DEFENSIVE
LAB AGENCY

FinSpy Android technical analysis

- Amnesty International -

August 2020

Defensive Lab Agency
31 avenue de Ségur
75007 Paris

contact@defensive-lab.agency

Executive summary

By analyzing the sample we found what we suspect to be a new version of the FinFisher's malware FinSpy for Android. Even though the malware behavior and capabilities seem to be the same as what it has already been described in the past, this version goes a step further to hide the malware configuration and its capabilities. This new version we named DexDen has very likely been released between May 2017 and October 2019.

Command and control server associated to the malware configuration is still alive by the time we wrote this report.

In terms of capabilities, the sample we have analyzed is meant to exfiltrate SIM card information, SMS log, call log, calendar events, address book, messages and files from 12 popular messenger applications and to track victim's location.

This report provides details on how strings are obfuscated, how the communication protocol has evolved and how the extraction of three technical aspects of the malware can give insights on the malware code-base evolution.

Threat:

FinSpy for Android

Key findings:

New version of FinSpy

Alive C2s

Defensive Lab Agency
31 avenue de Ségur
75007 Paris

contact@defensive-lab.agency

Table of Contents

1. Overview....4
2. IOC....5
3. A suspected new FinSpy version....6
    1. Configuration storage....6
    2. String obfuscation....6
    3. Local socket address generation....7
    4. Unknown TLV types....9
    5. Conclusion....10
4. Sample behavioral analysis....11
    1. Configuration parsing....11
    2. Emergency reconfiguration....12
    3. Privilege escalation....13
    4. Communication with C2....14
    5. Self destruction capability....14
    6. Data collection mechanism....15
    7. Data collected and sent by default....15
    8. Messenger applications data exfiltration....15
    9. Call log exfiltration....16
    10. SMS log exfiltration....16
    11. Calendar events exfiltration....16
    12. Address book exfiltration....16
    13. SIM information exfiltration....17
    14. Location tracking....18

1. Overview

This report focuses on the analysis of the sample described below.

- File Name `WIFI.apk`
- Size `2.87MB`
- MD5 `79ba96848428337e685e10b06ccc1c89`
- SHA1 `51b31827c1d961ced142a3c5f3efa2b389f9c5ad`
- SHA256 `854774a198db490a1ae9f06d5da5fe6a1f683bf3d7186e56776516f982d41ad3`

For this analysis we use the following tools:

- Aether (https://defensive-lab.agency/en/products/aether/) to analyze CFG
- Javalang (https://github.com/c2nes/javalang) to parse the Java code
- Smalisca (https://github.com/U039b/smalisca) to analyze the Smali code
- Yara (https://virustotal.github.io/yara/) to detect FinSpy variants
- FinSpy tools (https://github.com/devio/FinSpy-Tools/) to parse the FinSpy configuration

We share the following Python scripts with the community:

- `java_parser.py` to extract obfuscated from Java code
- `string_decoder.py` to decode obfuscated strings

We share the following documents with the community:

- the report you are reading now
- `table.ods` containing TLV types and decoded strings
- `FinSpy.yar` Yara rules detecting FinSpy variants

2. IOC

Sample file	
File Name	WIFI.apk
Size	2.87MB
MD5	79ba96848428337e685e10b06ccc1c89
SHA1	51b31827c1d961ced142a3c5f3efa2b389f9c5ad
SHA256	854774a198db490a1ae9f06d5da5fe6a1f683bf3d7186e56776516f982d41ad3

Android application	
App Name	wifi
Package	org.xmlpush.v3
Main Activity	org.xmlpush.v3.StartVersion

Sample certificate	
Subject	CN='MITAS Ltd.'
Signature Alg,	rsassa_pkcs1v15
Valid From	2017-05-27 07
Valid To	2023-05-26 07
Issuer	CN='MITAS Ltd.'
Serial Number	0x4d53ca56
Hash Alg.	sha256
MD5	b99ac605872a55e609854176413e603c
SHA1	7c6e4f2e84ebaa8d25040f63d840e14f6f822125
SHA256	8052584eacfd199602b348ef60e20c246ec929d62bc5b85fd0e60ba3205b05a2

3. A suspected new FinSpy version

FinSpy capabilities and technical aspects are widely documented online. In this section we focus on what we suspect to be clues of a new version of FinSpy for Android.

To do so, we investigate on the following parameters:

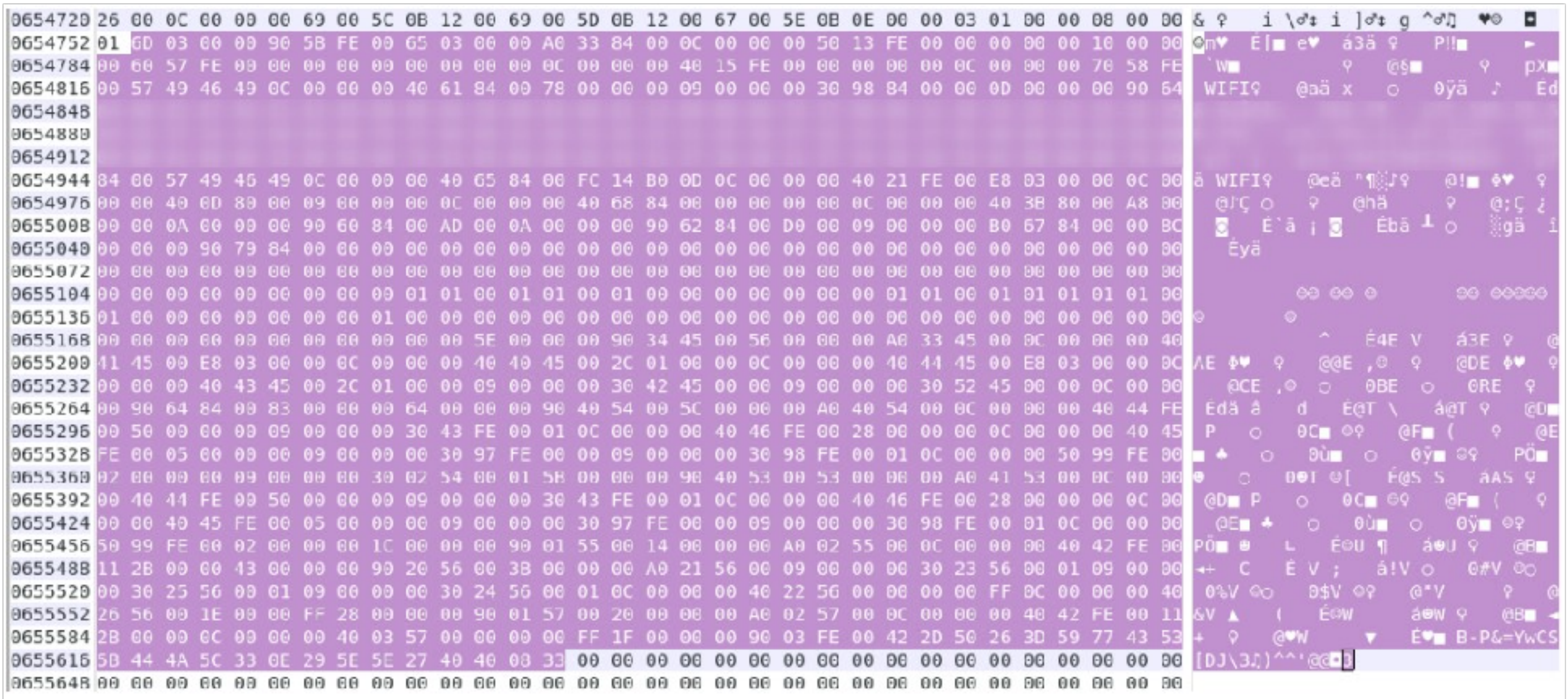
- location of the FinSpy configuration
- string obfuscation
- local socket address generation
- unknown TLV types

1. Configuration storage

As far as we know, FinSpy stores its configuration into APK metadata. It was well documented and extraction tools are available online:

- https://github.com/devio/FinSpy-Tools
- https://github.com/SpiderLabs/malware-analysis/blob/master/Ruby/FinSpy/

