DiscoWall: Design And Implementation Of A Firewall For Android Phones

Brian Tewanima Löwe

Bachelor Thesis

DiscoWall: Design And Implementation Of A Firewall For Android Phones

vorgelegt von

Brian Tewanima Löwe

29. Oktober 2015

Technische Universität Kaiserslautern
Fachbereich Informatik
AG Verteilte Systeme

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Zitate sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Kaiserslautern, den 29. Oktober 2015

Brian Tewanima Löwe

Contents

1 Motivation

While the use of powerful firewalls is common practice for protecting hosts within a network, this mentality has not yet found its way into the world of handheld mobile devices. Looking at Android – the most used operating system for smartphones today (Statista.com, 2015) – the lack of widespread security through firewalls can be explained due to the system's design: A built-in firewall is not part of any Stock-ROM to-date and cannot be installed without having root privileges (which no Stock-ROM grants by default). In fact the use of Android firewalls is only possible by users having rooted the device themselves. But even then, the existing software solutions revolve around providing a GUI for the command-line tool "iptables" which in turn only allows for static firewall rules. The on-demand interaction of a firewall – as seen on desktop systems – where the user may decide upon each individual connection of each application, does not exist nor can it be implemented using static rules only. Therefore the user has to manually create rules for each app and connection, which requires a list of trustworthy hosts to be imported into the firewall. This tedious act is the main reason why the user ends up simply allowing an application full network access – or none at all. Obviously this greatly reduces the security-aspect of using a firewall in the first place, as the trustworthiness of a connection is dependent upon the remote client itself and cannot be generalized by knowing the local application alone.

The goal must therefore be to design an Android firewall which queries the user on how an unknown connection should be treated as it is being initiated, and create a static rule for it only if he decides to do so. This implies, however, a more sophisticated approach than building a GUI for iptables. Seeing as Android uses a Linux kernel, this can be achieved by accessing the kernel-module "Netfilter", which is part of any modern Linux-kernel (starting 2.4.x) – including those available for Android.

2 Introduction

2.1 Terminology

Rooted Device / Superuser

By default (i.e. factory settings) an Android device does not offer root/superuser privileges to the user. All commands issued by the user are run with non-root privileges, whereas root functionality is executed by root services, which communicate through interfaces (mainly Android API). In order to gain root privileges on an Android device, a "superuser app" has to be installed, which provides similar functionality as the Linux's "sudo" or "su" tools.

Rooting:

Rooting is the process of achieving root privileges on an Android device. It is usually done by using exploits – i.e. exploiting security issues within a certain Android ROM to execute code as root. While **Custom-ROM**s often are **rooted** by default, **Stock-ROMs** usually are not.

AOSP / AOSP-ROM:

The Android Open Source Project (AOSP) is the bare Android operating system which does not include any *branding* (i.e. manufacturer added software) or Google software. Therefore AOSP is the base for most open-source Android versions, such as **CyanogenMod**.

Stock-ROM:

Stock-ROMs are preinstalled Android versions on smartphones. They usually include *branding* (i.e. manufacturer added software) and small changes to the GUI, designed to create some kind of "individuality" for the OS, so that it sets itself apart from Android versions of other manufacturers.

Custom-ROM:

A Custom-ROM is a modified or enhanced version of the Android operating system, based on an existing ROM. This ROM base is either (a) an **AOSP** or (b) **a Stock-ROM**. The most Custom-ROMs are **rooted** by default.

CyanogenMod:

CyanogenMod is a widely used **Custom-ROM** and a branch of the **AOSP** which includes a large set of features. One of those features are the **Netfilter** kernel-modules.

2.2 Relevant Technologies

Netfilter Framework:

The Netfilter framework (Netfilter.org, 2015) is an extension for Linux-Kernels of version 2.4.x and later, which offers package-filtering capabilities in kernel-space. It contains multiple libraries and command-line tools – and can even be ported to Android.

Netfilter Iptables:

Iptables is a Netfilter module and command-line frontend for configuring rooting-decisions as well as verdicts (accept or block) for individual packages or connections.

2.3 Goal

It is the goal of this thesis to provide a proof of concept for an interactive Android-firewall, which is able to decide the verdict (accept/block) of a connection as it is being established. While there may be different approaches towards achieving this goal, performance is of the essence. The resources of a mobile device are very limited in comparison to those of desktop machines. Additionally every millisecond spent deciding on the package's verdict increases the latency of the connection – up to the point where a package would instantly receive a timeout when the firewall application is too slow. Therefore the actual decision must be made efficiently and – if possible – static (as soon as the user agrees to a permanent rule). Those two requirements imply a dual-approach by using static iptables-rules on the one hand, and dynamic Netfilter nfqueue-verdicts on the other. For the latter, Netfilter's nfqueue library (libnetfilter) has to be compiled for the Android Kernel and accessed using a native binary registering kernel-level callbacks.

At last the performance of the firewall will be benchmarked in regards to its energy-consumption, package loss and latency, which are introduced when processing each package in user space (as opposed to kernel space). By analyzing the results it can be shown, that the use of such a dualistic firewall is feasible and does not impact the daily functions of a mobile Android device.

3 Firewall Requirements

For achieving the described goal (see *2.3*), as well as due to the constraints resulting from the use of the Android platform, a set of requirements has been defined. The resulting firewall application is named “DiscoWall”.

3.1 Functional Requirements

3.1.1 Operating-System

1. **Root** privileges in order to allow access to the Linux-Kernels Netfilter framework.
2. Access to **Netfilter** modules nfqueue as well as iptables. The kernel of the ROM therefore has to be compiled with specific flags (see *5.2.1*).
3. Execution of **custom binaries**, which are not part of the ROM itself (see *5.1.5*).

3.1.2 Firewall

1. **Interactive user-decisions / interactive rules**: Any package which does not belong to a *known connection* (i.e. a connection for which there exists is a firewall rule), can be accepted or dropped by the user as it comes in or leaves the device.
2. **Static rules**: A static rule may be configured for any connection. Such a rule will be added to the iptables chains and therefore be handled within kernel space, without the need to query a verdict from the DiscoWall application.
3. **Rule scopes**:
    a. **Rule per app**: A firewall rule effects only one Android application installed on the device.
    b. **Rule per interface**: A firewall-rule must react to at least one interface. The available interfaces are:
        i. WiFi
        ii. Mobile data (GSM/UMTS/LTE)
    c. **Rule per protocol:** A firewall-rule must react to at least one protocol (TCP/UDP) or both.
4. **Rule types**: A rule must be of one of the following types:
    a. **Policy rule**: Accept/Block a specific connection, using a **Rule Policy** (see below).
    b. **Redirection rule:** Redirect a connection to a specific target (host-port pair).
5. **Rule Policies**: When a package has been filtered, one of the following policies will be applied:
    a. **ACCEPT**: The package is accepted and freely traverses the device.
    b. **BLOCK**: The package will be dropped issuing the ICMP response “port unreachable”.
    c. **INTERACTIVE**: The user will be asked to decide between whether to **accept** or **block** the package, while he also has the choice to create a static iptables rule for the connection.
6. **Firewall Policies:** When there is no rule defined for a package, a *default-policy* has to be invoked. This policy has to be one of the **Rule-Policies**.
7. **Block traffic of unknown installed apps:** An installed but unknown application (i.e. it is not listed by the Android’s *package manager*) must not be allowed to communicate when the firewall is enabled.

3.1.3 Performance

An Android firewall may only provide additional security, as long as it is permanently running. To that end, the software has to be very efficient – both in latency and energy consumption. Therefore the traffic running through the firewall should be minimal, which is achieved by only filtering for **SYN** and **FIN** packages as only those control the lifecycle of a TCP connection. With UDP being a stateless protocol, no such improvement can be made.

3.2 Non-Functional Requirements

Since the firewall needs to be configured by the user at a certain point in time (i.e. at least rules have to be created), different requirements for user-interaction must be met.

1. **Rules management**: Create/Edit/Delete rules
2. **Data persistency**: Save/Load/Export rules and configuration
3. **User settings**: Firewall can be configured by the user. Minimal settings:
   - Port for firewall app and nfqueue communication (see *4.1*)
   - Default-verdict for unknown connections
   - Timeout until using default-verdict
4. **Application persistency**: The firewall must run as a persistent process:
   - The process must never be stopped nor killed by the Android system
   - The process must be restarted when it is force-closed for any reason
   - The firewall must be started with the system

4 Architecture & Design

4.1 Software-Components

4.1.1 Overview

In order to allow on-demand verdicts on package level, the Linux-Kernel module nfqueue of the Netfilter framework needs to be accessed. Therefore a native C-application is required, as it can be run with superuser privileges and connect to the nfqueue library. This native binary has been named "Netfilter-bridge" and must run continuously as long as the firewall is active.

At the same time the firewall frontend process must maintain its connection to the Netfilter-bridge, as a new package might be received at any time. Therefore the GUI part of the firewall must have a persistent component, which is never terminated by the Android OS – and has to be restarted even if is force-closed (e.g. crash, resource-problem, killed by user etc.). Next to the persistent component, the Android app requires a GUI which enables the configuration of the firewall. The entire visible part of the firewall is called "DiscoWall".

The firewall components consist of:

- DiscoWall: The Android app and graphical frontend of the firewall
  - "DiscoWall GUI/frontend": graphical interface / frontend
  - "DiscoWall service": persistent Android service
- Netfilter-Bridge: The native C-application registers its callback function for intercepting packages by the use of nfqueue (Netfilter). It needs to be executed by the root user (see *5.2*).

4.1.2 Communication between components

As the two components run within separate processes, IPC (inter-process communication) is required. This is done by implementing an application-layer protocol on top of TCP.

Rolls of software components:

Since the user starts the Android app DiscoWall via the OS GUI (e.g. the "Launcher") the frontend service will always be started first. Only then the Netfilter-Bridge may be started by the DiscoWall service, which in turn makes the Netfilter-Bridge the client.

- DiscoWall service: server listening for connections but also executing the client program.
- Netfilter-Bridge: client which is started via a root shell by the DiscoWall service.

Protocol:

The basic idea of the protocol is simplicity and minimal communication for avoiding unnecessary latencies. Therefore the client starts with an initial message, signaling the connection to the server. After this point only the client (i.e. the Netfilter-Bridge) will initiate contact, after which the server will respond.

1. Netfilter-Bridge initial contact
   - Message: `#COMMENT#Netfilter-Bridge says hello.`
2. DiscoWall initial contact
   - Message: `#COMMENT#DiscoWall App says hello.`

3. Netfilter-Bridge on package-received:
    - Format: #[actionType]##[protocol]##[ip.source]##[ip.destination]##[ttl]##[tos] ##[ip.version] ##[ID] ##[length]
        - Action-Types (flag only exists for further extensibility):
            - Packet.QueryAction
            - Packet.QueryAction.Response (only for response to message)
        - IP-fields: All IP-fields relate to the fields contained in an IP package.
    - Message example:
      `#Packet.QueryAction##protocol=tcp##ip.src=192.168.178.27##ip.dst=95.100.248.113##ip.ttl=64##ip.id=2781##ip.tos=0##ip.version=4##ip.length=2`
4. DiscoWall on verdict-decided (by user or per rule):
    - Case **accept**: #Packet.QueryAction.Response##ACCEPT#
    - Case **drop**: #Packet.QueryAction.Response##DROP#

Communication-Example:

The following excerpt contains the messages communicated from the start of both components until a package is (1) intercepted by the Netfilter-Bridge and (2) accepted by the user via the DiscoWall-Frontend.

1. *Netfilter-Bridge:* `#COMMENT#Netfilter-Bridge says hello.`
2. *DiscoWall:* `#COMMENT#DiscoWall App says hello.`
3. *Netfilter-Bridge:*
   `#Packet.QueryAction##protocol=tcp##ip.src=192.168.178.27##ip.dst=95.100.248.113##ip.ttl=64##ip.id=2781##ip.tos=0##ip.version=4##ip.length=2`
4. *DiscoWall:* `#Packet.QueryAction.Response##ACCEPT#`
5. *Netfilter-Bridge* is idle until new package is being intercepted…

4.2 Iptables Structure

Static rules are directly added to iptables, as this results in the minimal latency for applying a verdict (accept/drop) to package while reducing the energy consumption. A rule exists for each interface it should be applied to.

