1 Management Summary

Mobile phones and especially smartphones are ever growing in speed and capabilities, the latest trend is the mobile operating system called Android. As Android is based on the operating system Linux, it allows for porting applications which are traditionally run on computers. This gives users the capabilities to run their applications on smartphones. We ported two so called ‘Covert Channel’ applications; Hans and Iodine. These two applications ‘misuse’ protocols if you will, to set up a communication channel which is somewhat hidden. By changing bits and bytes in a specialized way it can bypass security policies, allowing for communication even though it is normally blocked or pay-only.

The impact of all this is that wireless access points which deploy a pay portal to earn money can be bypassed, allowing users to browse the Internet for free. As the Android market is booming business right now, this threat will only increase over the coming years.

Even though the application is accessible for everyone, it does require some initial steps and work from the user. For example the user needs a server which is always up, and has hans tunnel, Iodine and iperf configured and running. The user also needs ‘root access’ to his Android phone, this kind of access is not standard on consumers devices. Thus, this requires quite some technical work as well. Overall, because of additional technical steps we feel that the threshold is still somewhat high, resulting that only the technical people would end up using this application.

However, this does not mean that owners of pay portals should simply ignore this technology, instead they should actively increase the security.
2 Introduction

Nowadays phones are ever increasing in capabilities and smartphones should be regarded as a full-fledged computer. To prove this point, we ported two applications to Android which are traditionally run on computers. These applications can bypass pay portals on wireless hotspots. The direct impact of this is that people who use our application will not need to pay the portal, instead they can simply access the Internet for free. And with the increasing number of Android-powered phones, this threat will only get bigger.

3 Research

We ported two applications which create so called covert channels, they rely on other protocols to transmit their data. This means that the change and use legitimate protocols to transfer data. Humans are bad with numbers but pretty good with words and names. Computers and networks mostly work with numbers, so a link for needed for bridge the gap between people and computers. This link is called ‘Domain Name System’, it translates a domain like ‘www.os3.nl’ which people know into ‘145.100.96.70’ which computers know. Without this DNS system, the Internet would not work or even exist in the way and form it does now. However, ‘iodine’ can use that DNS protocol to transmit and request data, like websites. By porting iodine to Android, it allows users to circumvent pay portal because the portal is not blocking DNS, hence traffic can freely flow.

However, setting up, enabling, testing the tunnels can be slow and cumbersome process, especially if done on a phone. To make this whole process easier and friendlier for the users, we built an easy to use user interface. With a single click of a button the tunnels are set up and checked for speed, the best tunnel will then be selected to use.

4 Covert Channels

A covert channel is a communication channel which is capable of bypassing security policies to transferring information. By modifying certain unused bits is it possible to transfer data over existing protocols, which could be viewed as misuse or even unlawful. In our case the Android phone is the client, and you need a computer which is always on to be used as a server. This way a full-fledged tunnel can be built; the entrance on the Android, tunneling over the access point and the exit on the server.

4.1 Iodine

Iodine is a DNS-based covert channel, this means it tunnels data through DNS requests and replies. The server side listens on a specific domain, which is set up by Iodine itself, for example tunnel.example.com. The client who
wishes to surf the web, requests webpages by asking the local DNS about ‘X.tunnel.example.com’, where ‘X’ consists of specially crafted requests to transfer data. The replies are specially crafted as well to send the requested data back. Because the first requests are to the local DNS server, it in essence turns that local DNS into a man-in-the-middle, unknowingly an accomplice. All the DNS requests and replies is in itself bogus data, trying to masquerade as normal DNS traffic while sending back and forth other data or information. Blocking DNS to block this tunnel will most likely break the ability to even reach the pay portal of the access point, so blocking this can be rather tricky.

4.2 Hans tunnel

Hans tunnel is an ICMP-based covert channel, this means that it tunnels data through ICMP packets. ICMP stands for ‘Internet Control Message Protocol’ and has a broad usage, ranging from checking if a host is up to retrieving a timestamp for time synchronization. Checking if a host is up is used by the ‘ping’ command, which sends a single ICMP packet out and expects the same back. This packet can be filled with data in a pattern to see if not only the transmission works, but also if the communication is error free. In the case of Hans tunnel, it puts the requests and replies directly into that data field. This covert channel does not rely on a local server or any extra node, as this is a direct link between the client and server. However, most access points block this kind of traffic making this application less useful.

