DroidGuard: A Deep Dive into SafetyNet

Romain Thomas – me@romainthomas.fr

Abstract

SafetyNet is the Android component developed by Google to verify the devices' integrity. These checks are used by the developers to prevent running applications on devices that would not meet security requirements but it is also used by Google to prevent bots, fraud & abuse.

In 2017, Collin Mulliner & John Kozyrakis made one of the first public presentations about SafetyNet and a glimpse into the internal mechanisms. Since then, the Google anti-abuse team improved the strength of the solution which moved most of the original Java layer of SafetyNet, into a native module called DroidGuard. This module implements a custom virtual machine that runs a proprietary bytecode provided by Google to perform the devices integrity checks.

This paper aims at providing a state-of-the-art of the current implementation of SafetyNet. In particular, it presents the internal mechanisms behind SafetyNet and the DroidGuard module. This includes an overview of the VM design, its internal mechanisms, and the security checks performed by SafetyNet to detect Magisk, emulators, rooted devices, and even Pegasus.

Keywords: Android, Obfuscation, Remote Attestation, SafetyNet, VM-based Obfuscation, Mixed Boolean-Arithmetic

1 Introduction to SafetyNet

SafetyNet aims at providing information about the integrity of an Android device to make sure that applications which have to deal with sensitive assets, are not running in an environment that could threaten or weaken the security of these assets.

From a developer's point of view, SafetyNet can be seen as an oracle that basically outputs two information about the device's integrity[1]:

CTS Profile Match: detect unlocked bootloader, custom ROM, uncertified device, …

Basic Integrity: detect emulator, rooted devices, hooking frameworks, …

Depending on the values of *Basic Integrity* and/or *CTS Profile Match*, the developers could perform specific actions like disabling functionalities or stopping the application.

The current implementation of SafetyNet relies on a Google's internal component named DroidGuard. This component is quite obscure with very few information about its internal functionalities but it seems widely involved for detecting misuse of the Android platform (bot, spam, root state, ad fraud, …).

By trying to understand how SafetyNet works, I ended up with reverse-engineering the virtual machine implemented by DroidGuard. This analysis of SafetyNet was motivated by the end-of-life of Magisk-hide.

The analysis of Android applications — especially in the gaming and banking industries — can require to circumvent SafetyNet checks and this article aims at providing a better understanding about the strength and the weaknesses of SafetyNet.

2 SafetyNet Workflow

When an application requests a SafetyNet attestation, different layers, and different processes are involved in the generation of this attestation. The Figure 1 depicts an overview of the attestation process.

