informationen so ein Framework sammeln kann, desto genauer werden die Nutzungsprofile. Solche Nebeneffekte zu Unterbinden wird schwierig. Dennoch wird es für das Framework vermutlich auch schwierig, unbemerkt solche Nutzungsprofile anzufertigen, da dafür auch wieder Ressourcen des Systems benötigt werden. Somit würde der Ressourcenverbrauch

für das Framework steigen, was wiederum leicht erkannt werden würde.

4 POWER-SANDBOX - APPLIKATIONEN ISOLIERT MESSEN

Hinweis: Folgendes Kapitel bezieht sich auf die Vorstellung von Power-Sandbox der Wissenschafter Liwei Guo, Tiantu Xu, Mengwei Xu, Xuanzzhe Liu und Felix Xiaoz-hu Lin [9]. Aufgrund zahlreicher Quellenverweise wird von einer expliziten Nennung einiger Verweise auf diese Arbeit abgesehen.

Im Gegensatz zum in Kapitel 3 beschriebenen theoretischen Ansatz, verschiedene Komponenten genau zu messen, ist es den oben genannten Wissenschaftlern der *Purdue ECE* und der *Peking University* gelungen, ein Betriebssystem zu entwickeln, welches abgekapselt vom Rest des Systems den Stromverbrauch einer Anwendung möglichst genau messen kann. Die herkömmliche Art und Weise über das Messen des Stromverbrauchs von Applikationen funktioniert wie folgt: Der gesamte Stromverbrauch des Systems wird durch zuvor definierte Parameter der Applikationen geteilt, sei es zum Beispiel der prozentuale Anteil der CPU-Auslastung. Ein derartiger Ansatz kann allerdings sehr ungenau werden, da zum Beispiel eine Telefon-Applikation neben der vergleichsweise geringen CPU-Auslastung, einen sehr hohen Energieverbrauch des GSM-Moduls aufweist [7]. Die in diesem Kapitel beschriebene Power-Sandbox (kurz: *psbox*) versucht den Energieverbrauch einer einzelne Applikation abgeschirmt vom Rest des Systems zu messen. Die erhaltenen Messwerte stehen dann der Applikation als Information zur Verfügung, wodurch diese gegebenenfalls darauf reagieren kann. Allerdings ist der Einsatz von *psbox* einer Applikation nur temporär gedacht. Da *psbox* aktiv in Kernel und Treiber eingreift, und auch verschiedene neue Probleme *(siehe unten)* mit sich bringt, soll *psbox* im Alltagsgebrauch nur in gewissen spezifischen Phasen einer Applikation genutzt werden. Damit kann die Applikation in den übrigen Phasen ihre gewohnte Performance aufweisen, und das Betriebssystem selbst läuft wieder mit normaler Leistung.

Insbesondere müssen zwei Herausforderungen gemeistert werden: Zum Einen das Erzwingen der Abgrenzung der zu messenden Applikation, damit diese möglichst abgeschirmt vom Rest des Systems gemessen werden kann. Zum Anderen muss der Performance-Verlust der gemessenen Applikation so gering wie möglich sein. Denn auch *psbox* benötigt verschiedene Ressourcen, welche der Applikation dann fehlen, und somit ein anderes Verhalten der Applikation provizieren könnten.