4.3 Ptunnel

Ptunnel is just like Hans tunnel ICMP-based, so it is based on the same protocol to communicate. However, internally it does work differently. Whereas Hans tunnel and Iodine create a new tun device in Linux, ptunnel does nothing of that sort. This means you need to push the traffic on your own through the tunnel, for example by setting the tunnel as the proxy. Because of these extra steps and difficulties we left it out of the initial release of our application.
5 The Android Platform

The Android is an open software platform for mobile development and it consists of the Android Operating system and the accompanying software stack (middleware and key applications) [1]. It was released in November 2007 as an open source project and at the time of writing this document (May 30, 2010) the current version was 2.2 (released in May 2010). The project was initiated by Google which, together with other 65 hardware, software and telecom companies form the “Open Handset Alliance”, the consortium that now supervises the development of the Android Platform.

The Android OS is an open source product, licensed under the Apache License.

5.1 Android Architecture

![Android System Architecture](image.png)

At the foundation of the Android OS lies a Linux 2.6 kernel that acts as an abstraction layer for the underling hardware. The Linux kernel provides a
lot of the basic features: memory management, process management, a security model, networking, etc. The driver model used by the Linux kernel has been proven over time and most manufacturers already provide drivers for it.

Up the stack we find the native libraries, most of them written in C or C++. These libraries take care of core tasks like 2D/3D graphics, multimedia features (video, sound), persistent data storage (SQLite) or web rendering (WebKit).

Alongside the native libraries stands the Android Runtime which consists of the Dalvik Virtual Machine and on top of that the Core Libraries. The Dalvik Virtual Machine is a virtual machine optimized for embedded devices. At build time .class and .jar files are transformed to a special format .dex which contain optimized bytecode for the Dalvik Virtual Machine (DVM). The DVM allows a more efficient use of memory and also allows the share of data structures over different processes, whenever that is possible. In the Android OS each process runs in its own Dalvik Virtual Machine. The Core Libraries are written in Java and contain basic functionalities needed by applications: I/O, collection classes, etc.

The Applications Framework is written in Java and is the API available to all the the Android applications, whether they were bundled with the device or written by a 3rd party. For example the Content Providers allow applications to share data: the Contacts application can share phone numbers, addresses, names to other interested applications. The View System handles the building blocks of the UI: buttons, lists, etc. but also handles event dispatching, layout and drawing.

On top of these layers are the regular Android applications.

5.2 Android Applications

Android applications make use of “building blocks” to perform different tasks and to interact with the Android ecosystem.

Activity An Activity is the Desktop OS equivalent of a window. It is the basic UI component with which the user can interact. It corresponds to exactly one screen, so that when a user switches to a new screen (e.g. from “Main Menu” to “Settings”) in fact it switches to a new Activity.

IntentReceiver Intents are used for inter-component signaling and communication. An IntentReceiver is a component that responds to notifications or status changes. It is a way of registering code that will be running only when a certain external event is triggered via an intent.

Service A service is a task that doesn’t have an UI component and it runs independently of any Activity. Because of the Activity Lifecycle, Activities can be destroyed whenever the OS needs more resources or can be stopped when the user presses back for example, so no long running tasks should be performed by an Activity. Instead it should run in the background, in a Service. A good
example of a service is the “Music Player”. The play command is triggered by an Activity but once the music starts the user will want to switch to another activity. The playback operation should be implemented in a Service. An activity can connect at any time to a service and interact with it or can send intents addressed to the service. The main difference between using intents and directly calling the service is that data transfer is more easily achieved when calling services directly.

**ContentProvider**  The ContentProvider allows the application to share data system wide. It give complete control to the application on how the data is shared and with whom.

Figure 2: Android Activity Lifecycle
5.3 Native Components

Native components represent libraries that are developed in a native-code, specific to the instruction set of the device, language, such as C and C++. Using native components can have several benefits to certain classes of applications in the form of reuse of existing code or speed increase.

The Android Native Development Kit (NDK) provides a complete package that contains:

- cross-toolchains - compilers, linkers, etc
- a set of system headers to stable native APIs:
  - libc (C library) headers
  - libm (math library) headers
  - JNI interface headers
  - libz (Zlib compression) headers
  - liblog (Android logging) header
  - OpenGL ES 1.1 and OpenGL ES 2.0 (3D graphics libraries) headers
  - libjnigraphics (Pixel buffer access) header (for Android 2.2 and above)
  - A Minimal set of headers for C++ support
- build system that allows the developer to write short build files that call the required tools.
- samples and documentation

The C compiler included in the NDK is the GNU C compiler.

Using the Android NDK one can port existing code on the Linux platform to the Android OS with reduced effort. Alternatives to NDK include Embeddian (http://www.emdebian.org/) or CodeSourcery (http://www.codesourcery.com). Both provide an environment that allows the creation of statically linked binaries for the Android. The disadvantage is that statically compiled files are larger and consume more memory as each depending library is included in the binary and loaded into memory at runtime. On the other hand, statically linked binaries are not affected by updates that possibly break compatibility with existing applications as they have no dependencies.

