24 - 26 September, 2025 / Berlin, Germany

HUNTING POTENTIAL C2 COMMANDS IN ANDROID MALWARE VIA SMALI STRING COMPARISON AND CONTROL FLOW ANALYSIS

JunWei Song

Recorded Future, Taiwan

junwei.song@recordedfuture.com

ABSTRACT

Identifying command-and-control (C2) commands in *Android* malware is crucial for understanding its intent and enhancing threat mitigation strategies. Traditional dynamic analysis methods rely on network traffic analysis and often struggle against encryption, obfuscation, or unavailable C2 servers. Additionally, advanced anti-analysis techniques further hinder dynamic approaches. To address these challenges, we propose a static analysis method to efficiently locate functions that may contain C2 commands, significantly accelerating malware reverse engineering and increasing the likelihood of discovering previously unknown malware containing C2 commands.

Our approach detects structured patterns in malware code, particularly multiple string comparisons within if-else and switch statements, and HashMap-based C2 command mappings that associate command strings with specific functions. Even when C2 commands are encrypted or obfuscated, these structures remain identifiable through Smali opcode analysis. By analysing the frequency and distribution of specific opcode patterns (e.g. const-string, invoke-virtual, if-eqz) and *Android* API calls (e.g. equals, contains), we establish detection thresholds to identify suspicious functions. This method allows analysts to quickly pinpoint areas of interest, reducing the time spent on manual code inspection and improving overall analysis efficiency.

We use Python and the Androguard package to analyse Smali instructions and API calls. Our tool flags functions when predefined thresholds are exceeded and extracts potential C2 commands from string comparison patterns, streamlining the malware analysis process. In our research, we identified a new version of the TgToxic *Android* banking trojan, demonstrating the effectiveness of our methodology in real-world scenarios. We also validate our approach across various malware families such as Octo, Copybara, Cerberus, XLoader and more, confirming its effectiveness.

Our technique significantly narrows the investigation scope, making it a valuable tool for threat hunting and malware classification in *Android* malware analysis. Although our approach does not ensure that every flagged function contains C2 commands due to the varying number of C2 commands across malware families, it provides a strong basis for further research. We will share the tool implementation and comprehensive data on C2 command occurrences across malware families to support the design of detection thresholds.

Looking ahead, we will refine these thresholds and enhance our automated analysis to extend our methodology across diverse malware families. It will strengthen our ability to hunt for previously unknown malware with C2 command structures.

INTRODUCTION

Our investigation into *Android* malware C2 command structures involved reverse engineering recent malware families, including TgToxic, XLoader, Copybara, Nexus and Cerberus. Our goal was to identify and understand the C2 command structures within these malware families, as manual code inspection to locate them typically requires a significant amount of time and effort.

We have identified several consistent code patterns for managing C2 commands across different malware families. The four primary structures used for C2 command handling are the following:

1. **Dense if-else chains and switch-case blocks with multiple command string comparisons**:

   This typical C2 command-handling structure in *Android* malware, such as TgToxic, involves multiple comparisons between C2 command strings and the input data received from the C2 server. The functions corresponding to the matched commands are then executed accordingly.

2. **HashMap mappings between command strings and functions**:

   Some malware samples, such as XLoader, use the HashMap to associate C2 command strings with their corresponding functions. This approach separates the commands from the execution logic, making it more difficult to trace and debug.

3. **Command strings are stored in a large array**:

   Specific malware samples, such as Copybara, define multiple C2 commands in a single array, further complicating reverse engineering.

4. **Multiple constant command strings embedded in a single class**:

   Malware such as Nexus and Cerberus often includes numerous constant strings defining C2 commands, all of which are placed within a single class in certain instances. This centralization obscures their purpose and complicates static analysis, especially when combined with obfuscation or encryption techniques.

These recurring code patterns suggest the feasibility of developing a static detection framework that can identify such behaviours and accelerate the process of locating C2 commands for malware analysts. Unlike traditional signature-based matching, this approach emphasizes matching C2 command structure patterns, allowing for detection based on code organization rather than specific strings, APIs, or known indicators.

Building on these common code patterns in C2 command handling, we propose a novel static analysis method to identify functions likely involved in C2 command processing. This method significantly reduces the manual effort required for code inspection, enabling malware analysts to pinpoint malicious functions warranting further investigation swiftly. This approach not only accelerates the reverse engineering process but also facilitates the discovery of previously unknown malware families with similar C2 command structures, thereby providing a more efficient workflow for malware analysts.

C2 COMMAND STRUCTURES IN ANDROID MALWARE AND PATTERN RULE EXTRACTION

To illustrate these four C2 command structures, we analyse real-world malware samples from recent *Android* banking trojan families. Drawing from several malware families reported in recent years, this analysis provides practical insights into how these malware samples implement code structures for managing C2 commands and their corresponding *Android* Smali instructions. We will then demonstrate how to extract critical *Android* Smali opcodes and *Android* APIs from the Smali instructions to identify patterns in C2 command handling, enabling the identification of functions that may contain C2 commands.

1. Dense if-else chains and switch-case blocks with multiple command string comparisons: TgToxic case study

Figure 1 shows a code snippet from a newly identified TgToxic variant that demonstrates C2 command handling. This *Android* banking trojan family was initially reported by *Trend Micro* [1] in early February 2023 and re-emerged in the wild with new variants in late 2024 [2]. In this variant, C2 commands are processed in plaintext through a large switch-case structure within com.example.mysoul.KszahaVmkrjij. The input string is first mapped using its String.hashCode() value to determine the appropriate case block, followed by an exact comparison using String.equals() to confirm the matching C2 command.

```
String POoIo0IIOi1 = PlI0lo0i00iO.POoIo0IIOi1(objArr[0].toString(), KszahaVmkrjij.HI0IOiOlo);
if (POoIo0IIOi1 != null) {
    try {
        JSONObject jSONObject2 = new JSONObject(POoIo0IIOi1);
        try {
            String string = jSONObject2.getString("action");
            switch (string.hashCode()) {
                case -1949226861:
                    if (string.equals("updateApk")) {
                        c = 29;
                        break;
                    }
                    c = 65535;
                    break;
                case -1884362643:
                    if (string.equals("stopCam")) {
                        c = '\n';
                        break;
                    }
                    c = 65535;
                    break;
                case -1858379769:
                    if (string.equals("restartMe")) {
                        c = '-';
                        break;
                    }
                    c = 65535;
                    break;
                case -1858379585:
                    if (string.equals("restartSc")) {
                        c = ',';
                        break;
                    }
                    c = 65535;
                    break;
                case -1827548105:
                    if (string.equals("walletList")) {
                        c = '\\';
                        break;
                    }
                    c = 65535;
                    break;
```

Figure 1: TgToxic code snippet for handling C2 commands.

