Karlsruher Institut für Technologie

Analysing Page Duplication on Android

Studienarbeit
von

Jonas Julino

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dipl.-Inform. Konrad Miller

Bearbeitungszeit: 24. Oktober 2011 – 27. März 2012

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Jonas Julino
Karlsruhe
March 26, 2012

Deutsche Zusammenfassung

Gegenstand dieser Studienarbeit ist die Analyse der Seiten-Duplikation auf einem Android Mobiltelefon und die Entwicklung der dafür nötigen Werkzeuge. Des Weiteren gibt diese Studienarbeit einen Überblick über das Betriebssystem Android, Seiten-Duplikation und bereits existierende Ansätze zu deren Messung.

Es nicht ohne Weiteres möglich Kernel-Module unter Android zu laden, da es dem Benutzer auf unmodifizierten Geräten normalerweise nicht möglich ist root-Zugriff zu erlangen und darüber hinaus der Kernel einiger Geräte das Laden von Kernel-Modulen nicht unterstützt. So bietet der mit dem Android SDK [4] gelieferte Emulator zwar root-Zugriff, aber der mitgelieferte Kernel ermöglicht es nicht, Module zu laden. Deshalb sind die bestehenden Lösungen für die Messung von Seiten-Duplikation nicht ohne Modifikationen am Gerät beziehungsweise Betriebssystem anwendbar.
Es ist aber möglich an das Speicherabbild eines virtuellen, im Emulator laufenden, Android Gerätes zu gelangen, ohne das virtuelle Gerät zu modifizieren. Um diese Datenquelle nutzen zu können, haben wir ein eigenes Programm entwickelt, das alle nötigen Informationen aus einem Speicherabbild mittels Methoden der Computer-Forensik extrahiert.
Die hierbei gewonnen Informationen bezüglich der einzelnen Seiten umfassen unter anderem die ID des zugehörigen Prozesses, virtuelle und physische Adressen, zugeordnete Dateien (wenn vorhanden) und Zugriffsrechte. Des Weitern können im Kernel als unbenutzte geführte Seiten markiert und aussortiert werden.

Mittels des entwickelten Programms ist es uns möglich die Seiten-Duplikation zu untersuchen und das daraus folgende Einsparungspotential zu benennen. Unsere Messung ergibt, dass es sich in einem normalen Anwendungsfall bei 9,5% der benutzten, das heißt nicht als frei im Kernel markierten, Seiten um Duplikate handelt. (Genullte Seiten und Seiten aus dem Benutzer-Adressraum, die für memory-mapped I/O genutzt werden, sind in dieser Betrachtung nicht enthalten.)
Um die Quellen für diese Duplikate näher zu betrachten, unterteilen wir das System in mehrere Gruppen (u.a. in *Kernel* und *Dalvík Virtual Machine (Dalvík* VM*)*) und analysieren die Duplikate innerhalb dieser Gruppen. Wir identifizieren die Dalvík VM hierbei als Hauptquelle der Duplikate.

Abstract

Page duplication is often referred to as a problem of virtualized environments. In this theses we argue that there are memory saving opportunities on mobile devices – in this case on an Android device, too.
As loading kernel modules is problematic on Android devices (no root access and/or the kernel does not support loading modules) we decided to measure by analyzing the memory dump of the device. We use an emulated Android device [4], therefore we can easily acquire the memory content from outside of the (virtual) device. Our measurement method does not require any modification to the device or manual search in the dump; all information is automatically extracted by reverse engineering the structures found in the memory dump.
The information we can assign to each page includes the process id(s), virtual and physical addresses, attached file (for named pages) and access rights. We are also able to distinguish between used and free pages (i.e. marked as unused in the kernel).
Our evaluation depicts the total sharing opportunities and the opportunities within certain groups (e.g., the Dalvík Virtual Machine (Dalvík VM)).

Contents

1. Introduction

As software for mobile phones becomes more complex the amount of needed random access memory (RAM) increases, too. While enlarging RAM-size of common desktop computers is unproblematic and cheap, it is difficult for mobile device as it effects two of their important attributes: power consumption and size.
RAM is a constant power consumer; even when a mobile phone is in standby mode its RAM must be powered. For small amounts of RAM[1] its power consumption is low compared to other components [13], nevertheless increasing the RAM-size raises the power consumption of mobile devices and increases their size, in case of using a larger battery. As the single parts of mobile devices are packed together extremely narrowly, adding a single (memory) chip might imply a larger housing in any case.
Multitasking operating systems (OSes) are prone to load the same pieces of data into multiple physical pages (page duplication) as lots of programs, accessing partially equal data, run in parallel. Reducing page duplication yields a reduction of memory consumption, as all but one page containing the duplicated page content can be freed.
In this theses we measure the page duplication on Android mobile phones and analyze the memory saving potential. We chose Android as it is a very widespread OS for mobile devices and its source code can be freely obtained.
The announcement of VMware to develop a virtualization environment for Android [21] is an interesting aspect of this platform in future as running multiple OSes on a single device is likely to significantly increase the page duplication.

This theses covers the design decisions made during the development of our measurement tools (§ 3), implementation details of this tools (§ 4) and an evaluation of page duplication on a (virtual) Android device (§ 5).

[1] The device whose RAM power consumption is measured in [13] is equipped with 128 MB of RAM.

2. Background and Related Work

This chapter defines often used terms and provides an overview of existing tools (§ 2.2), deduplication approaches (§ 2.3) and the Android OS (§ 2.4).

2.1. Terms

Table 2.1 defines the most important terms used in this theses, for further definitions see the glossary at page 27.

Term	Description
Page Duplication	Situation in which the content of multiple physical pages is equal.
Page Deduplication	Reducing the count of duplicated pages.
Sharable Pages	Pages whose content occurs at least twice in the system.
Duplicated Pages	A subset of shareable pages, containing all but one page with the same content from the set of sharable pages; i.e., all pages in this set could be freed without loosing the page *content*.
Shared Pages	Pages referred to by at least two virtual addresses.
Offset	Address in the *target system*. Refers to physical addresses, unless stated otherwise. (We introduce this term to avoid confusion with addresses on host system)
Delta	Distance from the start of a structure to an entry inside this structure.

Table 2.1.: Terms & Definitions

2.2. Existing Tools

This section covers tools that are able to measure page duplication and the algorithms we can use as a base for own measurement approaches.

A suite for online page duplication measurement is available for Linux based systems. A kernel module acquires the data inside, whereas a userland application does the evaluation offline. The information it passes to the userland covers meta data of the pages

(e.g., owner, access rights, ...) and hashes of the page content, but not the content itself. [19]
As loading kernel modules is problematic on Android, we cannot use this approach.

