Android Kernel Builder

Student Research Paper

submitted: February 2012
by: Tobias Brentrop

Advisor: Michael Spreitzenbarth

Department of Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg
D – 91058 Erlangen
Internet: `http://www1.informatik.uni-erlangen.de`

Contents

List of Figures

Listings

1. Introduction

Computer forensics focus on gathering as much evidence as possible from any kind of memory. Usually that includes internal or external hard-disks, RAM, USB sticks or optical mediums like DVDs. While those are obvious targets for law enforcement agencies, memory cards with huge capacities became so small that they fit in almost every appliance, even pocket knives (1). Of course it doesn't make sense for every device to hold personal and sensitive data, but in recent years there has been a trend resulting in the spread of a device that is rich on traces of the owners activities: Smartphones.

2011 of all sold cell phones about 30% were smartphones and it's estimated that they reach over 50% in 2015 and Android, the open-source operating system developed by Google, currently has a 50% market share (2, 3). On older phones the only relevant data stored was the contact list and SMS messages. Smartphones, however, are basically as powerful as desktop computers and are used as such. This leads to many different sources of traces (e.g. web-browsing, social networks, email and more). Another benefit is that disk encryption, specifically pre-boot encryption, isn't as widespread as on computers and that Android stores a lot of data unencrypted (4).

Generally in forensics, when you get your hands on any kind of storage device, the first step is to make a complete image of the system. When you work exclusively on a copy of the image the integrity of the data can be maintained, even when data is accidentally overwritten or deleted.

For this paper we wrote a python script that modifies an Android kernel to include tools that allow basic forensic operations like 'md5sum' and system image creation. In Chapter 2 we describe the general layouts of Android phones, kernels and our Python script. In Chapter 3 we introduce you to the tools the script is able to add to a stock kernel. After we present our conclusions in Chapter 4 we provide a step-by-step tutorial in the appendix.

2. Kernel Builder

We give an overview of the basic layout of the Samsung Galaxy S2 in Section 2.1. We will also take a quick look at the kernel image in Section 2.2 and it's *Initial Ram File-system* (InitRamFS) in Section 2.3. Then, in Section 2.4, we show a few details of our Python script.

2.1. Samsung Galaxy S2 Partition Layout

The Samsung Galaxy S2 has 13 different partitions on it's internal memory, including the internal SD-card. The information about size and location of these partitions is stored in the *Partition Information Table* (PIT), depicted in Figure 2.1, and is needed when you flash an image to the phone. The size of the individual partitions may vary through different devices, but the general layout stays the same. The *PIT* is easily obtainable with the software used for flashing (see Figure A.1). We concentrate in the following on the kernel partition.

2.2. Kernel Image

The kernel image consists of two major parts. One is the kernel itself. The kernel can be compiled from the publicly available source (5), but we use already compiled stock kernels, which you can either download from the web (6), extract yourself (Appendix A.1) or take from the few provided with the enclosed CD (Appendix B). The configuration Samsung includes with the source uses different flags (mostly debug related) compared to the production build, but while this may affect the phone's performance the functionality of the kernel is the same (7).

The second part is the *Initial RAM File-system*. It's a basic file-system that is loaded into the RAM by the boot-loader and includes the critical 'init' process as well as 'init.rc' and some manufacturer dependent drivers (e.g. Wi-Fi, Vibrator)

Partition Name	LEN in BLK (512 B)	OS Partition	Physical Partition
GANG	0	-	0
BOOT	0	-	1
EFS	4,096	0p1	4
SBL1	2,560	0p2	2
SBL2	2,560	0p3	3
PARAM	16,384	0p4	5
KERNEL	16,384	0p5	6
RECOVERY	16,384	0p6	7
CACHE	204,800	0p7	8
MODEM	32,768	0p8	9
FACTORYFS	1,048,576	0p9	a
DATAFS	4,194,304	0p10	b
UMS	24,133,632	0p11	c
HIDDEN	1,048,576	0p12	d

Figure 2.1.: Layout: Samsung Galaxy S2 Partitions

that are not compiled into the kernel (8). The *InitRamFS* is the only interesting part for us, because all our modifications are made there.

2.3. Initial Ram File-System

Figure 2.2 shows the general directory structure of the *InitRamFS*. As you can see, even though the Galaxy S2 has a recovery partition, Samsung decided to include the recovery on the kernel partition. This has to be considered, if we want switch the stock recovery for a custom one later on. To add new modules we need to include the designated binaries and a small shell script that handles the installation process. To avoid any confusion we put all our files in the '/res/misc/' directory. We edit the 'init.rc' file, which provides the generic initialization instructions, to execute the shell-script at boot-time. The *InitRamFS* is extracted from the kernel image every time the device boots, so any modifications to it will be reverted once the phone is turned off. We copy the files to the '/system' partition, so the tools are still available after flashing a different kernel again.