The following code snippet demonstrates the *Android* Smali instructions for performing string comparisons as part of switch-case logic. It uses const-string to load C2 command strings, followed by invoke-virtual to call String.equals(), then stores the result with move-result, and uses if-eqz to check whether the input string matches any of the predefined C2 commands received from the C2 server.

```
0026f432: sparse-switch       v10, :s_switch_1278
0026f438: goto/16             :goto_0485

0026f43c: const-string        v10, "installPermission" # string@4209
0026f440: invoke-virtual      {v7, v10}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
0026f446: move-result         v7
0026f448: if-eqz              v7, :cond_0485
0026f44c: const/16            v7, 0x4c
0026f450: goto/16             :goto_0486

0026f454: const-string        v10, "gestureB" # string@3b0f
0026f458: invoke-virtual      {v7, v10}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
0026f45e: move-result         v7
0026f460: if-eqz              v7, :cond_0485
0026f464: const/16            v7, 0x58
0026f468: goto/16             :goto_0486

0026f46c: const-string        v10, "requestfloaty" # string@5009
0026f470: invoke-virtual      {v7, v10}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
0026f476: move-result         v7
0026f478: if-eqz              v7, :cond_0485
0026f47c: const/16            v7, 0x16
0026f480: goto/16             :goto_0486

0026f484: const-string        v10, "admLockRule" # string@3288
0026f488: invoke-virtual      {v7, v10}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
0026f48e: move-result         v7
0026f490: if-eqz              v7, :cond_0485
0026f494: const/4             v7, 0x5
0026f496: goto/16             :goto_0486
...
```

Opcode and Android API pattern rule

The *Android* Smali opcodes start with sparse-switch, followed by a sequence of const-string, invoke-virtual, move-result, and if-eqz. The *Android* API String.equals() consistently appears in a fixed order, indicating a repeated check to determine whether a string matches a predefined command set by the malware author.

This pattern suggests that the C2 command handling in the malware follows a recognizable structure. From this observation, we can extract a rule for these critical opcodes and *Android* API patterns as follows:

```
{
  "opcode": {
    "sparse-switch": 1,
    "const-string": 10,
    "invoke-virtual": 10,
    "move-result": 10,
    "if-eqz": 10
  },
  "api": {
    "Ljava/lang/String;->equals(Ljava/lang/Object;)Z": 10
  }
}
```

This rule specifies the Smali opcodes and *Android* API used to process C2 commands. Each opcode and API is assigned a value representing the minimum occurrence threshold. For instance, based on our experience, C2 commands typically

exceed a count of ten, so we set the threshold at ten as an example. When the defined occurrence of each opcode and API in our rule meets or exceeds these thresholds, we will flag the function name and output all strings within that function.

In this TgToxic sample, the malware utilizes intricate switch-case structures to process C2 commands. This structure, featuring multiple command string comparisons and the execution of corresponding functions, indicates that this block of code warrants further investigation by the analysts. The consistent pattern of opcodes, including sparse-switch, const-string, invoke-virtual, move-result and if-eqz, as demonstrated in the Smali instructions and the *Android* API String.equals(), serves as the foundation of our static analysis patterns.

However, not all malware relies on direct string comparisons. Some malware families utilize data structures, such as HashMap, to associate command strings with their corresponding functions, thereby enhancing modularity in command handling and presenting new challenges for malware analysts. Next, we detail this HashMap-based C2 command handling structure through a case study of the XLoader malware.

2. HashMap mappings between command strings and functions: XLoader case study

Here is another C2 command-handling structure that utilizes the HashMap. The following snippet is from XLoader (also known as MoqHao [3]), an *Android* banking trojan. It shows how C2 commands are managed within a large HashMap in the com.Loader.start method. Each C2 command string serves as a key in the map, linking it to a new instance of a class that implements the corresponding C2 functionality.

```
J j2 = new J(this, 7);
d0 d0Var = this.f393f;
d0Var.f234c.put("sendSms", j2);
J j3 = new J(this, 8);
HashMap hashMap = d0Var.f234c;
hashMap.put("setWifi", j3);
hashMap.put("gcont", new J(this, 9));
hashMap.put("lock", new J(this, 10));
hashMap.put("bc", new J(this, 11));
hashMap.put("setForward", new J(this, 12));
hashMap.put("getForward", new J(this, 13));
hashMap.put("hasPkg", new J(this, 14));
hashMap.put("setRingerMode", new J(this, 15));
hashMap.put("setRecEnable", new J(this, i6));
hashMap.put("reqState", new J(this, i4));
hashMap.put("showHome", new J(this, i3));
hashMap.put("getnpki", E.f157c);
hashMap.put("http", E.f158d);
hashMap.put("call", new J(this, i5));
hashMap.put("get_apps", new J(this, 4));
hashMap.put("ping", new J(this, 5));
hashMap.put("getPhoneState", new J(this, i2));
File file = new File(Environment.getExternalStorageDirectory().getAbsolutePath() + "/DCIM/Camera");
hashMap.put("get_gallery", new C0023y(this, file));
hashMap.put("get_photo", new P(file, i5));
```

Figure 2: XLoader code snippet for handling C2 commands.