The *Red Hat Crash Utility* is a tool for debugging the Linux kernel by investigating memory dumps. [5]
The supported architectures include ARM, however, there is currently no support for investigating an ARM dump on an architecture other than ARM. [6] Furthermore, we did not find any evidence of a successful usage of the crash utility on an Android memory dump.

Volatilitux is a python tool for computer forensics which is able to detect structures and their deltas in a memory dump of a Linux system. It provides functions for listing running process and accessing applications' memory, but it does not provide direct access to all structures (i.e. the kernel's free-lists) we need for our measurement.
The developer of Volatilitux successfully tested his program on Android devices. [17]

2.3. Page Deduplication

There are several approaches to merge duplicated pages; we divide them into active and passive ones. While active approaches merge already existing duplicated pages, passive approaches prevent duplicates from being established.

Active deduplication reduces the memory consumption at cost of CPU-time, as this approach requires active scanning for merge-able pages. A widely used implementation is Kernel Shared Memory (KSM), a Linux kernel module, which is able to merge duplicated anonymous pages. It detects duplicates by periodically comparing the content of pages. A common use case for KSM is the reduction of memory consumption on a Linux host which runs multiple similar guest OSes as virtual machines. [7, 26]

Creating a new process by calling `fork` or mapping a file into multiple address spaces can create a lot of page duplicates; *passive deduplication* avoids this by introducing copy-on-write (COW). All passive approaches rely on a priori knowledge of the sources of page duplication; we can only merge duplicated pages whose establishing we can observe.
COW avoids page duplicates which would arise upon a fork. Instead copying the complete address space, the pages of the creator can be marked as COW and mapped into the address space of the created process. A page is only copied if one of both processes tries to modify it (the split pages are normally no longer equal and therefore not a target for deduplication anymore).
If a *memory mapped file* is mapped into multiple address spaces as *shared* or the single pages are unmodified, it is not required to create multiple copies of the file's pages in the physical memory, as the content of the pages is equal in all address spaces. But in case of an unmodified private mapped file, the pages must be marked as COW. [12, 24]

2.4. Android

Android is an OS for mobile devices aimed to provide, among others, low power consumption. It consists of a modified Linux kernel and userland programs known from other Linux distributions, e.g., *wpa_ supplicant*.
An essential part of Android is the Dalvík VM. All programs normally directly executed by a user on Android are written in Java and run in this virtual machine (VM). The Dalvík VM does not directly execute `jar`-files; they must be converted to a more dense file format called `dex`. [11] Each Java applications is assigned a different user id and started in its own VM. [1]

On consumer devices the user's access to the OS is often restricted; acquiring root access is only possible by exploiting software bugs[1]. The kernels of some devices (like the emulator contained in the Android SDK [4]) are not configured to support loading of modules. [4]
It is, however, possible to save a memory dump of a virtual Android device which is run inside the Android emulator from the SDK. Acquiring a dump of a physical Android device might also be possible e.g., by using a cold boot attack [20].

2.4.1. Peculiarities in Memory Handling

Every non file-mapped page containing valid data resides in the RAM as a common Android device does not use swap. We validated the absence of swap space on our test device by verifying there is not `/proc/swaps`.

To avoid the termination of current foreground applications the Android kernel replaces the *out of memory killer* from Linux by its own implementation, the *low memory killer*. It is started early before a low memory situation to allow applications to save their context; the selection for termination is based on multiple attributes including the current visibility to the user. [11]

Android OS provides a device `/dev/ashmem` (*Anonymous Shared Memory*) which programmers can use to establish an anonymous shared memory region. As a region is set up by memory mapping the device above, this region appears to be named. [11]

2.4.2. Page Deduplication in Dalvík VM

On Android a special process called *Zygote* is responsible for starting VMs; on system boot this process loads the system libraries into its memory and runs their initialization routines.
Anonymous pages created by the initialization are automatically shared among the VMs as Dalvík VM starts a new VM by forking (pages are not actually copied but marked as

[1] This is not true for the devices run inside the emulator, here it is possible to acquire root access. But as mentioned in the text, the kernels of the emulated devices do not support loading modules.

copy-on-write by the kernel). Shared memory mapped files (libraries) are also shared among the processes. [9]

3. Design

This chapter describes several decisions we made during the development of our tools, concerning the choice of a measurement method (§ 3.1) and the page content representation (§ 3.2). As we choose to use hashes for the representation of page content, we also cover the consequences of hash collisions (§ 3.3).

3.1. Choosing a Measurement Approach

We want to measure the amount of page duplication on an Android device. We are also interested in the processes the duplicates belong to and information about their source (e.g., named/anonymous).
The measurement can be done from inside, by using a kernel module [19], or from outside the (simulated) Android device, by implementing a tool to make an offline analysis of a RAM dump.

Table 3.1 describes the advantages and disadvantages of both approaches:

Kernel Module	Offline Analysis
+ already implemented and tested on amd64 and x86 [19]	+ root access not required[i]
+ ability to use kernel API	+ kernel sources/headers not required
- module support must be activated	+ access to the complete page content
- kernel sources/headers required	- more difficult to implement
- measurement influences target	- only snapshot of one moment in time

[i] Root access might be necessary for acquiring the memory dump, if target is not run inside an emulator.

Table 3.1.: Comparison of Kernel Module and Offline Analysis

We use the offline analysis approach, as Android devices do not normally provide root access (§ 2.4) and loading kernel modules is not supported on all devices. It is possible to circumvent this limitations but this requires modifications to the target system. Another

reason is that the hack of kernel.org made it difficult[1] to attained the official sources of the Android kernel at the time of writing this theses.

We base our implementation on the algorithms used by *Volatilitux* (§ 2.2), as we known it works using an Android dump. In addition to that Volatilitux allows us to analyze an ARM memory dump on a foreign architecture, whereas this is not possible with the *Crash Utility* (§ 2.2).

To easily gain a copy of a device's RAM (§ 2.4) we decided to measure a virtual Android device running inside the Emulator from the Android SDK and not a physical one.

3.2. Page Content Representation

When working with duplicates it is useful to have an identifier referring to all pages with the same content, e.g., to request information on a certain page content and listing all pages containing this content. As we want to interact with our tools, the identifiers for the page content should be usable on the command-line.

An OS does generally not provide such an identifier (besides the page content itself) as it is not needed for the basic tasks of an OS; we therefore need to introduce an identifier on our own.

We considered the usage of the identifiers listed in Table 3.2:

Identifier	Description
Hex Representation	Page content (which is equal for all pages) represented by hexadecimal characters.
PFNs	All PFNs of pages inside the set.
Hash	A hash of the page content.

