Università degli Studi di Genova

Facoltà di Ingegneria

Laurea Magistrale in Ingegneria Informatica

Energy-Aware Security: Strumenti e Modelli Analitici per stimare il Consumo Energetico nel Sistema Operativo Android

Paolo Fontanelli

Relatore:

Chiar.mo Prof. Ing. Mauro Migliardi

Correlatore:

Dott. Alessio Merlo

Indice

Introduzione

Negli ultimi anni si è vista una crescente diffusione dei dispositivi mobili in tutti gli aspetti della vita delle persone, infatti al giorno d'oggi con uno smartphone o con un tablet si hanno a portata di mano molte delle funzionalità che in passato caratterizzavano esclusivamente i personal computer.
Questa rapida evoluzione ha portato questo tipo di dispositivi ad assumere un ruolo importante sia nella vita privata che in quella lavorativa delle persone, due ambiti in cui sono presenti problematiche di sicurezza legate principalmente ai dati che vengono memorizzati nei dispositivi; infatti se la sicurezza dei dati dell'utente è un aspetto fondamentale quando si fa un uso privato di questo tipo di dispositivi, la rilevanza dell'argomento crescerebbe notevolmente se il dispositivo fosse un dispositivo aziendale e i dati trattati riguardassero la vita lavorativa delle persone.
Oltre a questi problemi legati alla sicurezza, sono presenti anche altri aspetti, infatti, a causa della natura e delle caratteristiche di questo tipo di dispositivi diventa necessario considerare anche altri punti di vulnerabilità.

Un mobile device ha infatti la caratteristica, banale ma fondamentale, di non essere vincolato all'alimentazione elettrica ma di dipendere da un'alimentazione a batteria; questo costituisce probabilmente il principale vantaggio di questo tipo di dispositivo, ma costituisce anche una sua possibile debolezza: si potrebbe infatti pensare di effettuare attacchi di tipo DoS sfruttando la dipendenza del dispositivo dalla batteria.
Questo tipo di attacchi prevedono di forzare il dispositivo ad effettuare comportamenti che accelerino il consumo della batteria, rendendone così impossibile l'utilizzo, questi attacchi hanno un nome specifico: si tratta di

attacchi ti tipo “battery drain” e possono essere di vario tipo, da attacchi provenienti dalla rete, ad attacchi che forzano il dispositivo a fare calcoli inutili, ad attacchi che forzano lo schermo a rimanere acceso anche quando l’utente non utilizza il dispositivo.

Sebbene alcuni di questi attacchi siano facilmente riconoscibili, basti pensare allo schermo che si accende quando l’utente non sta utilizzando il dispositivo e non vi sono eventi che giustificano questo comportamento, altri attacchi di questo tipo non sono così facilmente riconoscibili, soprattutto ad un utente inesperto. Da qui nasce la necessità di avere tecniche e metodologie che permettano di eseguire un’identificazione di questi attacchi anche quando non siano evidenti per l’utente.

Queste tematiche rientrano nell’ambito della **Green Security**[1], che si occupa di studiare questi (ed altri) aspetti di sicurezza, con lo scopo di effettuare rilevazione di attacchi al dispositivo; si pensa infatti che partendo dalla conoscenza del consumo di un determinato pattern di applicazioni che sono in esecuzione su un dispositivo, si possa procedere con l’identificazione di attacchi ed anomalie.

Obiettivi

Le problematiche descritte pongono come obiettivo riuscire ad identificare eventuali attacchi “battery drain” in corso sul dispositivo, tuttavia per arrivare a questo risultato è necessario prima comprendere a fondo il modo in cui le risorse che si hanno a disposizione sul dispositivo consumano la batteria e, soprattutto, come si comportano le applicazioni che utilizzano queste risorse.

È quindi utile conoscere il consumo energetico delle applicazioni che sono installate sul dispositivo. Si parla, in questa situazione, dello studio della “firma energetica” delle attività, ove, con questo termine, ci si riferisce all’andamento dei consumi provocato da un’applicazione o da un insieme di applicazioni in esecuzione su un dispositivo. La conoscenza della firma energetica può permettere di capire quando vi siano comportamenti anomali o maligni sulla base dei consumi del dispositivo.

Per distinguere la firma energetica di un’applicazione serve un modello che descriva i consumi delle risorse che questa applicazione utilizza, in questo modo è possibile stimare quello che dovrebbe essere il comportamento teorico dell’applicazione ed andare a verificare se il comportamento reale è effettivamente quello atteso.

L’obiettivo del mio lavoro è stato quello di sviluppare una parte di questo sistema di controllo dei consumi, in particolare mi sono occupato di studiare il comportamento e realizzare il modello che riguarda la CPU, componente fondamentale in questo studio poiché si tratta di una risorsa che è sempre utilizzata all’interno del dispositivo.

Per affrontare questo problema mi sono concentrato sui dispositivi Android-based, che presentano il vantaggio di essere open source, semplificando così l'approccio al problema.

Stato dell'arte

In letteratura sono presenti numerosi studi volti a d analizzare i meccanismi di sicurezza del sistema operativo Android, in alcuni lavori sono state proposte soluzioni per l'individuazione dei malware[4], ma senza fare alcuna considerazione di carattere energetico.
Vi sono lavori che trattano il problema al di fuori del contesto dei mobile devices, ma anche in questi casi difficilmente si approccia il problema degli attacchi da un punto di vista energetico, probabilmente a causa della minor criticità del problema in quei contesti.

Sono quindi molto pochi i lavori che trattano il problema con un approccio di tipo energetico, in particolare esiste un lavoro che si è occupato di fare rilevamento di attacchi DoS, riuscendo a riconoscere tre tipi di attacchi, ma basandosi su hardware esterno per fare le misurazioni[5].

Esiste un lavoro chiamato PowerTutor[6] che adotta un approccio senza nessun tipo di misurazione hardware esterna, tuttavia è assai poco sensibile e permette soltanto di riconoscere alcuni attacchi di tipo battery drain di forza bruta.

Un altro lavoro di sicuro interesse in questo campo è AppScope[7]. Questo lavoro utilizza un approccio assai invasivo nei confronti del sistema operativo ma che arriva ad una misura precisa dei consumi. Tuttavia esso è legato ad un singolo dispositivo, il Google Nexus One, ed i suoi sorgenti non sono open source; per questo motivo non è facile replicare questo approccio su altri dispositivi.

Esistono inoltre lavori che trattano l'identificazione di malware su altri sistemi operativi[8 e 9] quali Symbian e Windows Mobile OS, accennando anche alcuni aspetti energetici.

Alcuni lavori hanno pensato di valutare il consumo di un dispositivo mobile andando a verificare il consumo di ogni singolo componente del dispositivo che abbia la possibilità di andare a richiedere energia alla batteria[10], ad esempio GPS, Wifi, modulo 3G, CPU ecc…

Avendo analizzato questi lavori, si possono distinguere diversi tipi di approcci al problema. Ad esempio, all'utilizzo di strumenti hardware esterni per realizzare la misura si contrappone una misura che si basa solamente sulla strumentazione e sui tool che sono presenti sul dispositivo; un'altra possibile distinzione riguarda il livello dello stack android all'interno del quale si interviene, infatti sono possibili approcci di "basso livello" che interessano il kernel Linux oppure approcci di "alto livello" che riguardano le applicazioni User-level e le API messe a disposizione dal sistema.

Il reale punto di partenza per il mio lavoro è però costituito da un lavoro che si è occupato di realizzare un modello di consumo per il modulo Wifi[3] attraverso la progettazione di moduli per il kernel Linux che si occupavano di campionare il consumo a "basso livello". Questo lavoro riesce ad effettuare l'identificazione di alcuni tipi di attacco senza l'ausilio di alcun dispositivo hardware esterno e senza perciò imporre alcuna limitazione alla mobilità del device.

Descrizione dell'approccio scelto

Visto che il mio lavoro è un'estensione di quello relativo al componente Wifi[3], ho studiato la possibilità di proseguire sulla strada che era già stata tracciata, ossia di affrontare il problema a basso livello, realizzando moduli che potessero essere inseriti ed utilizzati nel kernel Android. In realtà mi sono ben presto reso conto che questo tipo di approccio porta con sé i vantaggi dell'azione a basso livello, ossia il poter accedere direttamente alle misure provenienti dai driver, tuttavia presenta gli svantaggi di essere meno portabile su differenti devices rispetto a soluzioni di livello più alto, in quanto maggiormente legato alle caratteristiche hardware del dispositivo.
Per questi motivi ho provato a verificare se c'era la possibilità di realizzare lo stesso lavoro ad un livello più alto, rendendomi conto che il kernel Android mette a disposizione dello sviluppatore tutto quello che è necessario per andare a verificare quali sono i consumi del dispositivo e per andare a configurare la frequenza massima e minima alla quale far lavorare la CPU. Così ho deciso di provare a lavorare a livello più alto, pertanto nell'approccio che ho seguito non è previsto l'inserimento di nessun modulo kernel, ma l'installazione di alcune applicazioni che permettono di ottenere tutti i dati necessari.

L'approccio seguito segue l'idea presentata in letteratura[10] di considerare il consumo di ogni componente del dispositivo che possa fare richiesta di energia alla batteria, costruire un modello che descriva il consumo del componente in questione e considerare il consumo del dispositivo come composto dai contributi forniti da ogni componente.
Per quanto riguarda il supporto di strumenti hardware esterni per effettuare le misure, ho proseguito con l'idea presentata nel lavoro relativo al componente

Wifi, ritenendo che basarsi su un dispositivo hardware esterno potesse essere una forte limitazione per il genere di dispositivi verso i quali questo lavoro è indirizzato.

Ho deciso però, rispetto all'approccio di "basso livello" presentato nel lavoro relativo al modulo Wifi, di spostare l'attenzione ad un livello più alto, in modo da non essere così tanto dipendente dal driver della batteria utilizzato all'interno del dispositivo. Ovviamente una dipendenza di questo genere rimane, ma diventa un problema facilmente risolvibile ad alto livello.

Questo tipo di lavoro prevede che, per avere un modello che riesca a descrivere il comportamento energetico del dispositivo in un caso reale, servano i modelli che descrivono il comportamento di ogni componente del dispositivo in questione. Vista la grande quantità di componenti che possono essere contemporaneamente attivi su un dispositivo mobile e considerato il fatto che alla fine del lavoro saranno disponibili soltanto i modelli per due di questi componenti, difficilmente si potrà descrivere il comportamento del dispositivo in uno scenario d'uso reale, ma ci si dovrà limitare al caso in cui i componenti operativi sul telefono siano quelli il cui comportamento può essere descritto con i modelli a disposizione: Wifi e CPU.

Android

Google ha definito Android come "*software stack for mobile phones including the operating system, the middleware, and the applications*". È basato sul kernel Linux e le applicazioni (java) non sono eseguite direttamente sul sistema operativo, ma in un livello intermedio chiamato Davilk VM, una virtual machine che ha il ruolo di interpretare il codice e di creare una sorta di sandbox per le applicazioni.

Android ha una gestione del power management molto particolare, le applicazioni ed i servizi fanno richiesta di risorse CPU attraverso l'uso dei "wake lock" per mantenere la CPU in uno stato di power on, se nessun wake lock è stato acquisito Android tenderà a portare la CPU in uno stato di basso consumo energetico.

Il comportamento di default di Android è cercare, quando possibile, di portare la CPU in uno stato di sleep o di basso consumo nel minor tempo possibile, in modo da preservare energia per il dispositivo. Se qualche applicazione ha attivo qualche wake lock, la CPU non sarà portata in uno stato di basso consumo poiché ha dei calcoli da eseguire.

Vi sono quattro tipi di wake lock così definite[11]:

- PARTIAL_WAKE_LOCK: Garantisce che la CPU rimanga in uno stato di attività, permettendo però di spegnere lo schermo.
- SCREEN_DIM_WAKE_LOCK: Assicura che lo schermo sia acceso, ma la retroilluminazione della tastiera potrebbe disattivarsi, e quella dello schermo potrebbe non metterlo a piena luminosità

- SCREEN_BRIGHT_WAKE_LOCK: Garantisce che lo schermo sia alla massima luminosità, ma la retroilluminazione della tastiera potrebbe spegnersi.
- FULL_WAKE_LOCK: Il dispositivo è completamente acceso, compresi la retroilluminazione e lo schermo.

In realtà quelli appena descritti non sono gli unici tipi di wake lock possibili, ma sono quelli utilizzabili da user-space, ve ne è poi un altro tipo, utilizzabile da kernel-space, chiamati Kernel Wake Lock, che impediscono alla CPU di entrare negli stati di basso consumo, con un comportamento analogo a quello descritto per il PARTIAL_WAKE_LOCK del livello applicativo; questo tipo di wake lock possono essere acquisiti all'interno del kernel.