Auch wenn der Stromverbrauch des Systems genau gemessen wird, wird sich aber bei einzelnen Applikationen das Problem der Energie-Verschränkung *(power entanglement)* ergeben. Diese Schwierigkeit ist in den meisten modernen Betriebssystemen vorhanden: Prozesse sind ineinander verschränkt, und haben gleichzeitig Einfluss auf den Energieverbrauch, welcher nicht strikt den einzelnen Prozessen zugeordnet werden kann. Einer der drei bedeutensden Gründe dafür ist die räumliche Nähe auf der Hardware: Verschiedene Applikationen nutzen bestimmte Hardware-Elemente oder Speicher-Räume gemeinsam. Ein weiterer Grund sind nicht eindeutige Anfrage-Grenzen: Viele Hardware-Komponenten führen, aus unterschiedlichen Gründen, Anfragen der CPU nicht sofort aus, was diese oft nicht vorausberechnen kann. Schließlich kann eine bestimmte Software-Auslastung den allgemeinen Hardware-Zustand verändern, und damit, ohne Einfluss des Prozesses, unterschiedliche Energie-Profile der Hardware hervorrufen. Zum Beispiel wurde der gleiche Prozess auf einer CPU einmal nach einer längeren Leerlauf-Periode, und einmal direkt nach einer anderen Berechnung ausgeführt. Der Stromverbrauch hatte erkennbare Unterschiede. Derartige Unstimmigkeiten konnten bis jetzt nicht verlässlich gelöst werden. Die Ansätze versuchen an die Thematik heuristisch heran zu gehen, was für das Messen des Energie-Verbrauchs auf System-Level gut funktioniert, aber für das genaue Messen einer einzelnen Applikation nicht ausreichend ist. Heuristische Ansätze können die Energie-Verschränkung nicht beheben, zumal jedes Modell einer Hardware-Komponente sein eigenes Verhalten in dieser Hinsicht hat. Angenommen der Verbrauch einer Applikation wurde genau gemessen, dann besteht das Problem der Energie-Verschränkung bis zu einem kleineren Grad weiterhin: Wie soll eine Applikation damit umgehen, wenn beispielsweise Netzwerk-Übertragungen gleicher Länge jeweils signifikant andere Energie-Verbräuche aufweisen können? Man kann dies im Alltag oft erkennen, wenn man die sich ändernde WLAN-Signalstärke betrachtet.

Die zwei größten Herausforderungen für *psbox* sind zum einen wie schon oben beschrieben das Eliminieren der Energie Verschränkungen, und das Integrieren der Funktionalitäten in gängige Kernel-Mechanismen, sodass tiefgreifende Änderungen im bestehenden Kernel vermieden werden können. Um die Energie-Verschränkungen zu minimieren, wurde die Art und Weise, wie der Kernel gleichzeitig laufende Prozesse auf der Hardware verschaltet, verändert. Dies wurde durch Erweitern vorhandener Kernel-Treiber realisiert:

Es wurden zwei leichtgewichtete Änderungen übernommen: Die erste ist das sogenannte *Resource Ballooning*. Der Kernel teilt seine Ressourcen in kleine Abschnitte *(Ballons)* ein, die zugewiesen werden. Die Benötigte Leistung für einen Ballon wird schon im voraus berechnet. Der Vorteil von Ressourcen Ballons ist, dass man für diese normale Scheduling-Einheiten des existierenden Kernels verwenden kann, wodurch nötige Änderungen am Kernel gering ausfallen.

Es werden verschiedene Arten von Ballons unterschieden: Erstens räumliche Ballons, für die gewissen räumliche Ressourcen der Hardware vorgesehen sind. Vorrangig ist damit die CPU und der Arbeitsspeicher gemeint. Dies soll verhindern, dass die zu messende Applikation mit anderen Apps gemeinsam gleichzeitig gewisse Hardware nutzt. Zweitens Zeitliche Ballons: Sie beheben das oben beschriebene Problem der asynchronen Verarbeitung verschiedener Hardware-Elemente, wobei sie sicherstellen sollen, dass die gemessene und die übrigen Applikationen nicht gleichzeitig auf Hardware zugreifen können. Bewerkstelligt wird das, indem das System den Zugriff zeitlich abhängig verweigert. Bei I/O Geräten wird beispielsweise mittels Interrupt Handling sichergestellt, dass gleichzeitige Anfragen nacheinander *(in verschiedenen Ballons)* abgehandelt werden. Die zweite kleinere Änderung im Kernel ist die Virtualisierung von Leistungs-Zuständen. Um die Privatsphäre des Systems weitestgehend aufrecht zu erhalten, speichert sich das System eine virtuelle Kopie des Leistungs-Zustandes für jede einzelne *psbox* ab. Bei dem nächsten Ressourcen Ballon wird dieser gespeicherte Zustand auf der Hardware wiederhergestellt, sodass

die Applikationen *(wenn sie beispielsweise abwechselnd sequentiell laufen)* durch keine äußeren Einflüsse gestört werden. Um die Idee auch umzusetzen, werden verschiedene Leistungs-Zustände definiert:

(1) Der erste ist der Zustand, in dem das Hardware-Element *ausgeschaltet* beziehungsweise *suspendiert* wurde. Diese Zustände wiederherzustellen kann sehr Energie-hungrig sein. Ein Beispiel dafür ist das mehrmalige An- und Ausschalten des GPS-Moduls [9].
(2) Meist bleiben dann eingeschaltete Komponenten für längere Zeit in einem *Arbeits-* oder dem *Leerlaufzustand* welche die Performance bei einem arbeitenden Hardware-Element oder das Strom-Sparen bei einem sich im Leerlauf befindenden Hardware-Element kontrollieren. Beispiele hierzu wären unterschiedliche Prozessor-Frequenzen [9].

Diese Virtualisierung soll sicherstellen, dass eine bösartige Applikation nicht erkennen kann, ob beispielsweise in letzter Zeit auf das GPS-Modul zugegriffen wurde. Wie schon in Kapitel 3 aufgezeigt können schon wenig Parameter viel über das Nutzungsverhalten aussagen. Auch kann die Virtualisierung der Applikation mitteilen, dass sich ein Hardware-Element im Leerlauf befindet, obwohl es reell suspendiert/ausgeschaltet wurde.

Da jeder moderne Prozessor in alltäglichen Smartphones und Computern mehrere Prozessor-Kerne hat, muss dieses Thema auch von *psbox* behandelt werden. Für eine bessere Skalierbarkeit kommunizieren die einzlenen Scheduler untereinander sehr selten. Um räumliche Ballons den CPU-Kernen für *psbox* zuzuweisen, wird ein Mehrkern-Scheduler eingeführt, welcher auf allen Kernen der CPU Aufgaben der Applikation über die räumlichen Ballons verteilt (*coscheduling*). Sollte es weniger Tasks als Kerne geben, werden den übrigen Kernen Dummy-Tasks zugeteilt. Am Ende soll vermieden werden, dass während der aktiven Laufzeit von *psbox* gleichzeitig die zu messene App mit den übrigen gemeinsam berechnet wird. Zur Realisierung dieser Vorgehensweise, muss der Scheduler entscheiden, zu welchen Zeitpunkten das *coscheduling* stattfinden soll, und dann durch Abziehen von Scheduling Credits entscheiden, wann wieder die restlichen Applikationen berechnet werden sollen. Ein faire Behandlung aller Applikationen wird hier angestrebt.

Zur Umsetzung wird ein *Scheduling-Darlehen* eingeführt: Es wird dem Scheduler erlaubt, eine bestimmte Aufgabe auszuwählen, auch wenn sie nicht die höchste Punktzahl gegenüber den anderen anstehenden Aufgaben vorweisen kann. Dafür wird dieser Aufgabe ein Darlehen von Punkten vergeben, mit dem sie die höchste Punktzahl hat. Später wird sie Darlehen an erhaltenen Punkten wieder „abbezahlen“. Die Aufgaben der zu messenden Applikation haben ein gemeinsames Punktekonto. Somit wird sichergestellt, dass keine einzelne Aufgabe der zu messenden Applikation neben anderen beliebigen Aufgaben läuft, sonden nur neben anderen Aufgaben der zu messenden Applikation.

Neben der CPU müssen auch noch andere Hardware-Elemente wie zum Beispiel die GPU berücksichtigt werden. Die GPU bekommt Befehle von der CPU, und führt diese asynchron aus. Auch wenn die CPU selber darauf achtet, dass die gemessene Applikation abgekapselt von den anderen berechnet und gemessen wird, kann dies bei asynchron ausführenden Hardware-Elementen wie zum Beispiel auch beim WLAN-Modul nicht garantiert werden. Um dieses Problem einzudämmen werden zeitliche Ballons für den Treiber eingeführt. Auch wird der Treiber die Ausführung von Aufgaben der zu messenden Applikationen vermeiden, solange Aufgaben von anderen Applikationen bestehen. Im Scheduling behandelt der Treiber die Aufgaben der zu messenden Applikation als eine einzelne Aufgabe. So kann die Leistungs-Verschränkung möglichst gering gehalten werden. Es müssen viele verschiedene Treiber für *psbox* angepasst werden, wenn das Prinzip von *psbox* breitflächig angewendet werden soll, was viel Arbeit mit sich bringt.