Any application which is not handled by the firewall should be:

a) Ignored and package-filtering should not be applied.
b) Blocked if this behavior has been specified in the DiscoWall settings.

4.2.1 Handling the different WAN interfaces of Android devices

A rule applies to either the **WiFi** or **mobile-data** (GSM/UMTS/LTE) interface. However, Android supports multiple kinds of those interfaces (for example WiMax vs. WLAN) so the actual interface used by the device is not known during design-time. This fact would result in a rule being duplicated not only once, but multiple time for all **WiFi** adapters, and then again for all **mobile-data** adapters.

To avoid this, filter-rules for each possible interface (WiMax, WLAN, GSM, LTE, etc.) are being created which forward the package to a chain containing rules only for the interface group (WiFi or Mobile Data) of interest.

The following command forwards any package, received via the WiMax interface, to the "discowall-if-wifi" chain (a custom chain created for the functionality of the firewall). For each other WiFi interface an analogue rule is necessary.

```
# iptables -A discowall -i wimax+ -j discowall-if-wifi
```

As described, the same logic is being used for mobile Data, too:

```
# iptables -A discowall -i rmnet+ -j discowall-if-3g
```

Note: The "+" on the interface name instructs iptables to ignore the interface number, as interfaces are numbered on Linux systems (e.g. wlan0, wlan1, …). This is being done as all interfaces of a certain type should be filtered and not only a specific one.

4.2.2 Filtering packages by application

Filtering packages by application is not a supported function of iptables. However, it is possible to filter packages by the UID of the user who is running the application which is listening on the port:

```
# iptables -A <chain> -m owner --uid-owner <uid>
```

Android creates a user for each application, which is installed on the device, as part of the application sandboxing and security (developer.android.com: Manifest, 2015). This prohibits an app from accessing another app's data (a separate *shared data* functionality exists for that purpose). This circumstance is being used to filter a package by the UID (user ID) of a certain application, therefore limiting the rule to the specific app only.

4.2.3 Applying the policy for a package

When the package traverses the iptables chains of the firewall, one of two things can happen:

a) A matching rule is found for the package; therefore the specified policy (accept/block/user-decision via GUI) will be invoked.
b) No matching rule exists for the package; therefore the default-policy (accept/block/user-decision via GUI) will be invoked.

Invoking a policy for a package implies the execution of a specific iptables rule. This is being done by creating a custom iptables chain for each policy, which will perform the intended action.

- discowall-action-accept
- discowall-action-reject
- discowall-action-interactive

4.2.4 Resulting iptables structure

The following chain structure is being used within two tables of the iptables framework:

1) The **filter** table, where rules for accepting/dropping and manipulation of packages are located.
2) The **nat** table, where redirection-rules are located.

It is required to distinguish between those two goals, as the iptables table **filter** does not allow redirecting packages, while the table **nat** does not allow manipulation of packages. Therefore the following paradigm is used within both tables:

1) Fetch any TCP/UDP package in chains **INPUT** or **OUTPUT**.
2) Check whether the package belongs to an application which is to be secured by the firewall (i.e. check user-id of process) and ignore the rest (chain **discowall-prefilter**).
3) Move package to chain (according to interface) **discowall-if-3g** or **discowall-if-wifi**.
4) Apply matching rule (if any) by moving package to **discowall-action-accept/-reject/-interactive/-redirect**. Apply default-policy (configured by user) if no rule matches.

Chain-Overview:

Chain	Description	Actions	Filter	Table
INPUT	Standard chain for incoming packages. Only within **nat** table.	Jump to **discowall-prefilter**	Incoming packages	filter
OUTPUT	Standard chain for outgoing packages	Jump to **discowall-prefilter**	Outgoing packages	filter, nat
discowall-prefilter	Selects packages of apps which should be secured by the firewall.	Jump to **discowall**	Packages of user-selected apps	filter, nat
discowall	Main chain of firewall which jumps according to interface and applies default policy if no rule matches.	Filtering by network-interface, invoking default policy, jump to **discowall-if-***, **discowall-action***	*	filter, nat
discowall-if-3g	Contains rules for interfaces GSM/UMTS/LTE.	Applies user-defined rules	Packages originating from a mobile data interface	filter, nat
discowall-if-wifi	Contains rules for interfaces WLAN/WiMax/Ethernet	Applies user-defined rules	Packages originating from a WiFi interface	filter, nat
discowall-action-accept	Performs the accept policy action	Jump to **ACCEPT**	*	filter
discowall-action-reject	Performs the block policy action	Jump to **DROP**	*	filter
discowall-action-interactive	Performs the interactive policy action	Jump to **NFQUEUE**	*	Filter
discowall-action-redirect	Redirects the package to user-specified host	Jump to **DROP**	*	nat

Table 1: Iptables structure of firewall

5 Implementation Challenges

5.1 Android

5.1.1 Choosing OS/API version

To date there are more than 10 different Android versions in daily use worldwide. When developing a new application, the intended target platform (i.e. required Android version) is of major concern, as it is a tradeoff between functionality (and bug-fixes) vs. availability. As will be described later on, newer versions also add strong constraints to native binary code due to additional security features such as “SELinux”.

API version tradeoff:

The more recent the API (dependent upon Android version) required for the application, the more bug- and security-fixes and the more functionality. However, it is a well-known problem that most manufacturers do not (or very little) support their devices with upgrades once they have left the shop. The result is a very slow upgrade process to newer OS versions, resulting in the majority of devices running older Android versions (see *Table 2: Distribution of Android versions*), which in turn implies the requirement of always working with the oldest feasible API version.

Distribution of Android versions (October 5, 2015):

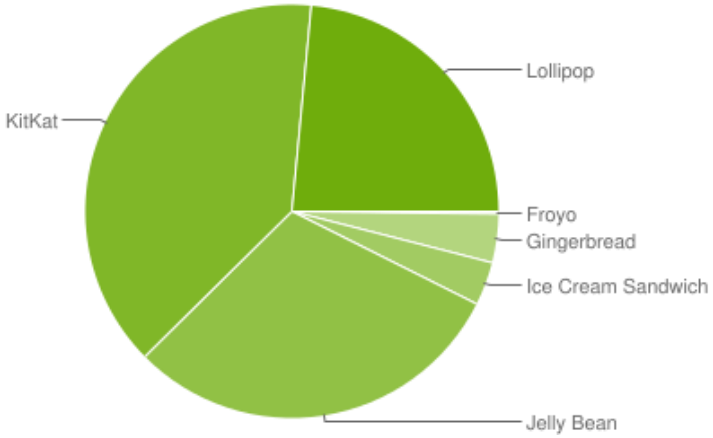

Figure 1: Distribution of Android versions (chart)

Version	Codename	API	Distribution	Notes
2.2	Froyo (FY)	8	0.2 %	
2.3.3 – 2.3.7	Gingerbread (GB)	10	3.8 %	
4.0.3 – 4.0.4	Ice Cream Sandwich (ICS)	15	3.4 %	SELinux: *permissive* mode
4.1.x	Jelly Bean (JB)	16	11.4 %	
4.2.x	Jelly Bean (JB)	17	14.5 %	
4.3	Jelly Bean (JB)	18	4.3 %	
4.4	KitKat (KK)	19	38.9 %	SD-Card mounted as readonly (only app local storage writable)
5.0	Lollipop (LP)	21	15.6 %	SELinux: *enforcing* mode
5.1	Lollipop (LP)	22	7.9 %	

Table 2: Distribution of Android versions

Source: (developer.android.com: Android versions distribution, 2015)

Decision:

As the majority of Android devices (92.6%) run on Android 4.1 or newer, this version has been decided as the API requirement which provides the minimum required functionality (developer.android.com: Android 4.1, 2015) while being supported by most hardware to date. The deciding factors are:

1) As described in *Android Framework/API* services cannot access the GUI thread, conflicting with the requirement for the user to decide upon the connection verdict (accept/block). As a solution for this problem an *extended notification* will be used. This kind of notifications has been greatly enhanced in API version 16.
2) The *Dialog* functionality has also been enhanced, allowing for passing the connection information to the dialog instance without the need of serialization within a *Bundle* instance.

5.1.2 Compiling shared libraries for Android app

The most efficient way of accessing the Netfilter nfqueue library functions is to call them from within the Android application itself (using Android's version of JNI, Java Native Interface). However, referencing non-standard shared libraries is not possible with the current Android framework implementation, as the list of allowed libraries is fixed. Project member "morri" of the Android Open Source project stated: "The NDK supports only APIs listed as supported in the NDK docs; we don't generally make changes to support other libraries." (code.google.com: Issues, 2015) This indicates that no fix for the problem is to be expected soon.

Due to this limitation an intermediate component – the Netfilter-Bridge – is necessary to handle the none-Android library calls to Netfilter nfqueue.

5.1.3 Compiling native binaries for Android

In order to use a native library (such as Netfilter nfqueue) on Android, it has to be cross-compiled on the development-system first. This can be done using of Android's Native Development Kit (NDK). NDK is "a toolset that allows you to implement parts of your app using native-code languages such as C and C++" (developer.android.com: NDK, 2015).

In theory this would even allow for direct invocation of activity-code (i.e. code within the regular Android app) from the native binary or library. In reality the NDK-support of Android Studio (the Android IDE) is very limited, resulting in deployment-bugs (e.g. compiler-errors which do not occur when compiling from command-line) and the inability to compile native binaries when they are referencing other libraries. This problem occurs due to the missing option for configuring the Gradle-builder's (Gradle.org, 2015) GCC compiler for changing the default include-paths, which is necessary for references to Netfilter-libraries. This can be solved by compiling the native binary manually through issuing the *ndk-build* command together with a customized "Android.mk" build-file.

Note: How to set up the directory-structure and compiling native binaries manually is described in this resource: (kvurd.com: compiling a cpp library for android).

5.1.4 Android Framework/API

No modal dialogs:

The Android GUI framework only supports modeless calls and dialogs (busy waiting results in a "not responding" message being displayed). Therefore it is impossible to **wait for a user to do something** – instead listeners have to be implemented in order to **react to the fact the user did something**. This means waiting for a user-decision for an unknown connection cannot be done explicitly (i.e. by using a modal dialog). Instead the method handling the user-action (of deciding the connection verdict) has to directly communicate its result to the Netfilter-Bridge, which has to block until a response is being received.

Thus packages have to be handled sequentially and the frontend cannot be informed about new packages until the user *(a)* decided a verdict or *(b)* an auto-response has been invoked by the DiscoWall to prevent permanent blocking. The overall speed of the firewall is being reduced and the package latency will be increased.

Persistent Services:

The Android framework provides *services* in order to allow persistent threads (developer.android.com: Services, 2015) to run independently of the application's lifecycle. Any *normal Activity* will be paused or stopped in order to save energy or when the devices resources are running out. In the case of the DiscoWall firewall, new packages could not be handled and as a result would be blocked without the user being informed. Therefore a **persistent** service has to be used for performing the firewall functionality.

When the user has to decide the verdict for a new connection, a dialog would be the common GUI element. However, this is not allowed due to the GUI thread access restriction imposed on Android services. As a workaround the *status bar* is opened, displaying a *notification* from which the user can open a dialog or directly select the verdict.

Network restrictions:

The Android framework does not allow any network access within the main (GUI-) thread of the application. However, a workaround for relieving this restriction exists:

```
StrictMode.setThreadPolicy(
   new StrictMode.ThreadPolicy.Builder().permitAll().build()
);
```

5.1.5 SELinux & pie security check

Starting from Android 5.0 onward, the mode of the SELinux security mechanism (source.android.com: SELinux, 2015) has been set to *enforcing*. This will stop many shell commands from being executed – even when running as root (source.android.com: Security Enhancements, 2015). Therefore adding rules to iptables may cause the application to crash. This can be worked around by using the application "SELinuxModeChanger" (forum.xda-developers.com: SELinuxModeChanger, 2015) to change the SELinux mode to *permissive*.

Additionally, any binary has to support PIE (position-independent executables) – which is not the case when compiling a native binary with the current NDK (Android Native Development Kit). Google officially states: "Android now requires all dynamically linked executables to support PIE". As this cannot be circumvented, the SELinux version has to be manipulated in order to execute the Netfilter-Bridge-Binary on Android 5 devices. A patch has been written by XDA-Developers member "shinyquagsire23" (forum.xda-developers.com: SELinux Patch, 2015), which currently only works for Google Nexus devices.