Table 3.2.: Identifiers for Page Content Representation

The features of these approaches are listed in Table 3.3:

[1] We would have needed to access a certain version matching the kernel of our device, and a special branch called *goldfish* when using the emulator, which was not possible.

	hex representation	PFNs	hash
length (for PFNs: highest number)	page size · 2	$\frac{\text{size of RAM}}{\text{page size}}$	length of hash
hash collision possible	no	no	yes
multiple identifiers for same content	no	yes	no
identifiers are stable[i] over time	yes	no	yes

[i] The identifiers of a content stays the same, regardless of a reboot, a closed application or a freed page.

Table 3.3.: Features of Identifiers for Page Content Representation

We dismiss PFNs as they are not stable over time and this is very confusing when working with multiple snapshots from different points in time; e.g., if we require information on a page content known from dump a, we do not know if the PFN is still the same in dump b. The hex representation is very long for common page sizes (e.g., 8192 characters long for a 4096 byte page) and therefore hard to handle.
We chose using hashes as identifiers; the algorithm we use is Message-Digest Algorithm 5 (MD5), resulting in a 32 hex character long identifier. The possibility of a hash collision and the resulting effects are covered in § 3.3.

As we already use hashes to describe the page content and we must therefore calculate the hashes in any case, we use them to detect duplicated pages, too. This improves the speed of the duplicate detection as comparing the hashes is faster than comparing the complete page content.

3.3. Hash Collision Consequences

We currently use the MD5 as a hash function and because we use a hash there is the possibility of a hash collision.
There are only minimal consequences of this possibility; see the following list of reasons:

Reason 1: Extremely Low Collision Probability

MD5 is not save to use for cryptographic issues [27], but as we do not have an active attacker, we do not mind the possibility of intendedly created collisions.
The collision probability in our scenario (ignoring weakness of MD5, an active attacker and assuming an equal distribution of hash-values) is extremely low, it can be estimated by using the Formula 3.1 from [25].

$$P_{\text{collision}}(N, n, s) \leq \frac{1}{N^{s-1}} \binom{n}{s} \tag{3.1}$$

N = Count of different hash values (for MD5: 2^{128})
n = Count of pages to hash (we assume 4 GB of RAM and 4KB page size $\rightarrow 2^{20}$ pages)
s = Number of pages with the same hash to consider as a collision (2)

$$P_{\text{collision}}(2^{128}, 2^{20}, 2) \leq \frac{1}{2^{128}} \binom{2^{20}}{2} \simeq 0.16 \cdot 10^{-26}$$

Reason 2: Little Effect On Measurement Results

If at all only a few pages should be detected as false-duplicates as the probability for a collision is low; this would pose only a minor modification to our results.

Reason 3: Active Check For Hash Collision

Our tools are able to detect hash collisions as we have access to the complete page content. (See § 4.4.2 for how this detection is implemented.)

4. Implementation

Our tools reverse engineer an (Android) memory dump to provide information on the pages and they detect and classify duplicates based on this information. The algorithms we use for the reverse engineering are extended/modified reimplementations of *Volatilitux*'s algorithms [17]. Our tools are implemented in C/C++ to reduce their runtime[1] compared to *Volatilitux*.

As the basic detection approaches are adopted from *Volatilitux* they are only addressed shortly and the main focus is on the extended detection abilities and the measurement code.

4.1. Target

The machine dependent part of our code is implemented for the (simulated) hardware listed in Table 4.1:

CPU	ARM 32-Bit (ARM926EJ-S rev 5)
OS	Android 2.3.1

Table 4.1.: Target (Simulated) Hardware

4.2. Assumptions/Constants

The algorithms used for detecting deltas and offsets of structures need some low-level constraints. It is possible to reduce the number of constants but this reduction significantly increases the amount of work for the detection.

Table 4.2 contains the most important constants we use:

[1] E.g., listing all processes on a 256 MB dump takes approx. 0.30 seconds using Volatilitux whereas our tools need about 0.05 seconds. Printing details on duplicates inside the same dump takes about 4.4 seconds with our tools. As Volatilitux does not implement this functionality we cannot measure its runtime, but if we assume the same speed up ratio from the previous measurement, we can suppose that Volatilitux would need about 26.4 seconds. We do not consider such a delay to be interactive. (We do not use configuration files, which could speed up their execution time when executed multiple times on the same memory dump, during these measurements named above.)

Name	Description	Default Value
PAGE_OFFSET	This offset describes the start of the kernel memory inside virtual address spaces.	0xC0000000
Bit-Count	The count of bits used to represent pointers, integers, etc. This is actual not a single value, but a value per data type.	32 bit
Page-Size	The size of a page	4096 byte
Endianness	The endianness of the target architecture	little-endian
Struct Max Size	The roughly (over-)estimated size of a structure	individual[i]

[i] We assign a different maximum size to every structure we detect. E.g., the maximum size for `task_struct` is 4096 byte.

Table 4.2.: Important Assumptions and Constants Used for Detection

To ease the detection of some structures needed to distinguish empty pages from other pages we assume that our target provides uniform memory access (UMA) and the kernel is configured to use `FLATMEM`; this implies that a single entity of the `pg_data_t` structure covers the whole physical memory.
This is a reasonable assumption *for an Android device*.

4.3. Preparation Stage

The preparation stage is responsible for the acquisition of entry points (§ 4.3.1), offset and delta detection (§ 4.3.2) and setting up structures which translate virtual addresses to physical ones (§ 4.3.4).
For an overview of the used structures and their connections see Figure A.1 on page 26.

4.3.1. Acquisition of Entry Points

An entry point describes an instance of a structure which we use as a starting point for detecting further structures and instances.
We currently need the offsets of two entry points for our tools. The first one is the `task_struct` owned by the thread *swapper*. Candidates, whose validation is addressed in § 4.3.3, are revealed by a string search for 'swapper' inside the memory dump (see [17] for details).
The search for the second entry point, the memory zone descriptor `pg_data_t`, is more time consuming as it does not contain a rarely used identifier such as a name and the detection algorithms are therefore more complex.
Table 4.3 describes the entries of `pg_data_t` which are used to find candidates for its offset. [23] The assumptions for their values are based on [17, 10, 22, 23].

Name	Description	Assumed Value [i]
`node_start_pfn`	Number of first frame in this node	small (< 255)
`node_spanned_pages`		$\sim \frac{\text{size of dump}}{\text{page-size}}$
`node_present_pages`		$\sim \frac{\text{size of dump}}{\text{page-size}}$
`node_id`	Global node identifier	0

[i] Assumed values are only valid if the target complies with the assumptions named in § 4.2.