Dennoch lohnt sich der Aufwand. Bei der Evaluation von *psbox* wurde das weitestgehende Eliminieren der Leistungs-Verschränkung erreicht. Die Ergebnisse von *psbox* haben sich weniger als 5% voneinander unterschieden. Ergebnisse anderer herkömmlichen Methoden haben sich bis zu 60% unterschieden. Somit können sich Applikationen bis zu einem gewissen Grad auf *psbox* verlassen, und können im Idealfall durch das genauere Feedback am Ende mehr Effizienz und Leistung gewinnen.

Eine zu 100% reproduzierbare genaue Messung einer Applikation in einem alltäglichen Betriebssystem ist höchstwahrscheinlich unmöglich, da Applikations-Verhalten und das Scheduling eines alltäglichen Betriebssystems in mehreren Durchläufen nicht immer gleich sein müssen.

Aufgrund dieser hohen Mess-Genauigkeit können theoretisch bestimmte Zustände einer Applikation abgelesen werden. Wie das zu bewerkstelligen ist, soll im nächsten Kapitel gezeigt werden. Man muss sich bewusst sein, dass alleine durch das Betrachten der GPU-Auslastung ein Ausspähen der Applikation mit 60% Genauigkeit gelingen kann, zum Beispiel welche Webseite von Zehn möglichen durch die Applikation geöffnet wurde. Deswegen haben sich die Entwickler von *psbox* dafür entschlossen, dass die Mess-Daten einer Applikation nur dieser Applikation selbst zur Verfügung gestellt werden. Somit schafft *psbox* in diesem Gebiet keine neuen Sicherheitslücken. Trotzdem müsste es aber möglich sein, eine gesamte Applikation als Kind einer anderen laufen zu lassen, wobei die erhaltenen Messdaten dieser Eltern-Applikation Rückschlüsse auf die Messdaten der Kind-Applikation zulassen. Damit kann die oben erwähnte Sperre zu einem gewissen Grad umgangen werden.

Nun stellt sich am Ende die Frage: Wofür kann man nun *psbox* nutzen? *Psbox* am Ende großflächig in Smartphones im Alltag einzusetzen, macht bei dem hohen Aufwand, viele Treiber anzupassen, wohl wenig Sinn. Andere konventionelle Verfahren existieren bereits, die etwas ungenau sind, vielen Applikationen aber wahrscheinlich dennoch ausreichen. Eher kann *psbox* für Software-Entwickler sehr interessant sein, um am Ende zum Beispiel bestimmte Komponenten einer Software zu messen. Das würde ohne ein System wie *psbox* nicht funktionieren. Auch in bestimmten professionellen Anwendungen in speziellen Software-Landschaften könnte *psbox* interessant werden. Beispielsweise könnten sich so verschiedene zusammenhängende Software-Komponenten, individuell ihr jeweiliges Verhalten besser im Gesamt-System anpassen. Am Ende ist *psbox* eine spannende Technologie, die große Probleme der konventionellen Energie-Messung, wie das *power entanglement* nahezu löst, und auch aufzeigt, dass verhältnismäßig genaue Messungen von einzelnen Applikationen prinzipiell möglich sind, aber zugleich viele Anpassungen eines Betriebssystems erfordern.

5 WAS KÖNNEN LEISTUNGS-SEITENKANALANGRIFFE VERRATEN?

Hat man also nicht die verfügbaren Ressourcen/Möglichkeiten, ein Betriebssystem anzupassen, um den Stromverbrauch einzelner Applikationen zu bestimmen und damit bestimmtes Nutzungsverhalten rückzuschließen, so wird in diesem Artikel eine andere Methode beschrieben. Diese setzt voraus, dass nur eine Applikation im Vordergrund läuft, was beispielsweise bei Android in den meisten Fällen erreicht wird.