5.2 Netfilter

Because the entire Netfilter framework modifies Kernel behavior, all provided commands have to be executed with root privileges – as it is the case on any other Linux system running the Netfilter framework. Therefore the Netfilter-Bridge (see *4.1*) has to be started by the root user. On Android this is being done by starting a root shell ("su terminal").

5.2.1 Netfilter on Android

For an Android device to run Netfilter nfqueue/libnetfilter the kernel needs to be compiled along with the following modules (see (roman10, 2011)):

- CONFIG_NETFILTER_ADVANCED
- CONFIG_NETFITLER_NETLINK
- CONFIG_NETFILTER_NETLINK_QUEUE

Adding those modules to the kernel later on is not possible and re-compilation is necessary. Currently most Android Stock-ROMs (which are also running Stock-Kernels) do **not** include the mentioned modules, thus using the DiscoWall firewall will not be possible on those systems. However, almost all Custom-ROMs (including CyanogenMod) are compiled with **iptables** and **Netfilter** modules already present.

Listing available kernel modules:

Because of the flexibility of Linux-Kernels, there is no standardized way for checking the available kernel modules. While most desktop distributions (such as Debian or Arch Linux) support the listing of those modules, there is no definite way to do this on Android. When the option is enabled during kernel compilation, the file "`/proc/config.gz`" will be created which contains a list of enabled modules. On most Stock-ROMs this file does not exist (e.g. on HTC devices) and the availability will only be known after fetching the exception caused by invoking the missing module via code or root shell.

5.2.2 Passing UID information of package to Netfilter-Bridge

Iptables supplies an extension for filtering a package by user-id (see 4.2.2), however the Netfilter module nfqueue does not. Therefore it is not possible to inform the DiscoWall app about the package's UID before retrieving the verdict (accept/block). In order to map the package to its owner application, this information is required as the rules are defined for each app individually. The solution to this problem is using the **MARK** target of iptables, which assigns an integer to a matching package.

```
# iptables -j MARK –set-mark <some-value>
```

Since each application, which is to be secured by the firewall, has its own iptables entry for jumping packages into the "discowall" chain, the mark can simply be added before doing that. The following example will first mark any package belonging to a root-user process and then jump the package into the main DiscoWall chain. Note that within the firewall a user-ID offset of 1000 is being used, as a mark value of zero means "no mark" and thus removes it.

```
# iptables -A discowall-prefilter -m owner --uid-owner 0 -j MARK --set-mark 1000
# iptables -A discowall-prefilter -m owner --uid-owner 0 -j discowall
```

5.2.3 Iptables mark extension bug

The Android implementation of iptables contains a bug within the marking-extension, which results in marked packages being dropped by the remote host. As the mark is not part of the package, i.e. the package is not being altered in any way, this behavior should not occur (see (linuxhowtos.org: marktarget, 2015)).

Recreating the bug:

1. Add iptables-rule for marking (mark here = 42) packages to Android device:
   `# adb root; adb iptables -A INPUT --set-mark 42 -j MARK`
2. Ping the device:
   `# ping <device-ip>`
3. ➔ The result will be a 100% package loss.

Workaround for the bug:

Because of the remote package only being dropped when a mark is set, removing the mark (i.e. setting its value to zero) before accepting/dropping the package solves the problem.

```
# iptables -A INPUT --set-mark 0 -j MARK
```

Aside from returning (accept/block) packages by using iptables rules, DiscoWall also accepts/blocks them by making use of the Netfilter-Bridge binary. After the decision has been made by the user, the frontend communicates the decision to the binary (see 4.1.2) where the function for assigning the verdict is being called. Therefore the mark also has to be removed there, as the described bug would still occur just like when using the iptables MARK target.

```
nfq_set_verdict2(qh, id, NF_ACCEPT, 0, 0, NULL);
```

5.2.4 Blocking nfqueue queue

As the name suggests, the “nfqueue” module of Netfilter handles intercepted packages within a queue. Since the package’s verdict is being decided by the DiscoWall frontend via user interaction, the package will be halted during that time. After the decision, the verdict will be set (within the Netfilter-Bridge) and the package will be either accepted or blocked.

However, as long as the verdict is pending the queue will not be processed any further, and any other package will have to wait, resulting in the inability to establish new connections while a pending one exists. The effect is limited to packages within the queue, which are those handled by the firewall’s “interactive” mode.

Possible solutions for this none-trivial problem are being described within the chapter *Future Work*.

6 DiscoWall

6.1 Functionality

6.1.1 Enabling the firewall for specific applications

As seen in *Figure 2: main activity*, the full list of installed applications is displayed in the DiscoWall's main activity. Checking an item will enable the firewall for the specified app. The list itself is retrieved via the Android *PackageManager* class and contains only those apps which have been installed via ".apk" file (i.e. via *Play Store* or filesystem). Selecting an item will allow to edit the app rules (see *Figure 3: edit application rules*).

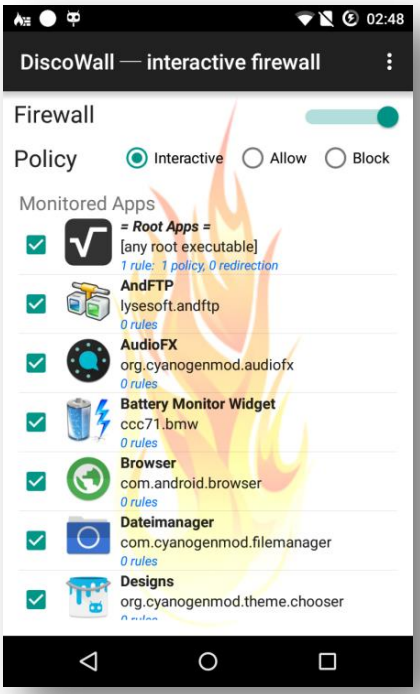

Figure 2: main activity

<u>The list item's information contains:</u>

1. First line (bold font): Name of application
2. Second line: Full package name of application
3. Third line: Number of rules defined for application

<u>Filtering system application traffic:</u>

The first item in the application-list (the "= Root Apps =" entry) does not represent an application, but any application being run by the root user (i.e. user ID "0"). This includes many of Android's system apps and services, but also any other command (e.g. "`su -c 'wget disco.cs.uni-kl.de'`") executed via superuser (see Terminology) app.

6.1.2 Firewall configuration

On the top of *Figure 2: main activity* the configuration of the firewall engine itself is being shown. The "Firewall" slider is set to enabled (i.e. to the right) and the "Policy" is set to interactive. The policy specifies the action to be taken for packages which do not match any rule. It may be changed at any time by selecting one of the radio buttons: "Interactive", "Allow", "Block".

Figure 3: edit application rules

6.1.3 Defining and editing application rules

The example in *Figure 3: edit application rules* shows customized rules for the CyanogenMod "Browser" application. The list contains all rules currently existing for the selected application. A new rule may be created by clicking the "Add" button, which results in at least one iptables entry for the rule (depending on the rule type).

<u>Information per rule as follows:</u>

1. The symbol representing the type of rule
   a. Green hook for an accepted connection
   b. Red cross for blocked connection

c. Yellow curved arrow for redirected connection
d. Blue user symbol for dynamic decision for each new connection

2. The interface for which the rule is being applied
   a. WiFi
   b. Mobile data interfaces: GSM/UMTS/LTE
3. Transport-Layer protocol
   a. TCP
   b. UDP
4. Source and destination pair

6.1.4 Dynamically allowing/blocking a connection

The main functionality of the DiscoWall firewall is the possibility for allowing/blocking a connection as it is about to be established without the need for earlier creation of static rules.

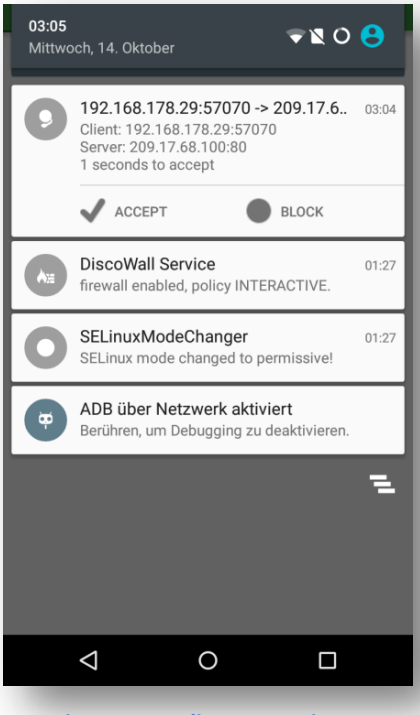

Figure 4: pending connection

<u>The user will be prompted for a new connection, when…</u>

a) There is no rule for the connection and the firewall policy is set to “interactive”.
b) The rule has the policy “interactive”.

As shown in *Figure 4: pending connection* the status bar will be opened (see “Android services” in 5.1.4) with the connection being shown. After a configurable countdown the connection will be either accepted or blocked (according to user settings, see *5.2.4*). By selecting the status bar entry the user may also create a permanent rule for the pending connection (*Figure 5: creating rule for connection*).

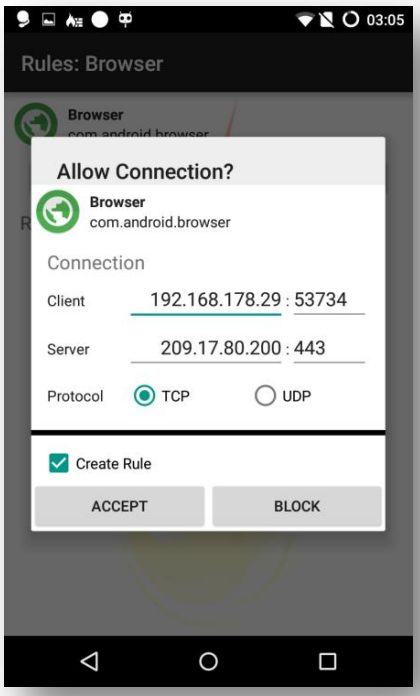

Figure 5: creating rule for connection

6.1.5 Importing/Exporting rules

In order to effectively use a firewall, it must be possible to batch-import rules. Therefore only the device-specific firewall configuration is stored within the application’s internal settings (provided by the Android API), while the rules are being stored as xml files in “/data/data/de.uni_kl.informatik.disco.droidwall/app_rules”. Thus a generic list of trusted/untrusted hosts may be imported into the firewall configuration without the use of the GUI.

6.2 Requirements

6.2.1 System requirements

The DiscoWall firewall can be executed on any device running Android 4.1 “Jelly Bean” (SDK version 16) or above, as long as the kernel has been compiled with enabled Netfilter modules. Only when using Android 5.0 or newer, some workarounds are required due to new non-optional security features Google introduced, which are making the use of native binaries or libraries difficult. The required workarounds are mentioned in chapter 5.1.

General requirements:

- ROM
    - Android 4.1 Jelly Bean (SDK version 16) or newer
        i. SELinux and PIE-check workarounds for Android 5.0
    - Kernel with Netfilter support (see 5.2.1)
    - Root permissions
- Software
    - Iptables binary (can be installed through Google market, e.g. (play.google.com: Iptables2, 2015))
    - Any superuser app (e.g. SuperSU or Superuser)

6.2.2 DiscoWall permissions

Even though DiscoWall requires root privileges, not everything the application does is executed as root user (this would be a great security risk). Therefore the usual operations such as accessing the filesystem etc. are performed with non-root permissions. More detailed information about the used permissions as well as explanations is given below.