The following section of *Android* Smali instructions represents a sequence of HashMap.put() operations, where each command string (e.g. ‘sendSms’, ‘setWifi’, etc.) is associated with a new instance of the La/J class. Each instance is constructed with the context object (p0, representing this) and a unique integer identifier. This design allows the malware to dynamically register various command handlers by associating each command string with its corresponding handler.

```
00021022: iget-object          v5, v4, La/d0;->c:Ljava/util/HashMap;

00021026: const-string         v6, "sendSms" # string@0895
0002102a: invoke-virtual       {v5, v6, v14}, Ljava/util/HashMap;->put(Ljava/lang/Object;,
Ljava/lang/Object;)Ljava/lang/Object;

00021030: new-instance         v14, La/J; # type@006a
00021034: const/16             v5, 0x8
00021038: invoke-direct        {v14, p0, v5}, La/J;-><init>(Lcom/Loader;, I)V

0002103e: iget-object          v4, v4, La/d0;->c:Ljava/util/HashMap;
00021042: const-string         v5, "setWifi" # string@08cd
```

```
00021046: invoke-virtual      {v4, v5, v14}, Ljava/util/HashMap;->put(Ljava/lang/Object;, 
Ljava/lang/Object;)Ljava/lang/Object;

0002104c: new-instance        v14, La/J; # type@006a

00021050: const/16            v5, 0x9

00021054: invoke-direct       {v14, p0, v5}, La/J;-><init>(Lcom/Loader;, I)V

0002105a: const-string        v5, "gcont" # string@064a

0002105e: invoke-virtual      {v4, v5, v14}, Ljava/util/HashMap;->put(Ljava/lang/Object;, 
Ljava/lang/Object;)Ljava/lang/Object;

00021064: new-instance        v14, La/J; # type@006a

00021068: const/16            v5, 0xa

0002106c: invoke-direct       {v14, p0, v5}, La/J;-><init>(Lcom/Loader;, I)V

00021072: const-string        v5, "lock" # string@07a2

00021076: invoke-virtual      {v4, v5, v14}, Ljava/util/HashMap;->put(Ljava/lang/Object;, 
Ljava/lang/Object;)Ljava/lang/Object;
...
```

Opcode and Android API pattern rule

Our analysis indicates that this type of C2 command registration in *Android* Smali typically follows a consistent pattern. After creating the initial HashMap instance earlier in the code, the malware retrieves the map reference using the iget-object opcode. It then executes a sequence of opcodes: const-string to load the C2 command string, new-instance to instantiate a handler class, and invoke-virtual to call the HashMap.put() method, which inserts the key-handler pair into the HashMap.

Based on this recurring opcode sequence, we propose the following pattern rule to identify HashMap-based C2 command registration structures:

```
{
  "opcode": {
    "iget-object": 1,
    "const-string": 10,
    "new-instance": 10,
    "invoke-virtual": 10
  },
  "api": {
    "Ljava/util/HashMap;->put(Ljava/lang/Object; Ljava/lang/Object;)Ljava/lang/Object;": 10
  }
}
```

In this XLoader sample, HashMap facilitates the dynamic linking of commands to their handling functions, resulting in a flexible and modular command processing mechanism. However, this approach can complicate static analysis, as command-to-function mappings are not as clearly defined as in dense if-else chains or switch statements.

In addition to HashMap, some malware families employ large arrays that consolidate multiple command strings into a single data structure for their C2 commands. This method complicates command extraction and introduces specific challenges for reverse engineering. The following section explores the detection of C2 commands stored in such extensive arrays through a case study of the Copybara malware.

3. C2 commands are stored in a large array: Copybara case study

In this section, we explore another method malware uses to handle C2 commands by analysing Copybara [4], an *Android* banking trojan that first appeared in November 2021 and which resurfaced in November 2023. The code snippet shown in Figure 3 illustrates how Copybara stores its C2 commands. These commands are defined in a large array, with each array index corresponding to a specific C2 function. These commands are implemented in com.plkmodulo261.newicon.backgroundservice._mqtt_messagearrived.

```
public static String _mqtt_messagearrived(String str, byte[] bArr) throws Exception {
    try {
    } catch (Exception e) {
        processBA.setLastException(e);
        Common.LogImpl("3801373", BA.ObjectToString(Common.LastException(processBA)), 0);
        _send_myerror_tosrv(Common.LastException(processBA).getMessage(), "mqtt_MessageArrived");
        return "";
    }
    if (!str.equals("commands_FromPC")) {
        return "";
    }
    B4XSerializator b4XSerializator = new B4XSerializator();
    new Map();
    Map map = (Map) AbsObjectWrapper.ConvertToWrapper(new Map(), (java.util.Map) b4XSerializator.ConvertBytesToObject(bArr));
    globalparameters globalparametersVar = mostCurrent._vvvvvvvvvvvvvv3;
    if (!globalparameters._vvvv7.equals(BA.ObjectToString(map.Get("deviceid")))) {
        return "";
    }
    switch (BA.switchObjectToInt(map.Get("fun"), "open_app_setngs", "hid_app_icno", "send_admn_lckdvcs_on", "send_inj_lst", "send_custom_opencam",
        case 0:
            _open_app_setngs_data(map);
            return "";
        case 1:
            _hid_app_icno(map);
            return "";
        case 2:
            _send_admn_lckdvcs_on();
            return "";
        case 3:
            _send_inj_lst(map);
            return "";
```

Figure 3: Copybara code snippet for handling C2 commands.