Seitenkanalangriffe wurden früher hauptsächlich eingesetzt, um kryptografische Geräte bei der Ausführung von kryptologischen Algorithmen zu beobachten, und durch Beobachten der Daten den verwendeten Schlüssel heruszufinden. Mittlerweile hat sich dies aber auf viele andere Anwendungszwecke ausgeweitet [10], wodurch mittels jeder erfassbaren Information Rückschlüsse auf die Inhalte gezogen werden können *(siehe Kapitel 2)*. Dieses Kapitel beschäftigt sich hauptsächlich mit dem Leistungs-Seitenkanalangriff. Es wird beschrieben, welche Informationen alleine aus dem Energieverbrauch eines herkömmlichen Smartphones erschlossen werden können.

Um den genauen Energieverbrauch des Smartphones zu messen wurde ein Leistungs-Zähler (*Monsoon Power Monitor*) mit einer konstanten Spannung von 4,2 Volt verwendet. Da in der Regel im alltäglichen Gebrauch ein Smartphone nicht an einen externen Leistungs-Monitor angeschlossen ist, bietet das System auch Informationen über den momentanen Leistungs-Verbrauch an, welche Daten aber nicht so detailliert sind, wie die des externen Gerätes [10]. Die meisten Betriebssysteme wie zum Beispiel Android stellen die Akku-Informationen der Battery Monitoring Unit ohne bestimmten Berechtigungen zur Verfügung, so dass jede Applikation darauf zugreifen kann. Da aber eine bösartige Applikation wohl eher nicht im Vordergrund laufen wird, benötigt sie die Berechtigung *Im Hintergrund ausführen* um Akku-Informationen zu erfassen [3].

An erster Stelle wird versucht, die momentan laufende Applikation herauszufinden. Dazu kann man die Leistungs-Kurve während des Ladevorgangs beim Start der Applikation näher betrachten. Es fällt auf, dass sich die Kurven bei Ladevorgängen der gleichen Applikation ähneln, die Applikationen aber untereinander unterschiedliche Kurvenverläufe aufweisen. Die Applikationen wurden über 20 Mal geladen, die Kurven sahen stets so wie in Abbildung 4 charakterisiert aus [10]. Auch sind im Leistungs-Verbrauch die Lade-Phasen unterschiedlich zu den normalen Betriebsphasen, und somit deutlich erkennbar [10]. Genau das gleiche Experiment wurde mit der Analyse der durch die von Android bereit gestellten Daten (also nur auf Software-Ebene) durchgeführt. Die Ergebnisse waren ähnlich zu den oben beschriebenen [10].

Ein ähnliches Experiment wurde diesmal nur auf Software-Basis (ohne den physischen Leistungs-Zähler) mit dem Aufrufen verschiedener UI-Elemente einer App durchgeführt, welches zum gleichen Ergebnis kam [10]. Damit könnte man zum Beispiel eine Passwort-Eingabe manipulieren: Erkennt die bösartige Applikation im Hintergrund den Moment, in dem eine Passwort-Eingabe einer bestimmten Applikation geöffnet wird, kann die Applikation ihre UI über der anderen anzeigen, und somit die Daten erfassen.

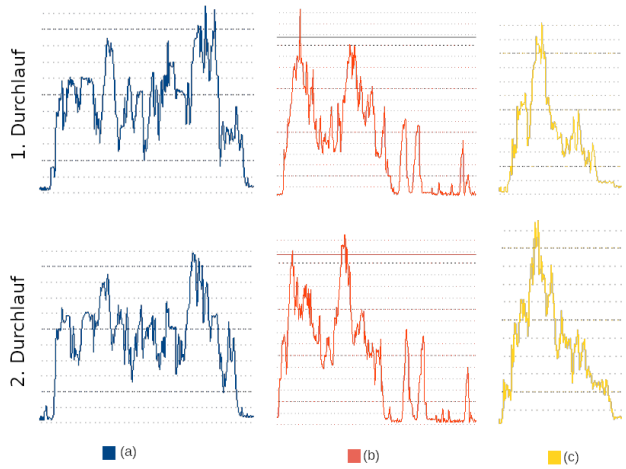

Abbildung 4: Leistungs-Verlauf während den Ladungsphasen von (a) WeChat, (b) Alipay, (c) Gmail [10]

Dies geht allerdings nicht so ohne weitere Berechtigungen. Dafür braucht die bösartige Applikation die Berechtigung, seine Anzeige über die der anderen Applikationen zu legen. [3]