Table 4.3.: Entries of `pg_data_t` Used for Our Detection of This Structure

Using these assumptions combined with the knowledge that this four entries should sequentially reside in memory we can generate a set of candidates.
The structure `pg_data_t`, and structures referred by it, often directly include other structures into their memory range. As avoidance of pointers significantly complicates the detection of deltas, we need multiple complex algorithms for the elimination of wrong offsets which can be found in our source code (see § 4.3.3 for the basic idea of validation).

4.3.2. Offset and Delta Detection

To extract information from the memory dump we need to know the layout of multiple structures used inside the kernel (e.g., `task_struct`, `pg_data_t`, ...) and where we can find instances of them.

The layout of structures is represented by the deltas of their entries. Based on the way the deltas are detected we can divide them into three groups (ordered by portability starting with best):

dynamic deltas These deltas are detected dynamically based on information on the entry they represent. This is the most portable solution across different kernel versions, as this does not make any assumptions on the order inside a structure.

Example: A delta refers to a pointer which points to a known string. To detect the value of this delta we assign all possible values to it and check whether the memory, the delta refers to, is a valid pointer to our known string.

mixed deltas Mixed deltas are dynamic deltas whose search area is restricted to only a part of the structure.

Example: A delta refers to an integer whose values is known, but this value occurs twice in the structure; one time at a delta of 0 and one time at the delta we search for. We can now ignore delta values 0-4 (bytes) to exclude the first integer (on a 32-bit system) from our detection.

static deltas As some parts of the kernel's structures stay stable over time, it is possible to hard-code the location relative to the start of the structure or an other delta inside the structure.

Although we call them static deltas they are often defined in units of *size of int, pointer, long, etc.* and not in bytes.

Example: The first entry in the `page` structure is known to be a long integer containing the flags for the page and this does not change for different kernel versions. So we know the delta for the flags is 0.

Choosing dynamic detection is not always possible as the high portability implies high requirements for the detection algorithm. It is generally a difficult and costly task to detect a delta, therefore it is advisable to reduce the amount of used deltas as far as possible.

The detection code for deltas traverses the structures inside the kernel, as a result we automatically detect most of the offsets, which represent the single instances of the structures, while we detect the deltas.

4.3.3. Offset and Delta Verification

The main problem of the preparation stage are the dependencies among different deltas and offsets.
The detection of a delta is done by assigning multiple values to it and validating the choice. In most cases it is not possible to validate each delta one by one, instead we have to make multiple assumptions and validate the assigned values all together.
The resulting code, containing n nested loops where n is the count of deltas to be detected at once, can still be executed in an acceptable amount of time due to the small ranges of the single loops, caused by the limits on the structures' sizes we assume in § 4.2.

The detection of offsets does not generally provide a range limit, in worst case it is necessary to search the complete memory dump byte-per-byte. But the amount of offsets without any range limits is extremely low.

Example: Validation using a Pointer

This is an example on how we are able to reduce the number of possible offsets for a structure which contains a pointer. The following items describe several situation in which we can make use of a pointer:

- We can exclude an offset if we know the address the pointer is to point to and this value does not occur within the area of our supposed offset.

- If we do not know the address the pointer should point to, we can try all possible values of the pointer's delta and check whether it points to a valid structure of a type we expected. Implementing this check is difficult but often we are interested in the structure pointed to, so this effort is necessary in any case.

- Without information about the pointer's target but with knowledge of the exact delta of a pointer, we are able to exclude every offset whose dedicated pointer is not valid. (This does only work if we know this pointer must be valid at any point in time.)

4.3.4. Resolving Virtual Addresses

We can translate virtual to physical kernel addresses by adding the offset PAGE_OFFSET. To resolve addresses of a userland process we walk its page-table, whose position is held by the entry `pgd` inside the `mm_struct` assigned to the process' `task_struct`.
If an architecture is hardware-walked like i.e. ARM, the structure of page tables is stable. We used an ARM manual [8] for implementing our table walking code.

We can traverse all pages of a process by translating every virtual address which is inside the region of an assigned `vm_area_struct`. For more details refer to [17].

4.3.5. Detection Error Avoidance

To prevent erroneous measurement results, caused by detection errors, we implement the following mechanisms:

Error Avoidance During Detection

We lower the risk of miss detecting deltas and offsets by following these rules while implementing our detection algorithms:

1. If we detect multiple candidates and we cannot validate that exactly one of them is valid, we print an error and interrupt the program.

   This implies we do not directly accept a delta when detected as valid by our algorithms, but instead we have to ensure that there is no other valid delta in the range, we are investigating.

2. Static deltas must be validated, if possible, as they base on assumptions and we cannot guarantee that these assumptions are valid on every kernel.

3. Detection algorithms, requiring multiple entities of a structure, have to ensure that the amount of available structures is sufficient for a correct detection; if requirements are not met they must raise an error.

4. The algorithms should issue a warning and exit the program on unusual values. An example for such a value would be that the size of the `page` structure is not a multiple of the size of an Integer.

In practice it is not possible to abide by these rules at all times.

An example:
Multiple instances of the same structure, whose exact size we do not know, reside next to each other in memory. While trying to detect a delta of this structure using instance I_1 we also search inside the memory region of I_1's neighbor I_2. This may result in detecting two deltas pointing to completely valid entries, one in the memory of I_1 and one in the memory of I_2.
If we cannot distinguish which entry belongs to which instance of the structure, we cannot follow rule one, but instead we must accept one of both deltas (e.g., the smallest positive one).

Manual Error Avoidance

We can output the detected values to allow users to manually validate them using the kernel configuration and sources if available.

Error Avoidance Using a Configuration File

Our tools can generate a configuration file which contains all detected deltas and some offsets based on multiple memory dumps of the same target. This generation reduces the probability of a detection error by ensuring all detected values are equal on all memory dumps or it signals an error.
This allows us, besides speeding up our tools (as the configuration file substitutes the detection), to use *one manually validated* configuration for all memory dumps of this target.

4.4. Output Stage

The output stage consists of one or multiple exchangeable output modules. In this theses we focus on the module for page duplication detection.

4.4.1. Details on Pages / What We Can Measure

We can acquire the following information for every userland page, for free and kernel pages the entities marked with an asterisk cannot be acquired because of their peculiarities[2]:

origin We assign every physical page to exactly one of the following classes:

- **userland** Every page mapped into any address space not associated to kernel.
- **free pages** Every page whose reference counter (`_count`) is zero. (This includes all pages in the free lists maintained by the kernel.)
  Warning: Not all currently unused pages are detected as free as not every unused page is instantly returned to the kernel and marked as free.