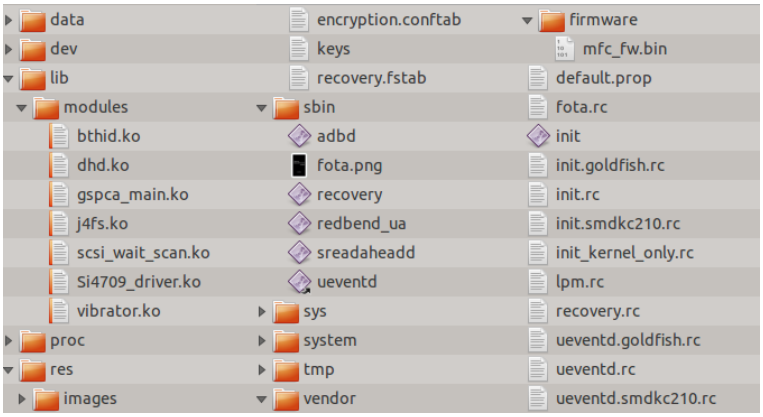

Figure 2.2.: Layout: Initial Ram File-System

2.4. Kernel Builder Python Script

The purpose of our script is to automatically add the desired modules and generate the shell installer script accordingly. The basic layout is depicted in Figure 2.3. The main script is only responsible for generating the command-line help and calling the methods that are adding the respective module. These methods are implemented in a device specific file. Although Android devices are similar, as they are all based on Linux, we already mentioned the recovery as one of the distinctions.

We implemented a helper method that makes the command line interface simpler to use. It returns the newest directory for a binary, if no explicit path is given (Listing 2.1). First it checks, if the path given is valid and returns it or, if not, looks for the newest path in the respective directory (e.g. 'binaries/Superuser'). It then creates a dictionary with directories and their timestamps and returns the maximum through a lambda (line 11-15).

Listing 2.1: Source: settings.py, getPath

```
1 def getPath(module, thisfile):
2   if not thisfile:
3     return False
4   if os.path.isdir(str(thisfile)):
5     log(thisfile +' seems valid')
6     return thisfile
7   else:
```

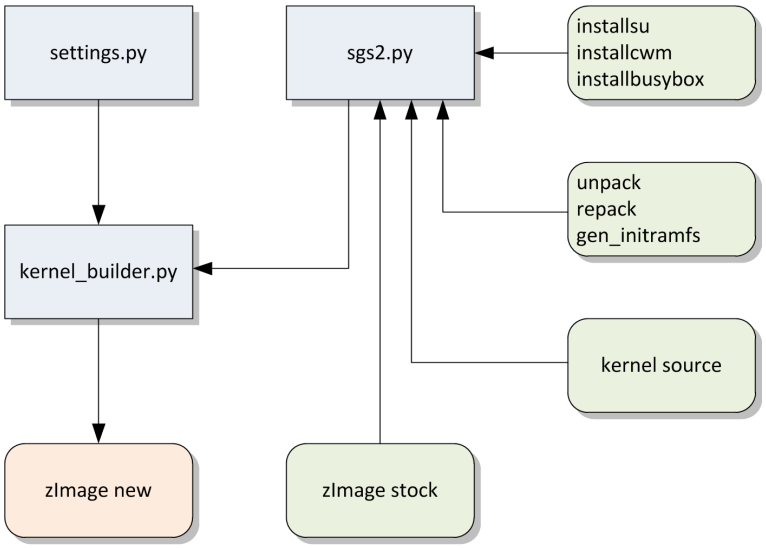

Figure 2.3.: Layout: Kernel Builder

```
8       log('no (valid) path: \''+str( thisfile )+'\' for '+module+'. using latest binary')
9     if not os.path.isdir(BINPATH+module):
10      exit('path to module \''+module+'\' does not exist. this should not happen')
11    files = os.listdir(BINPATH+module+'/')
12    dirlist = dict([(x, os.stat(BINPATH+module+'/'+x).st_mtime) for x in files])
13    log(BINPATH+module+'/'+max(dirlist, key=lambda k: dirlist.get(k)))
14    return BINPATH+module+'/'+max(dirlist, key=lambda k: dirlist.get(k))
```

When the actual script is started it first prepares the *InitRamFS* for our modifications. In Listing 2.2 you can see it first checks for the cross-compiler and extracts the *InitRamFS* from the kernel image (line 2-7). The tool **unpack** is freely available on the web and also comes with a kernel repacker (9). Then we include our own service, the shell script, in 'init.rc' (line 15-19) and copy the unfinished 'kernelpatch.sh' (line 25). The process for the individual modules is covered later in Chapter 3.

Listing 2.2: Source: sgs2.py, prepare

```
1  def prepare(kernel):
2    if not os.path.isdir(s.CROSSCPATH):
3      exit('cross compiler binary directory does not exist: '+s.CROSSCPATH)
4    if not os.path.isdir(s.CROSSLPATH):
5      exit('cross compiler lib directory does not exist: '+s.CROSSLPATH)
6    s.lsh('mkdir -p '+s.INITDIR)
```

```
7   s.lsh(DEVPATH+'unpack '+s.getPath('kernel', kernel)+'/zImage '+s.INITDIR)
8   s.log('add kernelpatch.sh to init.rc')
9   myfile = s.INITDIR+'init.rc'
10  shutil.move(myfile, myfile+"~")
11  destination = open(myfile, "w")
12  source = open(myfile+"~", "r")
13  for line in source:
14    if 'class_start' in line:
15      destination.write('    # kernel patch\n    start kernelpatch\n\n')
16      destination.write(line)
17      destination.write('\n\n# kernel patch\n'+
18                        'service kernelpatch /res/misc/kernelpatch.sh\n'+
19                        '    user root\n    oneshot\n    disabled\n\n')
20      continue
21    destination.write(line)
22  source.close()
23  destination.close()
24  s.lsh('mkdir -p '+s.INITDIR+'res/misc/')
25  s.lsh('cp -f '+DEVPATH+'kernelpatch.sh '+s.INITDIR+'res/misc/kernelpatch.sh')
```

After everything is added the method *finalize* in Listing 2.3 finishes the process. It adds the final line to 'kernelpatch.sh' and then executes the kernel repacker. The tool **repack** generates a CPIO file from the *InitRamFS* and runs the final steps from a kernel compilation. This is the reason the kernel source and cross-compiler are needed at all. The individual parts of the kernel image need to be linked properly, so that the phone will boot when the kernel is flashed.