Fig. 1: Lo Stack Android

Il Kernel Linux

Come abbiamo detto, Android è basato sul kernel di Linux, con il quale condivide moltissime caratteristiche. In particolare le componenti che sono state più utili per la realizzazione del mio lavoro sono state il driver cpufreq, utilizzato per manipolare la frequenza all'interno del kernel, il driver della batteria ds2784-battery, dal quale è stato possibile accedere ai valori elettrici riguardanti la batteria. Infine un ruolo importante è stato giocato da proc, uno pseudo-filesystem che contiene al suo interno le informazioni relative ai processi in esecuzione nella CPU.
Vediamo questi elementi nel dettaglio:

cpufreq

Cpufreq fa riferimento ad un'infrastruttura del kernel che permette di fare frequency scaling, si tratta cioè di un sottosistema di Linux che permette di andare ad impostare manualmente la velocità del clock. Si tratta di un sistema che è incluso in tutti i moderni kernels, ed è abilitato in tutte le distribuzioni più recenti.
Fare frequency scaling significa ridurre il numero di istruzioni che il processore può eseguire in un dato periodo di tempo, riducendo così le performances ed allo stesso tempo il consumo energetico.
Cpufreq prevede che la CPU adatti la frequenza basandosi sul carico di lavoro, questo viene realizzato attraverso meccanismi che prevedono di considerare solamente l'utilizzazione della CPU stessa.
Questo meccanismo è controllato dal governor che è in uso sul dispositivo. Il governor è un elemento del kernel che si occupa di regolare la frequenza della CPU; per gli scopi del lavoro eseguito ho deciso di utilizzare un governor di

tipo onDemand, il cui meccanismo prevede di innalzare la frequenza rapidamente a fronte di un incremento di utilizzazione, per poi diminuirla lentamente quando l'utilizzazione scende.

ds2784-battery

Si tratta del driver utilizzato per la batteria del dispositivo, questo driver effettua una misurazione di temperatura, tensione e corrente erogata dalla batteria, realizzando poi una stima della capacità residua.

Le caratteristiche di questo driver prevedono che la temperatura e la tensione vengano campionate ogni 440ms, mentre la corrente viene campionata su un periodo più lungo, questo è dovuto al fatto che il calcolo della corrente viene effettuato misurando la caduta di potenziale agli estremi di una resistenza di valore molto basso e convertendo i picchi di tensione ad un valore predefinito. I valori sono campionati ad una frequenza di 18,6kHz ed aggiorna il registro della corrente al termine di ogni ciclo di conversione; per questo motivo si ottiene un nuovo valore di corrente ogni 3,52s.

Come vedremo, questo dispositivo fornisce dati fondamentali per lo studio condotto, tuttavia il fatto che la carica residua sia stimata e non misurata impedisce di utilizzarne il valore.

proc

Il file system proc è uno pseudo-filesystem usato per accedere alle informazioni relative ai processi che sono fornite dal kernel. Solitamente è montato nella directory */proc*, e non essendo un filesystem reale non occupa spazio su disco rigido ed occupa una quantità limitata di memoria.

La maggior parte dei file presenti in */proc* sono file di sola lettura, ma alcuni di questi permettono all'utente anche la scrittura, consentendo così di modificare le variabili del kernel.

Tra le informazioni che sono reperibili all'interno di questo pseudo-filesystem cito solamente quelle che sono state di interesse per il lavoro che ho effettuato, ossia le statistiche del sistema relative all'esecuzione della CPU, contenute in un file chiamato *stat*, e quelle relative alla memoria del sistema, contenute in *meminfo*.

Dispositivo Utilizzato

Il dispositivo utilizzato per realizzare il mio lavoro è un HTC Desire, uno smartphone caratterizzato da un processore Qualcomm Scorpion single-core da 1GHz e schermo di tipo AMOLED. La batteria presente all'interno dello smartphone era una SEIDIO BASI16HNX1 caratterizzata da una capacità da 1600mAh con una tensione nominale da 3,7V.
Sul dispositivo ho installato una ROM modificata, esattamente la CyanoGen Mod 7.

La scelta del dispositivo è dovuta al fatto che era necessario un dispositivo che fosse dotato di un coulomb counter, in modo da avere una misurazione precisa dell'energia consumata. In una fase iniziale del lavoro avevo considerato anche la possibilità di utilizzare un Samsung Galaxy S, ma probabilmente non era dotato di coulomb counter e non esponeva all'utente la possibilità di ricavare i dati energetici di cui aveva bisogno.

Misurazione della potenza consumata

La potenza consumata dal dispositivo può essere ottenuta come il prodotto tra la corrente erogata dalla batteria per la tensione presente ai suoi capi.

$$P = V * A$$

Per calcolare la potenza consumata si rende pertanto necessario ricavare le informazioni di tensione e corrente erogata dalla batteria.
Nell'approccio di "basso livello" queste informazioni venivano recuperate attraverso alcuni moduli da inserire nel kernel Linux, questi si occupano di andare a leggere i dati memorizzati dal driver della batteria attraverso apposite funzioni già presenti all'interno del driver che venivano esportate appositamente.
Questo passaggio prevede tre passi: l'esportazione delle funzioni di interesse all'interno dei sorgenti del kernel Linux, la compilazione del kernel ed infine l'installazione del kernel sul dispositivo.
È intuitivo capire che se si dovesse implementare questa soluzione su un dispositivo che non utilizza lo stesso driver utilizzato sul dispositivo di riferimento, ebbene sarebbe necessario andare a modificare il codice del modulo sostituendo le funzioni del driver del vecchio dispositivo con quelle del nuovo dispositivo.

Utilizzare un approccio di "alto livello" risolve questo problema, in quanto sfrutta le funzionalità messe a disposizione dell'utente dal kernel Linux, infatti per capire quanto consuma il dispositivo è sufficiente andare a leggere i valori

di tensione e corrente erogata dalla batteria all'interno di appositi files posizionati nella directory del driver della batteria.
Il dispositivo che ho utilizzato per realizzare il mio lavoro è lo HTC Desire, il driver della sua batteria, chiamato ds2784-battery, scrive i valori di tensione e corrente nella directory */sys/devices/platform/ds2784-battery*, ed i due files che contengono i valori che sono di interesse per lo studio della potenza consumata sono:

1. */sys/devices/platform/ds2784-battery/getmAh* per la corrente
2. */sys/devices/platform/ds2784-battery/getvoltage* per la tensione

Qualsiasi applicazione utente può andare a leggere i valori contenuti in questi due files con poche e semplici righe di codice, riporto un esempio relativo alla lettura della corrente:

```
String currentPath = "/sys/devices/platform/ds2784-
battery/getcurrent";
FileReader currentFile = new FileReader(currentPath);
BufferedReader b1 = new BufferedReader(currentFile);
String current = b1.readLine();
b1.close();
currentFile.close();
```

Si noti che anche in questo caso permane una dipendenza dal driver, infatti la directory nella quale esso andrà a scrivere i valori di tensione e di corrente varia a seconda del driver presente sul dispositivo, si tratta però di una limitazione molto meno forte rispetto a quella che si avrebbe con una soluzione di "basso livello".

Per completezza, sebbene non fornisca dati utili per il calcolo della potenza, presento anche il file contenente la carica residua del dispositivo: si tratta di un file contenuto nella stessa directory dei due precedenti, che va sotto il nome di

/sys/devices/platform/ds2784-battery/getmAh

Come è facilmente intuibile dal nome del file, il contenuto è un valore numerico che riporta la carica residua nel dispositivo misurata in mAh. Purtroppo, come già specificato in precedenza, i valori contenuti in questo file sono frutto di una stima, pertanto non sono utilizzabili all'interno del nostro studio.

Modello Globale

Come ho già spiegato, il punto di partenza di questo lavoro è rappresentato da uno studio sul consumo del modulo Wifi[3] che ha utilizzato un approccio di "basso livello", questo lavoro ha descritto il modello di consumo generale di un dispositivo come sommatoria dei consumi di tutte le componenti che costituiscono il dispositivo:

$$C = \sum_i f_i + g_i = B + P_s$$

Dove:

f_i è il consumo base della i-esima componente

g_i è il consumo legato ad una specifica attività sull' i-esima componente

B è la somma dei consumi base di tutti i componenti

P_s è la somma dei consumi legati ad una certa attività su tutte le componenti

Secondo questa definizione del modello, calcolando la somma del consumo base di tutti i componenti (B) e sottraendo tale valore da quello globale misurato in un dato momento e con un determinato pattern di applicazioni in esecuzione (C), è possibile ricavare la firma energetica di una specifica attività (P_s) e sulla base di questa informazione andare ad identificare eventuali attività maliziose.

Da queste considerazioni discende chiaramente la necessità di sviluppare modelli appositi per misurare i consumi di ogni componente hardware del dispositivo, tali modelli dovranno tenere conto delle caratteristiche fisiche e di funzionamento del componente hardware in questione, dovranno individuare i

parametri che ne caratterizzano il consumo e stabilire in quale modo questi parametri entrano in gioco e, qualora esista, definire qual è la relazione che li lega.

Una volta definiti i singoli modelli sarà poi necessario andare a verificare se la loro interazione sia lineare, descrivibile come semplice somma dei consumi di ogni dispositivo, oppure se si tratti di qualcosa di più complesso, caratterizzato da una relazione non lineare.

Confronto tra approcci low level ed high level

Il lavoro riguardante il componente Wifi ha portato alla definizione delle firme energetiche di alcune applicazioni, tra cui Skype e Youtube, e di alcuni tipi di attacco quali il ping flood ed una serie ripetuta di HTTP GET request.
Attraverso il modello elaborato, note queste firme energetiche, è stato possibile riconoscere gli attacchi quando venivano effettuati sulle applicazioni in normale esecuzione.

L'identificazione degli attacchi è avvenuta in condizioni facilmente replicabili, ossia spegnendo ogni componente del dispositivo che potesse richiedere energia alla batteria e lasciando accesi solamente il modulo Wifi e la CPU e realizzando una misura a basso livello.

Un primo controllo che ho potuto fare riguarda proprio la possibilità di eseguire queste identificazioni ad "alto livello" anziché a "basso livello", ho voluto cioè verificare se gli strumenti messi a disposizione dal sistema operativo sono sufficientemente precisi per permettere di rilevare le stesse cose che si sono ottenute con un approccio di livello più basso.
Per fare questo ho posto il dispositivo nelle stesse condizioni descritte in [3] nei vari casi, ed ho realizzato misurazioni relative ad un attacco ping flood, una conversazione su skype senza attacco ed un attacco ping flood avvenuto durante una conversazione su skype, in questo modo con le prime due misurazioni ho cercato di raccogliere informazioni sulle firme energetiche dei due scenari, e con la terza ho osservato se l'arrivo di un attacco ping flood su uno scenario conosciuto era riconoscibile.

Il risultato di questo esperimento è stato che, analogamente a quanto scritto in [3], ho registrato un innalzamento del valore di corrente erogata della batteria dopo che l'attacco è incominciato. Il grafico rappresentante l'erogazione di corrente ottenuto con le mie misurazioni ha la stessa forma di quello ottenuto con le misurazioni presenti in [3]; i valori non sono esattamente uguali probabilmente a causa di un diverso amperaggio delle due batterie.

Il risultato di questo test preliminare ha perciò mostrato che un approccio di "alto livello" sembrerebbe riuscire ad ottenere gli stessi risultati che si ottengono con un approccio di "basso livello".

In realtà, oltre all'attacco in corso su una chiamata skype, di cui riporto i grafici per confrontarli con i risultati ottenuti dal lavoro di riferimento, ho effettuato anche altri tipi di test, sebbene non fossero presenti in [3]; in particolare ho provato a riconoscere un attacco di tipo ping flood realizzato in tre diversi scenari:

- Normale funzionamento del telefono con connessione Wifi attiva
- Visione di un video con l'applicazione di Youtube
- Chiamata Skype in corso

In tutti gli scenari si è verificato che la corrente erogata dalla batteria fosse maggiore quando di aveva l'attacco in corso rispetto alla normale esecuzione. Si noti ancora che mentre il primo attacco è stato possibile effettuarlo mettendo a zero il contributo energetico dello schermo, disattivandolo, per gli altri due scenari non è stato possibile prescindere dallo schermo, nonostante ciò l'inizio dell'attacco era chiaramente rilevabile.