[2] As the pages belonging to the kernel and the pages which are free do not generally have a `vm_area_struct`, we cannot extract information from this structure.

kernel Rest of the pages.

hash The hash of the page content.

content The content of the page.

access-bits* The access-bits of the *memory regions* the page belongs to.

memory mapped I/O state* Whether the page is part of a memory mapped I/O region. This information bases on the `VM_IO` bit in the member `vm_flags` of the dedicated `vm_area_struct` structure. [16]

content source* Anonymous vs. named pages: We consider a page to be named if `vm_file` in its `vm_area_region` is not null.

attached file* The filename of a file attached to a named page.

offset inside attached file* The offset inside this file.

process ids Ids of all user processes which own a mapping for a physical page.

4.4.2. Detecting Duplicates

We use hashes of the physical pages as keys in a hash-map (§ 3.2), this results in the structure shown in Figure 4.1

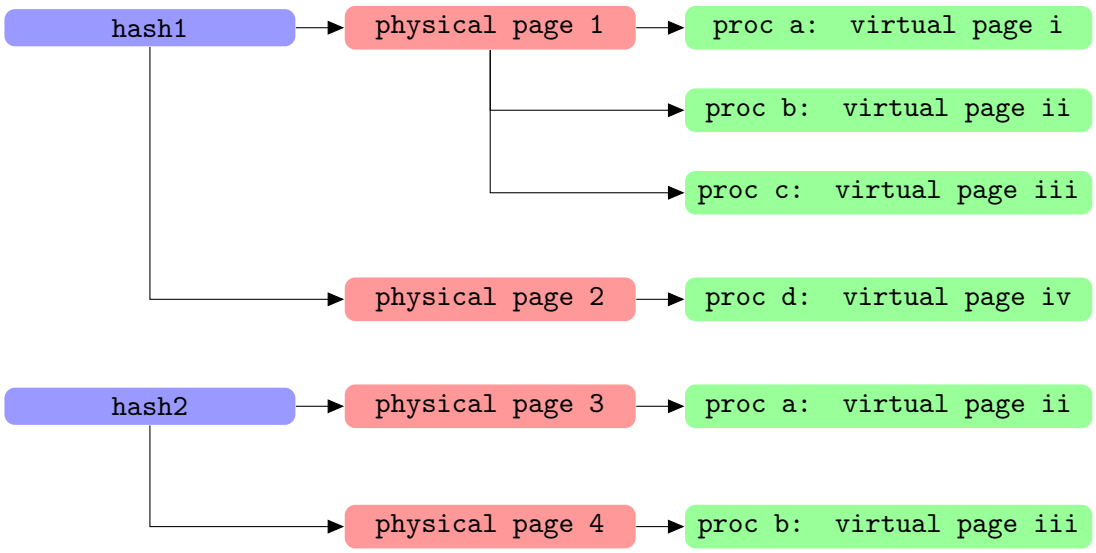

Figure 4.1.: Structure We Use for Duplicate Detection

Every hash pointing to more than one physical page is affected by page duplication; the origin of this pages can be traced because the physical pages point to address spaces they are mapped in.

Physical pages which are assigned to multiple virtual pages can occur because of page deduplication (e.g., copy-on-write or shared memory).

Active Check for Hash Collisions

Our tools can guarantee the absence of hash collisions inside a dump by comparing the content of all physical pages attached to the same hash (Figure 4.1). We do not automatically apply this check to all hashes, but one can manually initiate it.
We automatically execute this check for every hash, affected by page duplication, the user requests details on, as the local effect of a collision can be huge. (There might be no duplication for this hash at all.)

4.4.3. Limitations

We currently assume a single page size. If there is a superpage in the systems we currently handle it like multiple normal pages.
If we find a duplicate which is part of a superpage, it would be necessary to split the superpage up into normal pages to be able to deduplicate the page. This should normally not be done as this would lead to a lot of disadvantages (e.g., more entries in the TLB).

5. Evaluation

We ignore all pages filled with zeros in our measurements, as these pages might be free or unused but intentionally not reassigned to the kernel's free pool (e.g., to avoid overhead of releasing and reclaiming). Merging these pages would contradict this concept.
As merging unused pages does not provide additional storage we also ignore them in our evaluation (unless stated otherwise).
Our tools are able to detect userland pages which are used for memory mapped I/O (§ 4.4.1). Merging these pages is problematic and we therefore exclude these pages, too.

In our diagrams error bars do not represent the standard deviation but instead indicate the minimum and maximum values measured. All measurements are repeated five times and the average values are plotted.
Refer to Table 5.1 for a list of abbreviations used in our diagrams:

ppages	*physical pages:* Physical pages which are present in the system/group.[i]
vpages	*virtual pages:* Virtual pages which are present in the system/group.[i]
pdups	*physical duplicates:* Duplicated Pages; i.e, all pages in this set could be freed without loosing the page *content*.[i,ii]
vdups	*virtual duplicates:* Virtual pages which can be removed so that every previously mapped physical page is still mapped to by exactly one virtual page. Ignoring intentionally shared pages, we can use this as a rough estimation of already applied deduplication.[i]

[i] As mentioned before, we exclude multiple pages (e.g., pages filled with zeros). These pages are not regarded in *any* measured value (including this one).
[ii] Refer to § 2.1 for a more detailed definition of *duplicated pages*.

Table 5.1.: Abbreviations in Diagrams

Our measurements are run on an emulated Android device using the emulator from the Android SDK[4]. Table 5.2 describes the properties of the test system:

Attribute	Value
Android Version	2.3.1
CPU	ARM926EJ-S rev 5 (v5l)
RAM size	256 MB

Table 5.2.: Properties of Our Virtual Android Test Devices

We refer to the following scenarios in our measurements:

fresh We directly dump the memory after the OS is started; we do not manually execute any programs.

in use Like idle, but we start the programs named in Table 5.3 one after another with a delay of 10 seconds before dumping the memory.

Program	Version
AndFTP	2.9.8
AndSMB	1.8
AndExplorer	2.2
AdobeReader	10.1.1
ConnectBot	git-master-2011-12-19_18-30-54
K-9	4.001-release
Native Browser	2.3.1

Table 5.3.: Programs Started in Scenario *In Use*

5.1. Total Sharing Opportunities

Figure 5.1 depicts the overall sharing opportunities. In the *fresh* scenario approx. 7.8% of the non zeroed and non free physical pages are duplicates, in the *in use* scenario approx. 9.5%. Userland I/O pages are also excluded.