Listing 2.3: Source: sgs2.py, finalize, repack

```
1  def finalize(path):
2    s.log('finalize kernelpatch.sh')
3    dest = open(s.INITDIR+'res/misc/kernelpatch.sh', 'a')
4    dest.write('  toolbox mount -o remount,ro /system /system')
5    dest.close()
6    repackKernel(path)
7    s.lsh('mkdir -p '+s.OUTPATH)
8    s.lsh('cp -f '+s.TEMPDIR+'zImage '+s.OUTPATH)
9
10 def repackKernel(path):
11   s.lsh(DEVPATH+'gen_initramfs.sh -o '+s.TEMPDIR+
12         'initram.cpio -u 0 -g 0 '+s.INITDIR)
13   if __debug__:
```

```
14      result = s.lsh(DEVPATH+'repack '+str(path)+'/zImage '+
15                     s.TEMPDIR+'initram.cpio'+' '+s.CROSSCPATH+' '+
16                     s.CROSSLPATH+' true')
17    else:
18      result = s.lsh(DEVPATH+'repack '+str(path)+'/zImage '+
19                     s.TEMPDIR+'initram.cpio'+' '+s.CROSSCPATH+' '+
20                     s.CROSSLPATH)
21    s.log('repack exit code: '+str(result))
22    if result != 0:
23      exit('try to use different flags (e.g. don\'t install superuser.apk')
```

3. Modules

There are different tools that help to perform common forensic operations. This chapter gives an overview on Android Debug Bridge in Section 3.1, su in Section 3.2, Superuser in Section 3.3, ClockworkMod in Section 3.4 and finally BusyBox in Section 3.5.

3.1. Android Debug Bridge

Android Debug Bridge (ADB) is a tool that comes with the Android Software Development Kit (SDK) (10) and allows to interact with an Android device. It provides various commands to interact with the phone, including push, pull, logcat and shell. Push/pull allows you to move files to/from your phone's internal storage as well as the SD card without mounting it as a mass storage device. Logcat provides a way to look at real-time Android debug messages.

The most useful part of ADB is the shell. It's a stripped down version of a Linux shell and provides only basic commands. Some commands which are important for forensics, e.g. 'md5sum' to calculate hashes directly on the device, are missing, but can be added through BusyBox (Section 3.5). For security reasons the shell won't run as root if the phone is on a stock ROM. Without root the shell is only on user privilege level and thus some commands that modify the system are prohibited from being executed.

To gain root access there are different approaches. Temporary root can be achieved with an exploit. **ZergRush** for example works on various phones that are running on Android versions prior to *Honeycomb* (Android 3.0) (11). After pushing it on the phone and executing the binary the shell has root privileges until the phone is rebooted.

For a permanent solution the kernel image needs to be modified. The *InitRamFS* contains a file called 'default.prop'. It holds only a few lines of code, which you can see in Figure 3.1.

Listing 3.1: Source: default.prop

```
1 ro.secure=0
2 ro.allow.mock.location=0
3 ro.debuggable=1
4 persist.service.adb.enable=1
```

The first line determines if the kernel runs in secure mode. Permanent root is only possible, if the kernel is insecure. Additionally it's necessary that the ADB daemon is running (line 4). Our script only needs to edit those two lines.

Listing 3.2: Source: sgs2.py, addADB

```
1 def addADB(add):
2   for line in fileinput.FileInput(s.INITDIR+'default.prop', inplace=1):
3     line = line.replace('ro.secure=1', 'ro.secure=0')
4     line = line.replace('persist.service.adb.enable=0',
5                         'persist.service.adb.enable=1')
6     sys.stdout.write(line)
```

You can test for root, if you connect to the shell. It should display # rather than $. Also the command 'adb remount' should now work. It remounts the '/system' partition read-write instead of read-only.

3.2. SU

The standard Linux binary 'su' is not included within stock Android ROMs (12). In an Android environment every process is only allowed to access it's own application directory. With this Android tries to protect the user from malware collecting private data from other applications. The root-user has read-write access to everything (provided that the partition is not mounted read-only), so when an application like **ROOT-Explorer** (a file manager) (13) wants to provide access to protected directories it calls 'su', which in turn switches the context of the process to root. Now it's possible to browse and modify other applications files.

The binary is small enough to be included in the *InitRamFS* without any problems. We put 'su' in '/res/misc' and add the installation instructions to 'kernel-patch.sh'.

Listing 3.3: Source: installsu

```
1 mkdir /system/xbin
2 chmod 755 /system/xbin
3 rm /system/bin/su
4 rm /system/xbin/su
5 cat /res/misc/su > /system/xbin/su
6 chown 0.0 /system/xbin/su
7 chmod 6755 /system/xbin/su
```

In Listing 3.3 you can see that first it's ensured '/system/xbin' exists, is set to the proper access rights and possible older versions of 'su' are deleted (line 2-5). Since the command 'cp' is not included with Android we use 'cat' instead and direct the output to 'su'. Afterwards the owner of the file is changed to root and the file is also set executable. 'su' needs the Superuser application to work properly with Android applications. It would be possible to keep 'su' in the '/sbin' directory, but '/sbin' is part of the kernel image and only mounted as RAM-disk, so if you flash a different kernel you would lose the binary.