DiscoWall app permissions:

1. *Modify or delete contents of your USB storage*
    - Exporting firewall rules
2. *Read the contents of your USB storage*
    - Importing firewall rules
3. *Full network access*
    - Communication with Netfilter-Bridge through port on localhost
4. *Run at startup*
    - Required for automatically starting DiscoWall on device restart
5. *Expand/collapse status bar*
    - Expansion of status bar when user needs to decide on connection verdict

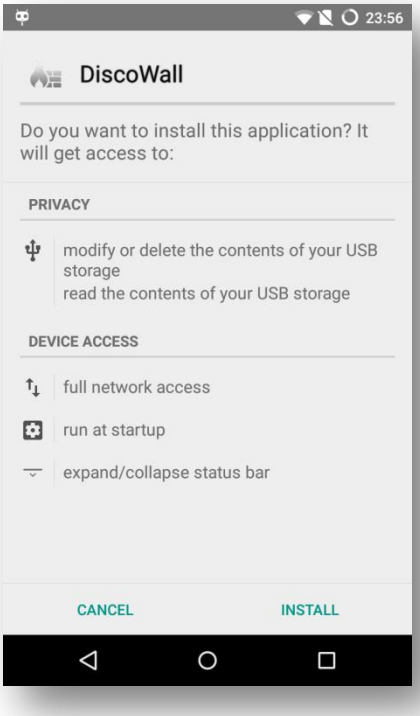

Figure 6: DiscoWall permissions

Root permissions:

Even though the Android API provides a permission called “ACCESS_SUPERUSER”, it can neither be tested during installation, nor does it have any impact on the functionality of the software requesting such root privileges

during runtime. DiscoWall declares "ACCESS_SUPERUSER" in its manifest and the superuser application installed on the device will prompt the user for granting root privileges to the app when enabling the firewall for the first time (see *Figure 7: DiscoWall root permission request*).

The firewall **exclusively** uses root permissions for the following operations:

- Executing "`iptables`" commands (create/delete rules and chains)
- Show "`ifconfig`" information to the user
- Starting/Terminating the Netfilter-Bridge process (see *Netfilter on Android*)

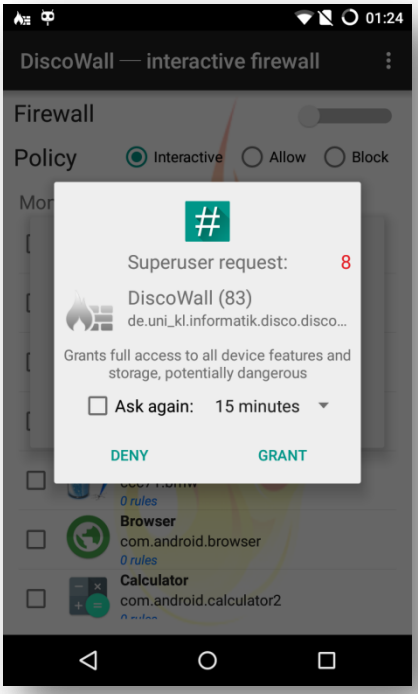

Figure 7: DiscoWall root permission request

6.3 Devices

The DiscoWall has been tested and used on two different Google Nexus devices: "Google Nexus S" (running Android 4.1.2) and "Google Nexus 4" (running Android 5.1.1). There is no general requirement for using Google Nexus devices when running the firewall. However, since Google changed the SELinux mode to *enforcing* starting with Android 5 (source.android.com: SELinux, 2015), the former mentioned unofficial patch is required for enabling the execution of arbitrary native binaries. As this patch is currently only available for Google Nexus devices, other workarounds have to be found for other devices running Android 5.

6.3.1 Google Nexus 4

- ROM
  - CyanogenMod 12.1 (20150512-NIGHTLY-mako)
  - Android v5.1.1
  - Kernel v3.4.0-cyanogenmod-gf5d2446
  - SELinux (*enforcing* mode by default)
- Software
  - SuperSU Free v2.46
  - Iptables v1.4.20
  - SELinuxModeChanger (used to change SELinux mode to *permissive*)

6.3.2 Google Nexus S

- ROM
  - CyanogenMod 10.0.0-crespo
  - Android 4.1.2
  - Kernel v3.0.50-cyanogenmod-gda17741
- Software
  - Superuser 3.1.4(47)
  - Iptables 1.4.11.1

7 Performance Benchmarks

In order to determine the feasibility of an nfqueue based Android firewall, its performance has to be tested. To that end *(1)* the energy consumption and throughput as well as *(2)* the latency will be benchmarked.

7.1 Testing Environment

7.1.1 Android device

- Hardware:
  - Device: Google Nexus 4
  - Battery capacity when full: 2099mAh
  - Device age: 4 months
  - WLAN: 802.11g/n
- ROM:
  - CyanogenMod 12.1 (20150512-NIGHTLY-mako)
  - Android v5.1.1
  - Kernel v3.4.0-cyanogenmod-gf5d2446
  - SELinux (*permissive* mode)
- Software
  - System tools
    - SuperSU Free v2.46 (granting root privileges – “superuser app”)
    - Iptables v1.4.20 (Android port of Netfilter iptables command-line tool)
    - SELinuxModeChanger (for setting SELinux mode to *permissive*)
  - Tools for performing tests
    - 3C Toolbox (analyzing energy consumption of idle device)
    - FTP Server (keeping port open)
    - Wake Lock (applying *partial wake lock*)
    - Battery Monitor (reading battery voltage)
- OS configuration
  - No SIM Card (i.e. no interrupts through provider communication)
  - Android synchronization disabled
  - Screen auto-dim after 1 minute
  - Wake Lock: Partial wake lock

7.1.2 Laptop

- Operating System: Linux Mint 17.3, KDE
- Network connection: 100mbit LAN
- Hardware:
  - CPU: Core i7
  - RAM: 12GB
- Benchmarking-tool: hping3 (packages.ubuntu.com: hping3, 2015)

7.1.3 Router

- Model: FritzBox! FonWLAN 7170

- WLAN:
  - WPA2, secured
  - Channel 11
  - Connection speed (to Android device): 24Mbps

7.2 Preparing Android device

7.2.1 Keeping port open

In order to assert the testing port staying open, it has to be connected to a running Activity. Due to the Android's "Activity Lifecycle" (developer.android.com: Activity-Lifecycle, 2015) an Activity might be closed when resources are required or the activity becomes idle. Therefore the testing-port has to be opened by a permanent service, as it will not be terminated by the system. The application "FTP Server" had been chosen as it employs such a service.

7.2.2 Keeping device awake

The Android energy saving mechanism will put the device into "sleep mode" as soon as it becomes idle for a short period of time. Thereby the WiFi module is being put into power saving mode, the device's CPU's voltage is being reduced to a minimum and in turn any operation will take longer than it normally would. Therefore the testing results would be distorted.

To prevent the problem a "partial wake lock" has been set (using the application "Wake Lock", see *Figure 8: Device prepared for* test), which prevents the device from switching to sleep mode while allowing the screen to turn off. (developer.android.com: WakeLock, 2015)

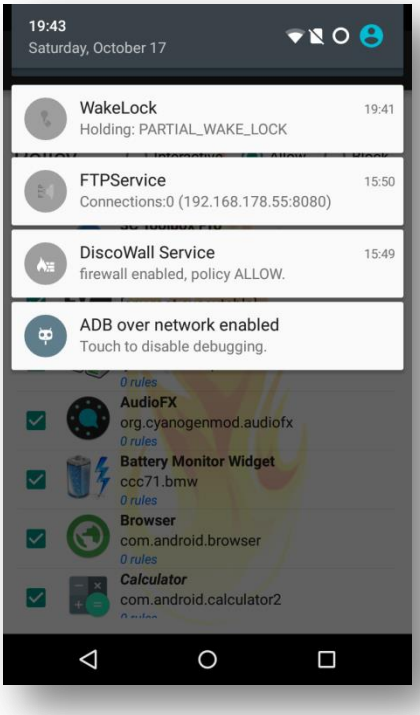

Figure 8: Device prepared for test

7.2.3 Stopping background services

Android performs different operations in the background, unnoticeable by the user. Those operations include RAM management (pause or stop idle activities), synchronization (e-mails, calendar etc.) and entropy-generation (forum.xda-developers.com: Seeder, 2015). Due to those tasks, the Synchronization is especially time-, CPU- and traffic-demanding and will therefore be disabled during all benchmarks.

7.3 Preparing laptop

For sending packages to the mobile device *hping3* (Ubuntu standard repository) is being used. It offers a wide range of functionality, including measurement of latency as well as package loss.

All packages are sent to an open port while having the flags **SYN** and **ACK** set. As DiscoWall does not filter all packages (see *3.1.3*), any other package having neither SYN nor FIN flag would not be processed by the firewall and thus falsify the measured performance.

Command:

- `# hping3 <IP> -S -p 8080 -i u<ms>`
- `<IP>` = IP of Android device

- `<ms>` = timeout in µ seconds between each sent package (default is 1s)

By varying the timeout between each package sent, different moderations of usage have been simulated and analyzed. The total runtime of each test is fixed to one hour.

7.4 Energy consumption and throughput benchmark

One of the biggest problems of modern Android devices is their short operation time before recharging the battery becomes necessary. Therefore the energy consumption of the DiscoWall decides the feasibility of the gained security when using the software.

Due to the device's package throughput limit, package loss occurs when exceeding this maximum package rate. This will influence the energy consumption of the device (as lost packages do not require CPU performance), which makes it necessary to analyze *(1)* the energy consumption and *(2)* the package throughput simultaneously.

7.4.1 Benchmarking process

Each test case is run for the duration of one hour while the *partial wake lock* is active. As described in paragraph *7.3* the transmitted packages are flagged with **SYN** and **ACK**.

```
# hping3 <IP> -S -p 8080 -i u<timeout-in-ms>
```

In order to accurately describe the energy consumption of the software, multiple usage scenarios are being analyzed. They are the product of testing different *firewall states* against different *package rates*.

Note: The application "3C Toolbox" has been installed only for finding the most active Android processes on an idle device with active DiscoWall. After this test has been completed, the utility has been removed in order to not affect any test results.

Tested firewall states:

1) Firewall disabled
2) Firewall enabled with 0 rules
3) Firewall enabled with 1000 rules (accepting packages using **iptables rules**)
4) Firewall enabled with 1000 rules (accepting packages using **Netfilter-Bridge**)

Tested network load:

1) 0 packages/s
2) 10 packages/s
3) 100 packages/s
4) 1000 packages/s
5) 10000 packages/s

Resulting data:

1) Energy consumption
2) Package loss

7.4.2 Results

7.4.2.1 Firewall CPU time

In *Figure 9: CPU time of apps during idle device (DiscoWall enabled)* the CPU time of all running Android applications has been measured (using "3C Toolbox") while no network traffic was active and the *partial wake lock* enabled.

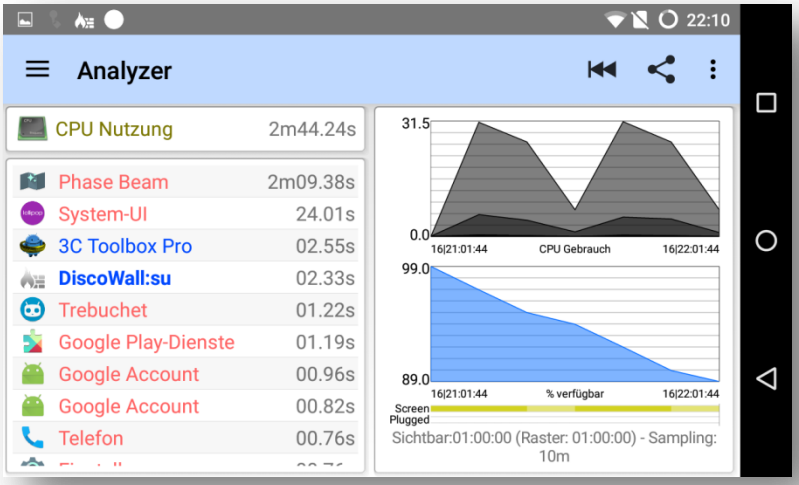

Figure 9: CPU time of apps during idle device (DiscoWall enabled)

The total CPU time amounts up to 2m44s, during which the "Phase Beam" *Live Wallpaper* (the default animated background image of the ROM) required by far the most CPU time (2m9s). It is followed by the "System-UI" (group of GUI applications including status bar, lock-screen, etc.) and third the "3C Toolbox" app itself. At the fourth place the DiscoWall application can be found, requiring 2.33s of CPU time.

This result shows the CPU time required by the firewall being negligible, as it amounts up to 9.7% of the time used by the ROM's "System-UI" process. When being compared to the background wallpaper ("Phase Beam"), the difference is even larger with only taking 1.8% of its CPU time. Therefore DiscoWall creates no noticeable battery drain for the device in idle state.