The sample we investigate on shows that the FinSpy configuration is stored into the DEX file.



```
0654720 26 00 0C 00 00 00 69 00 5C 0B 12 00 69 00 5D 0B 12 00 67 00 5E 0B 0E 00 00 03 01 00 00 08 00 00
0654752 01 6D 03 00 00 90 5B FE 00 65 03 00 00 A0 33 84 00 0C 00 00 00 50 13 FE 00 00 00 00 00 10 00 00
0654784 00 60 57 FE 00 00 00 00 00 00 00 00 00 0C 00 00 00 40 15 FE 00 00 00 00 00 0C 00 00 00 70 58 FE
0654816 00 57 49 46 49 0C 00 00 00 40 61 84 00 78 00 00 00 09 00 00 00 30 98 84 00 00 0D 00 00 00 90 64
0654848 [redacted]
0654880
0654912
0654944 84 00 57 49 46 49 0C 00 00 00 40 65 84 00 FC 14 B0 0D 0C 00 00 00 40 21 FE 00 E8 03 00 00 0C 00
0654976 00 00 40 0D 80 00 09 00 00 00 0C 00 00 00 40 68 84 00 00 00 00 00 0C 00 00 00 40 3B 80 00 A8 00
0655008 00 00 0A 00 00 00 90 60 84 00 AD 00 0A 00 00 00 90 62 84 00 D0 00 09 00 00 00 B0 67 84 00 00 BC
0655040 00 00 00 90 79 84 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0655072 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0655104 00 00 00 00 00 00 00 00 00 01 01 00 01 01 00 01 00 00 00 00 00 00 00 01 01 00 01 01 01 01 01 00
0655136 01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0655168 00 00 00 00 00 00 00 00 00 00 00 5E 00 00 00 90 34 45 00 56 00 00 00 A0 33 45 00 0C 00 00 00 40
0655200 41 45 00 E8 03 00 00 0C 00 00 00 40 40 45 00 2C 01 00 00 0C 00 00 00 40 44 45 00 E8 03 00 00 0C
0655232 00 00 00 40 43 45 00 2C 01 00 00 09 00 00 00 30 42 45 00 00 09 00 00 00 30 52 45 00 00 0C 00 00
0655264 00 90 64 84 00 83 00 00 00 64 00 00 00 90 40 54 00 5C 00 00 00 A0 40 54 00 0C 00 00 00 40 44 FE
0655296 00 50 00 00 00 09 00 00 00 30 43 FE 00 01 0C 00 00 00 40 46 FE 00 28 00 00 00 0C 00 00 00 40 45
0655328 FE 00 05 00 00 00 09 00 00 00 30 97 FE 00 00 09 00 00 00 30 98 FE 00 01 0C 00 00 00 50 99 FE 00
0655360 07 00 00 00 09 00 00 00 30 02 54 00 01 5B 00 00 00 90 40 53 00 53 00 00 00 A0 41 53 00 0C 00 00
0655392 00 40 44 FE 00 50 00 00 00 09 00 00 00 30 43 FE 00 01 0C 00 00 00 40 46 FE 00 28 00 00 00 0C 00
0655424 00 00 40 45 FE 00 05 00 00 00 09 00 00 00 30 97 FE 00 00 09 00 00 00 30 98 FE 00 01 0C 00 00 00
0655456 50 99 FE 00 02 00 00 00 1C 00 00 00 90 01 55 00 14 00 00 00 A0 02 55 00 0C 00 00 00 40 42 FE 00
0655488 11 2B 00 00 43 00 00 00 90 20 56 00 3B 00 00 00 A0 21 56 00 09 00 00 00 30 23 56 00 01 09 00 00
0655520 00 30 25 56 00 01 09 00 00 00 30 24 56 00 01 0C 00 00 00 40 22 56 00 00 00 00 FF 0C 00 00 00 40
0655552 26 56 00 1E 00 00 FF 28 00 00 00 90 01 57 00 20 00 00 00 A0 02 57 00 0C 00 00 00 40 42 FE 00 11
0655584 2B 00 00 0C 00 00 00 40 03 57 00 00 00 00 FF 1F 00 00 00 90 03 FE 00 42 2D 50 26 3D 59 77 43 53
0655616 5B 44 4A 5C 33 0E 29 5E 5E 27 40 40 08 33 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0655648 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 1: FinSpy configuration stored into the DEX

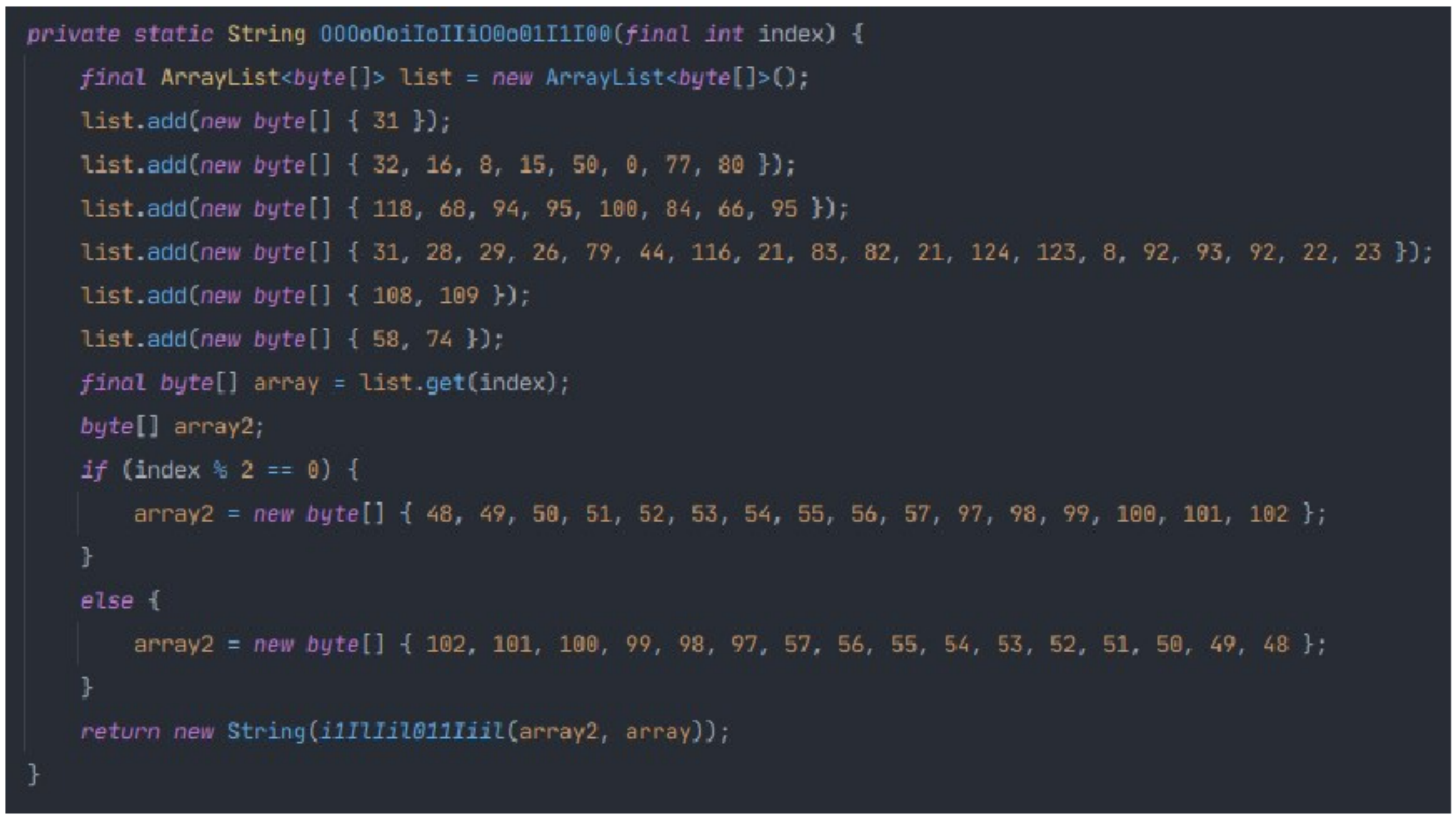
Even if existing extraction tools failed to extract the configuration from the DEX, parsing tools succeeded to parse it. The structure of the configuration remains the same, only its storage location has changed.

We name this FinSpy variant **DexDen**.

2. String obfuscation

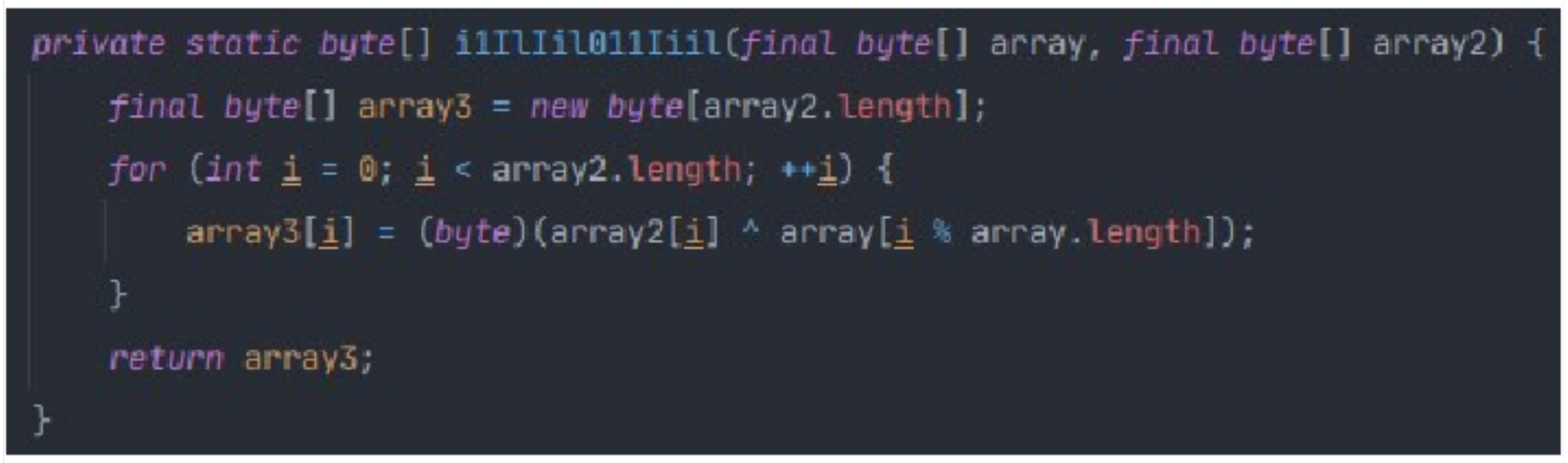
As far as we know, FinSpy strings defined in its code are not obfuscated. The sample we analyze is different, all Java strings are obfuscated. Each Java class using strings implements the following 2 Java methods:

- `String OOOoOoiIoIIiOOo01I1IOO(final int index)` returning the obfuscated string as bytes at the given index.
- `byte[] i1IlIil011Iiil(final byte[] array, final byte[] array2)` decoding an obfuscated string.