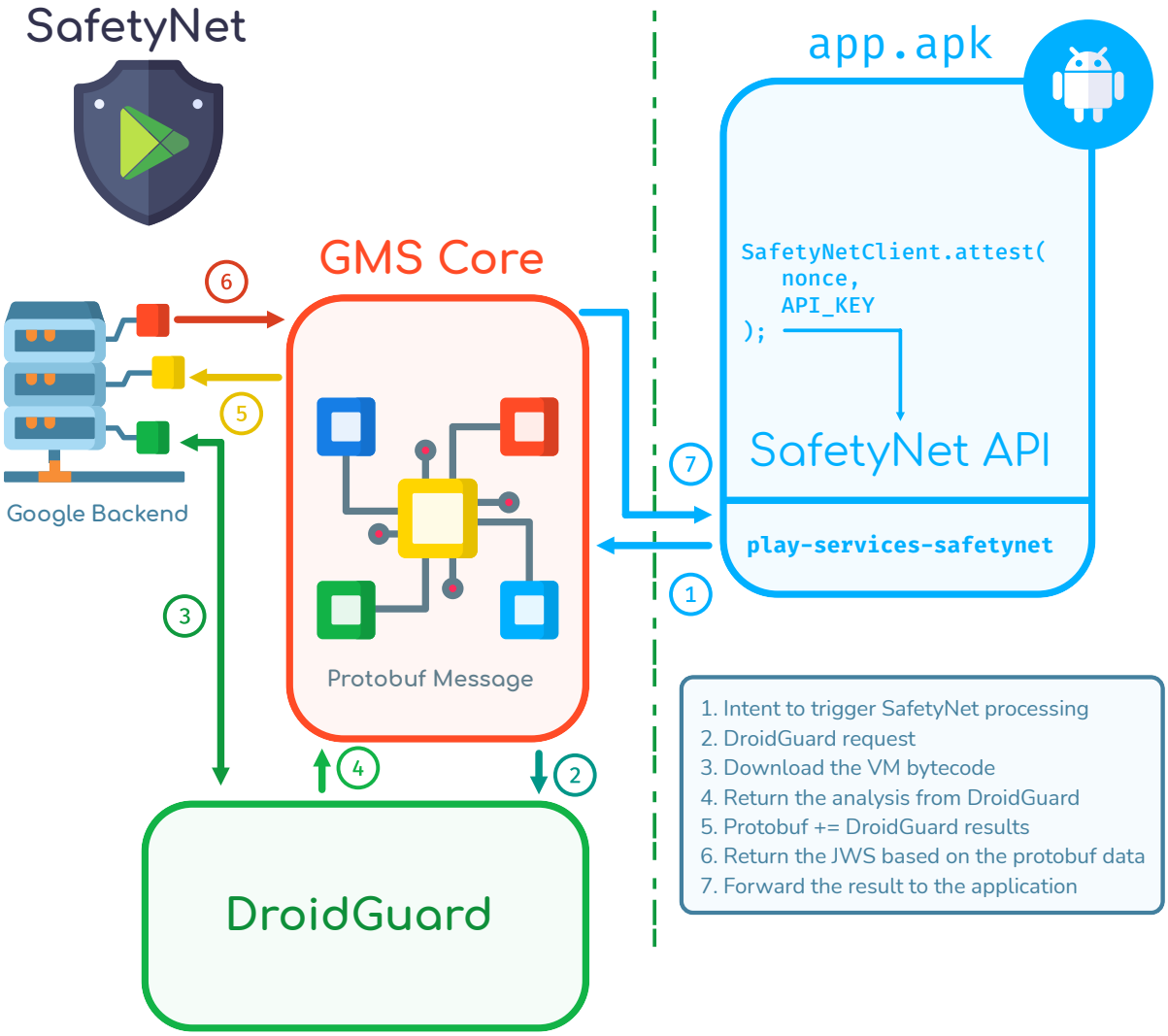

Figure 1: SafetyNet Workflow

Firstly, the application creates a SafetyNet request with the high-level API exposed by the Google SafetyNet SDK [1]. This API takes a nonce and an API key that are bundled into an Android intent which is sent to the **G**oogle **M**obile **S**ervice (**GMS**). The SafetyNet SDK adds other information to the Android intent such as the package name of the application.
The nonce is mostly used to prevent replay attacks while the API key is used by Google to identify the app developers.

When GMS Core receives the intent, it starts to build a Protobuf message that will be used by the Google backend to determine if the device has been tampered with or not. In particular, the information embedded in this message are used to **determine the values of** `Basic Integrity` **and** `CTS Profile Match`.

The structure of this Protobuf message has already been reversed and is publicly available on Github[3]. By monitoring the network communications going through Cronet[9], we can intercept the Protobuf message given in the Listing 1.

[1] com.google.android.gms:play-services-safetynet

```
SafetyNetData = {
  nonce           = [ca ee ...]
  packageName     = "com.demo.snet"
  signatureDigest = [66 49 ...]
  fileDigest      = [fa 0a ...]
  gmsVersionCode  = 213918046
  suCandidates = {
    fileName = "/system/bin/su"
    digest   = [25 53 ...]
  }
  seLinuxState = {
    supported = true
    enabled   = true
  }
  currentTimeMs = 1638672572674
  googleCn      = false
}
```

Listing 1: Protobuf Data Associated with a SafetyNet Request

As we can observe in the Listing 1, the Protobuf message embeds information about the application (package name, signature, APK checksum) as well as device healthy checks such as:

SELinux state: If SELinux is present and enforced

Root check: If `su` binaries are found on the device

The root checks are performed in a Java class of GMS Core and consist in checking predefined `su` paths:

```
package p000;
/* renamed from: aljb */
final class RootChecker {

  /* renamed from: a */
  private static final String[] f23781a = {
    "/system/bin/su",
    "/system/xbin/su",
    "/system/bin/.su",
    "/system/xbin/.su"
  };

  /* renamed from: a */
  public static List getRootFile() {
    ...
    return arrayList;
  }
}
```

There are similar checks for the SELinux status in another part of GMS Core.