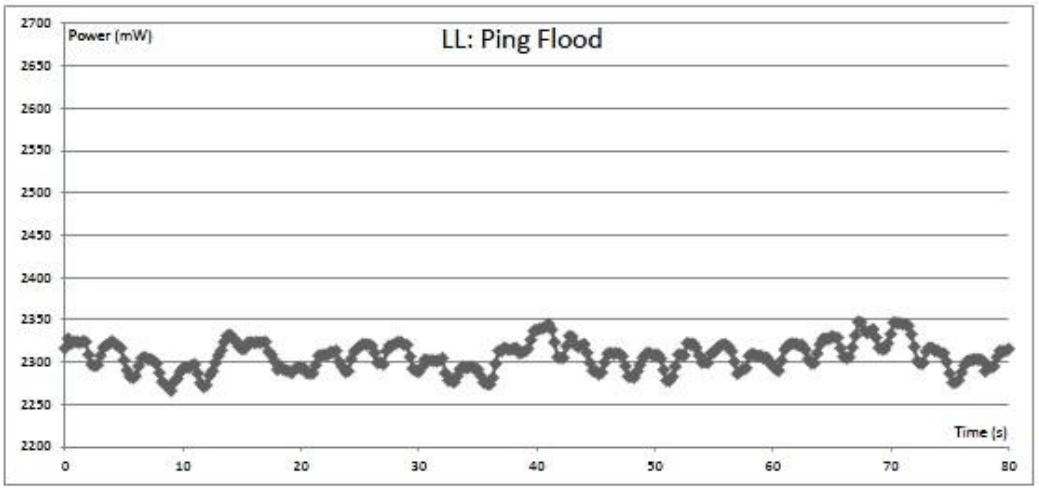

Fig.2: Misura basso livello Ping Flood

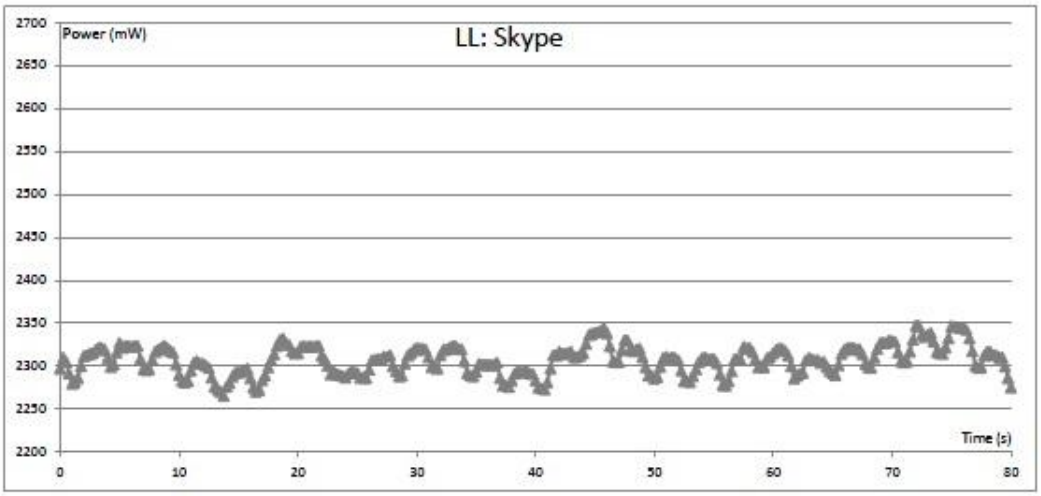

Fig.3: Misura basso livello Skype

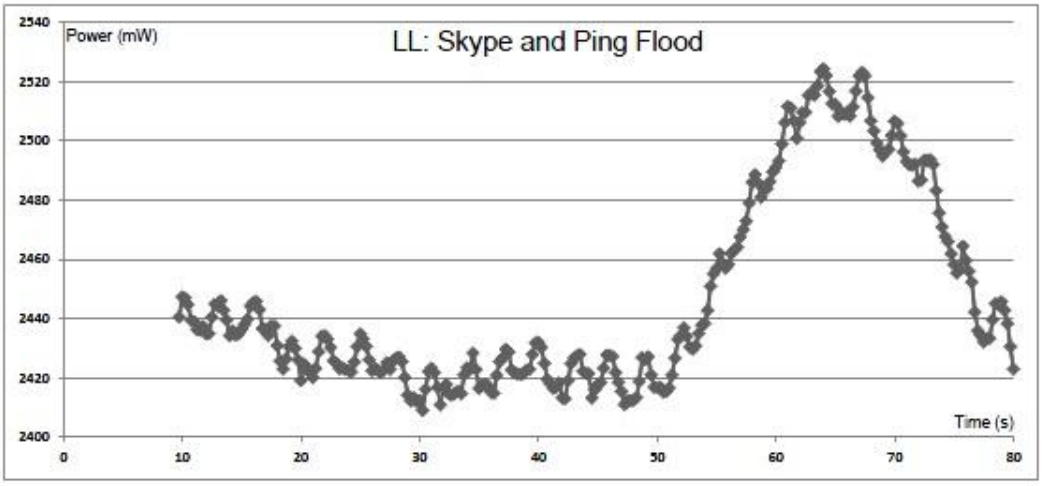

Fig.4: Misura basso livello Skype + Ping Flood

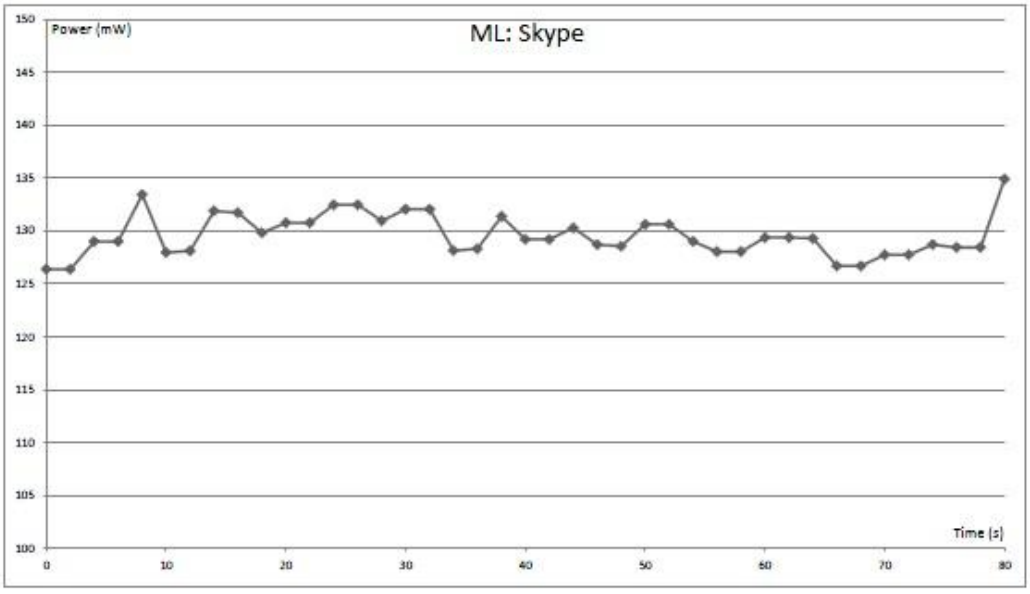

Fig.5: Misura alto livello Skype

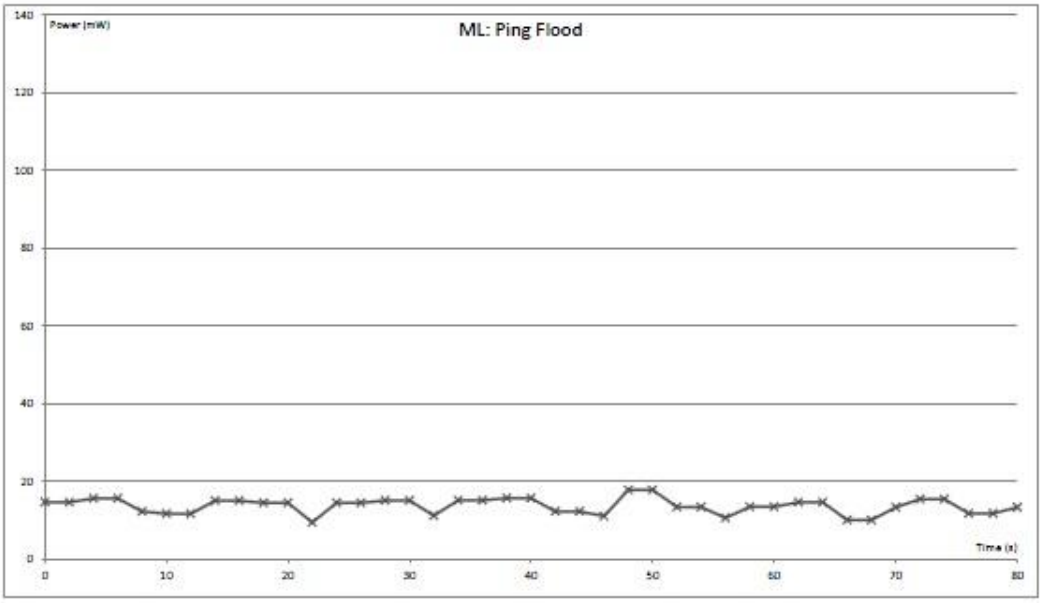

Fig.6: Misura alto livello Ping Flood

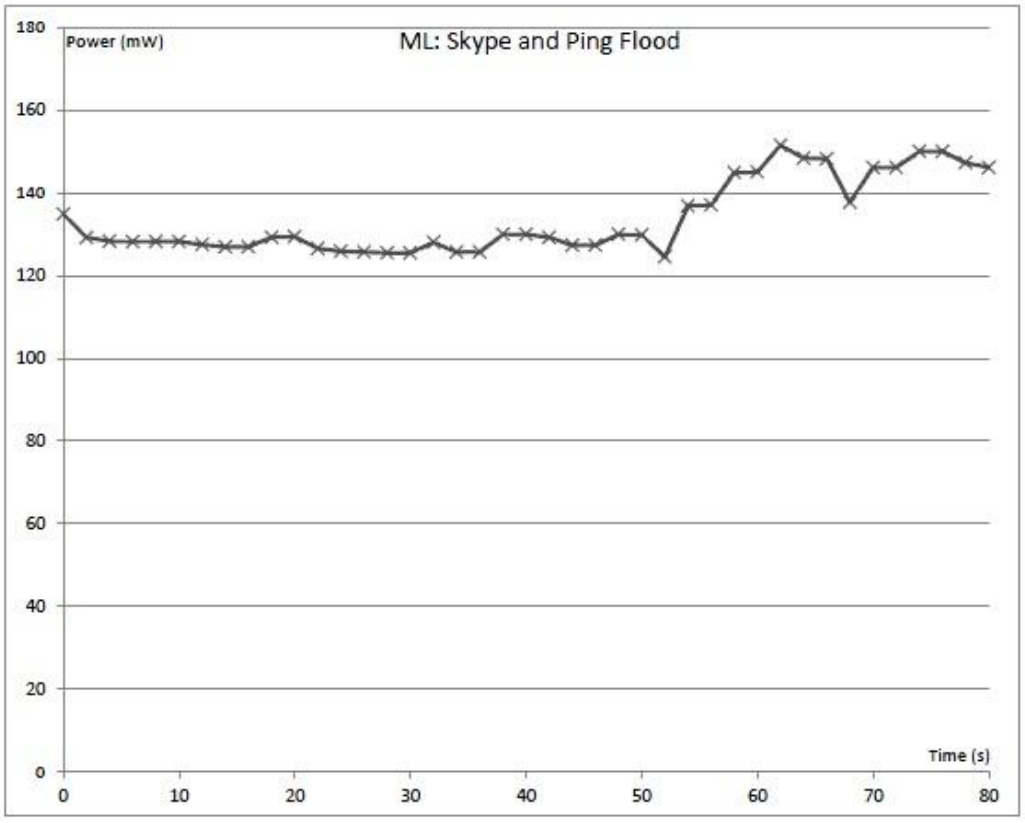

Fig.7: Misura alto livello Skype + Ping Flood

Per riuscire a misurare il consumo energetico ho scritto un'applicazione che permetteva di andare a leggere la corrente e la tensione campionata dal driver della batteria. Sperimentalmente mi sono reso conto che la tensione non aveva variazioni importanti durante il periodo di misurazione e si poteva considerare costante, pertanto ho successivamente modificato l'applicazione limitandola alla sola misurazione della corrente. Vediamo una parte significativa del codice:

```
private final class myThread extends Thread{

String curPath="/sys/devices/platform/ds2784-battery/getcurrent";
String frqPath =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq";
File sd = Environment.getExternalStorageDirectory();
File outputFile = new File(sd, "correnti.txt");

public myThread(){
}

public void run(){

      long startTime = System.currentTimeMillis();
      long endTime = startTime;
      int mSec = sec * 1000;

      try{
            FileWriter fw = new FileWriter(outputFile);
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(bw);
```

```
        while(endTime - startTime < mSec){

                FileReader currentFile = new FileReader(currentPath);
                FileReader curFreqFile = new FileReader(curFreqPath);
                BufferedReader b1 = new BufferedReader(currentFile);
                BufferedReader b2 = new BufferedReader(curFreqFile);

                String current = b1.readLine();
                String frequency = b2.readLine();
                b1.close();
                b2.close();
                currentFile.close();
                curFreqFile.close();

                String output = frequency+" "+current;
                pw.println(output);

                Thread.sleep(3000);

                endTime = System.currentTimeMillis();
        }

        pw.close();
        bw.close();
        fw.close();
}
catch(IOException e){
        e.printStackTrace();}
catch(InterruptedException e){
        e.printStackTrace();
}
wl.release();
stopSelf();
}}
```

Questo è il thread che si occupa delle misurazioni, è avviato da un servizio Android ed acquisisce il wake lock parziale al momento della sua creazione, si tratta del wake lock che permette l'esecuzione da parte della CPU mantenendo però lo schermo spento. Il thread si occupa di leggere i valori della corrente ogni 3 secondi: effettuare letture più fitte del file della corrente è inutile poiché viene aggiornato ogni 3,52 secondi.

Un comportamento di questo tipo permette di raccogliere i dati energetici di cui ho bisogno per il mio lavoro, senza che il campionatore rappresenti una fonte di consumo energetico tale da influenzare sensibilmente le misurazioni stesse.

Il thread esegue per una quantità di tempo che viene impostata, attraverso un Intent, dall'applicazione chiamante; esegue per questa quantità di tempo leggendo i valori della corrente dal file, dopodiché rilascia il Wake Lock ed interrompe l'esecuzione del servizio attraverso l'uso della funzione stopSelf().