3.3. Superuser

While 'su' is a Linux binary, there is no built-in way to see what application requests root or even deny those requests. Superuser is the corresponding Android application (Figure 3.1)(12). It prompts the user every time a process wants to have root access and also provides a way to block specific applications. It maintains a database to remember the user's decisions, logs all requests and additionally gives the option to update the 'su' binary.

Including Superuser in the installer script is similar to 'su' and shown in Listing 3.4. First old versions of Superuser are removed (line 3-4). However, Superuser is an Android application so it has to be put in the '/system/app' directory (line 4). Android applications (.apk) in this directory are automatically installed and displayed in the application drawer. Then again the script sets the proper rights for the file (line 5-6).

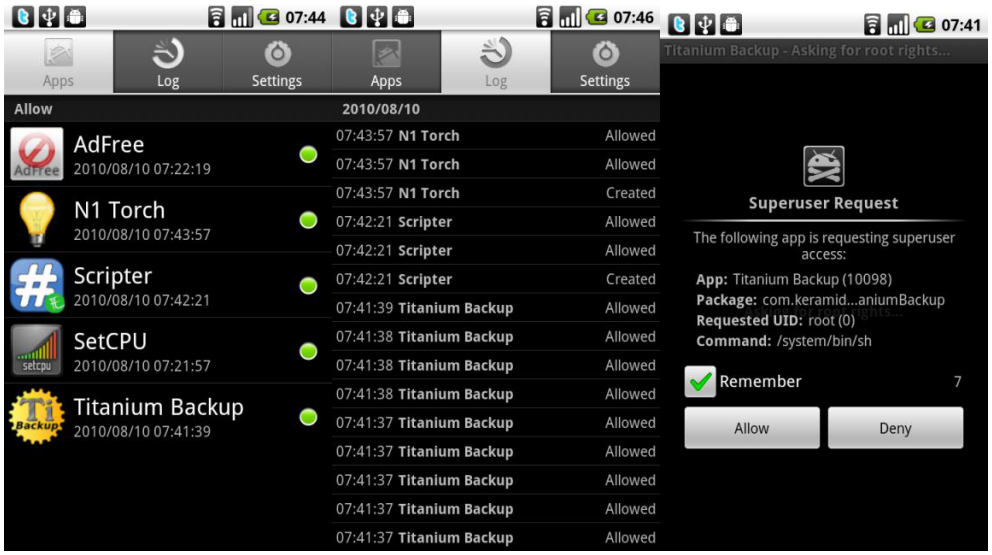

Figure 3.1.: App: Superuser

Listing 3.4: Source: installsuperuser

```
1 rm /system/app/Superuser.apk
2 rm /data/app/Superuser.apk
3 cat /res/misc/Superuser.apk > /system/app/Superuser.apk
4 chown 0.0 /system/app/Superuser.apk
5 chmod 644 /system/app/Superuser.apk
```

3.4. ClockworkMod

ClockworkMod (CWM)is an improved recovery available for many Android phones (14). It's being developed by Koushik Dutta and provides more functionality than the stock Samsung recovery. The main improvements over stock recovery are the ability to install unsigned '.zip' files, which prevents you from installing custom ROMs on stock recovery, and the option to make a full system image via **NANDroid**-backup. **NANDroid** will backup (and restore) '/system', '/data', '/cache', and '/boot' partitions. Usually this is used to make ROM testing easier, since you can just restore your image, if something is not working correctly. Obviously you can also use these images to examine them for any traces without using the actual phone.

Because ClockworkMod is replacing the stock recovery we don't need to add much to our installer script, as you can see in Listing 3.5. The script only needs to overwrite the stock 'recovery.fstab', which holds information about how partitions are mounted, with the one from CWM. Our Python script then extracts the files from 'cwm.zip' to the *InitRamFS* (Listing 3.6). We decided to use a '.zip' in this case, because CWM consists of a lot of files and it makes handling them simpler. Also, since the recovery is on the kernel partition and all files from ClockworkMod depend on it, there is no need to copy them over to '/system'.

Listing 3.5: Source: installcwm

```
1  rm /etc/recovery.fstab
2  cat /res/misc/recovery.fstab > /etc/recovery.fstab
```

ClockworkMod includes a stripped-down version of BusyBox as well, so if you add CWM you will rarely need to add BusyBox. The appropriate symlinks are already included in 'cwm.zip'. If you include both, CWM and BusyBox, be sure to invoke the right version. The ClockworkMod symlinks are in '/sbin' and, depending on the 'PATH' environment variable, usually called first.