```java
private static String OOOoOoiIoIIiOOo01I1IOO(final int index) {
    final ArrayList<byte[]> list = new ArrayList<byte[]>();
    list.add(new byte[] { 31 });
    list.add(new byte[] { 32, 16, 8, 15, 50, 0, 77, 80 });
    list.add(new byte[] { 118, 68, 94, 95, 100, 84, 66, 95 });
    list.add(new byte[] { 31, 28, 29, 26, 79, 44, 116, 21, 83, 82, 21, 124, 123, 8, 92, 93, 92, 22, 23 });
    list.add(new byte[] { 108, 109 });
    list.add(new byte[] { 58, 74 });
    final byte[] array = list.get(index);
    byte[] array2;
    if (index % 2 == 0) {
        array2 = new byte[] { 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 97, 98, 99, 100, 101, 102 };
    }
    else {
        array2 = new byte[] { 102, 101, 100, 99, 98, 97, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48 };
    }
    return new String(i1IlIil011Iiil(array2, array));
}
```

Figure 2: Example of obfuscated strings and the two decoding TippyPads



```java
private static byte[] i1IlIil011Iiil(final byte[] array, final byte[] array2) {
    final byte[] array3 = new byte[array2.length];
    for (int i = 0; i < array2.length; ++i) {
        array3[i] = (byte)(array2[i] ^ array[i % array.length]);
    }
    return array3;
}
```

Figure 3: Example of the Java method decoding obfuscated strings

Strings are decoded by XORing of the obfuscated one with one of the two pads. The pad is selected according to the index mod 2. Pads are the same for all Java classes using strings:

- `0123456789abcdef`
- `fedcba9876543210`

We have developed a Python script parsing the entire Java code to retrieve obfuscated strings `java_parser.py` and one to decode them `string_decoder.py`.

We denote this kind of string obfuscation TippyPad for short.

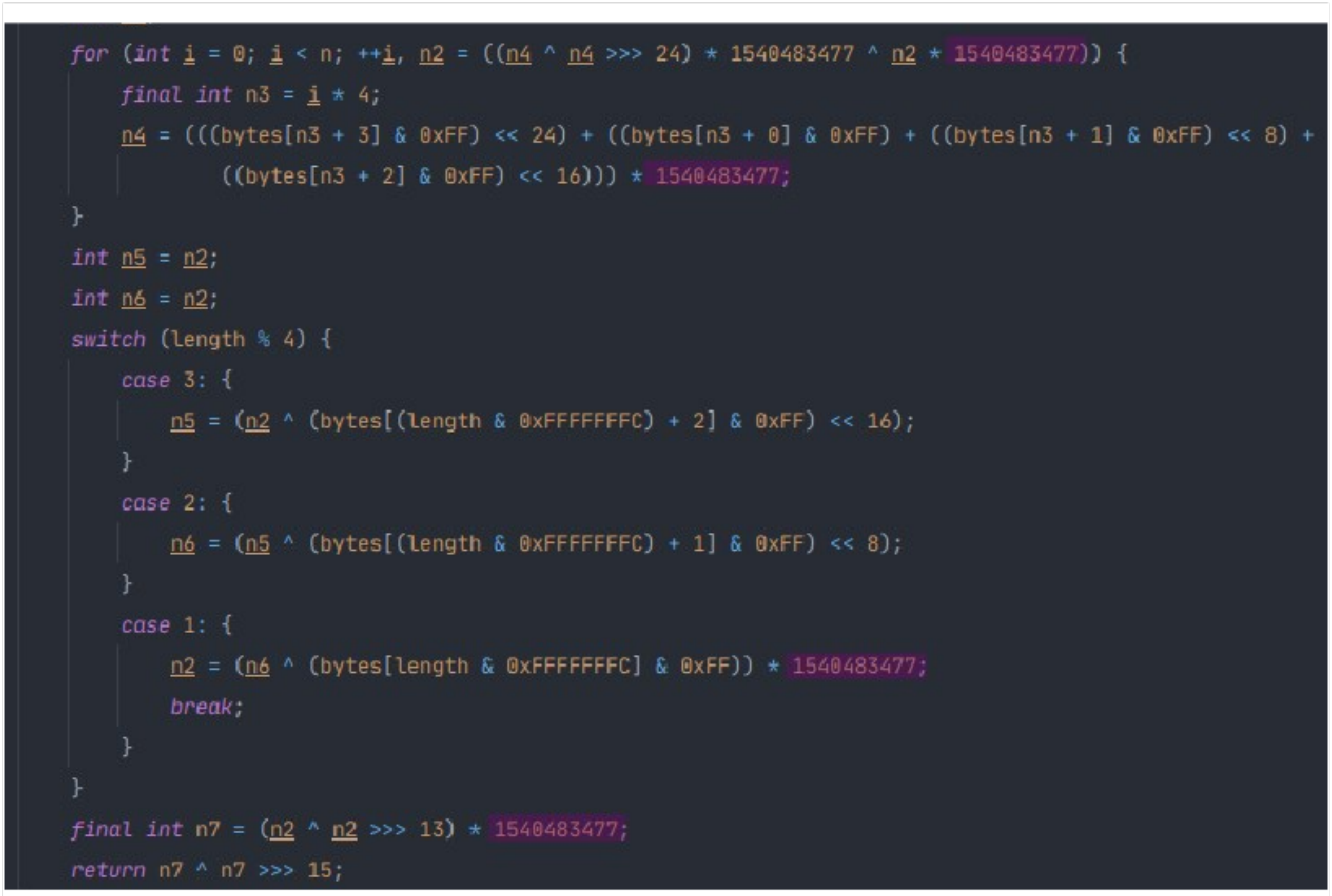
3. Local socket address generation

FinSpy uses Unix socket to communicate between threads. The local socket address is generated by hashing the values of the following system properties:

- `ro.product.model`

- `ro.product.brand`
- `ro.product.name`
- `ro.product.device`
- `ro.product.manufacturer`
- `ro.build.fingerprint`

An utility method meant to encode data and generate local socket address uses the timestamp `1540483477` corresponding to `Thu 25 October 2018 16:04:37 UTC`. Java method generating local socket address is listed below.