Attacchi di rete

Gli attacchi di rete provati in [3] sono sostanzialmente 3: Ping Flood, Port Scanning ed un attacco composto da un flood di pacchetti HTTP GET request. Vediamoli un po' più nel dettaglio:

Ping Flood

Si tratta di un attacco di tipo Denial Od Service (DoS) caratterizzato dal fatto che il dispositivo/sistema dell'utente si ritrova sommerso da una grandissima quantità di pacchetti ICMP Echo Request, o più comunemente ping.

Questo tipo di attacco ha successo solo se l'attaccante dispone di molta più banda rispetto al sistema attaccato. La speranza dell'attaccante è che il sistema attaccato risponda con pacchetti ICMP Echo Reply, consumando così banda in uscita, oltre a quella consumata per i pacchetti in entrata.

Eseguire un attacco di questo tipo è molto semplice, infatti è sufficiente conoscere l'indirizzo IP del bersaglio ed eseguire questo semplice script da terminale:

sudo ping – f –s 56500 192.168.1.100

In particolare il comando –f serve per indicare di non frammentare il pacchetto, -s invece indica la dimensione del pacchetto; l'ultimo campo è l'indirizzo IP del bersaglio.

Port Scan

Si tratta di una tecnica utilizzata per raccogliere informazioni su un computer connesso alla rete, stabilendo quali porte siano in ascolto su una determinata macchina. L'attacco consiste nell'inviare richieste di connessione al computer bersaglio: osservando le risposte è possibile verificare quali servizi siano attivi su una determinata macchina.

Una porta si dice aperta se c'è un'applicazione che la utilizza e si distingue se dall'host arriva una risposta, una porta si dice chiusa se l'host invia risposte di tipo ICMP Port Unreachable, ed infine si dice bloccata o filtrata se non si ha avuto alcuna risposta dall'host, cosa cha avviene se c'è un firewall o un ostacolo di rete che blocca l'accesso alla porta rendendone non identificabile lo stato. Solitamente si tratta di un attacco che viene fatto non per danneggiare il sistema ma per raccogliere informazioni su di esso.

HTTP GET Request flood

Questo non è uno degli attacchi standard eseguito da un malintenzionato nei confronti di un bersaglio, consiste nel mandare una serie di HTTP GET request sul sistema del bersaglio, provando ad ottenere un effetto simile al ping flood.

Parametri di interesse per il modello della CPU

In una fase iniziale del lavoro mi sono trovato a dover definire quali fossero i parametri che influenzano effettivamente il consumo della CPU: in primo luogo ho considerato la frequenza alla quale il processore sta lavorando, che è indubbiamente un parametro fondamentale per le CPU, quindi ho pensato che questo valore potrebbe essere legato al carico di lavoro cui il processore è sottoposto, ossia l'utilizzazione della CPU, intuizione che mi è poi stata confermata dalla documentazione riguardante il frequency scaling: infatti l'utilizzazione risulta essere una componente determinante per la gestione del power saving.

Si sarebbero potute fare anche altre considerazioni, ossia tenere conto del fatto che non sempre tutti i componenti della CPU sono attivi contemporaneamente, quindi un'ulteriore soluzione sarebbe potuta essere andare a controllare quali componenti sono attivi e verificare il consumo quando questi vengono disattivati, tuttavia si tratta di una soluzione difficilmente realizzabile, poiché le informazioni sui singoli componenti della CPU non sono facilmente reperibili tramite i files del sistema operativo.

Un'altra idea possibile sarebbe potuta essere quella che prevede di andare a verificare il comportamento della CPU nel momento in cui si passa ad uno stato di low consumption, questo sarebbe stato possibile andando a controllare in quale stato si trova il processore, intendendo per stati i possibili C-States. Non ho optato per questa soluzione poiché non su tutti i dispositivi è possibile andare a distinguere tra i vari C-State nel momento in cui si passa in uno stato di low consumption, inoltre il funzionamento descritto nella documentazione

di cpufreq dice proprio che il comportamento prevede di abbassare la frequenza con l'utilizzazione. Per questi motivi non ho approfondito questi aspetti, che però potrebbero rappresentare un'idea per un raffinamento del modello proposto.

Quindi il modello che ho sviluppato considera come parametri l'utilizzazione della CPU e la frequenza alla quale sta lavorando.

Ricerca dei dati di interesse per il modello

Un aspetto importante per il lavoro svolto è costituito dalla misura dei parametri al variare dei quali cambia il consumo, ossia il fatto di poter acquisire dal sistema operativo dati riguardanti la frequenza di esecuzione della CPU e la sua utilizzazione.

Per quanto riguarda la frequenza alla quale la CPU sta eseguendo, il kernel Linux mette a disposizione l'infrastruttura *cpufreq*, dalla quale è possibile leggere la frequenza corrente accedendo al file */sys/devices/system/cpu/cpu0/cpufreqscaling_cur_freq*.
Questo file è composto solo da una riga contenente la frequenza, pertanto è sufficiente effettuare una semplice lettura, ad esempio:

```
String curFreqPath =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq";
FileReader curFreqFile = new FileReader(curFreqPath);
BufferedReader b2 = new BufferedReader(curFreqFile);
String frequency = b2.readLine();
b2.close();
curFreqFile.close();
```

L'utilizzazione della CPU, invece, è reperibile in un file presente all'interno del filesystem *proc*, in particolare nel file */proc/stat*, che è costituito da una prima riga contenente i dati di esecuzione aggregati di tutti i processori del dispositivo; da un successivo numero di righe pari al numero di processori, ciascuna di queste righe contiene i dati di esecuzione relativi ad un processore, gli stessi dati che aggregati formano il contenuto della prima riga. Vi sono poi

altre righe che riportano altre statistiche che non sono di interesse al fine del lavoro che ho svolto, ad esempio il numero di interrupts serviti a partire dal boot time, il numero di context switches, il numero di fork dal momento del boot, il numero di processi running e blocked ed altro ancora. Si noti che il contenuto di questo file può leggermente cambiare a seconda dell'architettura, ma la parte di nostro interesse è una parte standard.

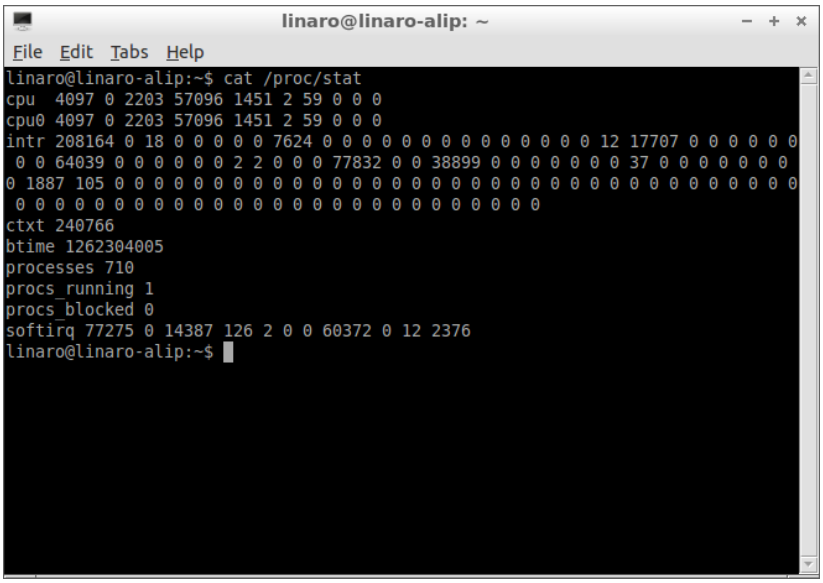

Fig.8: Il contenuto di proc/stat

Vista la struttura di questo file, è chiaro che per ottenere l'utilizzazione del processore è sufficiente andare a leggere la riga del processore di interesse, separare i tempi di esecuzione dai tempi di sleep e fare alcuni semplici calcoli. Ovviamente per l'utilizzo globale del dispositivo è sufficiente leggere la prima riga e lavorare sui dati in essa contenuti, nel mio caso questo problema non si è presentato poiché il dispositivo che ho utilizzato è un single-core, pertanto il file che sono andato a leggere era composto solamente da una riga per l'utilizzo complessivo ed un'altra riga uguale con l'utilizzo dell'unico processore a disposizione.

I valori contenuti nella prima riga e nelle righe successive relative ai singoli processori indicano, in ordine:

- Il tempo trascorso in User Mode
- Il tempo trascorso in User Mode con priorità bassa (processi nice)
- Il tempo trascorso in System Mode
- Il tempo trascorso in idle
- Il tempo trascorso in attesa del completamento di operazioni di I/O
- Il tempo trascorso per servire gli interrupt
- Il tempo trascorso per servire softirqs
- Il tempo speso per operazioni di sistema eseguite in ambiente virtualizzato
- Il tempo trascorso a virtualizzare una CPU per un S.O. guest
- Il tempo trascorso a virtualizzare una CPU niced per un S.O. guest

Quindi, per ottenere il tempo di esecuzione complessivo nel periodo è necessario sommare tutti i valori ad eccezione del quarto e del quinto, che congiuntamente costituiscono il tempo di sleep.
Vediamo ora un esempio del codice necessario per andare a recuperare i valori contenuti all'interno di questo file:

```
String usagePath = "proc/stat";
FileReader myFile = new FileReader(usagePath);
BufferedReader b1 = new BufferedReader(myFile);
String s = b1.readLine();
StringTokenizer st = new StringTokenizer(s);
st.nextToken();
long userExTime = Long.decode(st.nextToken());
```

```
long niceExTime = Long.decode(st.nextToken());
long systExTime = Long.decode(st.nextToken());
st.nextToken();
st.nextToken();
long irqExTime = Long.decode(st.nextToken());
long softirqTime = Long.decode(st.nextToken());
b1.close();
myFile.close();
long tot_ex_time = userExTime + niceExTime + systExTime +
irqExTime + softirqTime;
```

Come si può vedere, in questo esempio di codice si vanno a recuperare i valori relativi al tempo di esecuzione, pertanto vi sono due istruzioni st.nextToken() che permettono di saltare i valori relativi allo sleep.

L'utilizzo totale della CPU si ottiene aggregando i valori che indicano tempi di esecuzione, saltando come abbiamo appena detto quelli che riguardano tempi di sleep. Noti il tempo di esecuzione totale ed il tempo di sleep totale è possibile ricavare l'utilizzo percentuale della CPU e, di conseguenza, anche la percentuale di non utilizzo.

Imposizione di un carico di lavoro alla CPU

Da quanto visto finora si capisce che siamo in grado di riconoscere l'utilizzazione mantenuta dalla CPU in un certo intervallo di tempo, infatti leggendo le righe del file */proc/stat* in due istanti di tempo iniziale e finale, è possibile ricavare l'utilizzazione nel periodo considerato come una semplice differenza.

$$C_{CPU}(f , u) = C_{CPU}(f , u_f , u_i)$$

Ove f è la frequenza, u_f l'utilizzazione finale e u_i l'utilizzazione iniziale

A questo punto sorge però un problema: l'utilizzazione è monitorabile con le soluzioni proposte, ma si tratta di una quantità che varia in modo generalmente incontrollato, durante una normale esecuzione.

Per determinare se c'è effettivamente una relazione tra l'utilizzo della CPU e la potenza consumata dalla batteria è però necessario avere un modo per imporre un certo carico di lavoro alla CPU e mantenerlo costante durante un certo intervallo di tempo.