One important aspect regarding the NDK and Android is that the standard C library used is bionic, a light-weight C library with a smaller footprint than glibc. The core idea behind Bionic’s design is: KEEP IT REALLY SIMPLE. It consists of part BSD C library code and part Linux-specific code (threads, signals, etc). Bionic also comes bundled with its own PThread implementation.

Libraries are loaded in the Android application using JNI (Java Native Interface). In the source code of the application one must first declare all the methods available in the library as “native” (e.g. native byte[] loadFile(String filePath); ) and then load the library using System.loadLibrary.
5.4 Android Caveats

The kernel used by Android is a forked and modified version of Linux. This implies that not all the features that one would expect to be present are included in Android. Two examples of such features are the netfilter and tun kernel modules. Without these two modules running applications that interact with the network stack at a lower level is made impossible. The solution for this lack of features is to install a custom OS image that includes them. CyanogenMOD is such one unofficial custom OS image that provides missing but useful OS features.

The Android application security model implies that each application runs in its own Dalvik Virtual Machine instance and it has its own process and own user. Application IPC is done exclusively using Intents and the application can not gain a different set of privileges. Additionally the user must agree at installation time with the set of features that the application will access (location, wifi connection statistics, Internet access, etc). The application can not change any of these permissions at runtime.

While this security model proves to be very effective it also affects the use of the Android OS for other purposes other than the intended ones. No super user privileges can be obtained by regular applications and low-level interaction with the underlying operating system can not be achieved. While an application can use Runtime.getRuntime().exec() to execute any file available the privilege level is still the same as the parent (caller of exec function).

This limitation can be overcome by first executing su which provides the process super user privileges. Unfortunately, this is not available on regular Android devices and the user must manually intervene, often voiding the warranty of the device, to install an appropriate su binary file. Having access to super user privileges, applications can now full interact with the operating system. While this approach is not supported it can provide additional features to the device as it opens new opportunities for developers and users alike.

Below we present a code sequence that first starts su and then executes a given command with root privileges.

```java
1  cmdProcess = Runtime.getRuntime().exec("su");
2  
3  os = new DataOutputStream(cmdProcess.getOutputStream());
4  os.writeBytes(command + "\n");
5  os.flush();
6  os.writeBytes("exit\n");
7  os.flush();
```

Code Snippet 1: Executing an external command with root privileges
6 Porting Applications to Android

Besides writing native libraries the NDK can also be used to create native executables, enabling developers to port existing Linux applications and creating new ones.

This section will describe the porting process for an existing Linux application to Android.

Upon unpacking the source code archive the first thing that needs to be inspected is the associated makefile. The build steps described in the make file must be replicated to the NDK build script format.

The Android build system was designed in a modular fashion, where each application or library is part of a main build tree. This implies that the build files, or makefiles used are different than the regular GNU make or equivalent makefiles. The build script can contain equivalents for the make variables:

- \texttt{CFLAGS} → \texttt{LOCAL\_CFLAGS}
- \texttt{INCLUDES} → \texttt{LOCAL\_C\_INCLUDES}
- \texttt{LDLIBS} → \texttt{LOCAL\_LDLIBS}

The Android NDK documentation provides a good source on how to create Android specific build files.

After creating the required Android build file calling $\texttt{NDK\_ROOT/ndk-build}$ in the project directory will start the compilation process.

Because of the differences between the Linux system and Android the code might not compile cleanly from the first try and it is the developer’s job to edit the source code in order to reflect the differences between the two. Unfortunately this is a tedious process that requires time and perseverance. Upon completion the end result will be a binary executable of the ported application. In the following subsections we present this process in more detail using iodine and hans tunnel as examples.

6.1 iodine

Below is the iodine 0.6.3 \cite{iodine} makefile which is is a standard makefile for a small project. Iodine does not rely on autotools so the makefile is relatively compact and well structured. To enable portability the makefile uses\texttt{uname} to check the current architecture and enable the correct architecture flag at compile time.
COMMONOBJS = tun.o dns.o read.o encoding.o login.o base32.o base64.o base64u.o base128.o md5.o common.o
CLIENTOBJS = iodine.o client.o util.o
CLIENT = ../bin/iodine
SERVEROBJS = iodined.o user.o fw_query.o
SERVER = ../bin/iodined
OS = 'echo $(TARGETOS) | tr "a-z" "A-Z"'
ARCH = 'uname -m'

LIBPATH = -L.