```java
for (int i = 0; i < n; ++i, n2 = ((n4 ^ n4 >>> 24) * 1540483477 ^ n2 * 1540483477)) {
    final int n3 = i * 4;
    n4 = (((bytes[n3 + 3] & 0xFF) << 24) + ((bytes[n3 + 0] & 0xFF) + ((bytes[n3 + 1] & 0xFF) << 8) +
            ((bytes[n3 + 2] & 0xFF) << 16))) * 1540483477;
}
int n5 = n2;
int n6 = n2;
switch (length % 4) {
    case 3: {
        n5 = (n2 ^ (bytes[(length & 0xFFFFFFFC) + 2] & 0xFF) << 16);
    }
    case 2: {
        n6 = (n5 ^ (bytes[(length & 0xFFFFFFFC) + 1] & 0xFF) << 8);
    }
    case 1: {
        n2 = (n6 ^ (bytes[length & 0xFFFFFFFC] & 0xFF)) * 1540483477;
        break;
    }
}
final int n7 = (n2 ^ n2 >>> 13) * 1540483477;
return n7 ^ n7 >>> 15;
```

Figure 4: Java method generating the local socket address

We denote this kind of address generation TippyTime for short.

4. Unknown TLV types

After leaks about FinFisher and FinSpy, community has reversed the different TLV values used in data marshaling/unmarshaling to ensure a common data format between C2s and implants. These values are available online:

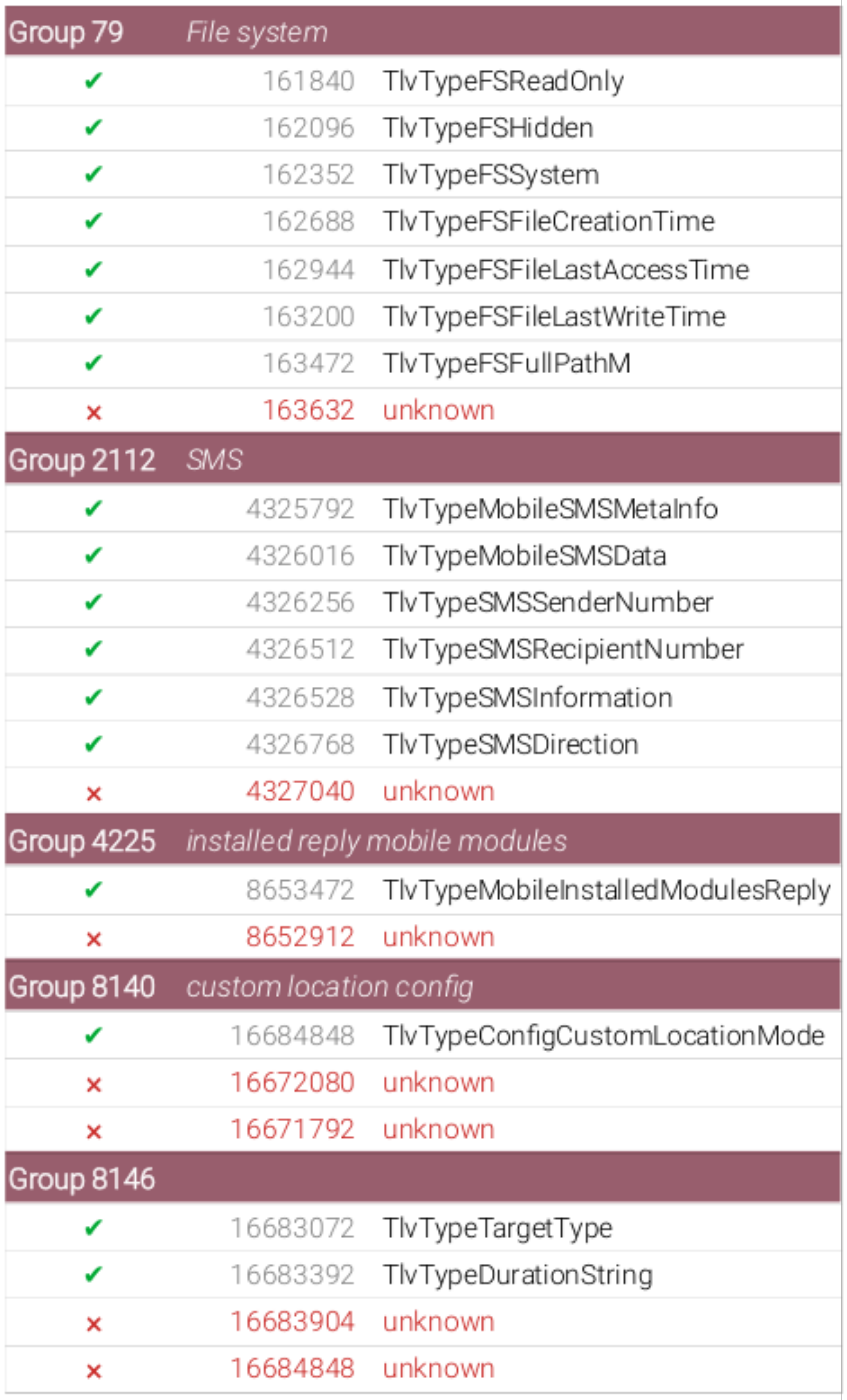
https://github.com/devio/FinSpy-Tools/blob/master/Android/finspyCfgParse.py

The FinSpy version we analyze seems to be using unknown TLV values. To get some meaning about the different unknown TLV values, we reversed existing values. We were able to detect semantic groups based on the binary representation of these values.

The Python script we developed recovers groups based on existing values. Then parses the sample Smali code to extract unknown TLV values. We used a patched version of Smalisca (https://github.com/U039b/smalisca) to do so.

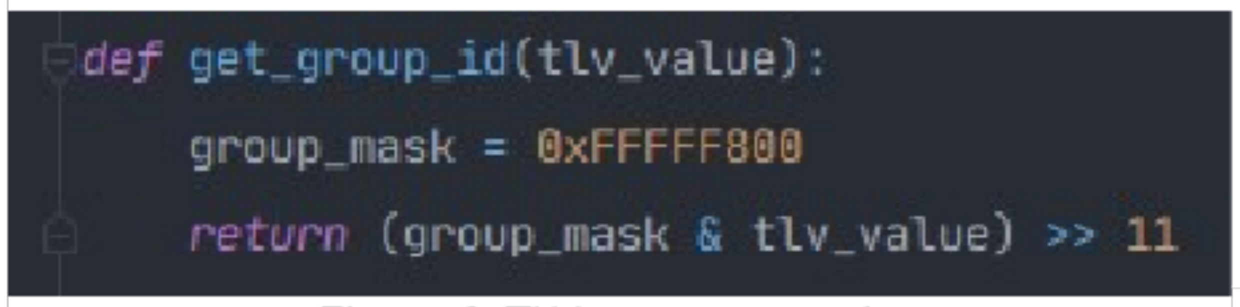
We have extracted the following suspected unknown TLV values. The entire list of TLV and groups is available in the GitHub repository.

To determine the group the TLV value belongs to just mask that value with `0xFFFFF800`.



Group 79	*File system*	
✓	161840	TlvTypeFSReadOnly
✓	162096	TlvTypeFSHidden
✓	162352	TlvTypeFSSystem
✓	162688	TlvTypeFSFileCreationTime
✓	162944	TlvTypeFSFileLastAccessTime
✓	163200	TlvTypeFSFileLastWriteTime
✓	163472	TlvTypeFSFullPathM
×	163632	unknown
Group 2112	*SMS*	
✓	4325792	TlvTypeMobileSMSMetaInfo
✓	4326016	TlvTypeMobileSMSData
✓	4326256	TlvTypeSMSSenderNumber
✓	4326512	TlvTypeSMSRecipientNumber
✓	4326528	TlvTypeSMSInformation
✓	4326768	TlvTypeSMSDirection
×	4327040	unknown
Group 4225	*installed reply mobile modules*	
✓	8653472	TlvTypeMobileInstalledModulesReply
×	8652912	unknown
Group 8140	*custom location config*	
✓	16684848	TlvTypeConfigCustomLocationMode
×	16672080	unknown
×	16671792	unknown
Group 8146		
✓	16683072	TlvTypeTargetType
✓	16683392	TlvTypeDurationString
×	16683904	unknown
×	16684848	unknown

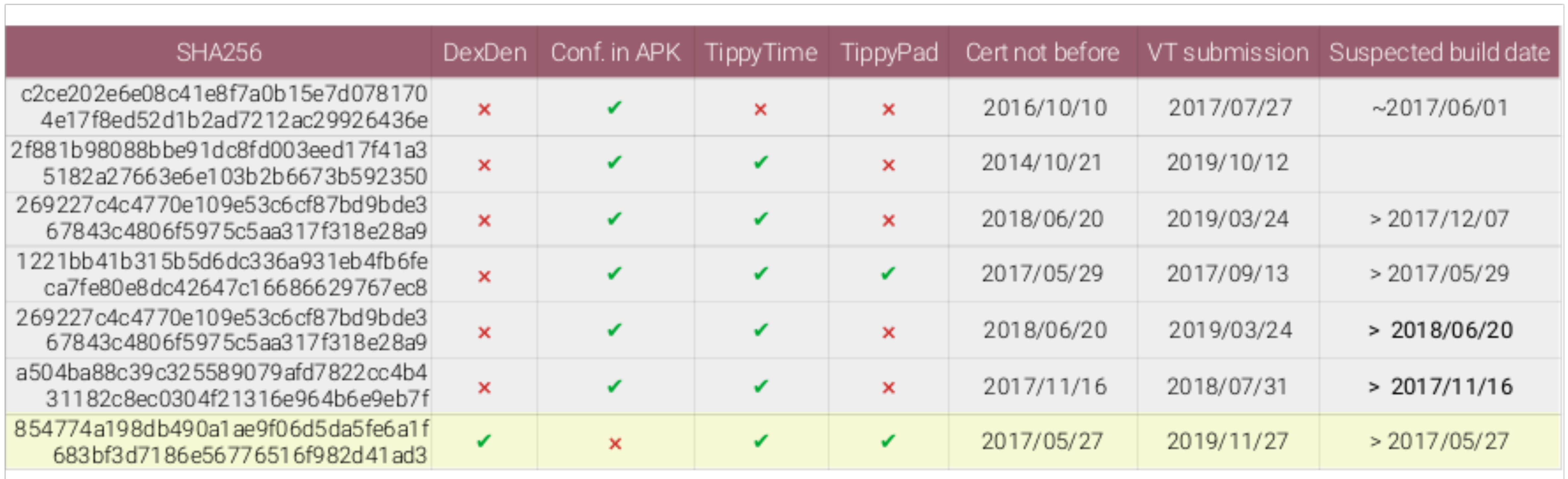
Figure 5: Few suspected new TLV values