The information of `SafetyNetData` are wrapped into another Protobuf message[4] that basically extends the previous information with data coming from *DroidGuard*.
The Listing 2 shows the layout of this extended Protobuf message.

```
{
  SafetyNetData    = { nonce = [ca ee ...], packageName = "com.demo.snet"}
  DroidGuardResult = "CgZpApMYiWYSi9cB [...]"
}
```

Listing 2: Protobuf Message with the DroidGuard Data

As it is explained in the next sections, DroidGuard is an APK that implements a custom virtual machine used to run a proprietary bytecode.

More concretely, and in the context of SafetyNet, DroidGuard is used to run a bytecode that collects evidence about the device's integrity. In particular, the running bytecode performs the advanced root checks, collects information about the bootloader, check if Frida is running ….

This bytecode is also used to encode and generate the `DroidGuardResult` attribute of the Protobuf message previously mentioned (Listing 2).

The Protobuf message wrapping both, `SafetyNetData` and `DroidGuardResult` is sent by GMS Core to the Google SafetyNet backend that returns a JWS with the following payload:

```
{
  "nonce":          "<base64 encoded>",
  "timestampMs":    1638672572674,
  "apkPackageName": "com.demo.snet",
  "apkCertificateDigestSha256": ["<base64 certs>"],
  "ctsProfileMatch": false,
  "basicIntegrity":  true,
  "advice":          "RESTORE_TO_FACTORY_ROM",
  "evaluationType":  "BASIC,HARDWARE_BACKED"
}
```

The values of `ctsProfileMatch` and `basicIntegrity` are determined by the results of DroidGuard and, to a lesser extent, by the early checks on SELinux and the `su` binaries. Finally, this JWS is forwarded by GMS Core to the application that created the request.

Key Points

- SafetyNet's detection logic relies on:
  1. Quick & simple Java checks (su candidates and SELinux status)
  2. Heavy and *expensive* DroidGuard checks
- DroidGuard implements a custom VM that runs proprietary bytecode provided by Google.
- The result of DroidGuard is used by the Google **backend** to determine the values of `ctsProfileMatch` and `basicIntegrity`.
- The final JWS is generated in the Google backend, not on the device.

This section highlighted the role of DroidGuard in the attestation process. The next section deals with the internal structures of DroidGuard.

3 DroidGuard: The VM behind SafetyNet

DroidGuard is part of the Google Mobile Service but it is not, strictly speaking, embedded in the GMS APK[2]. By looking at the manifest file of the GMS application, we can observe a service associated with DroidGuard:

```
<service android:name=".droidguard.DroidGuardService" android:process="com.google.android.gms.unstable">
  <intent-filter>
    <action android:name="com.google.android.gms.droidguard.service.INIT"/>
    <action android:name="com.google.android.gms.droidguard.service.PING"/>
    <action android:name="com.google.android.gms.droidguard.service.START"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</service>
```

We can also notice that this service runs in a different process (`com.google.android.gms.unstable`) than GMS Core (`com.google.android.gms`). When an application requests a SafetyNet attestation, at some point the `DroidGuardService` is triggered and spawn a new process if it is not already running.

`DroidGuardService` encompasses different functionalities, and one of those is to check if the device has the **latest** version of DroidGuard. It turns out that the *real* implementation of DroidGuard is actually located in an apk stored in `/data/data/com.google.android.gms/app_dg_cache/<hash>/the.apk` and dynamically loaded by `DroidGuardService`.

[2] `com.google.android.gms`

The value associated with the `DroidGuardResult` attribute of the Protobuf message mentioned in the Listing 2) is actually generated from this apk (`the.apk`).

As it will be discussed in this section, `the.apk` implements a virtual machine (VM) that is used to generate the `DroidGuardResult` value referenced in the Listing 2

The bytecode executed by the DroidGuard virtual machine is dynamically downloaded from the Google backend servers and **unique** for each attestation request.

SafetyNet Attestation Flow

1. Upon receiving a SafetyNet attestation's request, GMS Core download the latest version of DroidGuard (if not already downloaded)
2. DroidGuard downloads a unique VM bytecode from the Google's servers (unique for each attestation request).
3. DroidGuard runs the bytecode which performs integrity's checks, generates a token and, bind the token with the attestation's parameters.
4. The token is sent to the Google's servers to determine `ctsProfileMatch` and `basicIntegrity`

3.1 Overview

The APK (`the.apk`) embedding the VM is relatively small compared to GMS Core. It embeds about 60 classes (compared to ~ 63 000 classes in GMS Core) in which only a few of them are relevant. The important methods are implemented in the class:

`com.google.ccc.abuse.droidguard .DroidGuard`

This class declares a set of **native** methods among which we find:

- `long initNative(Context context, String flow, byte[] bytecode, ... )`
- `byte[] ssNative(long j, String[] strArr)`
- `void closeNative(long j, String[] strArr)`

On a typical attestation request, `initNative` is called first to initialize the DroidGuard VM and to run the bytecode provided in the third parameter. Most of the SafetyNet checks (root checks, bootloader status) are performed during this call.