The following snippet of *Android* Smali instructions demonstrates how to allocate and initialize an object array containing C2 command strings. The code first retrieves a value from a map object, then creates a new object array with a length of 0x3e and fills it with command strings such as ‘open_app_setngs’, ‘hid_app_icno’, and ‘send_admn_lckdvcs_on’ at designated indices.

```
003e4d54: const-string        v6, "fun" # string@9dab
003e4d58: invoke-virtual      {v5, v6}, Lanywheresoftware/b4a/objects/collections/Map;->Get(Ljava/lang/Object;)Ljava/lang/Object;
003e4d5e: move-result-object  v6

003e4d60: const/16            v7, 0x3e
003e4d64: new-array           v7, v7, [Ljava/lang/Object;

003e4d68: const-string        v8, "open_app_setngs"
003e4d6c: aput-object         v8, v7, v4
003e4d70: const-string        v8, "hid_app_icno"

003e4d74: const/4             v9, 0x1
003e4d76: aput-object         v8, v7, v9
003e4d7a: const-string        v8, "send_admn_lckdvcs_on"

003e4d7e: const/4             v10, 0x2
003e4d80: aput-object         v8, v7, v10
003e4d84: const-string        v8, "send_inj_lst"

003e4d88: const/4             v10, 0x3
003e4d8a: aput-object         v8, v7, v10
003e4d8e: const-string        v8, "send_custom_opencam"
...
```

Opcode and Android API pattern rule

Our analysis indicates that the Smali instruction sequence typically starts with a new-array opcode to allocate an array, followed by a consistent pattern of const-string and aput-object opcodes to initialize the array elements. Based on this, we define the following pattern rule:

```
{
  "opcode": {
    "new-array": 1,
    "const-string": 10,
    "aput-object": 10
  }
}
```

In this Copybara sample, C2 commands are organized into a large array that centralizes command strings while allowing for efficient lookup and processing within the code. Although this pattern centralizes commands, it may complicate analysis due to the large number and potential obfuscation or encryption.

Another related pattern we observe is the definition of numerous constant command strings within a single class, effectively embedding various commands as static fields. This centralization complicates static analysis, especially when combined with string obfuscation or encryption techniques. The following section explores this approach through a case study of the Nexus malware.

4. Multiple constant command strings embedded in a single class: Nexus case study

This section analyses Nexus [5], an *Android* banking trojan associated with the SOVA *Android* malware family, which was first identified in 2023. This variant employs a unique structure for managing its C2 commands, the fourth structure previously mentioned. As illustrated in the code snippet in Figure 4, commands are defined as individual static string constants. Each constant represents a specific functionality, allowing the malware to associate particular behaviours with predefined C2 command keywords. These commands are defined in the com.tapston.burgerking.Const class.

```
static {
    PERMISSION_LIST = Build.VERSION.SDK_INT >= 26 ? CollectionsKt__CollectionsKt.listOf((Object[]) new String[]{"android.permission.READ_SMS",
    get2fa = "get2fa";
    start2faactivator = "start2faactivator";
    stop2faactivator = "stop2faactivator";
    delbot = "delbot";
    openUrl = "openurl";
    startlock = "startlock";
    stoplock = "stoplock";
    admin = "getperm";
    delapp = "delapp";
    clearappdata = "clearappdata";
    startextraverbose = "startextraverbose";
    stopextraverbose = "stopextraverbose";
    starthidenpush = "starthidenpush";
    stophidenpush = "stophidenpush";
    hidesms = "starthidesms";
    stophidensms = "stophidesms";
    scancookie = "scancookie";
    stopcookie = "stopcookie";
    scaninject = "scaninject";
    stopscan = "stopscan";
    getsms = "getsms";
    clearsms = "clearsmslist";
    startkeylogs = "startkeylogs";
    stopkeylogs = "stopkeylogs";
    contactssender = "contactssender";
    sendsms = "sendsms";
    openinject = "openinject";
    getapps = "getapps";
    sendpush = "sendpush";
    enableinject = "enableinject";
    runapp = "runapp";
    callForward = "forwardcall";
    call = NotificationCompat.CATEGORY_CALL;
    disableinject = "disableinject";
    getcontacts = "getcontacts";
    startMute = "startmute";
    stopMute = "stopmute";
    gettrustwallet = "gettrustwallet";
    getexodus = "getexodus";
    remote = new Remote(null, null, null, 7, null);
}
```

Figure 4: Nexus code snippet for handling C2 commands.

The following snippet of *Android* Smali instructions demonstrates how constant C2 command strings are assigned to static fields in the com.tapston.burgerking.Const class. Each const-string opcode loads a command string into register v0, which is then stored in the corresponding static field using the sput-object opcode. This pattern associates command

strings, such as ‘get2fa’, ‘start2faactivator’ and ‘delbot’, with static constants for later reference in the malware’s command-handling logic.

```
004d9234: sput-object         v0, Lcom/tapston/burgerking/Const;->PERMISSION_LIST:Ljava/
util/List;
004d9238: const-string        v0, "get2fa"
004d923c: sput-object         v0, Lcom/tapston/burgerking/Const;->get2fa:Ljava/lang/String;
004d9240: const-string        v0, "start2faactivator"
004d9244: sput-object         v0, Lcom/tapston/burgerking/Const;->start2faactivator:Ljava/
lang/String;
004d9248: const-string        v0, "stop2faactivator"
004d924c: sput-object         v0, Lcom/tapston/burgerking/Const;->stop2faactivator:Ljava/
lang/String;
004d9250: const-string        v0, "delbot"
004d9254: sput-object         v0, Lcom/tapston/burgerking/Const;->delbot:Ljava/lang/String;
004d9258: const-string        v0, "openurl"
004d925c: sput-object         v0, Lcom/tapston/burgerking/Const;->openUrl:Ljava/lang/
String;
004d9260: const-string        v0, "startlock"
004d9264: sput-object         v0, Lcom/tapston/burgerking/Const;->startlock:Ljava/lang/
String;
004d9268: const-string        v0, "stoplock"
004d926c: sput-object         v0, Lcom/tapston/burgerking/Const;->stoplock:Ljava/lang/
String;
004d9270: const-string        v0, "getperm"
004d9274: sput-object         v0, Lcom/tapston/burgerking/Const;->admin:Ljava/lang/String;
...
```

Opcode and Android API pattern rule

Our observations indicate that the *Android* Smali instructions sequence in this case typically begins with a const-string opcode to load a string constant, followed by a sput-object opcode to store the string in a static field. Based on this observation, we define the following pattern rule:

```
{
  "opcode": {
    "const-string": 10,
    "sput-object": 10
  }
}
```

Cerberus

Another example is Cerberus [6], an *Android* banking trojan that manages C2 commands using encrypted strings stored within a single class, namely com.konybqplijaqcazz.nwclqe.a. In this variant, RC4 encryption with Base64 encoding is employed, which demonstrates how our research enables the rapid identification of specific functions in *Android* malware, even when C2 commands are encrypted or obfuscated. These strings must be decrypted dynamically to retrieve their original content, regardless of the encryption or obfuscation methods employed.