LDFLAGS += -lz 'sh osflags $(TARGETOS) link' $(LIBPATH)
CFLAGS += -c -g -Wall -D$(OS) -pedantic 'sh osflags $(TARGETOS) cflags'

all: stateos $(CLIENT) $(SERVER)

stateos:
  @echo OS is $(OS), arch is $(ARCH)

$(CLIENT): $(COMMONOBJS) $(CLIENTOBJS)
  @echo LD $@
  @mkdir -p ../bin
  @$(CC) $(COMMONOBJS) $(CLIENTOBJS) -o $(CLIENT) $(LDFLAGS)

$(SERVER): $(COMMONOBJS) $(SERVEROBJS)
  @echo LD $@
  @mkdir -p ../bin
  @$(CC) $(COMMONOBJS) $(SERVEROBJS) -o $(SERVER) $(LDFLAGS)

.C.o:
  @echo CC <$
  @$(CC) $(CFLAGS) <$ -o $@

base64u.o client.o iodined.o: base64u.h
base64u.c: base64.c
  @echo Making $@
  @echo '/* No use in editing, produced by Makefile! */' > $@
  @sed -e 's/^[Bb][Aa]64/\1u/g ; s/0123456789+/0123456789_/' < base64.c >> $@

base64u.h: base64.h
  @echo Making $@
  @echo '/* No use in editing, produced by Makefile! */' > $@
  @sed -e 's/^[Bb][Aa]64/\1u/g ; s/0123456789+/0123456789_/' < base64.h >> $@

clean:
  @echo "Cleaning src/"
  @rm -f $(CLIENT){,.exe} $(SERVER){,.exe} *~ *.o *.core base64u.*

Code Snippet 2: Original iodine makefile

Below is the same makefile simplified and rewritten for the Android NDK, it only generates the iodine client, omitting the server binary. The LOCAL_SRC_FILES variable contains a list of all the source files and LOCAL_CFLAGS the appropriate gcc compiler flags. LOCAL_MODULE defines the name of the output file and LOCAL_PATH sets the absolute path of the project. The last declaration specifies that the compiler should output an executable. We have removed any “autocon-
configuration” feature in the build script as we are targeting only one architecture, ARM, the Android Architecture.

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES :=
    src/tun.c
    src/dns.c
    src/read.c
    src/encoding.c
    src/login.c
    src/base32.c
    src/base64.c
    src/base64u.c
    src/base128.c
    src/md5.c
    src/common.c
    src/iodine.c
    src/client.c
    src/util.c

LOCAL_CFLAGS := -O2 -g
LOCAL_CFLAGS += -DLINUX

LOCAL_LDLIBS := -lz
LOCAL_MODULE := iodine

include $(BUILD_EXECUTABLE)
```

Code Snippet 3: iodine Android.mk

Specific to iodine is the use of the `<arpa/nameser.h>`, a header file specific to DNS resolving. While the “complete” libc’s available on the desktop operating systems include this header file, bionic ships a file with the following content:

```
/* this header intentionally blank
 *
 * the definitions normally found in <arpa/nameser.h> are
 * really a bunch of resolver’s internal declarations that
 * should not be exposed to client code in any way
 */
```

In order to compile iodine we had to provide our own version of this file and modify the include directives in the source code of the application.

To configure the tun device that it creates, iodine uses `exec` to calls the standard Unix commands, `ifconfig` and `ip`, but these utilities are not always available to the user on the Android device. We modified the code so that the `busybox` utility is called to perform these tasks. Because we used a custom Android OS image for the T-Mobile G1 device, we changed the hardcoded location of the tun device from `/dev/net/tun` to `/dev/tun`. This could have been
avoided by manually creating the tun device on the mobile using the following shell commands:

1. `busybox mkdir -p /dev/net`
2. `busybox mknod /dev/net/tun c 10 200`
3. `busybox chmod 600 /dev/net/tun`

**Code Snippet 4: Creating a tun device on the Android**

A diff run between the ported and original code is available in the Appendix A.1.

### 6.2 hans

Hans tunnel [3] is written in C++ and uses a make file similar in may ways to the one used by iodine and therefore the Android.mk creation process was similar with the one described above. Because the Android NDK is very limited in what regards C++ support, the application refused to compile invoking a missing C++ standard library.