```python
def get_group_id(tlv_value):
    group_mask = 0xFFFFF800
    return (group_mask & tlv_value) >> 11
```

Figure 6: TLV group extraction

5. Conclusion



SHA256	DexDen	Conf. in APK	TippyTime	TippyPad	Cert not before	VT submission	Suspected build date
c2ce202e6e08c41e8f7a0b15e7d078170 4e17f8ed52d1b2ad7212ac29926436e	×	✓	×	×	2016/10/10	2017/07/27	~2017/06/01
2f881b98088bbe91dc8fd003eed17f41a3 5182a27663e6e103b2b6673b592350	×	✓	✓	×	2014/10/21	2019/10/12	
269227c4c4770e109e53c6cf87bd9bde3 67843c4806f5975c5aa317f318e28a9	×	✓	✓	×	2018/06/20	2019/03/24	> 2017/12/07
1221bb41b315b5d6dc336a931eb4fb6fe ca7fe80e8dc42647c16686629767ec8	×	✓	✓	✓	2017/05/29	2017/09/13	> 2017/05/29
269227c4c4770e109e53c6cf87bd9bde3 67843c4806f5975c5aa317f318e28a9	×	✓	✓	×	2018/06/20	2019/03/24	**> 2018/06/20**
a504ba88c39c325589079afd7822cc4b4 31182c8ec0304f21316e964b6e9eb7f	×	✓	✓	×	2017/11/16	2018/07/31	**> 2017/11/16**
854774a198db490a1ae9f06d5da5fe6a1f 683bf3d7186e56776516f982d41ad3	✓	×	✓	✓	2017/05/27	2019/11/27	> 2017/05/27

Figure 7: FinSpy samples from CCC report & the sample we analyze (yellow row)

Our analysis based on 3 different parameters: configuration location, string obfuscation and local socket address generation tends to demonstrate that the sample we have analyzed is (as far as we know) the only known FinSpy for Android sample storing its configuration directly into the DEX file (DexDen). Reports FinSpy Dokumentation by Thorsten Schröder & Linus Neumann - CCC (Jan. 2020), AccessNow: FinFisher changes tactics to hooks critics by AccessNow (May 2018) and Hacking FinSpy by Sophos (2015) explain how the FinSpy configuration is stored in the APK file metadata. A retro-hunt on VT has found 0 samples (our sample excluded) storing the configuration the DEX. Changing the configuration location is a strong structural change indicating a suspected new version of FinSpy for Android.

A trend emerges when we focus on how the local socket address is generated and how strings are obfuscated. Old samples do not use a "magic" timestamp (TippyTime) in the generation algorithm nor pad-obfuscated strings (TippyPad). By analyzing briefly samples shared by CCC, we observed that since 2017, FinSpy seems to use TippyTime. However, only one sample use TippyPad string obfuscation.

Regarding unknown or undocumented TLV types, we have no clue indicating they are new or not since we have not analyzed other samples in deep and no unknown TLV types have ever been reported.

4. Sample behavioral analysis

The sample we analyze is heavily obfuscated:

- strings are encoded at the class level;
- Java methods are obfuscated (shortened);
- control flow graph is broken by the heavy use of threads and IPC;
- dummy calls are inserted between almost all the "useful" ones.

To analyze the sample, we firstly do a fast behavioral recon with Aether by extracting control flow graphs in which:

- sinks are Java methods of interest;
- sources are detected entry-points (*i.e.* services, threads, activities, …).

Secondly we extract TLV types involved in the different control flow graphs and then correlate the meaning of TLV with the meaning of actions done on the OS.

1. Configuration parsing

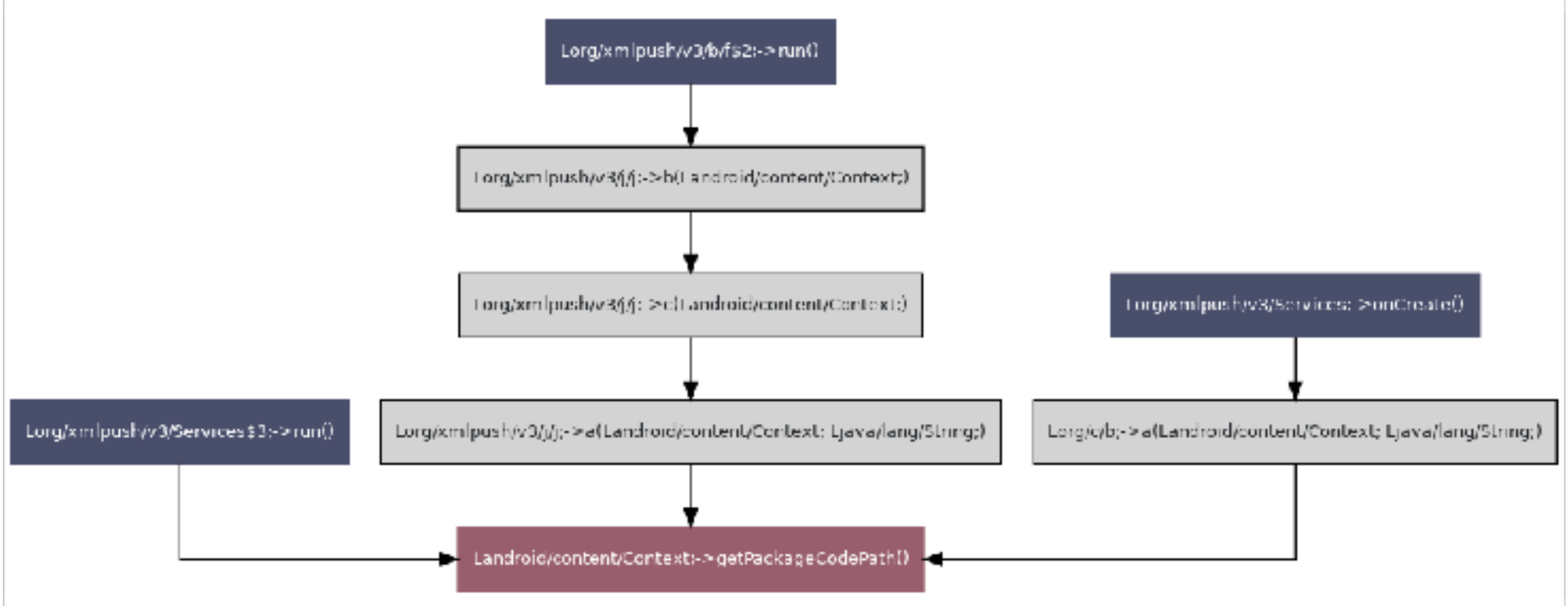

Lorg/xmlpush/v3/b/fs2;->run()
Lorg/xmlpush/v3/j/j;->b(Landroid/content/Context;)
Lorg/xmlpush/v3/j/j;->c(Landroid/content/Context;)
Lorg/xmlpush/v3/Services;->onCreate()
Lorg/xmlpush/v3/Services$3;->run()
Lorg/xmlpush/v3/j/j;->a(Landroid/content/Context; Ljava/lang/String;)
Lorg/c/b;->a(Landroid/content/Context; Ljava/lang/String;)