Our tool’s primary advantage lies in its ability to identify these functions and extract relevant strings from potentially suspicious functions. When a skilled malware analyst encounters strings similar to those found in the Cerberus sample shown in Figure 5, it typically suggests that those strings are likely encrypted or obfuscated C2 commands or configuration values, such as URLs, protocols, URIs, or other critical parameters utilized by the malware that warrant further investigation.

```
public class a {
    public boolean a = false;
    public String b = b("pejofjdnxlapNGU2NjNlZWI1NQ==");
    public String c = b("cnugorsjkayfOThhY2YxZjJhYjBkZDAyN2Y2Y2UyMmY4MzE2Mw==");
    public String d = b("dbebfxtwqutzYzA0YjcwNWI2NmFkOTYxNDRkZWMwZWY4YmM=");
    public String e = b("lnlanhaussdsMjlhYmQxN2E2MjdmNmVlYmY3ZTBhMjUyOWI1YQ==");
    public String f = b("nonyzppwyyjtNjBkOWM2ZjU3NmFmZWJjZDQ4ZWZlZmU1YjU=");
    public String g = b("tgjspamxouujYzg0ODBmYjA=");
    public String h = b("ocicvxpcwamrYTRkMmRkY2FlZDFhNTE3YmM0YjE3NzdmOWU2ZDNi");
    public String i = b("mkrpbwlqemmgYmVjNGRlMzA=");
    public String j = b("dmnlxjqwgsfjNjJmYTI2Zjg4Y2EzODg2NGM5ZTg3YWUzMTBkZDk1NzU1NzM5ZDk3YTY3ZWY1MWI1MmY2YTYyYTg=");
    public String k = b("pgegyigmdejmYmU0MTYwY2UxZDMzZDI5MDQ1ZTFhMGMxNjM2Mw==");
    public String l = b("ioxwkjcjsjerNDU0YWEyZWU3Y2NjNDc5YWNhY2VmOGQxZTM2YTI0NDU5Yzk1ZmE1Zg==");
    public String m = b("eraillolkwtrOWVhMGEyYmJmYjRiNTE5YmM2NTc0NjQ1NmJjZDA4MjA0OWUzZWYxMTIwZGM0ZGJk");
    public String n = b("zniscbjkwrtzODQ4YzY0YTk3MjUyZDNlODgwYWRlMTBmNGFmYTVmMWUwNzU2MzdmNzE3");
    public String o = b("cdgbwyvzuxxgNzVkZDA3NWUxNzY5YTFmMWE1M2QzMjMzYzhhMjBkZGRjOWEzY2FlY2U1");
    public String p = b("rvjozvnjeqpxMDNlNjU2NWEzNzk5");
    public String q = b("vwmkltdfezpqN2IzNmYzNmI2ZTdlMTk2OWEyZDM3MTkxOGFkMmVkZDU3MWFjZmE1YzNhYzIxZWM5NDdjNTAw");
    public String r = b("opuojndvvrgnMDZjMzc2OTBmN2M1YTFhN2Mw");
    public String s = b("lvmbvywdqipiZDdiZWM0YTZiNWFkMDZiYmYzMGM=");
    public String t = b("crohfnyrtuvhZjI2NGUyOGVhMmFlMDY5ZDMz");
    public String u = b("wiedsnqtytrdNTM1MGUwYjI0ZjFjNDdmYWQ1ODY1Zg==");
    public String v = b("ngceyqkycffnODY3YzkyMDBlOTRiMDljZDg3N2U=");
    public String w = b("ynreaupxyxgnODBhMTg4ZDFmMjMxOWM5ZDc5NzAyMWMwNmZkMDM3MDA=");
    public String x = b("xiuurgqvrzvuMTc1MDYxMjU2YTA0YTY=");
    public String y = b("vitolkujkfalN2U3MGQ0NmRmOGZlYTJjNjVh");
    public String z = b("wdpipmndgddjZmFkY2E2ZmZlNzYwMWU0ZTJk");
    public String A = b("dxkaydobvtvxNzAwYTNiMDUxNzlhNmEwOWU4");
    public String B = b("cqbimlyvylstYmZkYTEyOTlmZThkNjVmMTNjOTdmOGRm");
    public String C = b("gblyvknxgngeYjlhMWI1NmEzZTY0NzRhODFjMzJjNg==");
    public String D = b("qjiyznvfdjktMWViN2Y2NDQxNGM3OTc2NzAzYTI5YzExNzAzODY4MDQ5NjBh");
    public String E = b("aromwhxccdfgYzc0M2Q0ZTU3N2JhYjM0OGQ1ZTI=");
    public String F = b("bcuiefvqhhriMDkyZWE4ZTEwYzlmNDk2NQ==");
    public String G = b("kqfelqsmxymrMTQ2MWFhNDk0MjFjY2VjMGRi");
    public String H = b("gxiwczetgbhhMjRlOTQzYTRhYzM0ZWY1NzE4OWFiNmI5Mzk2ZjZj");
    public String I = b("ecpvetotqgkyZTkzODI1NzRkMDFiZTE5YzEwM2Q=");
    public String J = b("ijrxeniwhfgjN2ViNzQ0MGM2YWZk");
    public String K = b("xoijqrezrcfpYzkyZjFmNjljZjg1ZTRlOTE5");
    public String L = b("oagwaqmaoaleZmI5N2U5YzJlODZhYjE0YTkyMjcyYTJiMDQ=");
    public String M = b("jyxgmiawylkjMGYzNjA5OTcwYjIzYmIwYTRm");
    public String N = b("fbrrzlafpqvpMWMwZmYy");
```

Figure 5: Cerberus code snippet for handling encrypted constant strings.