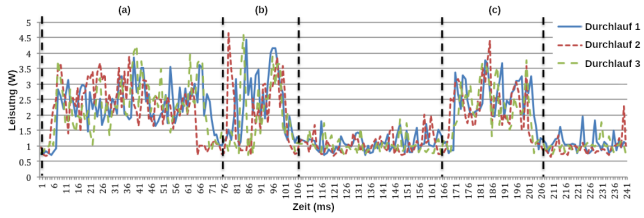

Abbildung 5: Leistungs-Verlauf in der App Amazon während dem Aufrufen der UI-Elemente (a) Übersicht, (b) Produktansicht, und (c) Benutzer-Anmeldung [10]

Falls die Applikation wie oben beschrieben keine Berechtigung hat, sich über anderen Applikationen abzubilden, kann sie aber auch nur durch Beobachten der Leistungs-Kurve herausfinden, wann eine Taste auf der Tastatur gedrückt wurde *(Abbildung 6)*. Es ist schwierig, die Tasten voneinander in der Kurve zu unterscheiden, man kann aber deutlich die Passwortlänge herauslesen. Dies setzt natürlich voraus, dass der Nutzer sein Passwort sofort richtig eingibt, und nicht währenddessen ein Zeichen korrigiert. Also eine korrekte Eingabe des Kennwortes beim ersten Versuch macht es Angreifern leichter, auf dessen Länge rückzuschließen.

Doch nicht nur die Passwort-Länge kann so ermittelt werden. Auch ist es Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandianund Dan Boneh und Gabi Nakibly gelungen, anhand der Leistungskurve, während das GPS-Modul aktiv ist, Routen anhand zuvor gesammelter Daten wieder zu erkennen (Abbildung 7). Es wäre laut diesen ein leichtes, mit einer größeren Datenbank Routen genau zu bestimmen, und so Standorte eines Smartphones alleine anhand der Leistungs-Kurve zu erkennen [4]. Dabei ist

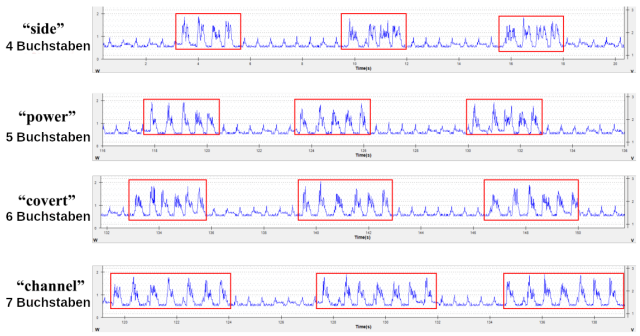

Abbildung 6: Leistungs-Verlauf während dem Tippen verschiedener Passwort-Eingaben [10]

noch nicht eingerechnet, dass man zum Beispiel anhand der eigenen IP-Adresse des Gerätes schon im Voraus eine grobe Standort-Bestimmung vornehmen kann. Es gibt auch Projekte, bei denen nur WLAN-Informationen für die Standort-Bestimmung eines Smartphones genutzt werden. Die Möglichkeiten für das Herausfinden des Standorts ohne direkten Zugriff auf die Standort-Funktion sind auf dem Smartphone sehr vielfältig [5].

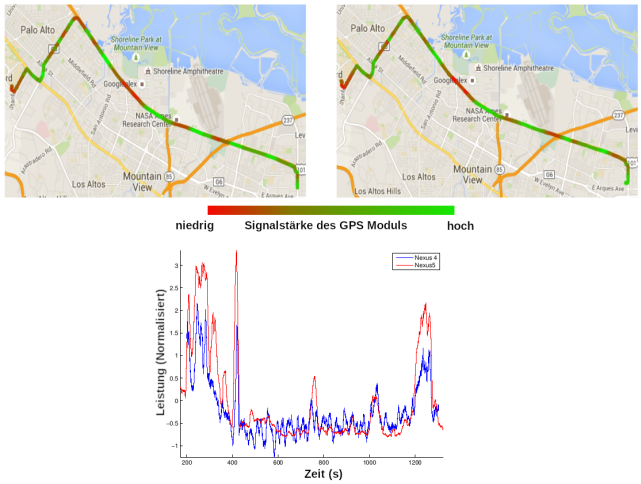

Abbildung 7: Leistungs-Verlauf während dem Ablaufen der gleichen Route an verschiedenen Tagen auf unterschiedlichen Modellen [4]