Listing 3.6: Source: sgs2.py, addCWM

```
1  def addCWM(path):
2    s.lsh('unzip -qo '+path+'/cwm.zip -d '+s.INITDIR)
3    source = open(DEVPATH+'installcwm', 'r')
4    dest = open(s.INITDIR+'res/misc/kernelpatch.sh', 'a')
5    data = source.read()
6    source.close()
7    dest.write(data)
8    dest.close()
```

3.5. BusyBox

BusyBox is a multi-call binary that combines many common Unix utilities into a single executable. Most people will create a link to 'busybox' for each function they wish to use and BusyBox will act like whatever it was invoked as (15). If you build it from source it's individually configurable so size and functionality can vary. You can get a list of commands that are included in your phones version

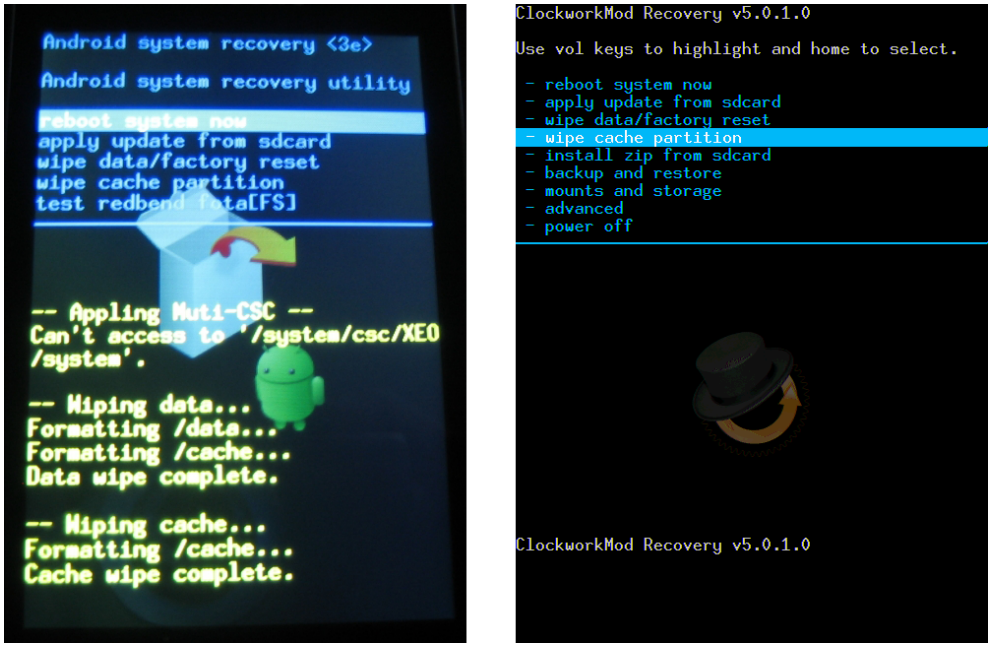

Figure 3.2.: Stock Recovery vs. ClockworkMod

by typing 'busybox –list' in a terminal. This feature is also used to generate the symbolic links, as you can see in Listing 3.7 (line 8-11), after we copied the binary to '/system/xbin'.

Listing 3.7: Source: installbusybox

```
1  mkdir /system/xbin
2  chmod 755 /system/xbin
3  rm /system/bin/busybox
4  rm /system/xbin/busybox
5  cat /res/misc/busybox > /system/xbin/busybox
6  chown 0.0 /system/xbin/busybox
7  chmod 6755 /system/xbin/busybox
8  for i in $(busybox --list)
9    do
10   ln -s busybox /system/xbin/$i
11 done
```

As expected, the method handling BusyBox is basically identical to the previous ones. Copy the binary and add the installation process to 'kernelpatch.sh' (Listing 3.8). As stated above, ClockworkMod needs some of BusyBox' tools to

work properly, so a limited version of BusyBox is included in CWM. If you need additional commands you need to compile your own version from source or get a pre-compiled version that supports them.

Listing 3.8: Source: sgs2.py, addBB

```
1 def addBB(path):
2     s.lsh('cp -f '+path+'/busybox '+s.INITDIR+'res/misc/')
3     source = open(DEVPATH+'installbusybox', 'r')
4     dest = open(s.INITDIR+'res/misc/kernelpatch.sh', 'a')
5     data = source.read()
6     source.close()
7     dest.write(data)
8     dest.close()
```

4. Summary, Limitations, and Related Work

The purpose of this work is to help with forensic investigation of smartphones running Android, specifically the Galaxy S2 from Samsung. Devices with a stock kernel lack important commands and functionality, which are required to perform certain operations, e.g. 'md5sum', system image creation and a root terminal.

4.1. Summary

After a short introduction in Chapter 1 we described our script and the involved parts of the phone in Chapter 2. Afterwards we introduced the different modules in Chapter 3 and provide a step-by-step guide for our script in Appendix A.

4.2. Limitations

While Android requires the manufacturers to have similar internal layouts for their phones to some extent they can still change things. One example would be the recovery binary that Samsung choose to include on the kernel partition instead on the existing recovery partition. This and other differences make it unlikely for our script, as it is, to work with something else than a Samsung Galaxy S2.

Also this script is only intended to work with stock kernels without any modifications. Our kernel builder might still work on custom kernels, but usually they already include all of the modules our script would add, so you wouldn't get any additional functionality, but, by chance break the kernel.

4.3. Related Work

There are a some specialized forums for phones, most notably *XDA developers* (16). Most of the site is dedicated to custom ROMs, i.e. altered versions of Android, which are not relevant for forensic purposes. Some threads provide close-to-stock kernels with some or all the modules added that you can find in this paper. Most of the time there is no documentation or source of what exactly has been added to the kernels, so while it is often the only source to get a modified kernel, we leave it in your discretion to decide if these files are suited for forensic investigations.