Landroid/content/Context;->getPackageCodePath()

Figure 8: CFG locating the DEX file

As we have seen before FinSpy stores its configuration into the DEX file. Thus, the first step for it is to locate the DEX file. On Android, the Java method `android.content.Context.getPackageSourceDir()` returns the location of the APK which contains the original DEX (not the optimized one).

Once located, the DEX file is copied at a randomly generated path into the cache directory. Once copied, the DEX loaded (or self-loaded since it is loaded by itself) using the Java method `dalvik.system.DexClassLoader.loadClass()`.

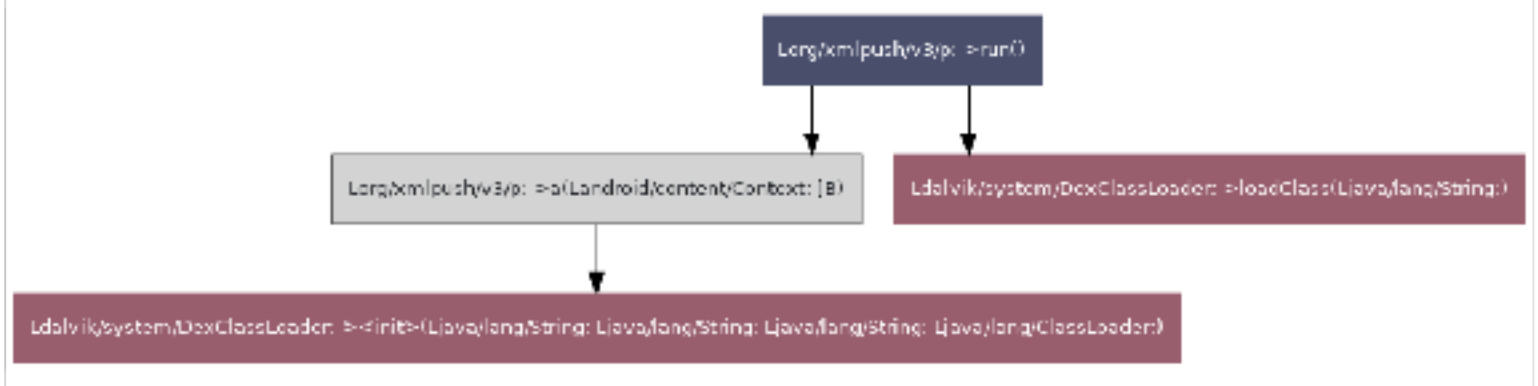

Lorg/xmlpush/v3/p;->run()
Lorg/xmlpush/v3/p;->a(Landroid/content/Context; [B)
Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)
Ldalvik/system/DexClassLoader;-><init>(Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/ClassLoader;)

Figure 9: CFG loading the DEX file

Finally, FinSpy parses its configuration stored into the loaded DEX using a large *switch-case* statement.

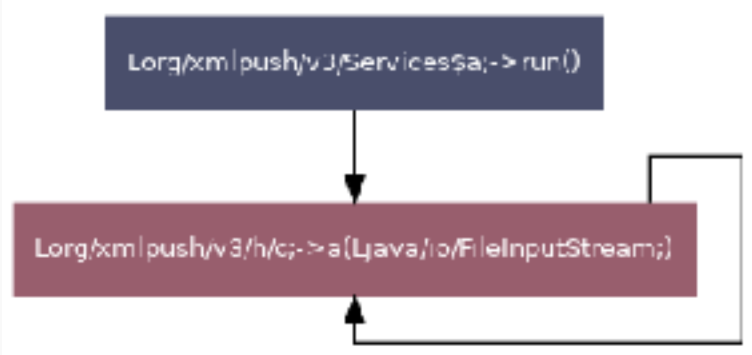

Lorg/xmlpush/v3/Services$a;->run()
Lorg/xmlpush/v3/h/c;->a(Ljava/io/FileInputStream;)

Figure 10: CFG parsing the configuration

The configuration stored into the current sample looks like:

- `TlvTypeMobileTargetID` = "WIFI"
- `TlvTypeMobileTargetHeartbeatInterval` = 120
- `TlvTypeMobileTargetPositioning` = b'\x82\x87\x86\x81\x83'
- `TlvTypeConfigTargetProxy` = "[redacted]"
- `TlvTypeConfigTargetProxy` = "[redacted]"
- `TlvTypeConfigTargetPort` = [redacted]
- `TlvTypeConfigSMSPhoneNumber` = "[redacted]"
- `TlvTypeMobileTrojanID` = "WIFI"
- `TlvTypeMobileTrojanUID` = b'\xfc\x14\xb0\r'
- `TlvTypeUserID` = 1000
- `TlvTypeTrojanMaxInfections` = 9
- `TlvTypeConfigMobileAutoRemovalDateTime` = Thu Jan 1 01:00:00 1970
- `TlvTypeConfigAutoRemovalIfNoProxy` = 168
- `TlvTypeMobileTargetHeartbeatEvents` = 173
- `TlvTypeMobileTargetHeartbeatRestrictions` = b'\xd0\x00'
- `TlvTypeMobileTrackingDistance` = 1000
- `TlvTypeMobileTrackingTimeInterval` = 300
- `TlvTypeInstalledModules` =
  - Logging: Off
  - Spy Call: Off
  - Call Interception: Off
  - SMS: On
  - Address Book: On
  - Tracking: On
  - Phone Logs: On

Note: Trojan UID is the AES sub-key used to encrypt/decrypt payloads exchange with the C2.

2. Emergency reconfiguration

FinSpy can be reconfigured by SMS, the Java method `org.xmlpush.v3.q.c.a()` is dedicated to that. FinSpy uses a lot of threads, probably not for performance purposes but to circumvent automatic reverse engineering of CFG. The following CFG shows the break in the CFG.

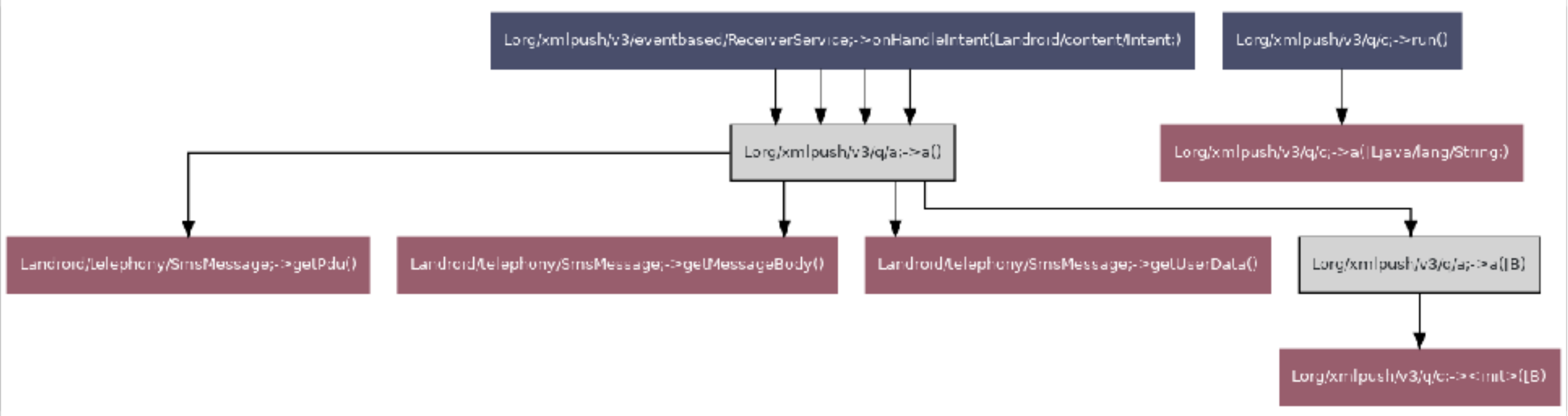

Lorg/xmlpush/v3/eventbased/ReceiverService;->onHandleIntent(Landroid/content/Intent;)
Lorg/xmlpush/v3/q/c;->run()
Lorg/xmlpush/v3/q/a;->a()
Lorg/xmlpush/v3/q/c;->a(Ljava/lang/String;)
Landroid/telephony/SmsMessage;->getPdu()
Landroid/telephony/SmsMessage;->getMessageBody()
Landroid/telephony/SmsMessage;->getUserData()
Lorg/xmlpush/v3/c/a;->a([B)
Lorg/xmlpush/v3/q/c;-><init>([B)

Figure 11: FinSpy reconfiguration

When an SMS corresponding to `TlvTypeMobileTargetEmergencyConfig` is received, FinSpy reconfigures itself by parsing the SMS payload.

The following attributes can be reconfigured:

- `TlvTypeConfigTargetPort`: port for C2 proxy
- `TlvTypeConfigSMSPhoneNumber`: phone number for SMS based C2 communications
- `TlvTypeMobileTrojanID`: unknown purpose
- `TlvTypeMobileTrojanUID`: unknown purpose
- `TlvTypeUserID`: unknown purpose
- `TlvTypeTrojanMaxInfections`: unknown purpose
- `TlvTypeConfigMobileAutoRemovalDateTime`: implant self-destruct past this date