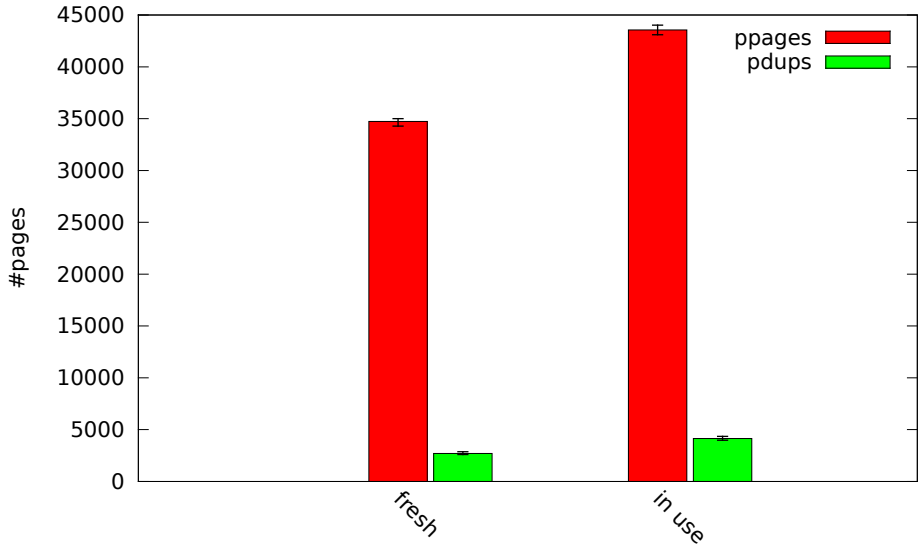

Figure 5.1.: Overview Scenarios, free pages excluded

The resulting saving opportunity is 10.6 MB in *idle* and 16.2 MB in *in use*.
As the *in use* scenario is the more common scenario we focus on this one.

5.2. Classification of Sharing Opportunities

We divide the system into the groups listed in Table 5.4, to isolate the sources for page duplication.
Inside these groups pages are only considered as duplicates if there is a page with equal content *within the same group*. The distribution of duplicates among the named groups can be found in Figure 5.2.

Group	Description
All	All pages (excluding zeroed pages)
Kernel	Pages containing Kernel's code and data.[i]
Dalvík VM	Pages assigned to a program whose parent is `Zygote` and the pages from `Zygote` itself
Remainder	Pages not included in *Kernel*, *Dalvík* VM or *Free*
Free	Pages which are marked as free.[i]

[i] Refer to § 4.4.1 for a more detailed definition.

Table 5.4.: Description of Groups for Classification

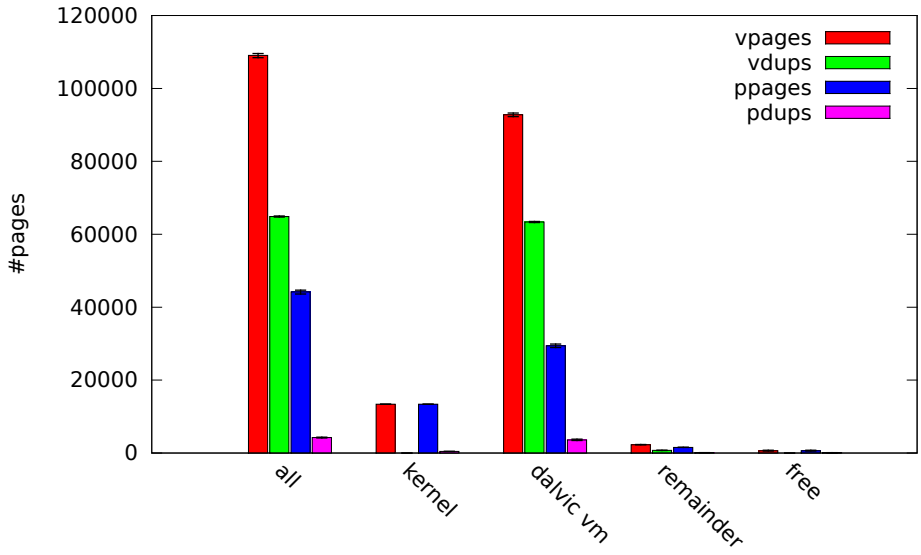

Figure 5.2.: Duplicates and Pages Per Group

As stated in Table 5.5 (containing details on Figure 5.2), this grouping results in only

ignoring a very small number of duplicated pages; we can therefore look at each group independently.

Description	Value
Physical duplicates in *All*	4250
Sum of all physical duplicates in the remaining groups[i]	4187.6

[i] Our measurement shows that no physical page affected by duplication is mapped into multiple of our groups, therefore it cannot happen that we count pages affected by duplication multiple times. As a result it is valid to add up the physical duplicates.

Table 5.5.: Comparison of Detected Duplicates With And Without Grouping in Figure 5.2

The amount of duplicates in groups *Kernel* and *Remainder* is small. We examine these groups in § 5.2.1 and § 5.2.2.

Most duplicates arise from the *Dalvík* VM group which we cover in § 5.2.3.

In Figure 5.2 we also plot the amount of virtual pages and virtual duplicates as these values hint the amount of already prevented duplicates. The figure states that the RAM consumption is more than halved by the already applied page deduplication techniques.

This is only a rough estimation as some of the virtual duplicates refer to shared memory and it is questionable whether this memory should be considered as already deduplicated memory or as memory which must be deduplicated by design.

5.2.1. Group: Remainder

The remainder group demands only a small amount of RAM (here: 1552.4 pages) as most processes are lightweight or do not even have an userland address space at all (e.g., swapper or kthread).

The small memory footprint of the single tasks result in a very small amount of shareable pages (here: 95.6). Most of the duplicates are named pages and often originate from libraries like `libc.so`.

For a list of processes in this group refer to Figure 5.3.

swapper	kseriod	rpciod/0	installd
init	kmmcd	ueventd	keystore
kthreadd	pdflush	mmcqd	qemud
ksoftirqd/0	pdflush	servicemanager	sh
events/0	kswapd0	vold	adbd
khelper	aio/0	netd	sh
suspend	mtdblockd	debuggerd	
kblockd/0	kstriped	rild	
cqueue	hid_compat	mediaserver	

Figure 5.3.: Processes Inside the *Remainder* Group

5.2.2. Group: Kernel

The group *Kernel* covers 13390.2 physical pages of which 441.2 ones are duplicates.
As the kernel does not own the data structures describing the address space of a process, we cannot use most capabilities of our tools to gain information[1] on the sources of duplication.
Only one hash of the about 60 hashes representing the page content involved in the kernel's duplicates does occur in another group than *Kernel*. This hash represents pages with all bits set to 1. These pages are likely to be unsuitable for deduplication, see next subsection (§ 5.2.3) for details.

5.2.3. Group: Dalvík VM

With 3591.8 duplicates of 29415.6 physical pages the *Dalvík* VM group is the most important source of page duplication on our android device.