```
1 In file included from apps/covert_service/project/jni/hanstunnel/auth.cpp:20:
2 apps/covert_service/project/jni/hanstunnel/auth.h:23:18: warning: vector: No such file or directory
3 apps/covert_service/project/jni/hanstunnel/auth.h:24:18: warning: string: No such file or directory
4 In file included from apps/covert_service/project/jni/hanstunnel/auth.cpp:20:
5 apps/covert_service/project/jni/hanstunnel/auth.h:31: error: ISO C++ forbids declaration of ‘vector’ with no type
6 apps/covert_service/project/jni/hanstunnel/auth.h:31: error: typedef name may not be a nested-name-specifier
7 apps/covert_service/project/jni/hanstunnel/auth.h:31: error: expected ';' before '<' token
8 apps/covert_service/project/jni/hanstunnel/auth.h:41: error: ‘Challenge’ does not name a type
9 apps/covert_service/project/jni/hanstunnel/auth.h:42: error: ISO C++ forbids declaration of ‘Challenge’ with no type
10 apps/covert_service/project/jni/hanstunnel/auth.h:42: error: expected ‘,’ or ‘...’ before ‘&’ token
11 apps/covert_service/project/jni/hanstunnel/auth.h:45: error: ‘string’ in namespace ‘std’ does not name a type
12 apps/covert_service/project/jni/hanstunnel/auth.h:46: error: ‘string’ in namespace ‘std’ does not name a type
```

**Code Snippet 5: Compiling hans with the official Android NDK**

After through researching, we found a patched version of the Android NDK [2] that includes C++ support. In order to achieve Android compatibility the compiler statically links some C++ components to the resulting binaries. This only affects the size of the application and not the performance or forward compatibility with future versions of Android.

As with iodine, we enabled the use of busybox and fixed errors regarding missing error logging functions.

A diff run between the ported and original code is available in the Appendix A.2.
7 Covert Droid

Covert Droid is an application that acts as a front-end for covert channel applications on Android.

7.1 Features

Covert Channels  Currently Covert Droid supports two covert channel applications: hans tunnel - ICMP based, and iodine,DNS based. While we have also ported ptunnel, another ICMP based application, we decided not to include it at this time. The Android OS does not provide an easy way to access the HTTP proxy setting and therefore the use of ptunnel is very limited. If the Android OS will provide such a feature, then ptunnel will become a good candidate for a general purpose covert channel application on Android.

Autodetect Covert Droid can autodetect the best suitable covert channel application by performing a series of tests that determine if the tunnel can be created, its bandwidth, average latency and packet loss. After performing the tests for each of the tunnels it starts the one with the best parameters.

Benchmarking Covert Droid can perform the same tests from autodetection phase, on user request. This enables the user to check the quality of the tunnel at any moment in time. This is achieved by using the iperf utility, a well known network testing tool.

Permanent Service Covert Droid includes a Service that starts and monitors the tunnel constantly informing the user about the health state of the tunnel.

Multi-language Covert Droid currently supports three languages: English, Dutch and Romanian. Additional languages can be easily added to the application.

Easy to plug in new covert channel applications Due to its modular design, Covert Droid can be easily extended to integrate new covert channel applications like nstx or icmptx. This can be done by extending the CovertApp class which provides the basic functionalities for tunneling applications.

7.2 Screenshots

The Covert Droid GUI has the following components:

- the Main Menu
- the Settings Menu

13
The Main Menu allows the user to control the application. From here the user can access the application settings, can start tunnels or can test the performance. The top button changes its function depending on the application state: start or stop.

The Settings activity allows the user to choose what tunnel application will be launched on the start command, the interface language and what benchmark parameters will be used.

The Benchmark Settings present the user the following editable fields:

- **ping host** - the target host that will be used for the ping command. This will provide the latency and link loss statistics.

- **ping count** - the number of ICMP echo_request packets sent by the application to the ping host.

- **ping timeout** - the number of seconds to wait for the ping command to execute. This setting allows the benchmark operation to complete in a timely manner.

- **iperf host** - address where an iperf server is listening. Iperf is used to determine the bandwidth of the tunnel.
• **iperf timeout** - the number of seconds to perform an iperf run. This setting allows the benchmark operation to complete in a timely manner. Increasing this value will provide more accurate bandwidth measurements.

If “Autodetect” is selected on the “Tunnels” list then the start and benchmark buttons will behave as following. The “Start” button will launch the Benchmark Activity 3(b), sending it a special parameter that signals that it should perform the tests and return the results back to the “Main Menu” activity.

### 7.3 Application Design

The Covert Droid application is composed of the following packages:

- **Covert GUI** - the nl.os3.ids.gui package - the user interface and application state
  - About.java - About Activity
  - Benchmark.java - Benchmark Activity
  - BenchmarkSettings.java - Benchmark Settings Activity
  - CovertDroid.java - CovertDroid Application class - it keeps application wide state
  - hans.java - hans tunnel settings Activity
  - Help.java - Help Activity
  - ICovertService.java - Interface implemented by Covert Service
Figure 3: Covert Droid Benchmark

- iodine.java - iodine settings Activity
- Main_Menu.java - The main activity
- Preferences.java - Preferences class that stores user settings
- ptunnel.java - ptunnel settings Activity
- Settings_Menu.java - Settings Activity

**Covert Service** - nl.os3.ids.gui.service - the service that keeps track of the application state and performs low level interaction with the Android OS

- CoreTask.java - Class that handles process execution, output and return value retrieval, performs latency and bandwidth tests
- CovertApp.java - class that abstracts the behaviour of a covert channel application
- CovertService.java - class that handles the lifecycle of the service

The flow diagram 4 presents the interaction between different components of the Covert Droid application.

First the Main Menu activity starts the CovertService service and stores a reference to it. CovertService exposes a public interface described in the ICovertService class interface definition. The Main activity calls the start() and stop() public methods exposed by the service. This triggers a series of actions inside the service.
Code Snippet 6: Starting and connecting to a service

Once the application state has changed, i.e. a tunnel is running, the service notifies the main activity via an intent about the state change. The Main activity registers a BroadcastReceiver that listens for state changed intents and which performs actions like button text replacing or disabling the start/stop button.
private BroadcastReceiver receiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        Log.d(MSG_TAG, "onReceive()");
        setToggleText();
    }
};

7.3.1 GUI

GUI design on the Android can be done in a manner that separates logic from presentation. In the res/layout project folder we create a XML file that will describe the visual appearance and position of the UI elements. Below we present a code listing that shows how a button is defined. The Button element’s attributes serve either for specifying visual properties (width or height) or for providing a handle (id) that will later be used to interact with the button. The string attribute is used to dynamically bind a string that is defined in another resource file, strings.xml. This file contains all the strings used in the application. Supporting a new language means requires just the translation of the string.xml file.