Per fare questo ho progettato un'applicazione che esegue calcoli fittizi con lo scopo di mantenere in esecuzione la CPU per il tempo necessario ad una certa percentuale di utilizzazione. Riporto di seguito il codice del servizio in cui risiede il thread che si occupa di sollecitare la CPU:

```
public class cpuOverhead extends Service{

	ThreadCounterOverhead tco = null;
	PowerManager pm = null;
	PowerManager.WakeLock wl = null;

	public IBinder onBind(Intent arg0) {
		return null;
	}

	public void onCreate(){
	 pm= (PowerManager)getSystemService(Context.POWER_SERVICE);
	 wl= pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "tag");
	}

	public int onStartCommand(Intent i, int flags, int strtId){
		wl.acquire();
		int usage = intent.getExtras().getInt("usage");
		int tempo = intent.getExtras().getInt("tempo");
		tco = new ThreadCounterOverhead(usage, tempo);
		tco.start();
		return super.onStartCommand(intent, flags, startId);
	}

	public void onDestroy(){
		super.onDestroy();
	}

	private final class ThreadCounterOverhead extends Thread{

		int usage, time, notUsage;
		String usagePath = "proc/stat";
		File outputFile = null;
```

```
            public ThreadCounterOverhead(int use, int t){
              super();
              usage = use;
              notUsage = 100 - usage;
              time = t * 1000;
              File sd= Environment.getExternalStorageDirectory();
              outputFile = new File(sd, "OverheadCPUoutput.txt");
            }

// Ritorna arraylist con le righe di proc/stat relative alle
singole cpu. (dalla seconda alla 2+#cpu).
            public ArrayList<String> readCpusStrings() throws
IOException{
                  ArrayList<String> cpuValues = new
ArrayList<String>();
                  FileReader useFile = new FileReader(usagePath);
                  BufferedReader b1 = new BufferedReader(useFile);
                  String s = b1.readLine(); // Salto la prima riga
                  s = b1.readLine();
                  while(s != null){
                        StringTokenizer st = new
StringTokenizer(s);
                        String firstWord = st.nextToken();
                        if(firstWord.startsWith("cpu")){
                              cpuValues.add(s);
                        }
                        else{
                              break;
                        }
                        s = b1.readLine();
                  }
                  b1.close();
                  useFile.close();
```

```
        return cpuValues;
}

public void run(){

  long timeUnit = 100;
  long start = System.currentTimeMillis();
  long current = start;
  long execTime = timeUnit * usage;
  long sleepTime = timeUnit * notUsage;
  int dummy = 0;

  try{

      ArrayList<String> listLine = readCpusStrings();
      long initialExTime = getTotalUsage(listLine);
      long initialSleepTime = getTotalSleep(listLine);

      while(current - start < time){

            long intStart= System.currentTimeMillis();
            long intCurr = intStart;

            if(execTime != 0){
               while(inteCurr-intStart < execTime){
                   dummy++;
                   intCurr=System.currentTimeMillis();
            }}

            if(sleepTime != 0){
                  try {
                        Thread.sleep(sleepTime);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
```

```
                    }


                    current = System.currentTimeMillis();
                    }// Chiusura del while

          ArrayList<String> listLine2 = readCpusStrings();
          long finalExTime = getTotalUsage(listLine2);
          long finalSleep = getTotalSleep(listLine2);
          long totUse = finalExTime - initialExTime;
          long totSlp = finalSleep - initialSleepTime;
          printResults(String.valueOf(totUse),
      String.valueOf(totSlp));
}catch(IOException e){
   e.printStackTrace(); }

wl.release();
stopSelf();

}// Fine metodo run



public long getTotalSleep(ArrayList<String> als){
      long totalSleepTime = 0;
      for(int x = 0; x < als.size(); x++){
            String s = als.get(x);
            StringTokenizer st = new StringTokenizer(s);
            st.nextToken();
            st.nextToken();
            st.nextToken();
            st.nextToken();
            long initSleep = Long.valueOf(st.nextToken());
            long initIOWait = Long.valueOf(st.nextToken());
```

```
                totalSleepTime = totalSleepTime + initSleep
+initIOWait;}
            return totalSleepTime;
      }

            // Ritorna l'utilizzo totale, ossia la somma degli
utilizzi su tutte le CPU.
            public long getTotalUsage(ArrayList<String> als){

                  long totalExTime = 0;
                  for(int x = 0; x < als.size(); x++){
                        String s = als.get(x);
                        StringTokenizer st = new
StringTokenizer(s);
                        st.nextToken();
                        long userExTime =
Long.decode(st.nextToken());
                        long niceExTime =
Long.decode(st.nextToken());
                        long systExTime =
Long.decode(st.nextToken());
                        st.nextToken();
                        st.nextToken();
                        long irqExTime =
Long.decode(st.nextToken());
                        long softirqTime =
Long.decode(st.nextToken());

                        totalExTime = totalExTime + userExTime +
niceExTime + systExTime + irqExTime + softirqTime;
                  }
                  return totalExTime;
            }
```

```
            // Stampa i risultati
            public void printResults(String esecuzione, String
sleep) throws IOException{
                  FileWriter fw = new FileWriter(outputFile,
true);
                  BufferedWriter bw = new BufferedWriter(fw);
                  PrintWriter pw = new PrintWriter(bw);
                  String myOutput = "Esecuzione: "+esecuzione+",
sleep: "+sleep;
                  pw.println(myOutput);
                  pw.close();
                  bw.close();
                  fw.close();
            }

      }

}
```

Il thread che genera overhead per la CPU riceve il tempo di esecuzione e la percentuale di utilizzazione da imporre al processore attraverso l'intent che avvia il servizio, è ovviamente necessario acquisire il Wake Lock parziale, in modo da permettere l'esecuzione al processore anche con lo schermo spento.
All'avvio del servizio viene generato un thread che si occupa di sollecitare la CPU con il carico desiderato; tale thread rimane in esecuzione per il tempo indicato nell'intent, e divide questa quantità di tempo in quanti da 100ms durante i quali esegue per una certa quantità di tempo e rimane in sleep per la

rimanente quantità di tempo, in accordo con le percentuali di utilizzazione che vengono stabilite nell'activity con la quale l'utente interagisce.

Un'applicazione di questo tipo induce alla CPU un carico computazionale che dovrebbe essere simile a quello indicato nominalmente. In realtà bisogna considerare che la CPU è un componente che non è caratterizzato soltanto dall'esecuzione delle applicazioni utente, ma risente molto dell'influenza del sistema operativo, che potrebbe avviare calcoli ed eseguire routine che potrebbero far variare l'utilizzazione nominale imposta dall'applicazione.

Per questo motivo è stato necessario verificare qual è il reale impatto del sistema operativo sull'utilizzazione utente, per fare questo è stato sufficiente imporre utilizzazioni nominali all'applicazione e farla eseguire per il tempo desiderato, dopodiché andare a verificare di quanto varia il comportamento effettivo rispetto a quello nominale.
Il risultato di questa verifica ha evidenziato il fatto che il sistema operativo non introduce una variazione del carico significativa ai fini della nostra analisi, pertanto si può considerare affidabile il carico nominale imposto dall'applicazione.

Nelle successive fasi del lavoro ho potuto verificare più attentamente qual'era la variazione nell'utilizzazione al variare dell'utilizzazione imposta nominalmente, osservando che il carico computazionale effettivo non si discostava mai dal valore nominale per più del 2%.

Imposizione di una frequenza di lavoro alla CPU

Il modello che ho definito prevede che il consumo totale della CPU in un certo periodo di esecuzione sia determinato dalla somma dei consumi alle diverse frequenze per il tempo in cui si rimane ad una determinata frequenza.
Alla luce di ciò, si rende necessario misurare l'energia consumata dal processore quando si trova alle diverse frequenze, per fare questa misurazione bisognerebbe che la frequenza della CPU non variasse per un prefissato periodo di tempo durante il quale effettuare la misurazione. Ovviamente il comportamento "naturale" del processore non è mai di questo tipo e le frequenze variano nel tempo, pertanto si rende necessario trovare un modo per fissare una determinata frequenza.
Una volta fissata la frequenza, sarà possibile effettuare la misurazione dei consumi alle varie utilizzazioni.
Il driver *cpufreq* mette a disposizione dell'utente due files attraverso i quali è possibile andare ad impostare la frequenza di lavoro massima e minima del processore, tutto semplicemente scrivendo sui due files:

per la frequenza massima:
/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq

per la frequenza minima:
/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq

L'unico problema che si riscontra è che questi files risultano di sola lettura per l'utente, è possibile tuttavia effettuare la scrittura attraverso un semplice comando da terminale, dopo aver acquisito i privilegi di root.

Ovviamente non è pensabile cambiare manualmente il valore scritto nei due files prima di ogni misurazione, si rende necessario un metodo più automatico che deleghi questo compito ad un'applicazione.
È possibile cambiare la frequenza di lavoro dal codice java di un'applicazione Android richiamando poche semplici istruzioni:

```
String maxFreqPath =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
String minFreqPath =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
Process su = Runtime.getRuntime().exec("su");
DataOutputStream os = new DataOutputStream(su.getOutputStream());
os.writeBytes("echo "+frequenza_massima+" >> "+maxFreqPath+"\n");
os.flush();
os.writeBytes("echo "+frequenza_minima+" >> "+minFreqPath+"\n");
os.flush();
os.writeBytes("exit\n");
os.flush();
os.close();
su.waitFor();
```

Il concetto è che ogni applicazione java si interfaccia con l'ambiente in cui sta eseguendo attraverso la classe *Runtime*; tale classe può essere ottenuta con il metodo *getRuntime()*.

Attraverso il metodo *exec(cmd)* è possibile eseguire specifici comandi di sistema, in particolare nell'esempio sopra viene utilizzato per acquisire i privilegi di root (comando "su") e per scrivere sul file desiderato.

Si noti che il driver *cpufreq* gestisce eventuali casi in cui si vada a cercare di scrivere una frequenza massima minore della minima o viceversa, impedendo di modificare i valori di frequenza attuali.
Questo introduce un problema, perché siccome devo aggiornare due files non posso scegliere di aggiornare prima un file dell'altro in maniera arbitraria, ma è necessario gestire il caso in cui si voglia impostare una frequenza massima minore della frequenza minima attuale, oppure il caso in cui si voglia impostare una frequenza minima maggiore della frequenza massima attuale, in questi casi è necessario non far "incrociare" mai la frequenza massima e minima, ossia non ricadere nel caso descritto prima con la frequenza massima minore della minima e viceversa, anche se temporaneamente, perciò ho dovuto gestire l'aggiornamento delle frequenze considerando lo spostamento richiesto.

L'unico inconveniente relativo a questo codice è il fatto che sono richiesti i permessi di root, cosa che richiede che il dispositivo possa concederli alle applicazioni. Di default sugli smartphone questa possibilità è negata, pertanto è necessario sbloccare il dispositivo. Vediamo brevemente cosa significa ottenere i permessi di root:

Il termine *root* deriva dai sistemi Unix/Linux ed indica l'utente dotato dei massimi privilegi di amministrazione del sistema, l'equivalente di quello che in ambiente Windows viene chiamato *Administrator*.

Si dice comunemente root (Oppure rooting o anche acquisizione dei permessi di root) il processo di modifica dei dispositivi volto a prenderne il pieno controllo. I benefici di questa procedura stanno, ai fini del mio lavoro, a poter accedere in scrittura ai files di frequenza massima e frequenza minima del sistema. In realtà i benefici consistono nella possibilità di eseguire l'applicazione Super User, infatti visto che i comandi che ho spostato dentro alle applicazioni i comandi che dovrebbero eseguire da terminale, saranno le applicazioni a dover beneficiare dei permessi in questione, attraverso Super User è possibile concedere i privilegi di root alle applicazioni.

Modello del consumo della CPU

Come abbiamo detto in precedenza, i parametri di interesse per determinare il consumo della CPU sono la frequenza del clock e l'utilizzazione, è perciò necessario capire in che modo questi due parametri influenzano il consumo energetico.

Come riporta la documentazione di *cpufreq*, a seconda del carico computazionale cui è sottoposto il processore si effettua una procedura di frequency scaling volta a ridurre il consumo energetico. Quindi al diminuire del carico computazionale, il sistema operativo diminuirà la frequenza del clock e di conseguenza il consumo energetico.
Il problema che si presenta di fronte alla conoscenza di questo funzionamento è capire esattamente come avvenga la procedura di frequency scaling; non avendo indicazioni esatte riguardo alle condizioni (si intende a che livello di utilizzazione) per cui si scala da una frequenza all'altra, diventa difficile definire quale sia il comportamento esatto.

L'idea che ho avuto, di fronte alla variazione non esattamente predicibile delle frequenze, è stata cercare qual è il comportamento delle sistema alle varie frequenze ed alle varie utilizzazioni, in modo da poter definire un andamento dei consumi al variare dei due parametri. In questo modo non ha più importanza sapere a che livello di utilizzazione si effettua lo scalamento di frequenza, infatti sapendo qual è il consumo energetico ad ogni frequenza ed utilizzazione è sufficiente sapere qual è stato il carico computazionale del processore nel periodo considerato e per quanto di questo tempo è stato in ogni frequenza.

Questo ragionamento si basa sul presupposto che le frequenze in cui può trovarsi la CPU sono un numero limitato, ovviamente se la CPU potesse variare in maniera continua tra la frequenza minima e massima ammissibile questo ragionamento non sarebbe probabilmente realizzabile in pratica.
Quindi il modello consiste nel creare una relazione che indichi per ogni frequenza il consumo ad una certa utilizzazione e, sulla base di questo consumo, calcolare il consumo totale della CPU come somma dei consumi parziali relativi ad ogni intervallo di tempo trascorso ad una certa frequenza.

$$C_{CPU}(f, u) = \Sigma_k C_{CPU}(f_k, u_f, u_i)$$

Il modello funziona per via del fatto che all'interno del kernel Linux è presente un file che indica il tempo passato dal sistema ad ogni frequenza. L'unità di misura temporale del valore contenuto in questo file è 10ms.
Questo file si trova alla directory:

/sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state

Questo file si presenta sottoforma di tabella di corrispondenze tra ogni frequenza disponibile e la quantità di tempo passata a quella frequenza.
Dato che questo file viene inizializzato a zero nel momento in cui il dispositivo viene acceso, per rendersi conto del tempo trascorso ad ogni frequenza è necessario leggerne il contenuto due volte: una all'inizio del periodo considerato ed una alla fine di tale periodo, dopodiché per differenza si otterrà il tempo reale trascorso ad ogni frequenza.