An important detail of the detected duplicates is that a single page content (a page with all bits set to 1) is the source for about 397 duplicates on average (max. 484). A lot of this pages are part of a `gralloc-buffer`[3] which contains memory for drawing accelerated graphics.
It is at least questionable if these pages can be and should be deduplicated.

Details on Duplicates

We partitioned the physical duplicates into anonymous/named pages and write allowed/not allowed, the results are shown in Table 5.6.
Please mind the peculiarity of Android concerning anonymous shared memory (see § 2.4.1 for details).

[1] E.g., assigned filename, access rights, involved processes, ...

anonymous pages:	∼ 52.7%	write forbidden	∼ 0.6%
		write allowed	∼ 52.1%
named pages:	∼ 47.3%	write forbidden	∼ 0.4%
		write allowed	∼ 46.9%

Table 5.6.: Details on Physical Duplicates in Dalvík VM

About 53% of the duplicates are anonymous and are therefore theoretically mergeable by KSM. As almost all duplicated pages are write-able we cannot make any statement on the chance that these pages stay unmodified for a reasonable amount of time (important for KSM to be work with an acceptable overhead).

The named duplicates are heavily affected by the peculiarity of Android concerning anonymous shared memory using `ashmem` (see § 2.4.1 for details); our further measurements show that about 95% of Dalvík VM's named pages affected by duplication are attached to a file starting with `dev/ashmem`.
As KSM currently cannot handle named pages, Dalvík VM must be modified to merge this duplicates. Whether it is possible to replace `ashmem` with a more KSM friendly approach is a target for further research.

5.3. Conclusion

The only group whose duplicates have a significant impact on the memory footprint is *Dalvík* VM.
There are two approaches for deduplication which might reduce the page duplication, but both need additional research to be done:

Modification of Dalvík VM To be able to name the reasons for duplication in Dalvík VM we need more information on the source of the duplicated pages inside the VM; to achieve this we could develop a module for our tools which reveals the internals of the Dalvík VMs running inside the target.

Usage of KSM We miss information on how often the duplicated pages change to known if running KSM is reasonable.

As mentioned in § 4.4.3 our tools assume a single page size and handle superpages like multiple standard-sized pages; merging duplicates located on superpages would possibly imply to split the superpage into normal pages – we consider this to be unwanted.
We modified our tools to name address spaces using superpages (referred to as Sections/Supersections on ARM), to estimate the effect on our measurements. As the kernel's address space was the only one containing superpages, the main source of page duplication (Dalvík VM) is not affected.

6. Conclusion

Mobile devices are mostly not equipped with huge amounts of RAM. One reason for this is the negative impact on the power consumption and the size of the device.
On Android devices as many applications as possible should fit into RAM, to allow fast application switching. Our evaluation on duplicated pages proved that there still exist memory saving potential, even though Androids OS already uses multiple passive deduplication techniques.

Our measurement yields that 9.5% of all non-free pages are duplicates in a common usage scenario. (Zeroed pages and userland pages, which are used for memory mapped I/O, are not included in the measurement.)
We found Dalvík VM to be the main source of duplication. Most duplicates originating from Dalvík VM are anonymous pages or anonymous memory represented as named pages in the kernel because of the `ashmem` (§ 5.2.3).
There are two approaches for deduplication which can be taken into account:
The first one is to modify Dalvík VM to further reduce the amount of duplicates. As Dalvík VM already uses multiple approaches to prevent the arising duplicates this is likely to be difficult.
The second one is a KSM like approach. Running KSM on a mobile device is critical and it would be absolutely necessary to reduce the computation time of KSM to a minimum – running KSM only on application switch and programmatically classifying the main sources of duplication and hinting only this pages to KSM might help reaching a low CPU usage.
Whether it is possible to merge the duplicated pages with a reasonable overhead (implementation time, CPU-time, battery runtime) is a target for further research.

The measurement tools we developed for this theses are able to perform an offline analysis of a memory dump from an ARM based Android device. Supporting other Linux based systems on other architectures requires only minor changes to the code.[1]
Our tools allow access to basic information on the address space layout, page properties and statistics on duplicates. In particulars we are able to exclude pages which are marked as free inside the kernel.
We currently assume a single page size in our implementation. Although it has no huge impact on our measurements (§ 5.3), this is a drawback. To remedy this the tools must be modified to record the pages' origin (normal page or superpage) and use this information in the statistics.

[1] This systems must still meet the requirements concerning the memory layout we name in § 4.2.