```xml
<Button android:id="@+id/StartStopButton" android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:text="@string/Start" />
```

Code Snippet 8: Defining a button GUI element

```xml
<string name="Start">Start</string>
```

Code Snippet 9: Defining a string element

```java
startstopButton = (Button) findViewById(R.id.StartStopButton);
startstopButton.setOnClickListener(StartStopListener);
```

Code Snippet 10: Obtaining a reference to and using a Button object

7.3.2 CovertService

The CovertService maintains the tunnel state and reacts to state changes: tunnel FAILED, tunnel STOPPED, tunnel RUNNING, tunnel STARTING. Based on
these states the service notifies the GUI to update its appearance.

CovertService is also responsible for creating the CovertApp objects, the ones that encapsulate the covert tunnel applications.

### 7.3.3 CovertApp

The CovertApp class is an abstract class that implements the main methods needed to interact with a tunnel application.

```java
//performs a benchmark and sends the results to the gui
public abstract void benchmark();

// performs a check to see if the app is running
public abstract int getPid();

//starts the tunnel and checks if it is up
public abstract void run();

//sets routes
public abstract void setRoutes();

//stops the tunnel
public abstract void stopTunnel();

//app name
public abstract String toString();

//removes the routes
public abstract void unsetRoutes();
```

Code Snippet 11: CovertApp interface

To start the actual covert application process CovertApp calls the CoreTask class that does all the low level “heavy lifting” like:

- launch processes with root privileges
- check return values
- parse output of commands
- check running processes
- add or remove routes
//delete a route
public void delRoute(String net, String via, String dev);
//execute given command as root and return stdout, stderr and exitVal
public Hashtable<String, ArrayList<String>> executeCommand(
    String command);
//perform bandwidth test
public Hashtable<String, String> getBandwidth(
    Hashtable<String, String> settings);
//get ip of given device
public String getDeviceIP(String dev);
//perform latency test
public Hashtable<String, String> getLatency(
    Hashtable<String, String> settings);
//check if a device is next hop interface for a given route
public boolean isDeviceExitIfaceForRoute(String route,
    String device);
//check if a given network device is present
public boolean isDevicePresent(String device);
//check if process name is running
public boolean isProcRunning(String proc);
//sends SIGKILL to a process
public void killProcesByName(String process);
//adds a default route
public void setDefaultRoute(String gateway, String dev);
//adds a route
public void setRoute(String net, String via, String dev);
//sends SIGINT to a process
public void sigintProcesByName(String process);

Code Snippet 12: CoreTask interface

Each supported tunnel application extends the CovertApp class to accommodate its own method of starting and monitoring its state. Below we present the Iodine class, which passes to its parent class a Hashtable object containing its specific settings. Also iodine needs a static route to the WiFi DNS server. In the DNS server and Android device are not on the same subnet, this route is added using the method onCovertAppRunning.
private class Iodine extends CovertApp {
    public Iodine(Hashtable<String, String> settings) {
        super(settings, getBaseContext(), CovertService.this);
        String appPath = appSettings.get("appPath");
        String device = appSettings.get("device");
        String dnsServer = appSettings.get("remoteServer");
        String password = appSettings.get("password");
        String tunnelDomain = appSettings.get("tunnelDomain");
        build the app string
        String cmd = appPath + " -d " + device + " -P " + password + " "
                     + dnsServer + " " + tunnelDomain;
        appSettings.put("startcmd", cmd);
        Log.d(MSG_TAG, "Iodine()");
    }
    @Override
    void onCovertAppRunning() {
        coreTask.setRoute(Preferences.iodine_relayIP,
                         CovertDroid.DHCP_GATEWAY, "tiwlan0");
    }
}

Code Snippet 13: Iodine class
8 Test Setup and Experimental Results

In this section we will present how we built our test environment and the results of our experiments.

For all of our experiments we used a XEN virtual machine running Fedora 13 Alpha1. The kernel version was 2.6.33. We will further refer to this machine as the “domain host”. It served as an endpoint for both hans and iodine.

hans and iodine setup tunnels that allow communication between the two tunnel ends. To enable Internet connectivity for the mobile device we had to enable IP routing and NAT on the domain host.