7.4.2.2 Disabled firewall

Packages/s	Firewall Status	Packages (loss, total)		Battery Voltage
0	Disabled	0%	0	2036mAh (97%)
10	Disabled	1%	36,000	1952mAh (93%)
100	Disabled	1%	360,000	1930mAh (92%)
1000	Disabled	48%	3,600,000	1890mAh (90%)
10,000	Disabled	100%	36,000,000	* 1952mAh (93%)

Table 3: energy consumption and package loss (firewall disabled)

* *Note:* As can be seen from the test case where 10,000 packages/s have been sent, there is a limit to the processing capabilities of the device's WiFi transceiver. When comparing the energy consumption of this result with the one of the 10 packages/s iteration, it can be observed that the same amount energy has been lost. Very similar behavior can also be observed at the next test case with the same package-rate.

7.4.2.3 Enabled firewall – static rules (Iptables)

Here the packages are being accepted using static iptables rules, created by the DiscoWall application. This is in contrast to the next test case, where each package is being accepted using the Netfilter-Bridge's nfqueue callback.

Packages/s	Firewall Status	Packages (loss, total)		Battery Voltage
0	Enabled, 0 rules	0%	0	2036mAh (97%)
10	Enabled, 0 rules	1%	36,000	1950mAh (93%)
100	Enabled, 0 rules	1%	360,000	1940mAh (92%)
1000	Enabled, 0 rules	57%	3,600,000	1889mAh (90%)
10,000	Enabled, 0 rules	100%	36,000,000	1909mAh (91%)

Table 4: energy consumption and package loss (0 rules, iptables)

Packages/s	Firewall Status	Packages (loss, total)		Battery Voltage
0	Enabled, 1000 rules	0%	0	2036mAh (97%)
10	Enabled, 1000 rules	1%	36,000	1952mAh (93%)
100	Enabled, 1000 rules	1%	360,000	1952mAh (93%)
1000	Enabled, 1000 rules	53%	3,600,000	1890mAh (90%)
10,000	Enabled, 1000 rules	100%	36,000,000	1913mAh (91%)

Table 5: energy consumption and package loss (1000 rules, iptables)

7.4.2.4 Enabled firewall – frontend rules (Netfilter-Bridge)

Packages are being accepted using the Netfilter-Bridge binary. The rules are exclusively defined within the frontend component of the firewall (see *4.1*), for which communication (via TCP) between both components needs to take place. The verdict decision is therefore made within user space, for which the energy requirement is bound to be greater than the one of the kernel space algorithms run by iptables. In addition, an iptables rule (i.e. the jump to the "NFQUEUE" target) is required for handling packages using the Netfilter-Bridge.

Packages/s	Firewall Status	Package Loss	Battery Voltage
0	Enabled, 0 rules	0%	2036mAh (97%)
10	Enabled, 0 rules	1%	1970mAh (94%)
100	Enabled, 0 rules	98%	1970mAh (94%)
1000	Enabled, 0 rules	100%	1958mAh (93%)
10,000	Enabled, 0 rules	100%	1929mAh (92%)

Table 6: energy consumption and package loss (0 rules, netfilter-bridge)

Packages/s	Firewall Status	Package Loss	Battery Voltage
0	Enabled, 1000 rules	0%	2036mAh (97%)
10	Enabled, 1000 rules	1%	1972mAh (94%)
100	Enabled, 1000 rules	97%	1973mAh (94%)
1000	Enabled, 1000 rules	100%	1951mAh (93%)
10,000	Enabled, 1000 rules	100%	1931mAh (92%)

Table 7: energy consumption and package loss (1000 rules, netfilter-bridge)

7.4.3 Analysis

The results are grouped within **test groups** by package speed (e.g. 10 packages per second) containing the individual **test cases** (e.g. "firewall disabled" or "0 rules (iptables)").

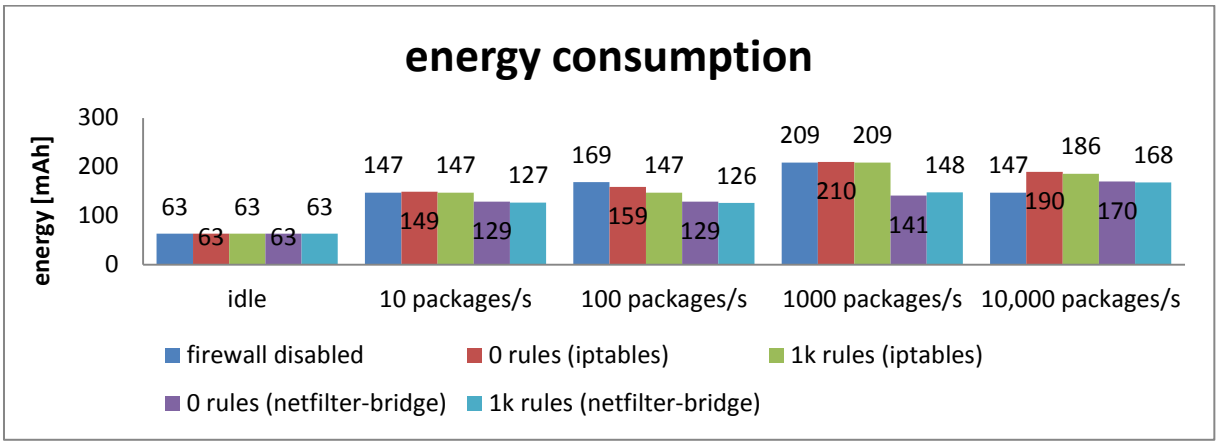

Figure 10: energy consumption comparison

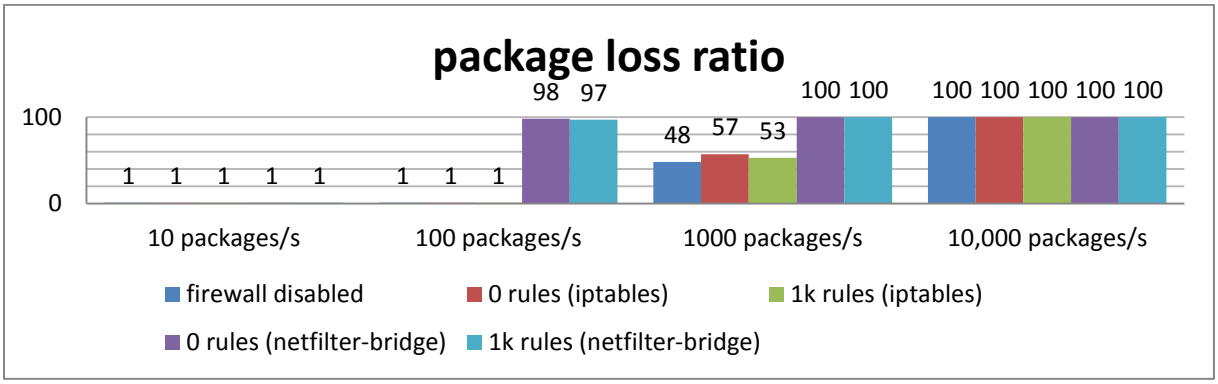

Figure 11: package loss ratio comparison

7.4.3.1 *Energy consumption vs. package loss ratio*

At first glance the energy consumption of the Netfilter-Bridge (which is using nfqueue callbacks in order to accept/block a package) seems extraordinarily low. In fact, it is **always below** iptables' energy consumption within each of the test groups. However, when comparing the values (see *Figure 10: energy consumption comparison*) with the package loss ratio (see *Figure 11: package loss ratio comparison*) one will discover the reason for this unlikely result: Due to the high package loss resulting from deciding the package verdict within user space (as opposed to iptables, which is working in kernel space), the actual amount of packages being processed by the Netfilter-Bridge is less than the one being processed by iptables. Thus the device requires less energy as it actually processes less packages.

Also when comparing the energy requirements for the "10,000 packages/s" test group with the ones using the slower rate of 1000 packages/s, the hypothesis is being confirmed: The iptables test cases using the higher rate, while having a package loss ratio of 100%, actually require **less energy** than those running with the slower rate (1000 packages/s) which have much less package loss. Only the Netfilter-Bridge has the same package loss ratio of 100% within both test groups – and here it can be observed that the tests running with **higher package rates** (> 1000 packages/s) actually require **more energy** (compared to the 100 packages/s case), just as one would expect.

Note: At 10,000 packages per second no package is being received by the Android device (100% package loss) – independent of the whether a firewall has been used or not. Therefore one can assume that there is a hardware limitation simply dropping packages once they reach a certain rate limit. Only then the costly CPU and RAM resource usage can be avoided and the device actually requires less energy than when receiving even 10% of those packages (i.e. 1000 packages per second).

7.4.3.2 Number of rules

When comparing the tests by **number of rules**, it becomes apparent that neither the **energy consumption**, nor the **latency** is being influenced by this parameter alone.

Especially when comparing the iptables results within the "1000 packages/s" group, it becomes quite obvious that the rules are not the significant factor for package loss ratio or energy requirement. Otherwise the iptables test using 1000 rules were bound to require more energy (or have a higher latency) than the one using no rules at all – where in fact a slightly inverse relationship can be seen.

7.4.3.3 Iptables vs. Netfilter-Bridge

In regards of energy consumption the Netfilter-Bridge requires less energy than iptables does. However, as described in the paragraph *Energy consumption vs. package loss ratio* (*7.4.3.1*), this comes at the price of a significant higher package loss ratio.

On the one hand while running at relatively low package rates around 10 packages per second, no significant difference between the two approaches can be observed and a firewall running within user space (Netfilter-Bridge) is just as feasible, as one running within kernel space (Iptables). On the other hand higher package rates, starting at 100 packages per second, come with a high penalty of almost 100% package loss (using Netfilter-Bridge), making the use of this user space implementation inefficient.

7.4.4 Conclusion

The energy consumption of DiscoWall, when being used for package filtering via Netfilter-Bridge, is relatively low. In fact, DiscoWall generally required less energy than iptables rules did – but only at the penalty of heavy package loss (100% at 1000 packages per second). The most efficient performance is reached at the relatively slow rate of 10 packages per second, when almost no package loss (1% and equal to using no firewall at all) occurs and the battery drain is still below iptables'.

As soon as the user creates static firewall rules, iptables is being used at kernel level for executing those rules. At this point there is no difference between DiscoWall and any other GUI frontend for iptables, which results in high performance when being compared to Netfilter-Bridge's nfqueue.

In general energy consumption of the DiscoWall – be it through user space algorithms via Netfilter-Bridge or kernel space algorithms via iptables – will never become significant enough for creating a relevant impact on the Android devices battery. In contrary – the energy consumption created by the device when using the firewall is not higher than it is without.

7.5 Latency benchmark

Just as important as the energy consumption resulting from using the DiscoWall software is the latency added to the networking performance of the mobile device. In the worst case scenario the latency resulting from handling each SYN package during connection setup could result in a timeout and thereby prevent connections from being established.

Independent from the additional latency created by the DiscoWall, any Android device has a maximum throughput, depending on *(1)* the network bandwidth, *(2)* the application receiving the data, *(3)* the CPU performance and finally *(4)* the network interface itself. When working with desktop systems, it is very rare for the limiting factor being anything but the network bandwidth. On mobile devices, however, the relatively low CPU performance (when compared to PCs) is a significant limitation when approaching the maximum network bandwidth.

7.5.1 Benchmarking process

As indicated above, the property being tested is the firewall latency when being confronted with a high number of **SYN-ACK** packages. Therefore the amount of data actually being transferred is negligible for this test and will not be analyzed any further. In each test case different *firewall states* will be tested for one hour.

Tested firewall states:

1) Firewall disabled
2) Firewall enabled with rules (accepting packages using **iptables rules**):
   10, 100, 1000, 10.000
3) Firewall enabled with rules (accepting packages using **Netfilter-Bridge**):
   10, 100, 1000, 10.000

Resulting data:

1) Latency per package
2) Minimum latency
3) Maximum latency
4) Average latency

7.5.2 Results

Each package's round trip time (RTT), out of the 3600 packages sent for each test case (one per second), has been individually logged. Due to this slow package rate no package loss occurred.

Note: The RTT diagrams use logarithmic scales for the y-axis, as the individual values are greatly distributed. The RTTs range from ~4ms to ~110ms, up to spikes between ~1100ms and ~4000ms.