La presenza di questo file è fondamentale per il corretto funzionamento del modello, infatti senza di esso sarebbe impossibile definire quanto tempo viene trascorso ad ogni frequenza, in quanto l'unico metodo alternativo sarebbe un campionamento del file contenente la frequenza corrente all'interno del file system *cpufreq*, tuttavia questa non è una soluzione percorribile poiché non si ha la conoscenza esatta del comportamento del governor, pertanto non è possibile sapere quando si ha una transizione di frequenza, in modo da poter misurare l'ora in corrispondenza delle transizioni e ricavare per differenza il tempo passato ad una certa frequenza. Allo stesso modo non è possibile pensare di andare a campionare il file contenente la frequenza corrente così fittamente da scoprire con precisione sufficiente quanto tempo è stato speso ad una determinata frequenza. Questo secondo metodo non è realizzabile poiché il carico computazionale introdotto dal meccanismo di lettura introdurrebbe troppo overhead e falserebbe il carico computazionale introdotto dall'applicazione scritta da me.

Raccolta dei dati per la realizzazione del modello

Per raccogliere i dati necessari a realizzare il modello ho utilizzato l'applicazione già presentata in precedenza per la lettura della corrente e l'applicazione per imporre al processore un carico di lavoro, anche questa già presentata in precedenza.

Per avere un buon numero di campioni e non far risentire troppo di possibili "sbalzi" nella richiesta di energia alla batteria ho deciso di effettuare misurazioni della durata di 10 minuti. Vista la quantità di frequenze a disposizione e considerato il fatto che molte di esse hanno valori non molto differenti, in un primo momento ho preso in considerazione soltanto frequenze abbastanza "distanti" tra loro, in questo modo è possibile costruire un andamento dei consumi e ricavare il consumo alle frequenze intermedie mediante interpolazione.

La granularità delle frequenze, come ho già detto, è stata definita scegliendo le frequenze che non erano "vicine", definendo come soglia di vicinanza una quantità di circa 100MHz, in questo modo ho eseguito le misurazioni per circa la metà delle frequenze disponibili ed ho ottenuto i valori relativi alle rimanenti mediante una semplice interpolazione. Per quanto riguarda la granularità delle utilizzazioni, invece, ho scelto di utilizzare step che aumentassero del 10% il carico in ogni misurazione, realizzando così prove dal a salire fino al 100% di utilizzazione nominale.

Gli esperimenti volti a raccogliere il consumo energetico ad ogni frequenza, quindi, sono stati svolti fissando una ogni frequenza con l'apposita

applicazione e misurando la corrente erogata mentre l'applicazione con il compito di generare carico computazionale eseguiva sul dispositivo.

I risultati ottenuti hanno mostrato che mantenendo fissata una determinata frequenza ed abbassando l'utilizzazione del processore, si ottiene un abbassamento dell'energia erogata dalla batteria.

Allo stesso modo, osservando i dati, si può notare che fissando una certa utilizzazione si ottiene un decremento della potenza erogata al diminuire della frequenza.

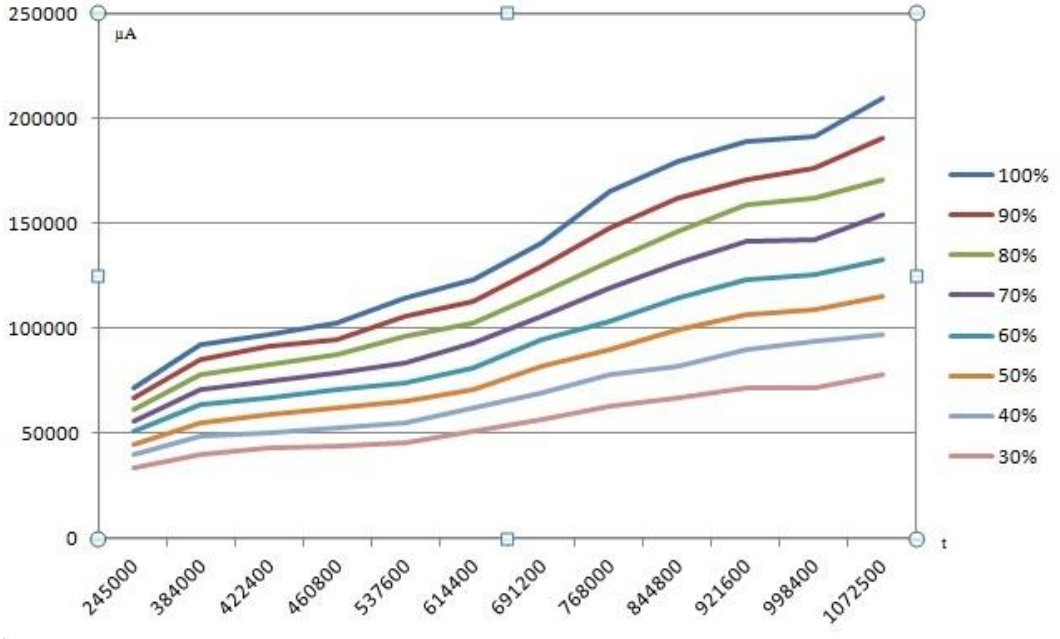

Fig.9: Andamento dei consumi alle varie utilizzazioni

Osservando il grafico in figura 9, nel quale sull'asse delle x abbiamo le frequenze cui il processore può trovarsi, mentre sull'asse delle y è riportato il consumo di corrente, si può notare che l'energia richiesta una volta fissata una certa utilizzazione cresce al crescere della frequenza.

Allo stesso modo, osservando che i grafici hanno tutti la stessa forma ma subiscono una sorta di traslazione verso l'alto al crescere dell'utilizzazione, possiamo concludere che fissata una determinata frequenza e facendo crescere il carico computazionale, cresce la corrente erogata dalla batteria.

Alla fine di questa fase di calibrazione del modello ho raccolto abbastanza dati per poter stimare il consumo energetico del dispositivo, una volta noto il comportamento in frequenza e l'utilizzazione nel periodo considerato.

Faccio altresì notare che un modello di questo tipo è device-dependent, questo è ovvio, dal momento che la calibrazione è stata eseguita su un determinato dispositivo, caratterizzato da una certa batteria ed un certo consumo energetico; questo significa che con questi dati sarà possibile eseguire dei test riguardanti il lavoro fatto solamente sullo stesso dispositivo su cui sono state effettuate le misure, infatti per poter testare il modello su un dispositivo differente, sarebbe necessario calibrare il modello sul dispositivo in questione, in modo da avere i dati che descrivono il suo comportamento energetico.

Considerazioni riguardo la validazione del modello

Il modello che ho costruito è applicabile a qualsiasi dispositivo mobile Android-based, infatti che richiede informazioni che sono facilmente reperibili andando a leggere i files descritti in precedenza, che sono comuni in tutti i dispositivi basati sul sistema operativo Android.
Sebbene utilizzazione e frequenze disponibili siano facilmente ottenibili su tutti i dispositivi, la relazione che le lega potrebbe variare al variare del dispositivo, pertanto si tratta di una relazione device-dependent.
Questa caratteristica fa si che non sia possibile, ovviamente, calibrare il modello su un certo dispositivo e poi validarlo su un dispositivo differente. Per questa ragione sono vincolato ad effettuare i test di validazione sullo stesso dispositivo che ho utilizzato per costruire il modello.

Per effettuare una prova realistica sarebbe stato necessario effettuare alcuni test utilizzando applicazioni "reali" realizzate da terze parti, in modo da simulare il più possibile un comportamento reale.
Fare questo presenta però il problema che servirebbe un'applicazione dal comportamento molto particolare, che sfruttasse solamente la componente processore, senza richiedere l'ausilio di altre componenti.
Sebbene sarebbe stato possibile trovare una componente che non abbia bisogno di collegarsi ad internet o effettuare scambi con il modulo Bluetooth (e pertanto che avrebbe potuto funzionare in modalità aereo), né che avesse bisogno del modulo GPS, era pressoché impossibile trovare un'applicazione reale che facesse completamente a meno dell'interazione con l'utente e quindi del contributo dello schermo.

Descrizione degli esperimenti di validazione

Le condizioni in cui sono stati eseguiti gli esperimenti di validazione prevedono che la frequenza di esecuzione del processore rimanga costante durante tutta la durata della prova, che l'utilizzazione prefissata sia raggiunta suddividendo il tempo totale della misurazione in intervalli di esecuzione e riposo più piccoli e più numerosi rispetto a quelli utilizzati in fase di calibrazione. Per quanto riguarda la frequenza di campionamento della corrente consumata, quella è comandata dal rate di aggiornamento del file contenente la corrente erogata dalla batteria, pertanto ho effettuato lo stesso campionamento dei valori utilizzato durante la fase di calibrazione, ossia una misura ogni 3 secondi.

Per testare il modello che ho ideato, ho creato un'applicazione che si occupa di sollecitare la CPU del dispositivo ad eseguire dei calcoli; a prima vista potrebbe sembrare che questa applicazione abbia lo stesso compito che aveva l'applicazione che si occupava di creare carico computazionale per permettere di raccogliere i dati per la costruzione del modello, in effetti il compito delle due applicazioni è lo stesso, ma funzionano in maniera un po' differente.

L'applicazione già presentata, che ho utilizzato per generare il carico necessario per le misurazioni e la costruzione del modello, prevedeva che l'intervallo temporale in cui effettuo la misurazione sia suddiviso in intervalli di tempo piuttosto lunghi, dal punto di vista computazionale, invece l'applicazione che si occupa di generare il carico computazionale per la validazione prevede di suddividere il tempo totale della misurazione in intervalli di dimensione inferiore.

Per rendersi conto della variazione basti pensare che in fase di calibrazione, un carico computazionale del 50% corrispondeva ad intervalli di lavoro e riposo da 500ms, invece in fase di validazione quel carico computazionale corrisponde ad intervalli di 50ms.

La scelta di costruire gli intervalli in questo modo è dovuta al desiderio di rendere l'andamento lavoro/riposo maggiormente altalenante, in modo che non possa verificarsi il caso in cui si vada a campionare principalmente valori in un momento di riposo piuttosto che in un momento di lavoro.

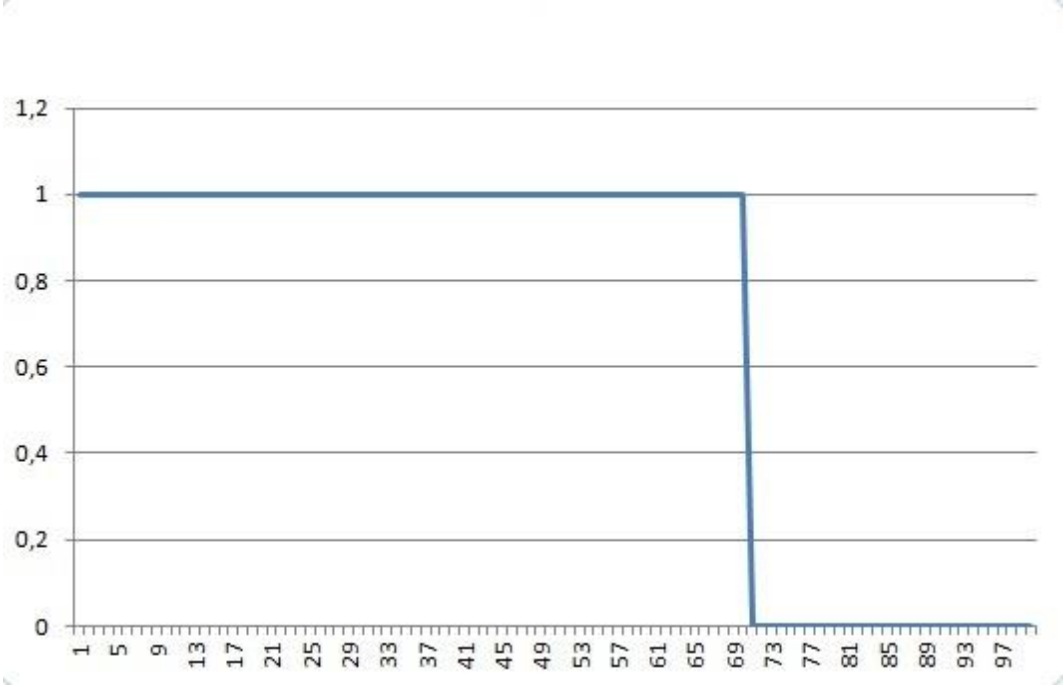

Fig.10: Intervallo di esecuzione in fase di costruzione del modello

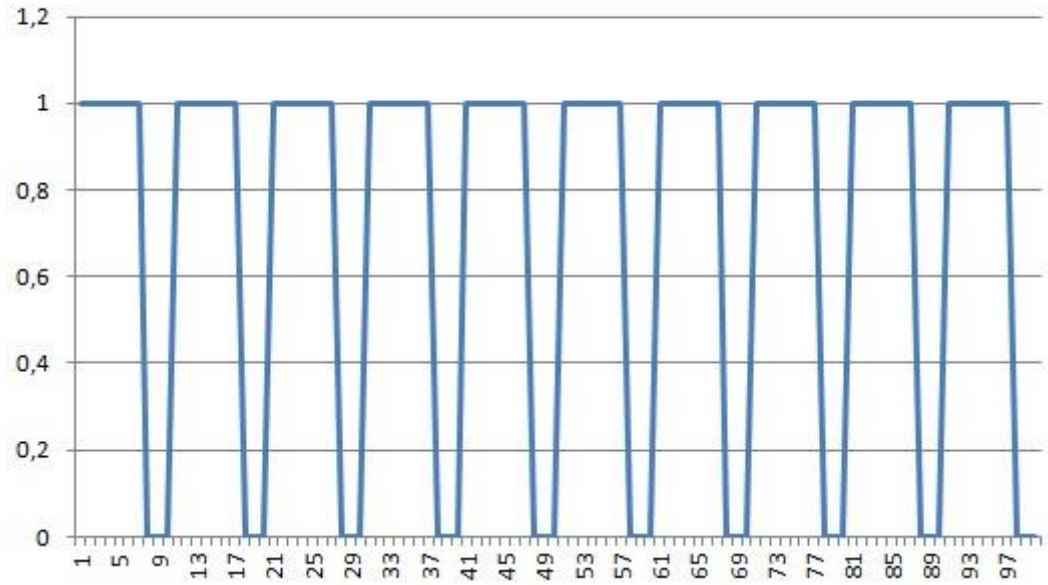

Fig.11:Intervallo di esecuzione in fase di validazione del modello

Osservando i grafici, si può vedere la struttura ingrandita di uno degli intervalli di esecuzione in cui è diviso l'intervallo temporale. È abbastanza chiara la differenza tra l'andamento costante per lunghi periodi che si ottiene con l'esecuzione della prima applicazione, e quello molto più altalenante che si ottiene con la seconda applicazione.