- `TlvTypeConfigAutoRemovalIfNoProxy`: implant self-destruct if C2 proxy is unavailable
- `TlvTypeMobileTargetHeartbeatRestrictions`: conditions to avoid callbacks
- `TlvTypeMobileTargetHeartbeatEvents`: events to trigger callbacks to the C2
- `TlvTypeMobileTargetLocationChangedRange`: trigger updates based on location changes
- `TlvTypeInstalledModules`: list of implant features and their configuration (SMS log, call log, etc.)
- and other unknown parameters

3. Privilege escalation

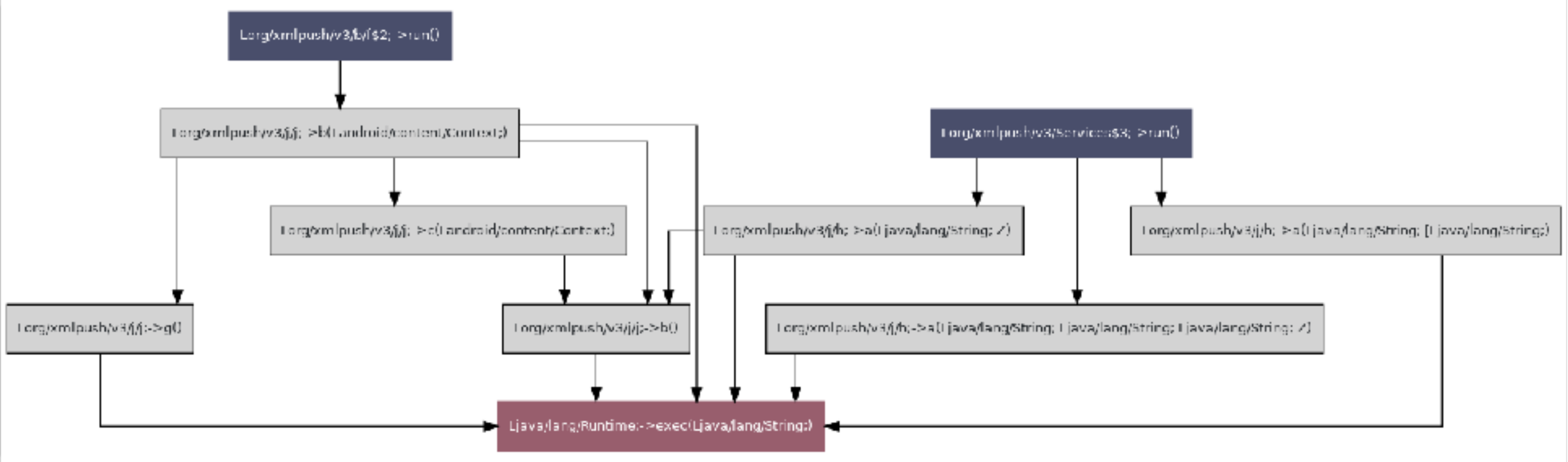

Lorg/xmlpush/v3/b/$2;->run()
Lorg/xmlpush/v3/j/j;->b(Landroid/content/Context;)
Lorg/xmlpush/v3/Services$3;->run()
Lorg/xmlpush/v3/j/j;->c(Landroid/content/Context;)
Lorg/xmlpush/v3/j/h;->a(Ljava/lang/String; Z)
Lorg/xmlpush/v3/j/h;->a(Ljava/lang/String; [Ljava/lang/String;)
Lorg/xmlpush/v3/j/j;->g()
Lorg/xmlpush/v3/j/j;->b()
Lorg/xmlpush/v3/j/h;->a(Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Z)
Ljava/lang/Runtime;->exec(Ljava/lang/String;)

Figure 12: Runtime command execution

FinSpy needs super user privileges to do things like access data of other applications. When started, the implant checks if `su` is available and then check if the user id is `0`. We found no evidence of vulnerability exploitation (DirtyCow or SELinux abuse) like the ones mentioned in other publicly available reports. We did not find ELF hidden into the APK, DEX or into natives libraries packaged in the APK.

Either we have missed something or this sample is tailored to be implanted after exploitation.

4. Communication with C2

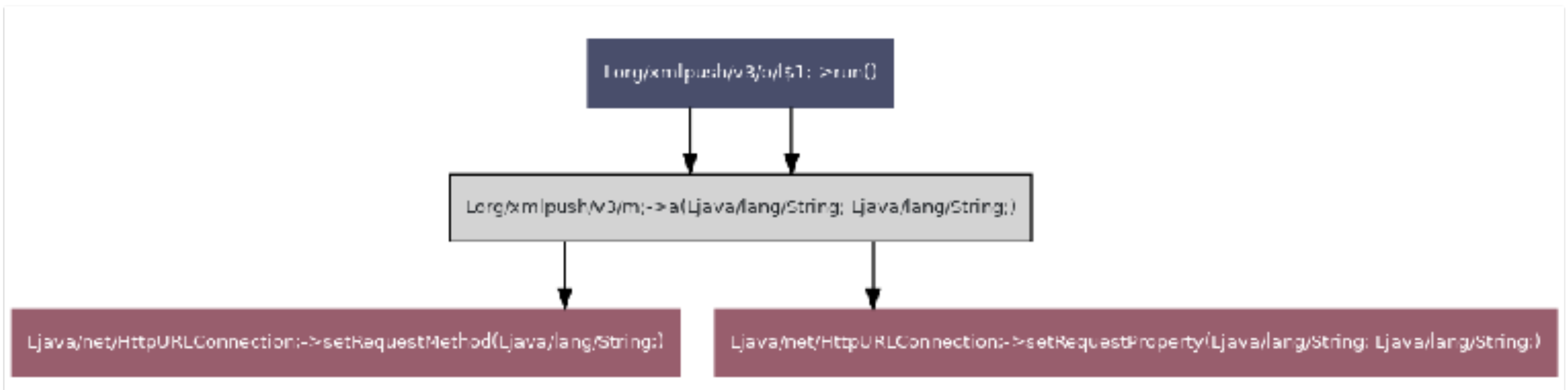

Lorg/xmlpush/v3/o/l$1;->run()
Lorg/xmlpush/v3/m;->a(Ljava/lang/String; Ljava/lang/String;)
Ljava/net/HttpURLConnection;->setRequestMethod(Ljava/lang/String;)
Ljava/net/HttpURLConnection;->setRequestProperty(Ljava/lang/String; Ljava/lang/String;)

Figure 13: SMS based exfiltration

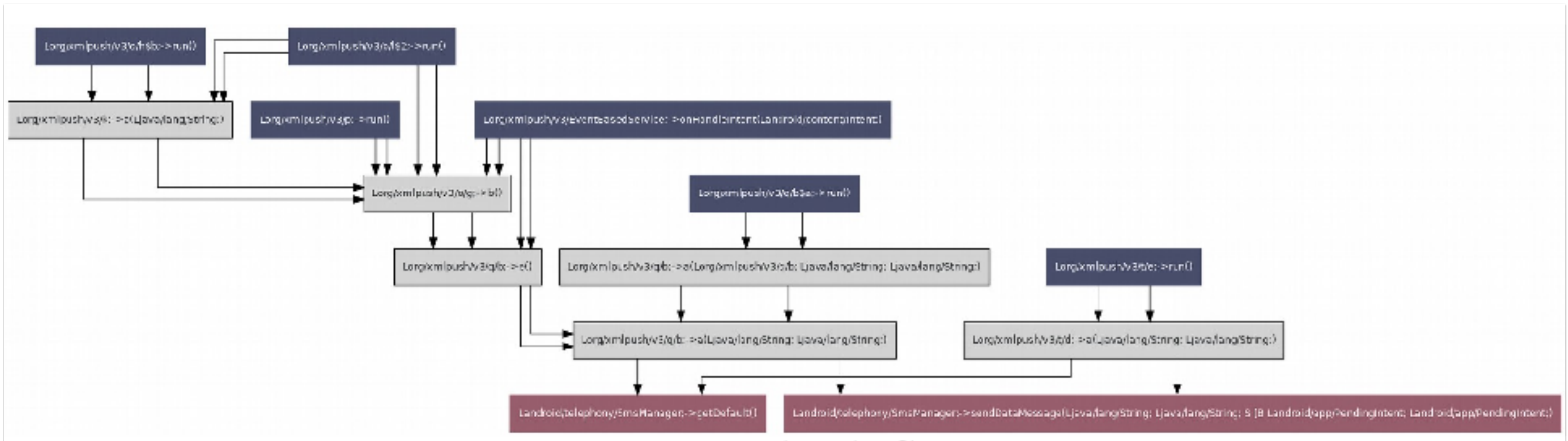

Lorg/xmlpush/v3/o/h$b;->run()
Lorg/xmlpush/v3/o/l$2;->run()
Lorg/xmlpush/v3/k;->a(Ljava/lang/String;)
Lorg/xmlpush/v3/g;->run()
Lorg/xmlpush/v3/EventBasedService;->onHandleIntent(Landroid/content/Intent;)
Lorg/xmlpush/v3/o/g;->b()
Lorg/xmlpush/v3/o/b$a;->run()
Lorg/xmlpush/v3/q/b;->c()
Lorg/xmlpush/v3/q/b;->a(Lorg/xmlpush/v3/o/b; Ljava/lang/String; Ljava/lang/String;)
Lorg/xmlpush/v3/b/e;->run()
Lorg/xmlpush/v3/q/b;->a(Ljava/lang/String; Ljava/lang/String;)
Lorg/xmlpush/v3/b/d;->a(Ljava/lang/String; Ljava/lang/String;)
Landroid/telephony/SmsManager;->getDefault()
Landroid/telephony/SmsManager;->sendDataMessage(Ljava/lang/String; Ljava/lang/String; S [B Landroid/app/PendingIntent; Landroid/app/PendingIntent;)

Figure 14: HTTP based exfiltration

The implant can use both SMS and HTTP requests to send collected data to the command and control server. Both SMS and HTTP communications use the same marshaling schema based on TLV types describing data. Payload are encrypted before being sent. The encryption mechanism is the same as the one described in AccessNow: FinFisher changes tactics to hooks critics by AccessNow.

5. Self destruction capability

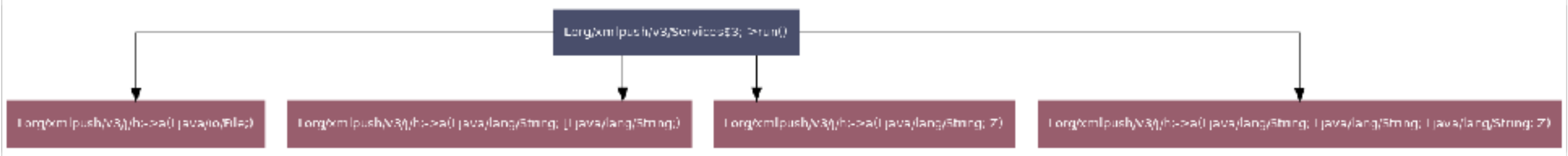

Lorg/xmlpush/v3/Services$3;->run()
Lorg/xmlpush/v3/j/h;->a(Ljava/io/File;)
Lorg/xmlpush/v3/j/h;->a(Ljava/lang/String; [Ljava/lang/String;)
Lorg/xmlpush/v3/j/h;->a(Ljava/lang/String; Z)
Lorg/xmlpush/v3/j/h;->a(Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Z)

Figure 15: FinSpy self-destruct script generation

Since FinSpy has the ability to remove itself, it generates a shell script `/system/etc/xrebuild.sh` listed below.