```
1 iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
2 echo 1 > /proc/sys/net/ipv4/ip_forward
```

Code Snippet 14: IP setup on the domain host

8.1 iodine

Iodine needs a valid DNS domain name delegated to the machine that runs the iodine daemon. Using bind we added the following to the cosu.ro parent zone:

```
1 tun NS tunhost ; subdomain delegation
2 tunhost A 145.100.105.107 ; ip address of iodined machine
```

Code Snippet 15: Iodine DNS setup

On the host machine we started iodined giving it as parameters the domain name, an arbitrary session password and the IP subnet that will be used over the tunnel.

```
./iodined -f 10.1.0.1 tun.cosu.ro -P test
```

Code Snippet 16: Iodine in server mode

8.2 hans

As hans uses ICMP we had to disable ICMP echo_reply from the kernel. This makes the OS ignore ICMP messages and allows hans to communicate with the remote end.
echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_all

Code Snippet 17: Turning off ICMP echo on the Linux kernel

We started hans in server mode giving it as parameters the IP subnet and an arbitrary session password.

Code Snippet 18: Hans server mode

8.3 iperf

To perform bandwidth measurements we used the iperf tool running in server mode on the domain host.

Code Snippet 19: Iperf server mode

8.4 Covert Droid Installation

To install Covert Droid a number of steps have to be performed. First, the iodine, hans and iperf binaries must be loaded on the device. To do this we used the **adb push** command from the Android SDK. Once the binaries have been compiled with the Android NDK we copied them on the device.

Code Snippet 20: Installing binaries on Android

The application package can be installed using the **adb install** command.

Code Snippet 21: Installing the Covert Droid Application Package
8.5 Testing

For hans to succeed, it needs a clear path to the domain host, on which it can send ICMP packets. If ICMP is blocked then hans will not succeed in creating a tunnel.

For iodine to succeed it either needs a clear path to the domain host or it can use the local DNS resolver to forward DNS queries to the domain host. The first option is called “raw login” and offers the best performance. The second option offers less bandwidth but it succeeds more often as it is less dependent on the restrictions employed by the WiFi hotspot owner.

<table>
<thead>
<tr>
<th>WiFi AP</th>
<th>Hans</th>
<th>Iodine</th>
<th>Iodine raw login</th>
</tr>
</thead>
<tbody>
<tr>
<td>UVA Guest</td>
<td>×</td>
<td>40KB/s</td>
<td>×</td>
</tr>
<tr>
<td>VU Guest</td>
<td>×</td>
<td>40KB/s</td>
<td>×</td>
</tr>
<tr>
<td>Home WiFi (Ziggo + SurfNET)</td>
<td>120KB/s</td>
<td>40KB/s</td>
<td>120KB/s</td>
</tr>
</tbody>
</table>

Our tests focused more on the functional aspects rather than precise values.

We tested the application in three locations: at Universiteit van Amsterdam in the OS3 lab, connecting to the UVA guest WiFi network, at Vrije Universiteit Amsterdam, connection to the VU guest WiFi network and in a private setup with a regular wifi router with an Internet connection provided by Ziggo and SurfNET.

The user experience varies from good to usable. We have not seen significant differences between hans and iodine in raw login mode. The surfing experience was in both cases very good.

With iodine in normal mode the performance degrades but we still found it usable for normal day-to-day activities (web browsing, email, instant messaging).
9 Conclusions and Future Work

During the development of Covert Droid we have observed certain obstacles in the way of developers like limited access to the device hardware or missing functionalities in the kernel. While Android proves to be a very productive and open platform for developing regular applications, special applications, like Covert Droid, are difficult to develop and are dependent on 3rd party enhancements to Android.

Given the popularity of Android devices we believe that Cover Droid could seriously affect WiFi Hotspot revenues in the future. Paid Wifi Hotspot owners need to be aware of covert channels applications and must employ detection and prevention mechanisms in their systems.

As future work we intend to do additional testing of the current codebase to improve the user experience. A simpler install procedure of the application is needed as most users will benefit from it. We have used busybox as a 3rd party tool to setup ip interfaces and install and delete routes. Ideally Covert Droid should not have 3rd party dependencies so that users will need just the main application package. The current autodetect function relies only on bandwidth measurements but future version will take into account other parameters such as latency or link loss. The modular application design allows developers to integrate new covert channel application into Covert Droid. Popular software like icmp or nstx could also be included.

Lastly Covert Droid will have a history feature that will associate a working tunnel configuration to a wifi network, so that the user will no longer have to perform the autodetect procedure each time that he connects to a previously visited WiFi network. This location aware feature will improve the user experience with Covert Droid.

In this report we presented a new Android application, Covert Droid, which enables the use of covert channels, ICMP and DNS, on the Android platform.

References


A Patches

A.1 iodine