7.5.2.1 Disabled firewall

Before analyzing the RTT induced by the firewall, the *system latency* of the device is being measured.

Firewall Status	Engine	RTT (min, Ø, max)		
Disabled	–	3.2ms	71.2ms	1113.7ms

Table 8: RTT (disabled firewall)

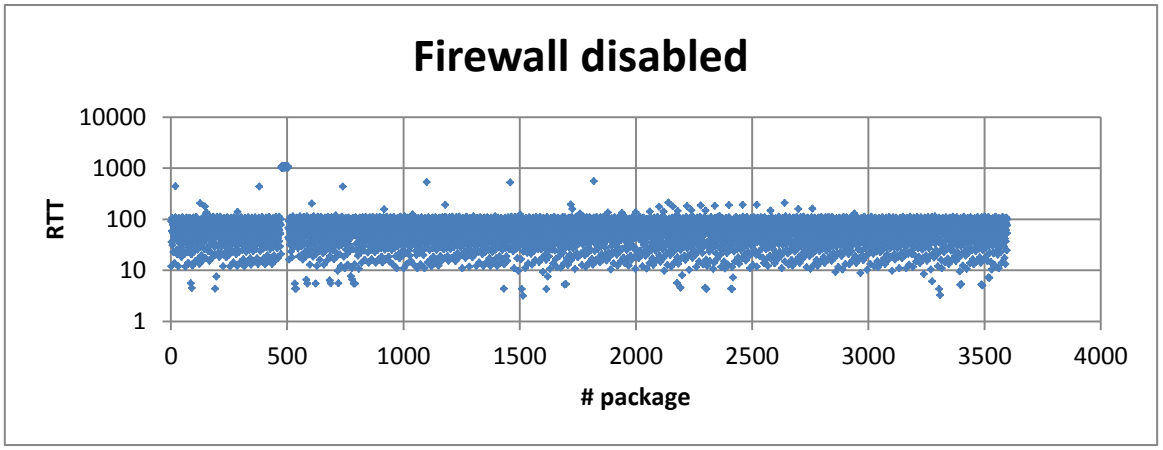

Figure 12: RTT per package (no firewall)

As depicted in *Figure 12: RTT per package (no firewall)*, the samples contain multiple spikes resulting from packages having extraordinary high RTTs. Before reaching a conclusion, the samples have to be *cleaned* by disregarding the spikes.

7.5.2.2 Static rules (Iptables)

Analogue to the energy consumption benchmark "*Enabled firewall – static rules (Iptables)*" (*7.4.2.3*), the latency of static iptables rules is being tested. No Netfilter-Bridge communication is required for this test.

Firewall Status	Engine	RTT (min, Ø, max)		
0 rules	Iptables	4.4ms	79.2ms	4052.1ms
10 rules	Iptables	4.4ms	71.1ms	1290.7ms
100 rules	Iptables	4.4ms	71.8ms	1130.6ms
1000 rules	Iptables	4.4ms	72.4ms	1111.9ms
10,000 rules	Iptables	6.7ms	82.3ms	1125.5ms

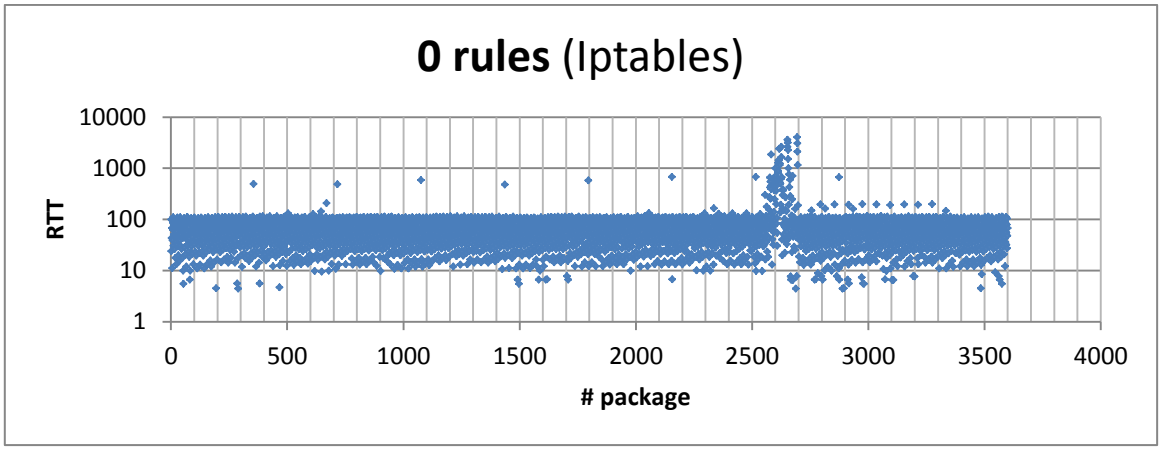

Figure 13: RTT per package (0 rules, iptables)

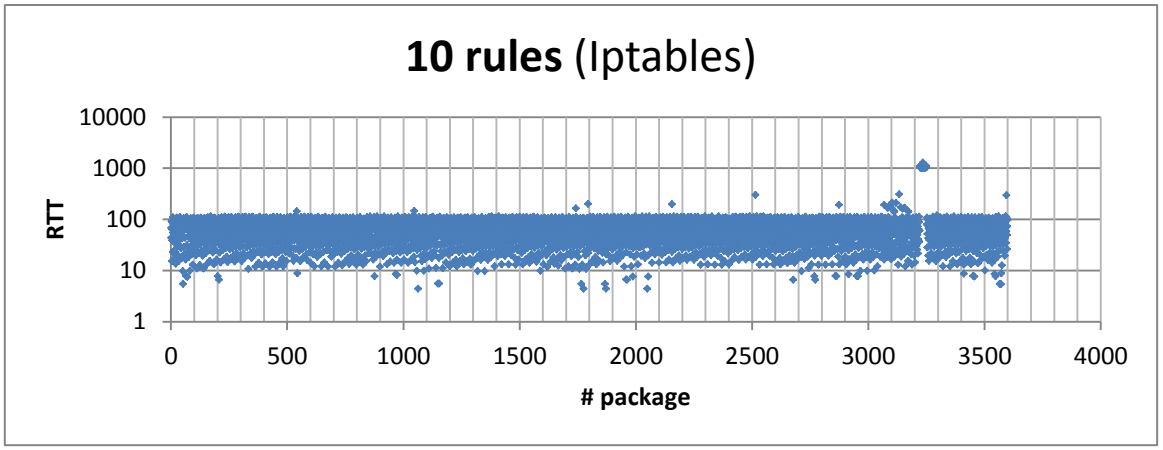

Figure 14: RTT per package (10 rules, iptables)

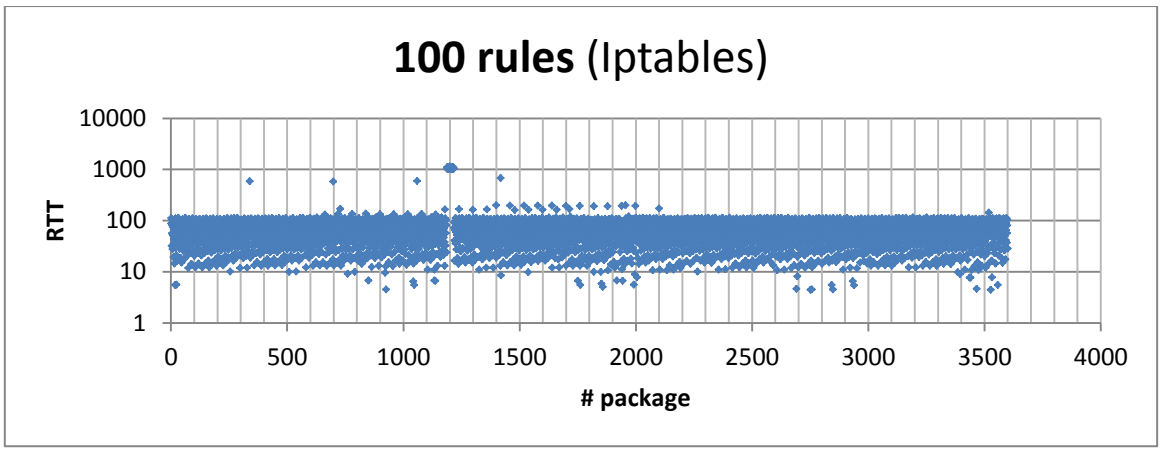

Figure 15: RTT per package (100 rules, iptables)

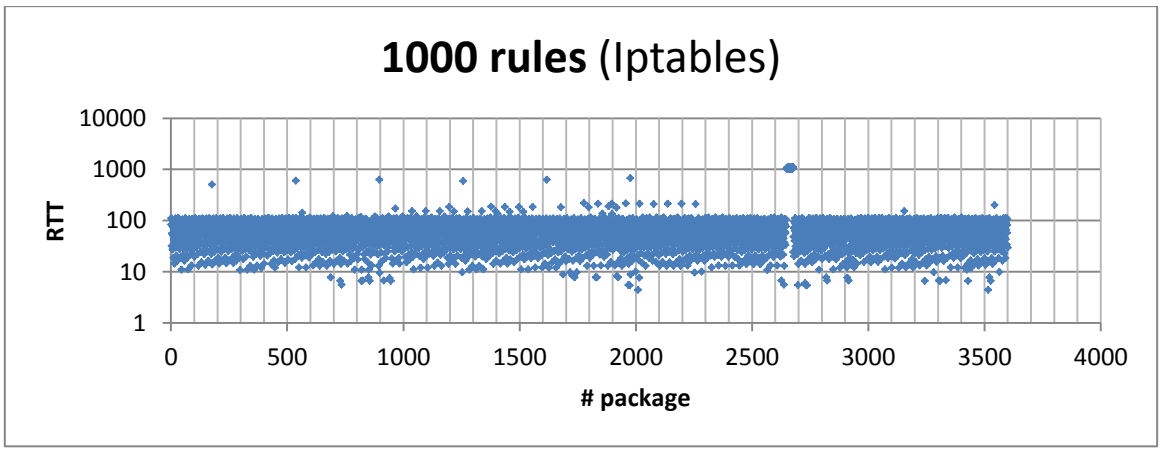

Figure 16: RTT per package (1000 rules, iptables)

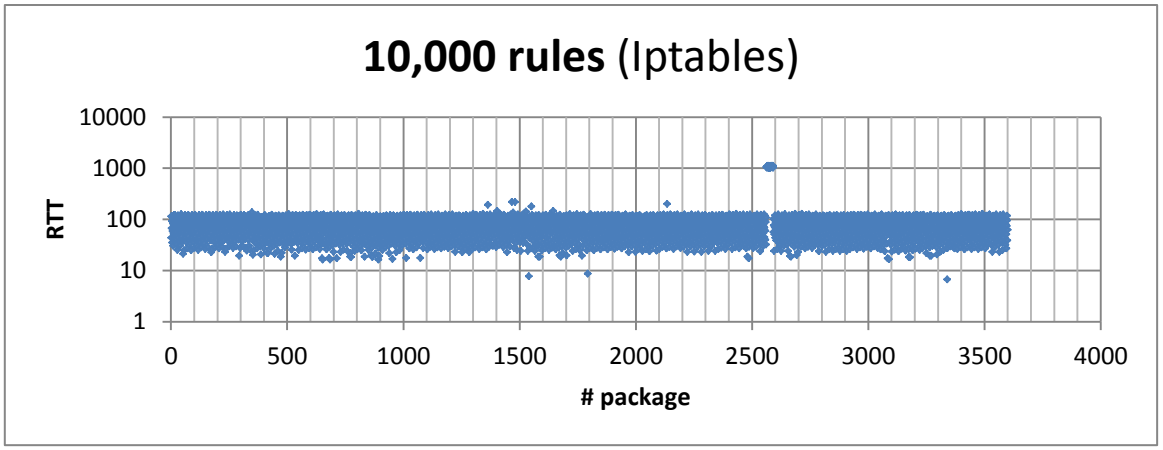

Figure 17: RTT per package (10,000 rules, iptables)

7.5.2.3 Frontend rules (Netfilter-Bridge)

Analogue to the energy consumption benchmark "Enabled firewall – frontend rules (Netfilter-Bridge)" (*7.4.2.4*), the latency of frontend rules is being tested. DiscoWall has to communicate with the Netfilter-Bridge in order to assign the verdict (here "accept") to the package. The latency is bound to be greater than the one resulting from the kernel space algorithms run by Iptables.