```
#!/system/bin/sh
mount -o rw,remount /system
am force-stop <package name>
dd if=/dev/zero of=<apk path> bs=1024 count=8192
find <path> | while read line; do dd if=/dev/zero of=$line bs=1024 count=8192; done
```

Then it makes the script executable and reboots the device.

The script writes zeros over the APK file and does the same for all files located into the application data directory. FinSpy can be configured to remove itself at a given date and time, when the C2 is not reachable for a given amount of time or when the implant receive a specific command. By filling all files with zeros, FinSpy prevents forensic investigation. The script generation takes in account the fact that the implant can be a system application or a regular application.

6. Data collection mechanism

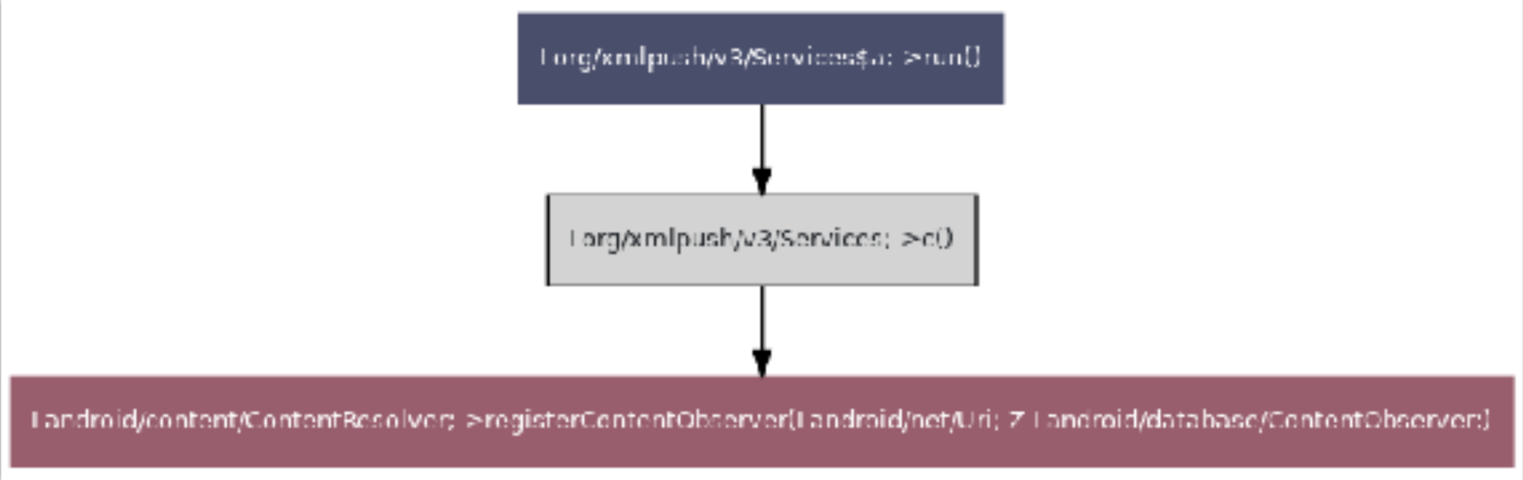

Lorg/xmlpush/v3/Services$a;->run()
Lorg/xmlpush/v3/Services;->c()
Landroid/content/ContentResolver;->registerContentObserver(Landroid/net/Uri; Z Landroid/database/ContentObserver;)

Figure 16: FinSpy content changes tracking

Java class `org.xmlpush.v3.Services` registers the following content observers on:

- changes on phone contact list
- changes on SIM contact list
- changes on SMS log
- changes on calendar

Java class `org.xmlpush.v3.eventbased.ReceiverService` listens to the following events:

- new outgoing phone call
- new data SMS received
- SIM card has been changed

Numerous threads are started to periodically check device location and messenger applications files. Every time a change occurs on observed data or an event occurs, FinSpy collects data related to that change/event and sends it to the C2 either over HTTP or SMS.

7. Data collected and sent by default

The FinSpy code shows that all payloads sent to C2s contain at least:

- trojan UID
- phone number
- timezone
- current date and time
- mobile operator name
- country code based on mobile network
- location area code
- mobile cell ID

8. Messenger applications data exfiltration

FinSpy is designed to exfiltrate contacts, messages, groups, location and files of the following applications:

- `com.viber.voip`
- `jp.naver.line.android`

- `com.skype.raider`
- `com.facebook.orca`
- `com.futurebits.instamessage.free`
- `jp.naver.line.android`
- `com.viber.voip`
- `com.skype.raider`
- `com.futurebits.instamessage.free`
- `com.bbm`
- `ch.threema.app`
- `org.telegram.messenger`

FinSpy looks at the content of each application data directory (i.e. `/data/data/com.futurebits.instamessage.free/`). This capability has already been documented in many public reports.

9. Call log exfiltration

FinSpy exfiltrates the following information each time a call is placed:

- caller's phone number
- callee's phone number
- caller's name
- callee's name
- call duration is seconds

10. SMS log exfiltration

FinSpy exfiltrates the following information each time a SMS received:

- date and time
- sender's phone number
- recipient's phone number
- SMS content

11. Calendar events exfiltration

FinSpy exfiltrates the following information each time a new event is added/edited:

- attendees' names
- attendees' emails
- event title
- event description
- event location
- event start and end date

12. Address book exfiltration

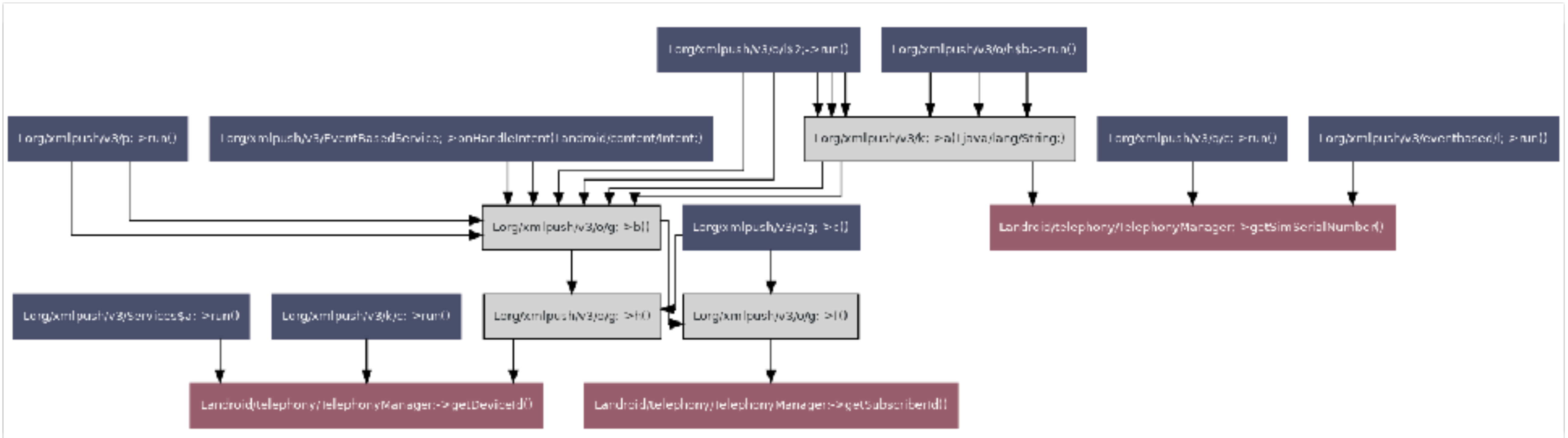
FinSpy exfiltrates the following information each time a modification is done on the address book:

- work phone number
- mobile phone number

- home phone number
- all other available phone numbers
- display name
- location
- email addresses
- postal addresses

FinSpy collects contacts stored in the phone memory and in the SIM card.

13. SIM information exfiltration


Lorg/xmlpush/v3/c/ls?;->run()
Lorg/xmlpush/v3/o/h$b;->run()
Lorg/xmlpush/v3/p;->run()
Lorg/xmlpush/v3/EventBasedService;->onHandleIntent(Landroid/content/Intent;)
Lorg/xmlpush/v3/k;->a(Ljava/lang/String;)
Lorg/xmlpush/v3/c/c;->run()
Lorg/xmlpush/v3/eventbased/l;->run()
Lorg/xmlpush/v3/o/g;->b()
Lorg/xmlpush/v3/c/g;->c()
Landroid/telephony/TelephonyManager;->getSimSerialNumber()
Lorg/xmlpush/v3/Services$a;->run()
Lorg/xmlpush/v3/k/c;->run()
Lorg/xmlpush/v3/o/g;->h()
Lorg/xmlpush/v3/o/g;->i()
Landroid/telephony/TelephonyManager;->getDeviceId()
Landroid/telephony/TelephonyManager;->getSubscriberId()
Figure 17: SIM information retrieval

Each time the SIM card is changed, FinSpy sends the following data to the C2:

- phone number
- SIM serial number
- IMEI
- IMSI
- network operator name

14. Location tracking

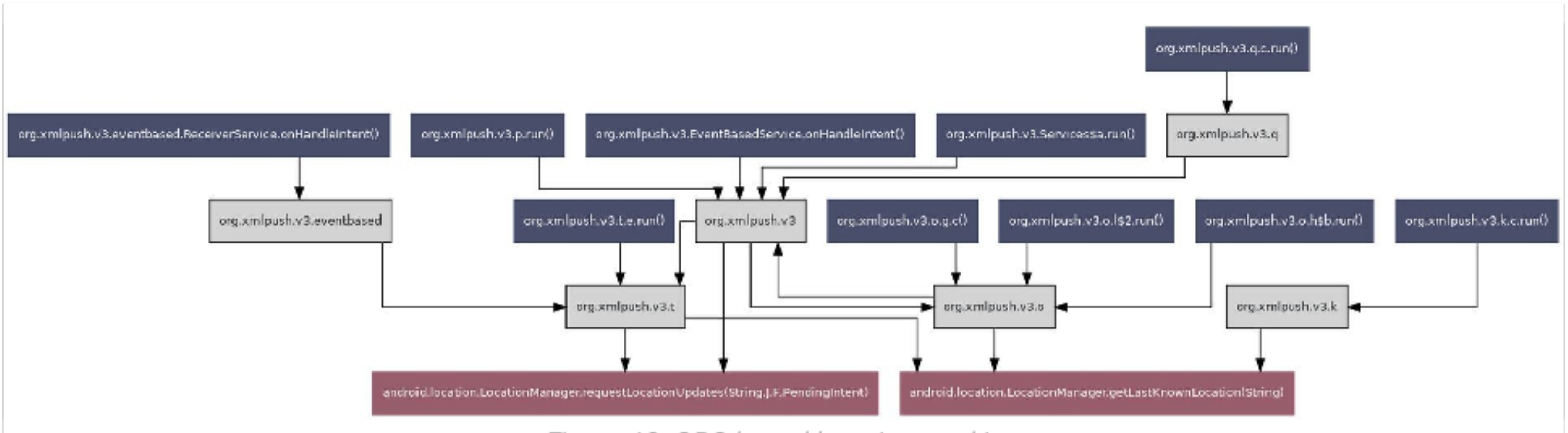

org.xmlpush.v3.q.c.run()
org.xmlpush.v3.eventbased.ReceiverService.onHandleIntent()
org.xmlpush.v3.p.run()
org.xmlpush.v3.EventBasedService.onHandleIntent()
org.xmlpush.v3.Servicessa.run()
org.xmlpush.v3.q
org.xmlpush.v3.eventbased
org.xmlpush.v3.t.e.run()
org.xmlpush.v3
org.xmlpush.v3.o.g.c()
org.xmlpush.v3.o.l$2.run()
org.xmlpush.v3.o.h$b.run()
org.xmlpush.v3.k.c.run()
org.xmlpush.v3.t
org.xmlpush.v3.o
org.xmlpush.v3.k
android.location.LocationManager.requestLocationUpdates(String,J,F,PendingIntent)
android.location.LocationManager.getLastKnownLocation(String)

Figure 18: GPS based location tracking

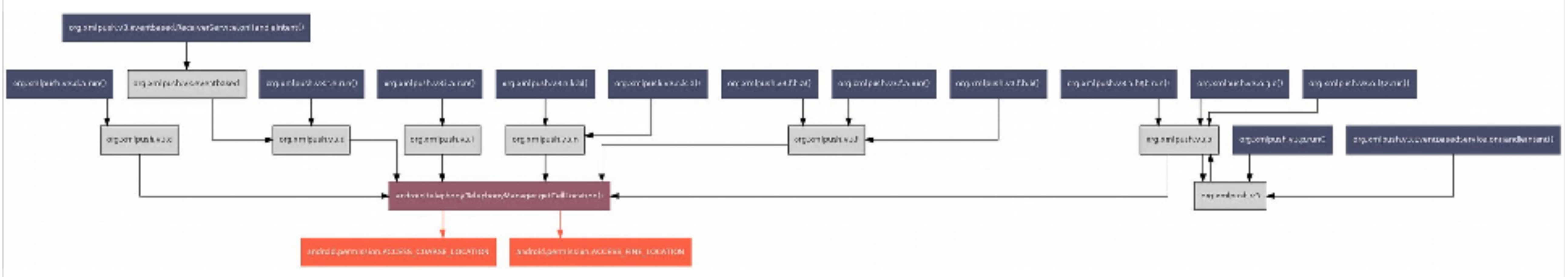

android.permission.ACCESS_COARSE_LOCATION
android.permission.ACCESS_FINE_LOCATION

Figure 19: Network based location tracking

FinSpy periodically collects and sends the device location. It collects both GPS based location and network based location by using cells.