```c
1  diff iodine/src/client.c iodine-0.6.0-rc1/src/client.c
2  36c36
3  < #include "nameser.h"
4  ---
5  > #include <arpa/nameser.h>
6  1
7  3d3d
8  < #include "nameser.h"
9  ---
10 > #include <arpa/nameser.h>
11  11c11
12  < #include "err.h"
13  ---
14 > #include <err.h>
15  2d2d
16  < /*#if !defined(WINDOWS32) && !(defined(BSD) && (BSD >= 199306)) && !defined(__GLIBC__)
17  ---
18 > #if !defined(WINDOWS32) && !(defined(BSD) && (BSD >= 199306)) && !defined(__GLIBC__)
19  19d19
20  < */
21  ---
22 >
23  diff iodine/src/common.c iodine-0.6.0-rc1/src/common.c
24  36d36
25  < #include "nameser.h"
26  ---
27 > #include <arpa/nameser.h>
28  34d34
29  < #include "err.h"
30  ---
31 > #include <err.h>
32  41d41
33  < /*
34  < #define vwarnx(fmt, args) ({ fprintf(stderr, "unifdef: "); vfprintf(stderr, fmt, args); fprintf(stderr, "\n"); })
35  < #define warnx(fmt, args...) fprintf(stderr, "unifdef: " fmt "\n", ## args)
36  < #define errx(exit_code, fmt, args...) ({ warnx(fmt, ## args); exit(exit_code); })
37  < #define err(exit_code, fmt, args...) errx(exit_code, fmt ": %s", ## args)
38  < #define warn(fmt, args...) fprintf(stderr, "unifdef: " fmt "\n", ## args)
39  < */
40 >
41 >
42 >
43 >
44 > #include <err.h>
45 > Only in iodine/src: err.h
46 > Only in iodine/src: find.sh
47 >
48  28d28
49  < #include "nameser.h"
50  ---
51 > #include <arpa/nameser.h>
52 > Only in iodine/src: nameser.h
53 >
54  20d20
55  < #include "iodined.h"
56  ---
57 > #include <iodined.h>
58 > Only in iodine/src: iodined.h
59  41d41
60  < #include "err.h"
61  ---
62 > #include <err.h>
63  44d44
64  < Only in iodine/src: err.h
65 > Only in iodine/src: err.h
66  47d47
67  < "busybox ifconfig %s %s %s netmask %s",
68  ---
69 > "busybox ifconfig %s %s %s netmask %s",
70 > "busybox route add %s/\d %s",
71 > "busybox ifconfig %s add %s",
72 > "busybox ifconfig %s add %s",
73 > "busybox ifconfig %s add %s",
74 > "busybox ifconfig %s add %s",
```

Code Snippet 22: Differences between the ported and original iodine
A.2 hans

```cpp
#include <unistd.h>
```

---

```
syslog(LOG_ERR, "gethostbyname: %s", hstrerror(h_errno));
```

---

```
syslog(LOG_ERR, "gethostbyname: ");
```