Firewall Status	Engine	RTT (min, Ø, max)		
0 rules	Netfilter-Bridge	74.6ms	164.5ms	310.5ms
10 rules	Netfilter-Bridge	82.7ms	187.3ms	1739.8ms
100 rules	Netfilter-Bridge	82.2ms	191.1ms	1579.5ms
1000 rules	Netfilter-Bridge	67.7ms	191.7ms	328.5ms
10,000 rules	Netfilter-Bridge	113.8ms	243.8ms	1358.6ms

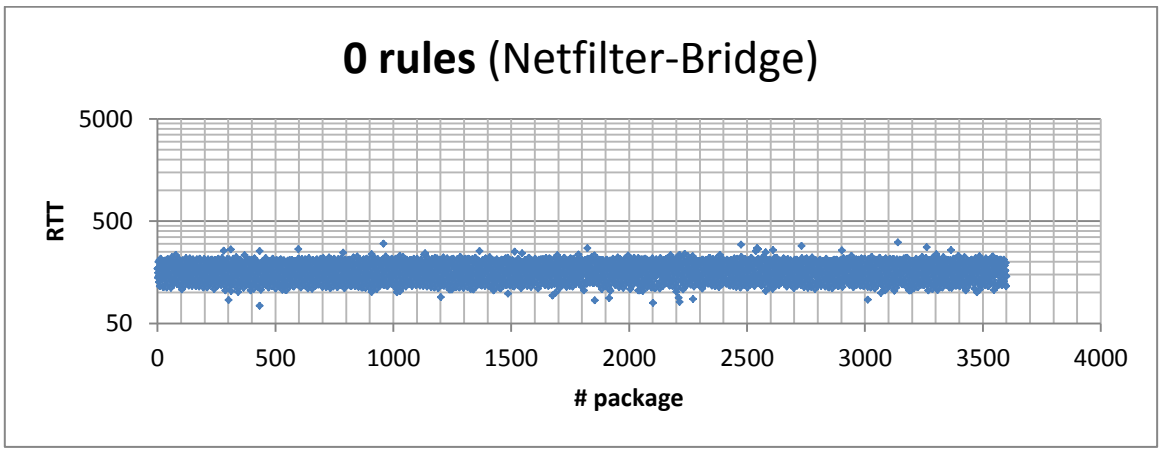

Figure 18: RTT per package (0 rules, netfilter-bridge)

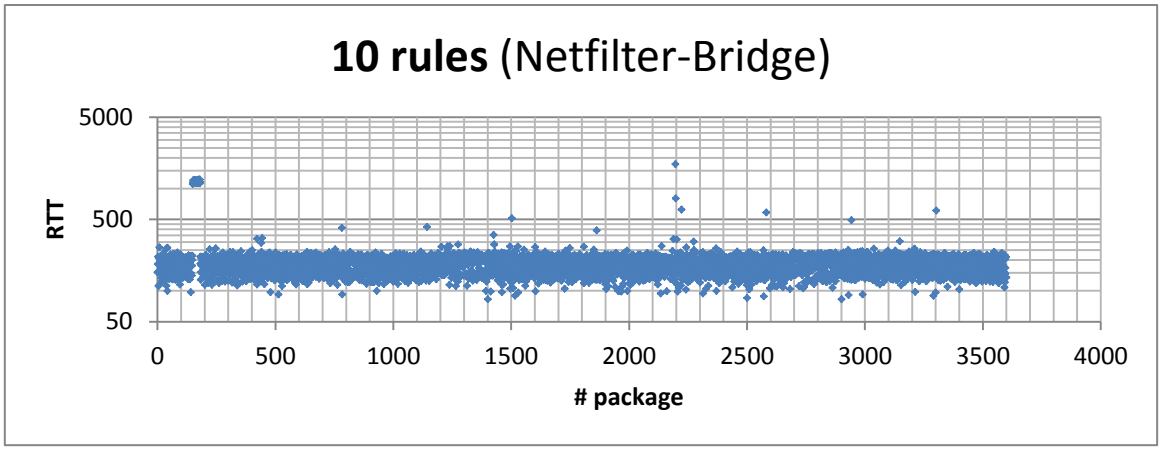

Figure 19: RTT per package (10 rules, netfilter-bridge)

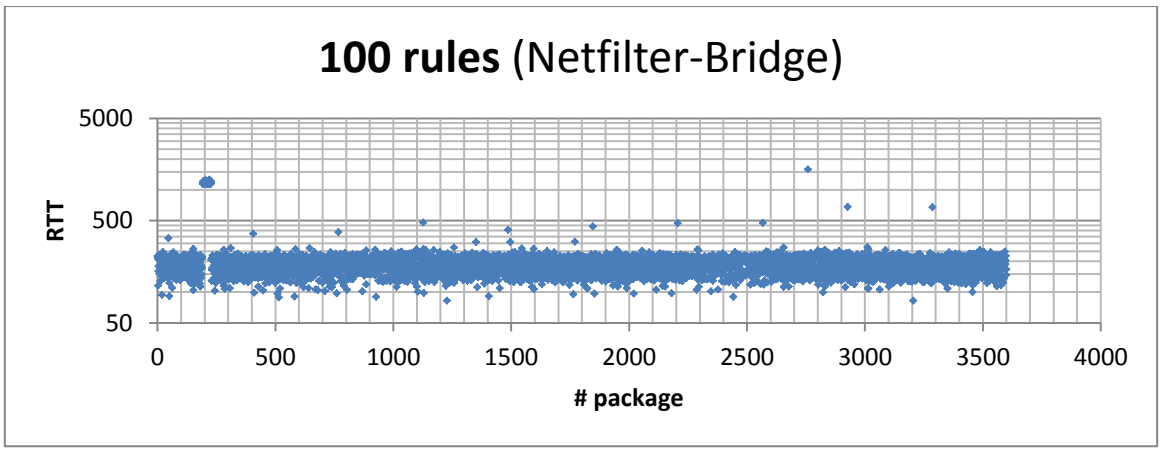

Figure 20: RTT per package (100 rules, netfilter-bridge)

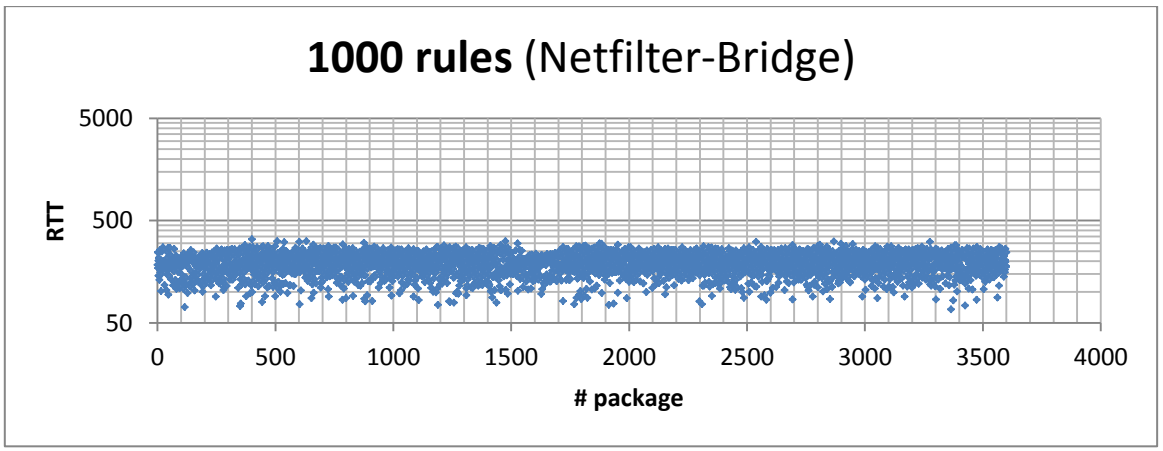

Figure 21: RTT per package (1000 rules, netfilter-bridge)

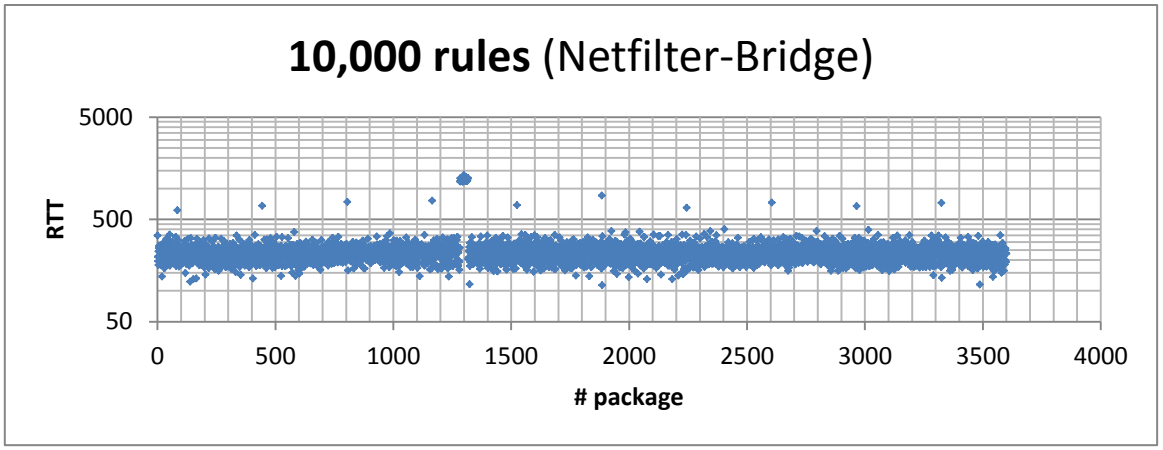

Figure 22: RTT per package (10,000 rules, netfilter-bridge)

7.5.3 Analysis

From the samples the **average**, **standard deviation** and **interval of convergence** (with 95% of all samples being contained in the interval) have been calculated.

Firewall Status	Engine	Average	Standard Deviation	Interval of Convergence
0 rules	Iptables	79.2ms	79.98ms	[14.4ms, 111.0ms]
10 rules	Iptables	71.1ms	176.20ms	[15.3ms, 110.2ms]
100 rules	Iptables	71.8ms	99.29ms	[15.3ms, 110.1ms]
1000 rules	Iptables	72.4ms	98.87ms	[15.4ms, 110.6ms]
10,000 rules	Iptables	82.3ms	95.43ms	[27.4ms, 121.2ms]

Table 9: latency benchmark analysis (iptables)

Firewall Status	Engine	Average	Standard Deviation	Interval of Convergence
0 rules	Netfilter-Bridge	164.5ms	32.16ms	[115.2ms, 212.3ms]
10 rules	Netfilter-Bridge	187.3ms	104.47ms	[125.5ms, 231.6ms]
100 rules	Netfilter-Bridge	191.1ms	109.80ms	[127.9ms, 233.6ms]
1000 rules	Netfilter-Bridge	191.7ms	43.90ms	[115.7ms, 261.0ms]
10,000 rules	Netfilter-Bridge	243.8ms	109.56ms	[173.6ms, 309.9ms]

Table 10: latency benchmark analysis (netfilter-bridge)

7.5.3.1 Standard deviations

Independent of the engine being used (iptables or Netfilter-Bridge) the **standard deviation** is always rather high. This indicates a software independent source of this behavior, as at least the test case without any iptables rules (“0 rules” case) should provide a very small standard deviation. However, even here the deviation is 79.98ms.

7.5.3.2 Average & Minimum

When comparing **average** and **minimum** of the used engines, it can be observed that both values are higher for the Netfilter-Bridge engine. This is the expected result, as the package’s verdict has to be decided within user space and TCP communication is required (see *Software-Components*), creating an unavoidable overhead.

7.5.3.3 Interval of Convergence

The **interval of convergence** shows the latency which statistically applies to 95% of all packages transmitted. In both engines the interval boundaries increase (i.e. higher minimum and maximum

latencies) with the number of rules specified in the firewall. However, in both cases the increase is non-linear. When comparing the intervals of each test case for 10 rules with the respective one having 10,000 rules, it becomes apparent that the Netfilter-Bridge engine is less efficient in finding the rule for an individual package. That causes the upper bound for the latency increasing roughly by the same fraction as the lower one.