Bibliography

1 Softpedia, "Ces 2012: Victorinox pocket knife doubles as 1tb usb 3.0/esata ssd." [Online]. Available: http://news.softpedia.com/news/CES-2012-Victorinox-Pocket-Knife-Doubles-as-1TB-USB-3-0-eSATA-SSD-245455.shtml

2 iSuppli, "Smartphones to account for majority of cell-phone shipments by 2015." [Online]. Available: http://www.isuppli.com/Mobile-and-Wireless-Communications/News/Pages/Smartphones-to-Account-for-Majority-of-Cellphone-Shipments-by-2015.aspx

3 Business Insider, "Android's market share collapses as apple surges thanks to the iphone 4s." [Online]. Available: http://articles.businessinsider.com/2012-01-09/tech/30606530_1_new-iphone-android-sales-verizon-customers

4 XDA developers, "Howto: Backup sms database." [Online]. Available: http://forum.xda-developers.com/showthread.php?t=448361

5 Samsung, "Samsung open source release center." [Online]. Available: http://androidsu.com/superuser

6 XDA developers, "Official i9100 firmwares." [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1075278

7 ——, "Compile modules for stock kernels." [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1123643

8 eLinux, "Android boot." [Online]. Available: http://elinux.org/Android_Booting

9 XDA developers, "Unpack/repack initramfs in zimage (i9100)." [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1294436

10 Google Inc., "Android debug bridge - android developers." [Online]. Available: http://developer.android.com/guide/developing/tools/adb.html

11 XDA developers, "zergrush local root 2.2/2.3." [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1296916

12 ChainsDD, "Superuser." [Online]. Available: http://androidsu.com/superuser/

13 Android Market, "Root explorer (file manager)." [Online]. Available: https://market.android.com/details?id=com.speedsoftware.rootexplorer

14 Koushik Dutta, "Clockworkmod." [Online]. Available: http://www.clockworkmod.com/

15 BusyBox, "Busybox." [Online]. Available: http://busybox.net/

16 XDA developers, "Xda developers." [Online]. Available: http://forum.xda-developers.com/

Appendix

A. User Manual

If you are new to the topic or just don't know what to do here is a step-by-step guide.

A.1. Dumping The Kernel

For forensic investigations it's important to alter as few data as possible and be able to return to the original state of the device you are investigating. So before you flash a modified kernel, backup the original kernel from the phone. If you don't have root on the shell the system will prevent you from reading the block devices, so try to get temporary root with ZergRush or similar methods. Then type 'dd if=/dev/block/mmcblk0p5 of=/data/zImage bs=512' in a terminal to obtain the kernel. This image can then be flashed to the phone after you are finished with your operation.

A.2. Prerequisites

1. Python kernel builder script

What this paper is about. Get it from the enclosed CD.

2. ARM cross-compiler

For example: Sourcery G++ Lite 2009q3-68 toolchain for ARM EABI. Download from:
http://sourcery.mentor.com/sgpp/lite/arm/portal/release1033

3. Flashing software

We used **Heimdall**, because it is cross-platform compatible.
http://www.glassechidna.com.au/products/heimdall/

A.3. Configuration

After you set everything up edit the paths to the cross-compiler in 'settings.py':

Listing A.1: Source: settings.py

```
21 # paths to cross compiler
22 CROSSCPATH = '/opt/toolchains/arm-2009q3/bin/'
23 CROSSLPATH = '/opt/toolchains/arm-2009q3/lib/gcc/arm-none-eabi/4.4.1'
```

If, for any reason, you want to change the default directories you will find them in the 'settings.py' as well.

A.4. Running The Script

First start the script with the '–help' argument. It will show you the options. *device* shows a list of supported devices. As of now only the Samsung Galaxy S2 is supported. The SGS2 is also the default target, if *device* is not specified.

Listing A.2: Output: kernel_builder.py –help

```
1  positional arguments:
2    device                sgs2
3
4  optional arguments:
5    -h, --help            show this help message and exit
6    -k [path], --kernel [path]
7                          path to the kernel image
8    -r, --root            give adb root permissions
9    -c [path], --cwm [path]
10                         install ClockWorkMod
11   -b [path], --busybox [path]
12                         install BusyBox
13   -s [path], --su [path]
```

```
14                              install SU
15    -S [path], --superuser [path]
16                              install SuperUser.apk
17    -o [path], --output [path]
18                              output path
```

Now every other argument is also optional, but it wouldn't make much sense to run it without adding any modules. When giving an option, 'path' is not necessary. The script will default to the newest (sorted by date) directory in the binary folder. You can specify a directory of your choice as well. I decided to make paths relative to the scripts root directory so that the shell auto-completion will work. Be sure to give the path only (e.g. use '-k /myzimage/' instead of '-k /myzimage/zImage').

Superuser and BusyBox are bigger applications and with the limited space they might not fit in the *InitRamFS* together. Remember that ClockworkMod has a custom version of BusyBox included, so unless you are missing specific shell commands, there is no need to install both, BusyBox and ClockworkMod.

Finally, if you're having problems or just want to see some debug output start the script with 'python kernel_builder.py', because per default it is run with the '-O' Python flag. For demonstration purposes you can find a terminal output from a successful run with debugging enabled in listing A.3. After the script exits you can find your 'zImage' in the script's root directory, if you didn't specify a different output path.

A.5. Flashing

We prefer an application called **Heimdall** to flash images on the Samsung Galaxy. Heimdall is a cross-platform, open-source tool and runs under Linux as opposed to **Odin**, the leaked proprietary Samsung software. Heimdall comes with a GUI that simplifies the flashing process.