The following *Android* Smali instructions snippet demonstrates the initialization of an object, where multiple encrypted strings are loaded, decrypted through a method call, and then stored in instance fields. This pattern usually indicates the runtime decrypting or decoding of embedded configuration data.

```
000173cc: invoke-direct        {v6}, Ljava/lang/Object;-><init>()V
000173d2: const/4              v0, 0
000173d4: iput-boolean         v0, v6, Lcom/konybqplijaqcazz/nwclqe/a;->a:Z

000173d8: const-string         v1, "pejofjdnxlapNGU2NjNlZWI1NQ=="
000173dc: invoke-virtual       {v6, v1}, Lcom/konybqplijaqcazz/nwclqe/a;->b(Ljava/lang/
String;)Ljava/lang/String;
000173e2: move-result-object   v1
000173e4: iput-object          v1, v6, Lcom/konybqplijaqcazz/nwclqe/a;->b:Ljava/lang/String;

000173e8: const-string         v1, "cnugorsjkayfOThhY2YxZjJhYjBkZDAyN2Y2Y2UyMmY4MzE2Mw=="
000173ec: invoke-virtual       {v6, v1}, Lcom/konybqplijaqcazz/nwclqe/a;->b(Ljava/lang/
String;)Ljava/lang/String;
000173f2: move-result-object   v1
000173f4: iput-object          v1, v6, Lcom/konybqplijaqcazz/nwclqe/a;->c:Ljava/lang/String;

000173f8: const-string         v1, "dbebfxtwqutzYzA0YjcwNWI2NmFkOTYxNDRkZWMwZWY4YmM="
000173fc: invoke-virtual       {v6, v1}, Lcom/konybqplijaqcazz/nwclqe/a;->b(Ljava/lang/
String;)Ljava/lang/String;
00017402: move-result-object   v1
00017404: iput-object          v1, v6, Lcom/konybqplijaqcazz/nwclqe/a;->d:Ljava/lang/String;

00017408: const-string         v1, "lnlanhaussdsMjlhYmQxN2E2MjdmNmVlYmY3ZTBhMjUyOWI1YQ=="
0001740c: invoke-virtual       {v6, v1}, Lcom/konybqplijaqcazz/nwclqe/a;->b(Ljava/lang/
String;)Ljava/lang/String;
```

```
00017412: move-result-object  v1
00017414: iput-object         v1, v6, Lcom/konybqplijaqcazz/nwclqe/a;->e:Ljava/lang/String;
...
```

Opcode and Android API pattern rule

Our observations indicate that the *Android* Smali opcode sequence often includes frequent occurrences of const-string, invoke-virtual, move-result-object and iput-object opcodes. These opcodes are typically utilized during object initialization, field assignments, and string decryption processes, where const-string loads encrypted strings and invoke-virtual calls the decryption method. Based on this, we define the following pattern rule:

```
{
  "opcode": {
    "const-string": 10,
    "invoke-virtual": 10,
    "move-result-object": 10,
    "iput-object": 10
  }
}
```

To date, we have identified several opcodes and *Android* API patterns that aid in detecting various structures of C2 command handling. This research outlines the common patterns observed in *Android* malware related to C2 command management. As new patterns may emerge in the future, they can be integrated into this adaptable, pattern-based rule framework.

IMPLEMENTATION

Our implementation identifies functions that may contain C2 commands by detecting C2 command structure patterns commonly observed in *Android* malware. These include:

1. Dense if-else chains and switch-case blocks with multiple command string comparisons
2. HashMap mappings between command strings and functions
3. C2 commands stored in large arrays
4. Multiple constant command strings embedded in a single class.

We define these patterns in a JSON-based rule format, enabling extensibility and customization for future enhancements. We analyse real-world *Android* malware samples from various families to refine these patterns and extract their distinct C2 command-handling structures.

Each function from a given APK or DEX file is analysed based on the patterns specified in our JSON-based rules, excluding *Android* framework APIs and third-party libraries. If the structure of a function meets or exceeds the defined thresholds, it is flagged as potentially containing logic for processing C2 commands. Since different malware families vary in the number and structure of C2 commands they implement, establishing a universal threshold is inherently challenging. However, our analysis shows that functions with a high density of string comparisons within multiple switch statements or if-else chains are strong indicators of C2 command handling, alongside the other structures we mentioned. These structures are typically rare in benign applications.

To assist analysts, we provide a dataset of recent *Android* malware families, along with the number of observed C2 commands in each sample, enabling data-driven tuning of detection thresholds rather than relying on static or arbitrary values.

We implemented a proof of concept using Python and the Androguard framework to parse Smali instructions and identify relevant API calls. When pattern thresholds are met or exceeded, our tool flags the corresponding function names and extracts associated string constants, proposing an efficient and adaptable method for detecting potential C2 command structures in *Android* malware.

The overall workflow of our static analysis tool is shown in the flowchart in Figure 6. The process begins by taking an APK or DEX file as input and extracting all functions, excluding system libraries, *Android* APIs, and third-party libraries. For each function, Smali instructions are extracted and compared against predefined opcodes and *Android* API pattern rules. Functions that match any of these rules and meet or exceed the defined thresholds are flagged as potentially containing C2 command logic. All string constants within these functions are then extracted for further analysis. The flagged functions and their associated strings are subsequently output for review.