Increase of minimum and maximum latency (10 rules vs 10,000 rules):

- Iptables engine: **90%** higher minimal latency, **9%** higher maximum latency
- Netfilter-Bridge engine: **38%** higher minimal latency, **34%** higher maximum latency

When comparing the minimum and maximum latency of both engines for the "0 rules" test case, the basic difference in latency can be observed:

- Minimum latency: **800%** increase (115.2ms vs. 14.4ms)
- Maximum latency: **191%** increase (212.3ms vs. 111ms)

7.5.4 Conclusion

The difference between deciding the package verdict (accept/block) within user space (Netfilter-Bridge) and kernel space (iptables) is significant in regards to the latency. Even though the Android device's performance fluctuates in both cases, the **extreme values** (minimum, maximum) deviate even further when using the DiscoWall's Netfilter-Bridge component.

The communication between frontend (DiscoWall GUI, service) and backend (Netfilter-Bridge) adds a heavy penalty to each package's RTT, increasing the **minimum RTT** from 4.4ms (iptables) up to 67.7ms (Netfilter-Bridge). Also the **interval of convergence** shows an increase of the expected minimal latency of approximately 800% as well as an increase of the maximum latency of 191% (when using no rules).

The results show the importance of efficient rule-matching algorithms within the DiscoWall service. The algorithms currently used for finding the rules associated to a package are non-optimized and simply iterate over the set of existing rules, checking each rule against the package one after another. This simple implementation creates a high impact on the package latency when using many rules and stresses the preconceptions known for implementing package filtering in a user space application. On the same time, it gives a hint to the solution of the problem (i.e. using hash maps or other associative mapping techniques).

8 Future Work

8.1 Root shell performance

The Android framework uses **modeless** dialogs only, resulting in code being executed convoluted without the developer having implemented this behavior purposely. Additionally many **threads** are being used, as work has to be done while showing "busy…" dialogs. These implementations result in the need for strict control over access to shared resources – and this can be accomplished by first limiting their usages. Due to this problem, each **root command** is being executed in a **separate shell** within the DiscoWall frontend.

Therefore each execution implies *(1)* starting an instance of the root shell "su", *(2)* executing the desired command by writing into the stream, *(3)* reading the result and *(4)* terminating the instance with "exit". As this has to be done **for each root command**, one can easily see that this is not very efficient. The entire process creates a big overhead and takes more time than needed for the execution of one desired root command.

The execution of all root commands (firewall start/stop, creation/deletion of rules) can be done faster, when knowing of this problem before designing the firewall architecture (or any complex Android root application) with this in mind.

8.2 Netfilter nfqueue and Netfilter-Bridge

8.2.1 Direct library access

The bottleneck of the DiscoWall's package-handling performance is the communication between the frontend component and Netfilter-Bridge (see *4.1*). To date the Android framework does not allow the compilation of shared libraries or applications accessing foreign native libraries (i.e. libraries which are not part of the NDK framework or Android system, see *5.1.2*).

Possible solutions:

a) Enhancing the communication protocol between DiscoWall frontend and Netfilter-Bridge
b) Recompiling Android ROM and adding the nfqueue library to the list of standard libraries

8.2.2 Multiple threads for nfqueue access

The current implementation of the Netfilter-Bridge waits for the user to decide the verdict of a package, which results in the package blocking the rest of the queue. Netfilter provides a solution for this problem by allowing multiple threads handling different packages at the same time. By using the iptables rule "`-j NFQUEUE --queue-balance`" instead of "`-j NFQUEUE --queue-num <n>`" automatic load-balancing can be enabled and thereby handling the rest of the otherwise blocked queue within different threads.

8.3 DNS support

Android does not use a DNS cache and therefore cannot perform a reverse DNS lookup of the package's IP. However, this is required for identifying a host after it has changed its IP or to show the hostname to the user when deciding the connection's verdict.

This may be solved by:

a) Installing a DNS cache on the Android device
b) Sniffing the DNS traffic on the Android device

8.4 Additional Protocols

Currently the DiscoWall application supports the transport layer protocols TCP and UDP, while the Netfilter-Bridge is already capable of handling ICMP traffic (as well as the layers above transport layer up to physical layer). In order to increase the security of an Android device, support of any other protocol may be implemented. Especially the decoding of DNS traffic can be used for the reverse lookup of package's IPs.

8.5 Advanced Rules

As nfqueue/libnetfilter allows access to the packages before deciding a verdict, "advanced rules" can be implemented, filtering packages **depending on their content**. Examples for such rules are:

- Block specific application layer protocols
- Limit traffic throughput per second

Another possibility is the **alteration of packages** as they are being received or sent. For such an alteration a package might simply be dropped and replaced by an altered copy. Resulting from such functionality would be rules like:

- Manipulation of webpages (e.g. removing links to large pictures in order to reduce traffic consumption)
- Transparently encrypt/decrypt data

9 Related Work

9.1 Existing Android firewalls

While differing in functionality, many Android firewalls exist to date. By looking at the technology used, they can be separated into two groups:

a) Firewalls being a frontend to iptables and requiring root privileges
b) Firewalls using a VPN connection for working without root

For each group some firewall implementations are listed below.

Iptables frontends:

- DroidWall (play.google.com: DroidWall, 2015)
- AndroidFirewall+ (play.google.com: AFWall+, 2015)

VPN connection:

- MobiWall (play.google.com: Mobiwol, 2015)
- Firewall without Root (Firewall ohne Root) (play.google.com: Firewall without Root, 2015)

9.2 Problems with existing firewalls

While the reasons for using a firewall are manifold, the main one is **security**. For users this does not only imply the usage of a firewall, but also its **configuration**. In terms of Android firewalls this means at the very least **defining firewall rules**.

9.2.1 Rule creation

However, neither the Iptables frontend firewalls nor the VPN connection firewalls allow for dynamic creation of rules during connection-setup. Both groups of firewalls rely on the principle mentioned in the *Motivation* section and define static routes for connections.

9.2.2 Technology

When using a VPN connection for filtering packages, one is limited to the use of exactly one VPN due to the Android API restrictions. Therefore it is not possible to establish an ordinary VPN connection to a remote network while also using a local firewall.

Additionally, the VPN API is part of the Android Framework and as such subject to great changes. As it is the case with many other API functions the VPN has been changed drastically – even resulting in a long VPN API bug-list for Android 4.4 KitKat (http://www.androidpolice.com - KitKat VPN Bug, 2015), making the use of a VPN firewall on such devices impossible.

9.2.3 Functionality

When using either Iptables or Android VPN for designing a firewall, the rules are limited to the information provided by those engines. However, by directly accessing the nfqueue/libnetfilter API, even the content of a package might be used for filtering (see *8.5*). Also, one might even decide to **alter** packages in order to remove advertisements from webpages, remove links to images in order to reduce the size of web content, etc.

10 Conclusion

Using user space algorithms such as within Android applications is feasible even for sensitive kernel space tasks like package filtering. As described in *Energy consumption vs. package loss ratio* (*7.4.3.1*) the package loss ratio is an important issue – but one which can easily be solved when using multiple nfqueue threads (see *5.2.4*) and an efficiency-optimized Netfilter-Bridge (i.e. communication between Android App and Netfilter nfqueue) implementation.

The increased transport layer latency (see *7.5.4*) is of far more pressing concern, as its solution requires a strongly optimized implementation which reduces the time spent on assigning the package verdict (i.e. accept or block the package). The communication with an Android service (or activity) creates an enormous bottleneck in this regards, which results in a large increase of a package's round trip times (RTT). Even when not applying any rules to a package (therefore no matching of rules has to be done) the expected RTT (i.e. after removing the spikes, see *7.5.3.3*) increases from [14.4ms, 111.0ms] to a range of [115.2ms, 212.3ms], adding additional 108ms to the package's minimal transmission time. Using a linear list-search algorithm for matching a set of 10,000 rules to each package creates an even higher latency increase of 146.2ms, revealing the expected sensitivity of running algorithms within user space.

The second important factor regarding the usability of such a firewall is the energy consumption. Independent of the employed algorithm's efficiency, the battery drain remains unaffected – instead, lack in efficiency creates additional package loss when transmitting with a rate higher than 100 packages per second (see *7.4.3.1*). Due to this loss, the packages will not be processed and additional energy will not be consumed.

Therefore it has been shown, that the efficiency of an Android firewall within user space is relevant to package loss (which may imply retransmission), but will not impact the energy consumption of the mobile device. As a result, such firewalls are not only feasible but the demand on their architecture and efficiency is relatively low.

Bibliography

code.google.com: Issues. (2015). Retrieved from https://code.google.com/p/android/issues/detail?id=14475

developer.android.com: Activity-Lifecycle. (2015). Retrieved from http://developer.android.com/training/basics/activity-lifecycle/index.html

developer.android.com: Android 4.1. (2015). Retrieved from http://developer.android.com/about/versions/android-4.1.html

developer.android.com: Android versions distribution. (2015). Retrieved from https://developer.android.com/about/dashboards/index.html

developer.android.com: Manifest. (2015). Retrieved from http://developer.android.com/guide/topics/manifest/manifest-element.html

developer.android.com: NDK. (2015). Retrieved from https://developer.android.com/tools/sdk/ndk/index.html

developer.android.com: Services. (2015). Retrieved from http://developer.android.com/guide/components/services.html

developer.android.com: WakeLock. (2015). Retrieved from https://developer.android.com/training/scheduling/wakelock.html

forum.xda-developers.com: pie security check. (2015). Retrieved from http://forum.xda-developers.com/showpost.php?p=56123703&postcount=13

forum.xda-developers.com: Seeder. (2015). Retrieved from http://forum.xda-developers.com/showthread.php?t=1987032

forum.xda-developers.com: SELinux Patch. (2015). Retrieved from http://forum.xda-developers.com/showpost.php?p=56123703&postcount=13

forum.xda-developers.com: SELinuxModeChanger. (2015). Retrieved from http://forum.xda-developers.com/showthread.php?t=2524485

Gradle.org. (2015). Retrieved from http://gradle.org/

http://www.androidpolice.com - KitKat VPN Bug. (2015). Retrieved from http://www.androidpolice.com/2014/03/31/bug-watch-vpn-issues-in-kitkat-are-interfering-with-connection-routing-fixes-are-planned-for-some-of-them/

linuxhowtos.org: marktarget. (2015). Retrieved from www.linuxhowtos.org/Security/iptables.htm#marktarget

Netfilter.org. (2015). Retrieved from http://www.netfilter.org/

packages.ubuntu.com: hping3. (2015). Retrieved from http://packages.ubuntu.com/de/precise/hping3

play.google.com: AFWall+. (2015). Retrieved from https://play.google.com/store/apps/details?id=dev.ukanth.ufirewall&hl=de

play.google.com: DroidWall. (2015). Retrieved from https://play.google.com/store/apps/details?id=com.googlecode.droidwall.free&hl=de

play.google.com: Firewall without Root. (2015). Retrieved from https://play.google.com/store/apps/details?id=app.greyshirts.firewall

play.google.com: Iptables2. (2015). Retrieved from https://play.google.com/store/apps/details?id=jp.ymatsumoto.iptables2

play.google.com: Mobiwol. (2015). Retrieved from https://play.google.com/store/apps/details?id=com.netspark.firewall

source.android.com: Security Enhancements. (2015). Retrieved from https://source.android.com/devices/tech/security/enhancements/enhancements50.html

source.android.com: SELinux. (2015). Retrieved from https://source.android.com/devices/tech/security/selinux/

Statista.com. (2015). Von http://de.statista.com/statistik/daten/studie/182363/umfrage/prognostizierte-marktanteile-bei-smartphone-betriebssystemen/ abgerufen

Intel.com: using NDK with android studio. (n.d.). Retrieved from https://software.intel.com/en-us/videos/using-the-ndk-with-android-studio

kvurd.com: compiling a cpp library for android. (n.d.). Retrieved from http://kvurd.com/blog/compiling-a-cpp-library-for-android-with-android-studio/

roman10. (2011, 12 1). *roman10.net: how to build libnetfilter-queue*. Retrieved from http://www.roman10.net/how-to-build-and-use-libnetfilter_queue-for-android/