Die oben gezeigten Experimente waren nur Beispiele. Es gibt wohl viel mehr Möglichkeiten, an private Informationen über einen Seitenkanalangriff heranzukommen. In der Praxis ist es wohl am Ende nicht so eindeutig, nur aus einer einzigen Kurve, direkte Informationen zu bekommen. Durch genügendes Training einer Künstlichen Intelligenz mit einer ausreichend großen Datenbank, ist so ein Vorgehen dennoch großflächig vorstellbar, zumal die dafür benötigten Berechtigungen auf Android auf den ersten Blick wohl keine große Besorgnis erregen. Künstliche Intelligenz wurde durchaus schon bei anderen Side-Channel-Attacken erfolgreich verwendet [6].

6 SCHLUSSFOLGERUNG

Es wurden zwei Methoden gezeigt, wie der Stromverbrauch eines Systems oder einer Applikation differenziert werden kann, und schon mit wenigen Informationen grob auf gewisses Nutzungsverhalten wie zum Beispiel die momentane Tätigkeit (E-Mail, Telefonieren, Web-Browsing, …) geschlossen werden kann. Leistungsinformationen in der Smartphone-Welt spielen eine sehr große Rolle, und helfen vielen Applikationen, sich besser anzupassen, um somit den Akku-Verbrauch eines mobilen Endgerätes zu senken. Dennoch kann wie in Kapitel 5 gezeigt auch schon die Leistungs-Kurve allein vertrauliche Informationen über Passwörter, verwendeten Apps, bis hin zu dem Standort verraten. Am Ende bleibt dem Endnutzer wohl nur übrig, der Software auf dem Smartphone zu vertrauen, bzw. die geladenen Applikationen auf ihre Notwendigkeit und Zuverlässigkeit zu hinterfragen, da Informationen über den Leistungs-Verbrauch auf einem mit Akku betriebenen Gerät für viele Applikationen unerlässlich sind.

LITERATUR

[1] Gnome Foundation. 2020. Gnome Network Manager Documentation. https://developer.gnome.org/NetworkManager/stable/NetworkManager.html [aufgerufen am 22.06.2020].
[2] Google. 2020. Android Documentation: Network Info. https://developer.android.com/reference/android/net/NetworkInfo [aufgerufen am 22.06.2020].
[3] Google. 2020. Android Documentation: Permission Manifest. https://developer.android.com/reference/android/Manifest.permission [aufgerufen am 19.07.2020].
[4] Yan Michalevsky und Aaron Schulman und Gunaa Arumugam Veerapandian und Dan Boneh und Gabi Nakibly. 2015. PowerSpy: Location Tracking using Mobile Device Power Analysis. (2015), 4–5, 8–9.
[5] Ali H. Sayed und Alireza Tarighat und Nima Khajehnouri. 2005. Network-Based Wireless Location. (2005), 1–2.
[6] Maria Mushtaq und Ayaz Akram und Muhammad Khurram Bhatti und Maham Chaudhry Muneeb Yousaf und Umer Farooq und Vianney Lapotre und Guy Goniat. 2018. Machine Learning For Security: The Case of Side-Channel Attack Detection at Run-time. (2018), 5.
[7] Aaron Carroll und Gernot Heiser. [n.d.]. An Analysis of Power Consumption in a Smartphone. ([n. d.]), 2–12.
[8] Dacian Tudor und Marius Marcu. 2009. Designing a Power Efficiency Framework for Battery Powered Systems. (2009), 2–6.
[9] Liwei Guo und Tiantu Xu und Mengwei Xu und Xuanzzhe Liu und Felix Xiaozhu Lin. 2018. Power Sandbox: Power Awareness Redefined. (2018), 1–10.
[10] Lin Yan und Yao Guo und Xiangqun Chen und Hong Mei. 2016. A Study on Power Side Channels on Mobile Devices. (2016), 1–5.