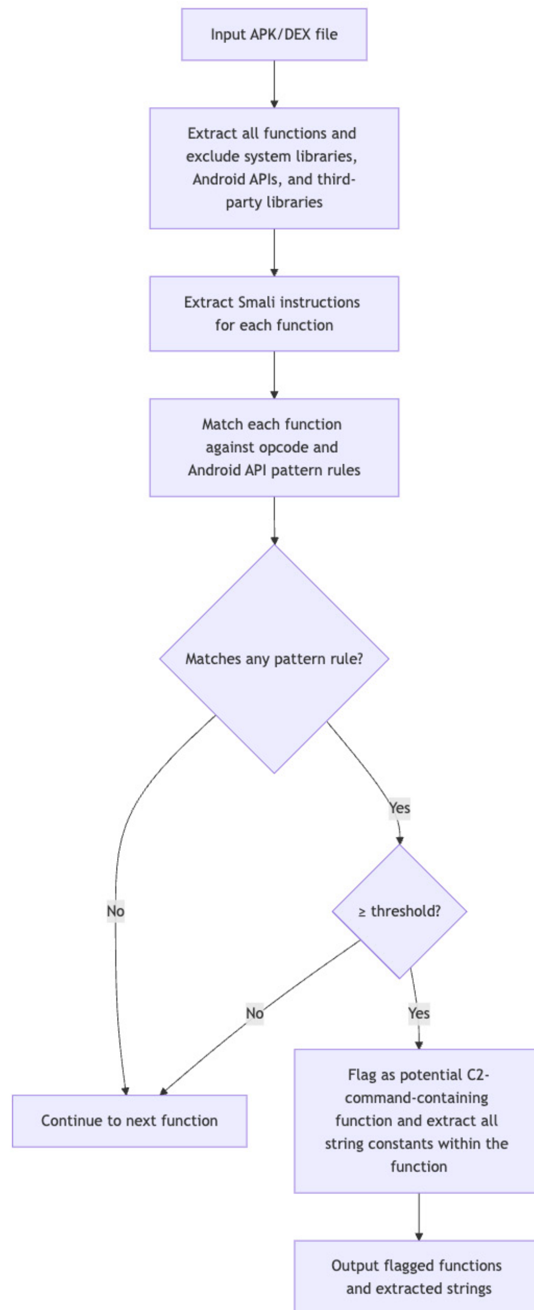

Figure 6: Workflow of our tool for detecting functions that may contain potential C2 commands in APK/DEX files.

The implementation is available as open-source code at [7]. The complete source code, along with detailed usage instructions and sample data, is available in this repository. We encourage researchers and analysts to explore, reproduce, and extend our tool for their own *Android* malware analysis.

Result output

The following is an example of the output generated by our research tool during the analysis of the TgToxic *Android* malware family. In this analysis, we applied the opcode and *Android* API pattern rule described in Section 1. This rule focuses on identifying dense if-else chains and switch-case blocks with multiple command string comparisons. It specifically targets the C2 command-handling logic commonly found in *Android* malware. The result demonstrates the effectiveness of our tool in extracting potential C2 command strings and their corresponding handler functions.

```
$ c2hunt -f malware_family/tgtoxic.dex -o custom-opcode/switch-equals.json
```

```
[INFO] Analyzing: malware_family/tgtoxic.dex
[INFO] Using OPcode & Android API Pattern Rule: custom-opcode/switch-equals.json
```

```
[INFO] Opcode & APIs threshold: {'sparse-switch': 1, 'const-string': 10, 'invoke-virtual': 10, 'move-result': 10, 'if-eqz': 10, 'Ljava/lang/String;->equals(Ljava/lang/Object;)Z': 10}

[+] The following functions potentially contain C2 commands:

Function: Lcom/example/mysoul/KszahaVmkrjij$UoO1i1liii0; call ([Ljava/lang/Object;)V
Opcode & APIs count: {'sparse-switch': 2, 'const-string': 219, 'invoke-virtual': 467, 'move-result': 446, 'if-eqz': 148, 'Ljava/lang/String;->equals(Ljava/lang/Object;)Z': 100}
=====[ C2HUNT RESULT ]=========================================
flag
homepage
action
screen_relay
walletList
installPermission
gestureB
requestfloaty
admLockRule
swipePwdScreenOff
inputSend
realtimeSet
showShortcuts
reqPerList
wallpaper
autoRequestPerm
readSmsList
autoBoot
backstage
setDebugMode
...

Function: Lc4; UoO1i1liii0 (I I J)V
Opcode & APIs count: {'sparse-switch': 10, 'const-string': 23, 'invoke-virtual': 211, 'move-result': 227, 'if-eqz': 71, 'Ljava/lang/String;->equals(Ljava/lang/Object;)Z': 68}
=====[ C2HUNT RESULT ]=========================================
alpha
elevation
rotation
transitionPathRotate
progress
rotationX
rotationY
transformPivotX
transformPivotY
scaleX
scaleY
translationX
translationY
translationZ
waveOffset
CUSTOM,
...
```

```
Function: Lcom/example/mysoul/MqcshiLrfqtluhuf; onAccessibilityEvent (Landroid/view/
accessibility/AccessibilityEvent;)V
Opcode & APIs count: {'sparse-switch': 2, 'const-string': 218, 'invoke-virtual': 545,
'move-result': 574, 'if-eqz': 240, 'Ljava/lang/String;->equals(Ljava/lang/Object;)Z': 76}
=====[ C2HUNT RESULT ]========================================
is_installApk_start
is_DeviceAdmin_enable
android.widget.EditText
android.view.ViewGroup
com.android.systemui
android.widget.FrameLayout
android.view.View
oppo
realme
android.inp##utmet##hodservice.SoftInp##utWindow
com.goo##gle.android.inpu##tmet##hod.la##tin
com.android.inp##utme##thod.keyb##oard.Key
com.goo##gle.android.inpu##tme##thod.lat##in:id/key_po##s_del
...
```

The analysis results show that the function com.example.mysoul.KszahaVmkrjij$UoO1i1liii0; call ([Ljava/lang/Object;)V stands out as a strong candidate for containing C2 commands, given the nature of the extracted strings and opcode patterns. It is essential to note that similar opcode and string patterns may also be found in other functions unrelated to C2 command handling, as shown in the additional results above.

The primary goal of our tool is to help analysts narrow down a vast number of functions to those that are most worthy of further investigation. By presenting these candidate functions along with their associated strings, our approach enables malware analysts to focus their efforts more efficiently and effectively.

CONCLUSION

Our research proposes a lightweight static analysis method based on *Android* Smali control flow and string comparison patterns, which are crucial for understanding the behaviour of *Android* malware in handling C2 commands. This method aims to create a scalable detection mechanism that is independent of malware families, specifically designed to identify potential C2 command processing logic in *Android* malware.

By analysing the specific frequency of Smali opcodes and control flow structures, such as string comparisons and switch statements, we present an automated technique for identifying functions likely involved in C2 command handling. This significantly improves the efficiency of malware static analysis and detection. Our practical and effective solution targets functions that may contain potential C2 commands, thereby streamlining the detection of previously unknown malware with these structures.