L'applicazione che genera il carico computazionale per la validazione del modello prevede inoltre di poter effettuare esperimenti replicabili, è infatti possibile andare a definire la percentuale di utilizzazione ed il tempo di esecuzione attraverso un file apposito che deve essere posizionato sulla scheda SD esterna contenuta nel dispositivo.
L'applicazione andrà a leggere da questo file al momento dell'esecuzione, per stabilire il modo in cui deve eseguire i calcoli che generano carico computazionale.

```
public class cpuOverhead extends Service{

	ThreadCounterOverhead tco = null;
	PowerManager pm = null;
	PowerManager.WakeLock wl = null;

	public IBinder onBind(Intent arg0) {
		return null;
	}

	public void onCreate(){
		pm = (PowerManager)
getSystemService(Context.POWER_SERVICE);
		wl = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
"My Tag");
	}

	public int onStartCommand(Intent intent, int flags, int
startId){
		wl.acquire();
		int usage = intent.getExtras().getInt("usage");
		int tempo = intent.getExtras().getInt("tempo");
		tco = new ThreadCounterOverhead(usage, tempo);
		tco.start();
		return super.onStartCommand(intent, flags, startId);
	}

	public void onDestroy(){
		super.onDestroy();
	}

	private final class ThreadCounterOverhead extends Thread{

		int usage, time, notUsage;
		String usagePath = "proc/stat";
		File outputFile = null;


		public ThreadCounterOverhead(int use, int t){
			super();
			usage = use;
			notUsage = 100 - usage;
			time = t * 1000;
			File sd =
Environment.getExternalStorageDirectory();
			outputFile = new File(sd,
"OverheadCPUoutput.txt");
		}
```

```
            // Ritorna arraylist con le righe di proc/stat
relative alle singole cpu. (dalla seconda alla 2+#cpu).
            public ArrayList<String> readCpusStrings() throws
IOException{
                  ArrayList<String> cpuValues = new
ArrayList<String>();
                  FileReader useFile = new FileReader(usagePath);
                  BufferedReader b1 = new BufferedReader(useFile);
                  String s = b1.readLine(); // Salto la prima riga
                  s = b1.readLine();
                  while(s != null){
                        StringTokenizer st = new
StringTokenizer(s);
                        String firstWord = st.nextToken();
                        if(firstWord.startsWith("cpu")){
                              cpuValues.add(s);
                        }
                        else{
                              break;
                        }
                        s = b1.readLine();
                  }
                  b1.close();
                  useFile.close();
                  return cpuValues;
            }


            public void run(){

                  long timeUnit = 100;
                  long start = System.currentTimeMillis();
                  long current = start;
                  long execTime = timeUnit * usage;
                  long sleepTime = timeUnit * notUsage;
                  int dummy = 0;
                  try{
                        ArrayList<String> listLine =
readCpusStrings();
                        long initialExTime =
getTotalUsage(listLine);
                        long initialSleepTime =
getTotalSleep(listLine);

                        while(current - start < time){

                              long internalStart =
System.currentTimeMillis();
                              long internalCurrent =
internalStart;

                              if(execTime != 0){
```

```
                                    while(internalCurrent -
internalStart < execTime){
                                          dummy++;
                                          internalCurrent =
System.currentTimeMillis();
                                    }
                              }

                              if(sleepTime != 0){
                                    try {
                                          Thread.sleep(sleepTime);
                                    } catch (InterruptedException
e) {
                                          e.printStackTrace();
                                    }
                              }

                              Log.i("OverheadCpu", ""+dummy);

                              current =
System.currentTimeMillis();
                        }// Chiusura del while



                        ArrayList<String> listLine2 =
readCpusStrings();
                        long finalExTime =
getTotalUsage(listLine2);
                        long finalSleepTime =
getTotalSleep(listLine2);
                        long totalUsage = finalExTime -
initialExTime;
                        long totalSleep = finalSleepTime -
initialSleepTime;
                        printResults(String.valueOf(totalUsage),
String.valueOf(totalSleep));
                  }
                  catch(IOException e){
                        e.printStackTrace();
                  }

                  wl.release();
                  stopSelf();

            }// Fine metodo run
```

```
	public long getTotalSleep(ArrayList<String> als){

		long totalSleepTime = 0;
		for(int x = 0; x < als.size(); x++){
			String s = als.get(x);
			StringTokenizer st = new
StringTokenizer(s);
			st.nextToken();
			st.nextToken();
			st.nextToken();
			st.nextToken();
			long initSleep =
Long.valueOf(st.nextToken());
			long initIOWait =
Long.valueOf(st.nextToken());

			totalSleepTime = totalSleepTime +
initSleep +initIOWait;
		}
		return totalSleepTime;
	}


	// Ritorna l'utilizzo totale, ossia la somma degli
utilizzi su tutte le CPU.
	public long getTotalUsage(ArrayList<String> als){

		long totalExTime = 0;
		for(int x = 0; x < als.size(); x++){
			String s = als.get(x);
			StringTokenizer st = new
StringTokenizer(s);
			st.nextToken();
			long userExTime =
Long.decode(st.nextToken());
			long niceExTime =
Long.decode(st.nextToken());
			long systExTime =
Long.decode(st.nextToken());
			st.nextToken();
			st.nextToken();
			long irqExTime =
Long.decode(st.nextToken());
			long softirqTime =
Long.decode(st.nextToken());

			totalExTime = totalExTime + userExTime +
niceExTime + systExTime + irqExTime + softirqTime;
		}
		return totalExTime;
	}
```

```
            // Stampa i risultati
            public void printResults(String esecuzione, String
sleep) throws IOException{
                  FileWriter fw = new FileWriter(outputFile,
true);
                  BufferedWriter bw = new BufferedWriter(fw);
                  PrintWriter pw = new PrintWriter(bw);
                  String myOutput = "Esecuzione: "+esecuzione+",
sleep: "+sleep;
                  pw.println(myOutput);
                  pw.close();
                  bw.close();
                  fw.close();
            }

      }
```

Risultati dei test di validazione

L'applicazione che ho utilizzato per realizzare il carico computazionale "fittizio" necessario per la validazione, come ho spiegato nel capitolo precedente, riduce la dimensione di un intervallino di esecuzione/riposo rendendo altalenante l'andamento lavoro/riposo.

Gli esperimenti che ho condotto per verificare la correttezza del modello prevedono di procedere con l'esecuzione di questa applicazione, cercando di capire se effettivamente i valori ottenuti con la calibrazione del modello sono consistenti, ossia se quando si va a mutare lo scenario esecutivo si ottengono valori di consumo simili a quelli di riferimento registrati in precedenza.

Gli esperimenti di validazioni sono stati eseguiti scegliendo determinate frequenze di lavoro ed impostando un determinato carico computazionale attraverso l'apposita applicazione. La granularità delle frequenze scelte per questi test è stata ovviamente più ampia rispetto alla quantità di dati in nostro possesso per la costruzione del modello, pertanto le frequenze che sono state interessate dal processo di validazione sono più distanti tra loro.

Per ciascuna frequenza considerata ho provato un carico computazionale scelto tra quelli per i quali avevo a disposizione dati provenienti dalla procedura di calibrazione del modello, ho impostato tale valore nel file di configurazione che guida l'esecuzione dell'applicazione di validazione e ho fatto eseguire i calcoli.

I risultati ottenuti dalle prove che ho effettuato hanno mostrato che i valori trovati in fase di validazione si discostano al più di 0,011A dai valori misurati in fase di calibrazione del modello.

Frequenza / Utilizzazione	Consumo medio misurato in fase di calibrazione	Consumo medio misurato in fase di validazione
360 MHz / 60%	64 mA	73 mA
460 MHz / 60%	71 mA	82 mA
537 MHz / 80%	96 mA	102 mA
614 MHz / 80%	103 mA	111 mA
844 MHz / 70%	133 mA	141 mA
921 MHz / 90%	177 mA	180 mA

Osservando i valori riportati nella tabella ed i grafici sottostanti possiamo vedere che i valori ottenuti in fase di validazione sono piuttosto allineati con i risultati ottenuti in fase di calibrazione.
Si noti che nei grafici riportano sull'asse delle ordinate il consumo in mA, mentre sull'asse delle ascisse c'è il tempo, indicato con la variabile t per semplicità di comprensione, ma indica un tempo discreto, corrispondente agli istanti di campionamento della corrente.

Si può perciò dire che il modello del comportamento della CPU descrive il funzionamento del dispositivo in maniera affidabile, con un errore massimo e medio piuttosto limitato.