Before you can use Heimdall make sure your device is in Download mode. Turn it off and hold Home + Volume Down while turning it on again. Now plug it in and start Heimdall. On the Utilities tab detect your device. If your device is not recognized start **zadig** (Windows only) and follow the following steps.

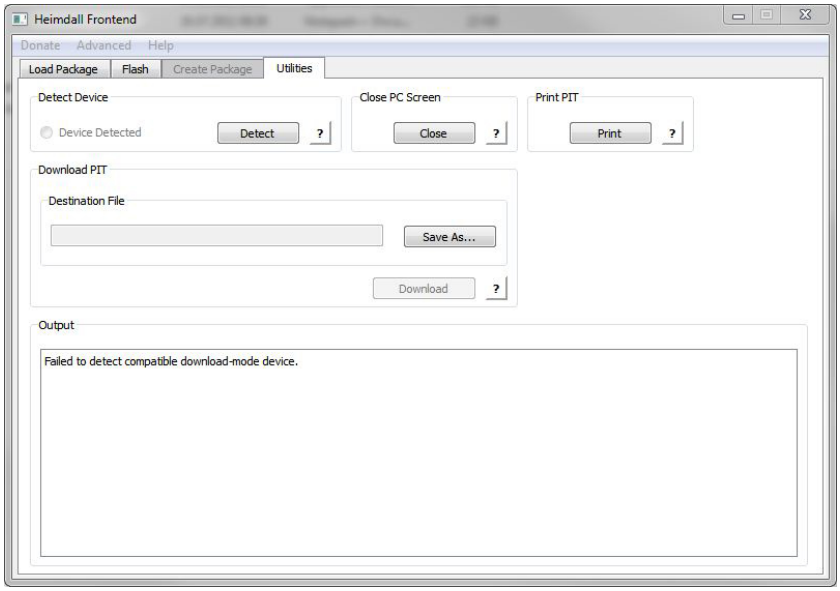

Figure A.1.: Heimdall: Utilities Tab

If your phone is now detected first download the PIT. That's the partition table for the device and tells Heimdall where to flash the boot image. In the next step switch to the Flash tab and select your PIT file. Then add a new partition file on the right side of the window, change the partition name to *KERNEL* and add a 'zImage' of your choice. Now press Start and the flashing should be finished in a few seconds. Congratulations, the phone will reboot automatically and run with your modified kernel.