Given the scarcity of samples with clearly labelled C2 logic, we aggregate these C2 command-handling structures from multiple malware families to derive heuristic thresholds using statistical averages. For example, by computing the mean number of conditional branches or switch-case structures per function, we establish baselines for anomaly detection. Functions exceeding these baselines in complexity are flagged for further inspection, as they may indicate sophisticated C2 command handling.

Furthermore, our research is not limited to *Android* and can be applied to other systems, such as *Windows* and *Linux*, as the structure of C2 command-handling logic is typically consistent across platforms, differing mainly in the underlying CPU instructions (e.g. x86 or ARM). As long as the structure patterns can be extracted, corresponding static detection rules can be developed, enabling cross-platform detection of potential C2 commands.

However, the primary challenge of this method is establishing reasonable and generalizable thresholds for various malware families or sample sets, as the complexity and frequency of C2 command processing logic often depend on the malware's functional design and data targets.

In conclusion, we reiterate that the primary goal of this research is not to pinpoint every C2 command processing function with absolute precision but to assist malware analysts in effectively refining their focus through the pre-screening and pattern identification of the C2 command-handling structure. This methodology allows the prioritization of functions that are more likely to harbour C2 commands, thereby accelerating the entire analysis process and enhancing efficiency.

REFERENCES

[1] Trend Micro. TgToxic Malware's Automated Framework Targets Southeast Asia Android Users. 3 February 2023. https://www.trendmicro.com/en_us/research/23/b/tgtoxic-malware-targets-southeast-asia-android-users.html.

[2] Song, J. The New Version of the TgToxic Android Banking Trojan Is Coming Back with More Advanced Techniques and Capabilities. Hatching.io. 1 November 2024. https://hatching.io/blog/triage-insights-ep3/.

[3] Trend Micro. XLoader Android Spyware and Banking Trojan Distributed via DNS Spoofing. 20 April 2018. https://www.trendmicro.com/en_us/research/18/d/xloader-android-spyware-and-banking-trojan-distributed-via-dns-spoofing.html.

[4] Nigam, R. Technical Analysis of Copybara. Zscaler. 21 August 2024. https://www.zscaler.com/blogs/security-research/technical-analysis-copybara.

[5] Cleafy. Nexus: a new Android botnet? 21 March 2023. https://www.cleafy.com/cleafy-labs/nexus-a-new-android-botnet.

[6] ThreatFabric. Cerberus – A new banking Trojan from the underworld. 1 August 2019. https://www.threatfabric.com/blogs/cerberus-a-new-banking-trojan-from-the-underworld.

[7] krnick / c2hunt. https://github.com/krnick/c2hunt.

[8] Stefanko, L. Android app breaking bad: From legitimate screen recording to file exfiltration within a year. ESET. 23 May 2023. https://www.welivesecurity.com/2023/05/23/android-app-breaking-bad-legitimate-screen-recording-file-exfiltration/.

[9] ThreatFabric. Hook: a new Ermac fork with RAT capabilities. 19 January 2023. https://www.threatfabric.com/blogs/hook-a-new-ermac-fork-with-rat-capabilities.

[10] ThreatFabric. Look out for Octo's tentacles! A new on-device fraud Android Banking Trojan with a rich legacy. 8 April 2022. https://www.threatfabric.com/blogs/octo-new-odf-banking-trojan.

[11] Günel, M.; Filik, H. Alien Technical Analysis Report. 2020. https://drive.google.com/file/d/1qd7Nqjhe2vyGZ5bGm6gVw0mM1D6YDolu/view.

[12] ThreatFabric. Exposing Crocodilus: New Device Takeover Malware Targeting Android Devices. 28 March 2025. https://www.threatfabric.com/blogs/exposing-crocodilus-new-device-takeover-malware-targeting-android-devices.

SAMPLE HASHES

The table below presents hashes of various *Android* malware families analysed during our research. All of these samples are publicly available on *Triage Sandbox*.

SHA256 hash	Malware family
b2c1517e4b0e0b3286a5cde06310b2277da7333f5ab3c2828f08272e3f85b260	AhRat
4ad14957f494891e4ebd0fd37e09b0a9e51ef6ca6e578443e80acab3f662cdb6	XLoader
c5996e7a701f1154b48f962d01d457f9b7e95d9c3dd9bbd6a8e083865d563622	Hook
125a320735cde5a0891dd19d89c41395b63fcf4c6a0ad1dd80cda439f53d256f	Octo
7a65b831c9f781e49c47393345bbe2ffff6859a608b9a5413d7f3b76683ff178	AlienBot
14a05dab2a8dce7d407cc8878927a1b7dcf5dff9078fcef6783f151663573136	Cerberus
753dd541e71bf92c2b09971d4f5285f7f0376c0eb1fc7e3900d6273ab447cb28	Copybara
3dcd8e0cf7403ede8d56df9d53df26266176c3c9255a5979da08f5e8bb60ee3f	Nexus
11d926b4e7068914d27200e1aebcbc5e255088ae588a50a1f8f0520771bb6b15	TgToxic
cbfe8e5b2021bfb5d3f5941b849cba4c4c5ebc13983c914e73d8bfcc78075e27	Crocodilus

C2 COMMAND COUNTS AND HANDLING PATTERNS IN ANDROID MALWARE FAMILIES

The following table presents the total number of C2 commands for various *Android* malware families based on research from multiple organizations. We sincerely appreciate their valuable contributions in making this information publicly available, which has been essential for our research. The table also includes the C2 command-handling patterns we observed for each malware family.

Malware family	Number of C2 commands	C2 command-handling pattern
AhRat [8]	6	if-else/switch
TgToxic [2]	94	if-else/switch
Hook [9]	48	if-else/switch
Octo [10]	31	if-else/switch
AlienBot [11]	27	if-else/switch
Cerberus [6]	13	Constant strings class (instance field)
Crocodilus [12]	25	Constant strings class (instance field)
Nexus [5]	39	Constant strings class (static field)
Copybara [4]	59	Array
XLoader [3]	20	HashMap