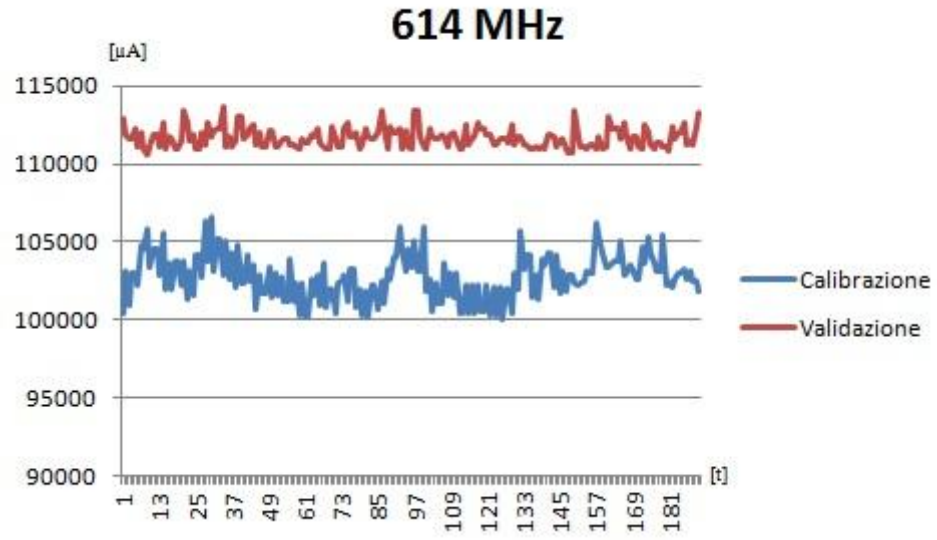

Fig.12: Test di validazione 614MHz utilizzazione 80%

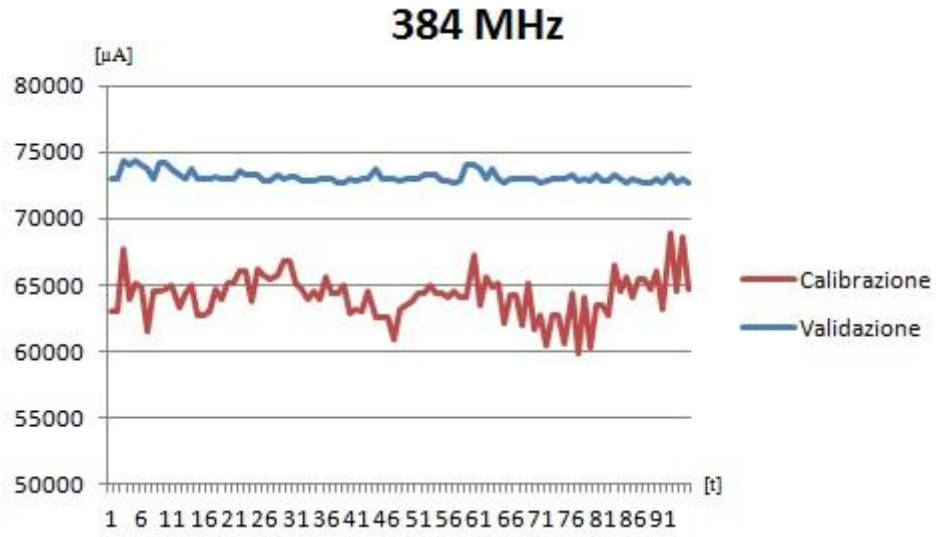

Fig.13: Test di validazione 384MHz utilizzazione 60%

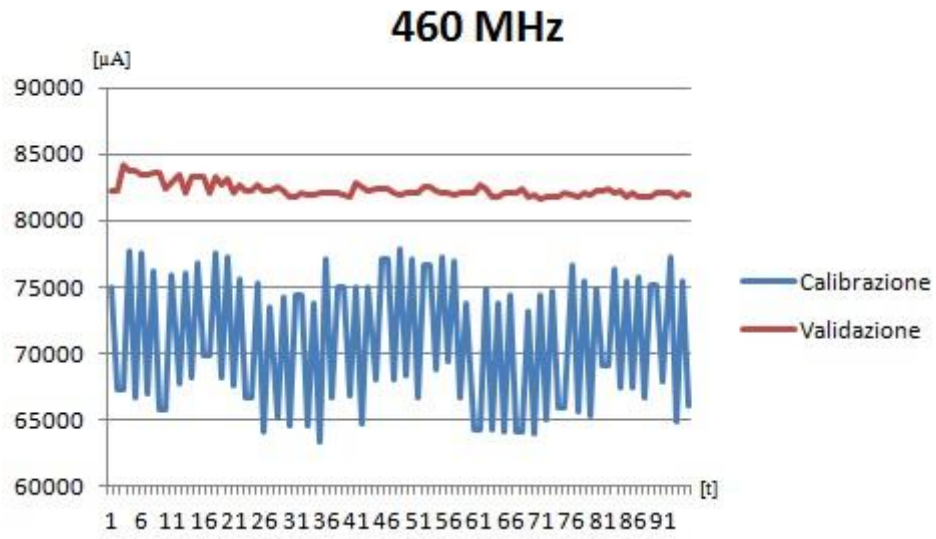

Fig.14: Test di validazione 460MHz utilizzazione 60%

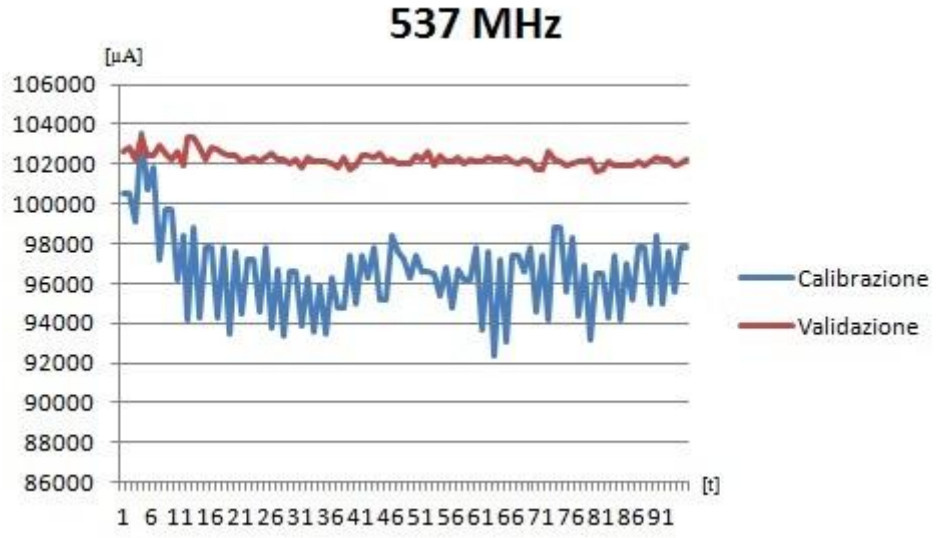

Fig.15: Test di validazione 537MHz utilizzazione 80%

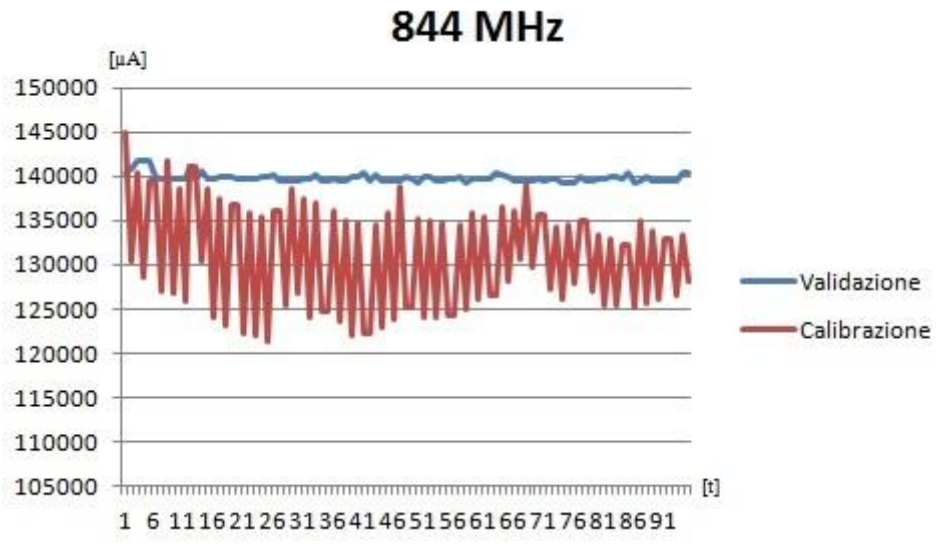

Fig.16: Test di validazione 844MHz utilizzazione 70%

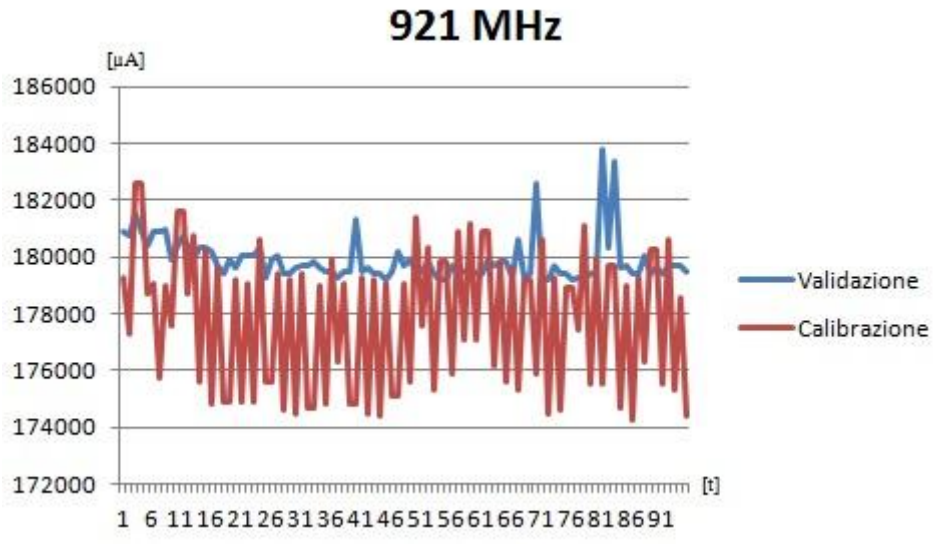

Fig.17: Test di validazione 921MHz, utilizzazione 90%

Conclusioni

Con la crescente diffusione dei dispositivi mobili dipendenti dalla batteria, nasce l'esigenza di estendere il concetto di sicurezza agli aspetti energetici che riguardano il dispositivo, in quanto questo tipo di dispositivi possono essere soggetti a determinati tipi di attacchi diretti allo scaricamento della batteria stessa.

Come spiegato in [3], il modello generale del consumo di un dispositivo può essere realizzato modellando le singole componenti che lo compongono e descrivendo i consumi totali come somma dei consumi di base (ossia i consumi che si hanno in assenza di specifiche attività) più i consumi legati ad una determinata attività. Questo modello permette di costruire le firme energetiche delle applicazioni che eseguono sul dispositivo, che saranno poi utilizzate per riconoscere eventuali comportamenti anomali; permette altresì di costruire le firme energetiche dei principali tipi di attacco, che serviranno per poter riconoscere l'attacco o quantomeno definire un insieme di attacchi possibili per un determinato comportamento energetico.

Il modello che descrive il comportamento del modulo Wifi era già stato sviluppato in [3], sebbene a basso livello, ed è in grado di riconoscere attacchi provenienti dalla rete; integrando questo modello con il modello relativo alla CPU che ho realizzato, diventa possibile distinguere qual è il contributo del processore e quale il contributo del modulo Wifi nel consumo energetico totale misurato durante gli attacchi considerati, raffinando ulteriormente la valutazione dei consumi energetici e compiendo un altro passo in direzione di una loro conoscenza puntuale ed analitica.

Inoltre, grazie all'estensione fornita da questa tesi, sarà possibile andare a verificare quale sia l'effetto degli attacchi via rete presi in considerazione sul consumo indotto dall'uso del processore, ossia, oltre a valutare il consumo del modulo Wifi, anche valutare quanto varia il consumo del processore nel momento in cui si ritrova sotto attacco.

In realtà il modello proposto descrive il comportamento della CPU, un componente del dispositivo che è sempre attivo, qualunque sia il pattern di componenti e di applicazioni in uso, per questo motivo ha una rilevanza che si estende a futuri sviluppi del lavoro, in quanto qualsiasi tipo di componente si andrà ad analizzare e modellare, sarà possibile predire esattamente il suo contributo energetico e distinguerlo dal consumo che sarebbe altrimenti forzatamente aggregato a quello del processore.

Il modello che ho sviluppato descrive il comportamento energetico del processore alle varie frequenze ed utilizzazioni; sebbene non sia stato possibile effettuare test che considerassero esclusivamente la CPU con applicazioni reali, il comportamento modellato è stato confermato dai test eseguiti con un'applicazione costruita appositamente per questo scopo.
I test sono stati eseguiti fissando la frequenza del processore e la sua utilizzazione per la durata della prova, ed hanno mostrato una piccola differenza rispetto ai consumi energetici registrati in fase di costruzione del modello, si tratta di una differenza media di 0,007A, mentre i valori massimi di questa differenza sono di 0,011A; si può pertanto dire che il modello descriva in maniera piuttosto accurata il consumo della CPU.

Sviluppi futuri

Il lavoro svolto ha portato alla realizzazione di un modello che descrive il comportamento del processore di un dispositivo mobile al variare dei parametri di frequenza ed utilizzazione, facilmente reperibili su un qualsiasi dispositivo Android-based.

Una possibile estensione del lavoro svolto potrebbe essere cercare di estendere il modello studiando il comportamento della CPU e considerando in particolare le sue transizioni attraverso i C-states, per fare questo si potrebbe considerare il sottosistema *cpuidle* presente in alcune versioni più recenti del kernel Linux; questo si potrebbe fare tenendo però in considerazione che sarebbe uno sviluppo limitato soltanto a determinati tipi di dispositivi, caratterizzati dalla presenza del sottosistema.

Considerando poi l'obiettivo globale di modellare i consumi di un dispositivo mediante i modelli di ogni componente che lo costituisce, ovvi ulteriori sviluppi possono essere pensati in tal senso, andando a sviluppare modelli riguardanti le componenti non ancora studiate (ad esempio il modulo 3G,oppure il GPS), cercando così di poter predire con maggior precisione il comportamento energetico del dispositivo mobile.

Riferimenti:

[1] Luca Caviglione, Alessio Merlo, Mauro Migliardi, *What is Green Security?*, Proc. Of the 7th International Conference on Information Assurance, Malacca (Malaysia) 5 – 8 December 2011, pgg. 366 – 371

[2] A.Armando, A.Merlo, L.Verderame, *An Empyrical Evaluation of the Android Security Framework*, in Proc. Of the 28th IFIP TC-11 SEC 2013 International Information Security and Privacy Conference (SEC 2013).

[3] Monica Curti, Alessio Merlo, Mauro Migliardi, Simone Schiappacasse, *Towards Energy Aware Intrusion Detection Systems on Mobile Devices*

[4] I.Burguera, U.Zurutuza, and S. Nadjm-Therani, *Crowdroid: behavior-based malware detection system for Android*, in Proc. Of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM11), 2011.

[5] T.Martin, M.Hisao, D.Ha, and J.Krishnaswami, *Denial of Service Attacks on Battery-powered Mobile Computers*, in Proc. Of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04), Whashington, DC, USA, p.309.

[6] L.Zhang, B.Tiwana, Z.Qian, Z.Wang, R.P.Dick, Z.Mao, and L.Yang, *Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones*, in Proc. of the Int. Conf. on Hardware/Software Codesign and System Synthesis, Oct. 2010.

[7] C.Yoon, D.Kim, W.Jung et al, *AppScope: Application Energy Metering Framework for Android Smartphone using Kernel Activity Monitoring*, in Proc. of the 2012 USENIX Technical Conference, June 13-15, 2012.

[8] H.Kim, J. Smith, and K. G. Shin, *Detecting energy-greedy anomalies and mobile malware variants*, in Proc. of the 6th International Conference on Mobile systems, applications, and services (MobiSys '08). ACM, New York, NY, USA, pp. 239-252.

[9] A. D. Schmidt, F. Peters, F. Lamour, C. Scheel, S.Ahmet, S. Albayrak, *Monitoring smartphones for anomaly detection*, Mob. Netw. Appl. 14, 1 (February 2009), 92-106.

[10] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. Mao and L. Yang, *Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones*, in Proc. of the Int. Conf. on hardware/Software Codesign and System Synthesis, Oct. 2010.

[11] developer.android.com

[12] wiki.archlinux.org

[13] www.kernel.org/doc/Documentation

[14] www.wikipedia.org