A. Appendix

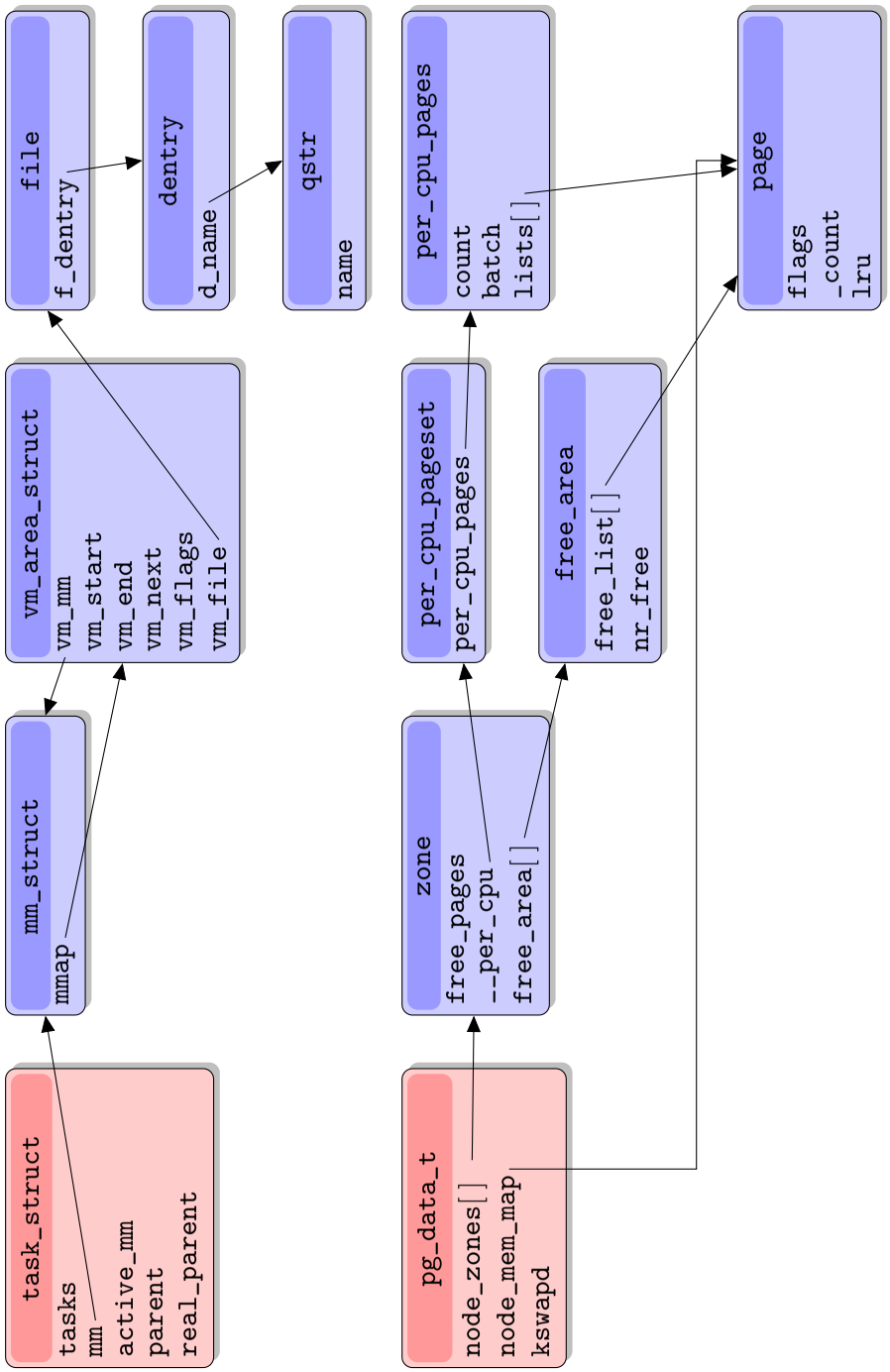

Figure A.1.: Kernel Structures Used In Our Tools (Based on Information from [2, 22, 23, 10])
Not all members of the structures are named

B. Glossary

COW copy-on-write. 3

KSM Kernel Shared Memory also known as Kernel Same Page Merging (see § 2.3). 3, 23, 24

MD5 Message-Digest Algorithm 5. 8

OS operating system. 1–4, 7, 19, 24

PFN A Physical Frame Number (PFN) is used to identify a physical page/frame. In this document we assume that the frame referred to by the PFN 0 covers addresses 0 to (page size −1). 7, 8

RAM random access memory. 1, 4, 6–8, 18, 21, 24

UMA uniform memory access. 11

VM virtual machine. 4, 23

Dalvík VM Dalvík Virtual Machine. iv, v, 4, 20–24

delta In this document: Distance from the start of a structure to an entry inside this structure. 2, 3, 10–15

offset In this document: Address in the *target system*. Refers to physical addresses, unless stated otherwise. (We introduce this term to avoid confusion with addresses on host system). 2, 10–15

Bibliography

[1] Android Developers: Security and permissions, . URL `http://developer.android.com/guide/topics/security/security.html`. last access: 11. March 2012.

[2] Android Kernel Headers, . URL `https://android.googlesource.com/platform/external/kernel-headers`. git commit 4879d6a8... from 7. December 2010.

[3] Android Libhardware, . URL `https://android.googlesource.com/platform/hardware/libhardware`. git commit f1d76bb7... from 28. November 2011.

[4] Android SDK Version r16, . URL `http://developer.android.com/sdk/index.html`. last access: 25. January 2012.

[5] David Anderson. White Paper: Red Hat Crash Utility, . URL `http://people.redhat.com/anderson/crash_whitepaper/`. last access: 24. January 2012.

[6] David Anderson. Red Hat Crash Utility: Sourcecode 6.0.2, . URL `http://people.redhat.com/anderson/`. last access: 22. December 2011.

[7] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. URL `http://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf`. last access: 12. January 2012.

[8] *ARM1176JZ-S Technical Reference Manual*. ARM Limited, 2009. Revision: r0p7.

[9] Dan Bornstein. Google I/O: Dalvik-VM-Internals, 2008. URL `http://sites.google.com/site/io/dalvik-vm-internals`. last access: 12. January 2012.

[10] D.P. Bovet and M. Cesatí. *Understanding the Linux Kernel*. O'Reilly Series. O'Reilly, 3rd edition, 2006. ISBN 9780596005658.

[11] Stefan Brähler. Analysis of the Android Architecture, October 6 2010. URL `http://os.ibds.kit.edu/`.

[12] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997. ISSN 0734-2071. doi: 10.1145/265924.265930. URL `http://doi.acm.org/10.1145/265924.265930`.

[13] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of 2010 USENIX Annual Technical Conference*, USENIX,

pages 271–284. USENIX Association, 2010. ISBN 978-931971-75-1. URL http://static.usenix.org/events/atc10/tech/.

[14] Andrew Case, Lodovico Marziale, and Golden G. Richard III. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7, Supplement(0): 32–40, 2010. ISSN 1742-2876. URL http://www.sciencedirect.com/science/article/pii/S1742287610000320. last access: 18. March 2012.

[15] Ben Cheng and Bill Buzbee. Google I/O: A JIT compiler for Android's Dalvik VM, 2010. URL http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html. last access: 12. January 2012.

[16] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, 2005. ISBN 978-0-596-00590-0.

[17] Emilien Girault. Volatilitux sourcecode 1.0. URL http://code.google.com/p/volatilitux/. last access: 15. January 2012.

[18] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004. ISBN 0-13-145348-3.

[19] Thorsten Gröninger. Analyzing Shared Memory Opportunities in Different Workloads, November 21 2011. URL http://os.ibds.kit.edu/.

[20] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. URL http://usenix.org/event/sec08/tech/full_papers/halderman/halderman.pdf. last access: 22. March 2012.

[21] Heise Zeitschriften Verlag GmbH & Co. KG. VMware virtualisiert Android-Smartphones. URL http://www.heise.de/mobil/meldung/VMware-virtualisiert-Android-Smartphones-1191196.html. last access: 11. March 2012.

[22] Wolfgang Mauerer. *Linux Kernelarchitektur : Konzepte, Strukturen und Algorithmen von Kernel 2.6*. Hanser, München, 2004. ISBN 3-446-22566-8.

[23] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Programmer to Programmer. Wiley, Indianapolis, Ind., 2008. ISBN 978-0-470-34343-2.

[24] Grzegorz Miłoś, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of 2009 USENIX Annual Technical Conference*, USENIX, pages 1–14. USENIX Association, 2009. ISBN 978-1-931971-68-3. URL http://static.usenix.org/event/usenix09/tech/.

[25] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday Paradox for Multi-collisions. In Min Rhee and Byoungcheon Lee, editors, *Information Security and Cryptology – ICISC 2006*, volume 4296 of *Lecture Notes in Computer Science*, pages 29–40. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-49112-5.

[26] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844146. URL `http://doi.acm.org/10.1145/844128.844146`.

[27] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-25910-7.