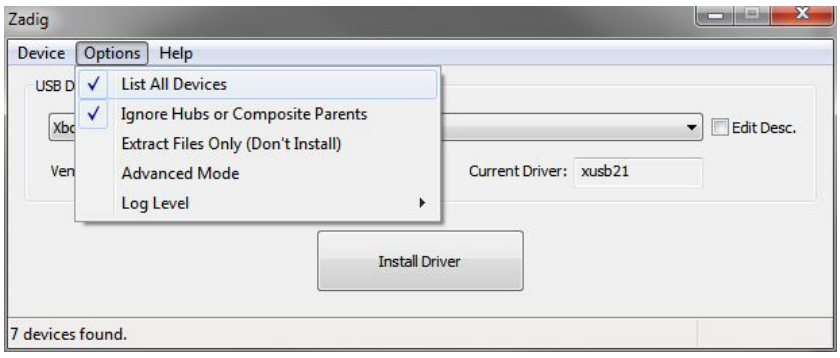

Figure A.2.: Heimdall: Zadig Options

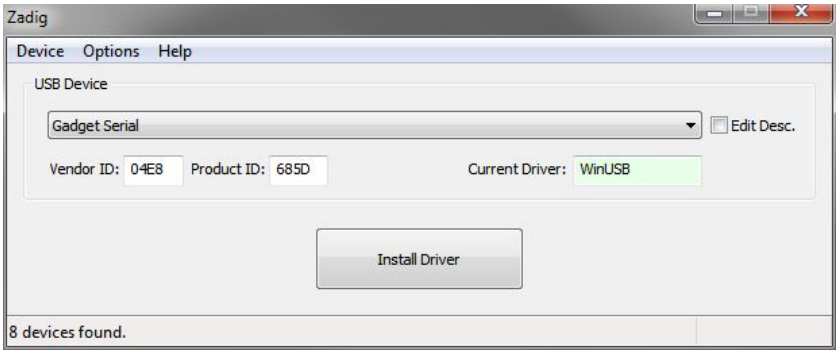

Figure A.3.: Heimdall: Zadig Main Screen

Listing A.3: Output: kernel_builder.py

```
1  python buildkern.py -k binaries/kernel/I9100XWKF3_Kernel/ -r -s -c
2  DEBUG:root:rm -rf ./tmp/
3  DEBUG:root:binaries/kernel/I9100XWKF3_Kernel/ seems valid
4  DEBUG:root:mkdir -p ./tmp/initram/
5  DEBUG:root:binaries/kernel/I9100XWKF3_Kernel/ seems valid
6  DEBUG:root:./devices/sgs2/unpack binaries/kernel/I9100XWKF3_Kernel//zImage ./tmp
       /initram/
7  DEBUG:root:add kernelpatch.sh to init.rc
8  DEBUG:root:mkdir -p ./tmp/initram/res/misc/
9  DEBUG:root:cp -f ./devices/sgs2/kernelpatch.sh ./tmp/initram/res/misc/kernelpatch.sh
10 DEBUG:root:modifying default.prop
11 DEBUG:root:no (valid) path: 'True' for cwm. using latest binary
12 DEBUG:root:./binaries/cwm/5.0.2.3
13 DEBUG:root:unzip -qo ./binaries/cwm/5.0.2.3/cwm.zip -d ./tmp/initram/
14 DEBUG:root:add cwm install script to kernelpatch.sh
15 DEBUG:root:no (valid) path: 'True' for su. using latest binary
16 DEBUG:root:./binaries/su/3.0.3
17 DEBUG:root:cp -f ./binaries/su/3.0.3/su ./tmp/initram/res/misc/
18 DEBUG:root:add su install script to kernelpatch.sh
19 DEBUG:root:binaries/kernel/I9100XWKF3_Kernel/ seems valid
20 DEBUG:root:finalize kernelpatch.sh
21 DEBUG:root:./devices/sgs2/gen_initramfs.sh -o ./tmp/initram.cpio -u 0 -g 0 ./tmp/
       initram/
22 DEBUG:root:./devices/sgs2/repack binaries/kernel/I9100XWKF3_Kernel//zImage ./tmp
       /initram.cpio /opt/toolchains/arm-2009q3/bin/ /opt/toolchains/arm-2009q3/lib/
       gcc/arm-none-eabi/4.4.1 true
23 [I] ----------kernel repacker for i9100
```

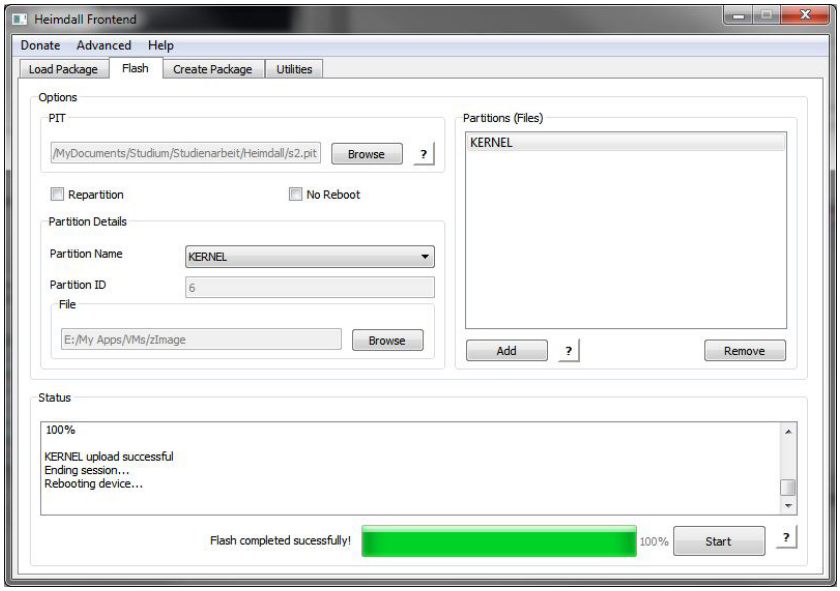

Figure A.4.: Heimdall: Flash Tab

```
-----------------------------
[I] Extracting gzip'd kernel from binaries/kernel/I9100XWKF3_Kernel//zImage (start = 16621)
[I] Non-compressed CPIO image from kernel image (offset = 163840)
[I] CPIO image MAX size:2982912
[I] Head count:3146752
[I] Making head.img ( from 0 ~ 163840 )
[I] Making a tail.img ( from 3146752 ~ 10554720 )
[I] Current ramdsize using cat : 4602880 with required size : 2982912 bytes
[I] Current ramdsize using gzip -f9 : 2731390 with required size : 2982912 bytes
[I] gzip -f9 accepted!
[I] Merging [head+ramdisk] + padding + tail
[I] Now we are rebuilding the zImage
[I] Image ---> piggy.gzip
[I] piggy.gzip ---> piggy.gzip.o
[I] Compiling head.o
[I] Compiling misc.o
[I] Compiling decompress.o
[I] Compiling lib1funcs.o
[I] Create vmlinux.lds
[I] head.o + misc.o + piggy.gzip.o + decompress.o + lib1funcs.o---> vmlinux
[I] vmlinux ---> zImage
[I] New zImage size:6072140
```

```
45 [I] zImage has been created
46 [I] Cleaning up...
47 [I] finished ...
48 DEBUG:root:repack exit code: 0
49 DEBUG:root:mkdir -p ./
50 DEBUG:root:cp -f ./tmp/zImage ./
```

B. Content Of The CD

- **binaries/:** External files used in the build process
  - **busybox/:** BusyBox binary
  - **cwm/:** ClockworkMod files stored in a .zip
  - **kernel/:** Collection of stock kernel
  - **su/:** Su binary
  - **superuser/:** Superuser Android application
- **devices/:** Device-specific files
  - **sgs2/:** Files for the Samsung Galaxy S2
- **LaTeX/:** LaTeX source for this document
- **prerequisites/:** Prerequisites for the script
  - Heimdall
  - SourceForgery cross-compiler
- **kernel_builder.py:** Main entry point for the script
- **settings.py:** Settings for the script

Eidesstattliche Erklärung

Hiermit versichere ich, dass diese Abschlussarbeit von mir persönlich verfasst ist und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

Erlangen, den 08.02.2012

Tobias Brentrop