Then, it follows `ssNative` that takes a pointer to the `C++` DroidGuard VM object as the first parameter (`long j`). The second parameter is a *content binding* which turns out to be the SHA-256 checksum of the `SafetyNetData` Protobuf message (cf. Listing 1). The output of this `ssNative` function is the actual `DroidGuardResult`. While `initNative` runs and generates integrity's information **independantly** of the application that triggered the request, `ssNative` ensures that the `DroidGuardResult` is bound and unique for the application.

Finally, `closeNative` cleans the VM, cleans the buffers dynamically allocated and the Java references.

By tracing the parameters of these functions, we get the following sequence:

```
DroidGuard.initNative(DroidGuardChimeraService@a5bdb0b,
                      flow: 'attest', vmBytecode: Bytes Array, ... )
DroidGuard.ssNative("{contentBinding=<Protobuf SHA-256 Hash}"): CgZpApMYiWYSi9cB[ ... ]
DroidGuard.closeNative();
```

These methods are implemented in a native library for which the name is not meaningful (e.g. `libd58FDD24B24CD.so`) but in which the ELF metadata is more relevant:

```
→ readelf -d libd58FDD24B24CD.so|grep SONAME

0×000000000000000e (SONAME)             Library soname: [libdroidguard.so]
```

`libdroidguard.so` and the running bytecode contain the **main logic of SafetyNet**. Compared to the analysis of J. Kozyrakis and C. Mulliner in 2017, it looks like the current architecture of SafetyNet drop most of the Java layers and only relies on DroidGuard[3]

From the section 2 SafetyNet Workflow, we identified that the content of `DroidGuardResult` is significantly used to determine the boolean values of `basicIntegrity` and `ctsProfileMatch`. Therefore at this point, the main challenge is to figure out how the output of `ssNative` is generated.

3.2 The Virtual Machine Internals

By Googling about DroidGuard, we find very few information about this component. Nevertheless, one blog post[5] references some keywords that ring the bell.

The blog post is two years old but it turns out that after analysis, the current implementation is still based on a VM. Compared to the blog post, we can notice some changes such as using `JNI_OnLoad` to register `initNative` and `ssNative` instead of exporting them through `JNIEXPORT`.

`libdroidguard.so` is pretty small, about 400 functions in a 350KiB file, and most of the strings are encoded. Needless to say that the library is stripped and does not contain metadata like RTTI. The analysis of `libdroidguard.so` has been performed in a pure blackbox approach.

While skimming over the library's functions, we can quickly figure out that `libdroidguard.so` is written in `C++` and mostly relies on two STL containers:

1. `std::vector`
2. `std::string` (or `std::basic_string<uint8_t>`)

Actually, the `std::string` container is far more used in the code than the `std::vector` container. One hypothesis is that this container is preferred by the DroidGuard developers to leverage the small strings optimization made by the STL[6]. The other hypothesis is that this container is used because it's a bit more complicated to deal with when reverse-engineering `C++` code (cf. 6 Reverse-Engineering `C++`: What We Should Be Aware Of?).

The main `C++` object implemented and managed by `libdroidguard.so` is the implementation of the VM itself which is a `C++` class. We will name this object `DroidGuardVM` in the rest of this paper.

To get a good understanding of the high-level functionalities behind the VM, we have to address at least two points:

1. Figure out the memory layout of the `DroidGuardVM` object
2. Understand the purpose of the VM handlers

Through static analysis, we can infer that the **first** class attribute of the `DroidGuardVM` object, is a **pointer** to the current registers frame. The `DroidGuardVM` can use up to 256 **typed** registers that are indexed by a `uint8_t` integer.

A typed VM register is defined as a pair of two values:

1. Its *effective* value (`uintptr_t`)
2. Its type (`enum:uint8_t`)

After the analysis of the library functions which manipulate these registers, we can figure out the different types supported by the `DroidGuardVM`:

DG Type	Raw Type
Long	`int64_t`
Int	`int32_t`
Double	`double`
Pointer	`void*`
JNI Object	`jobject`
String	`std::string*`

[3] DroidGuard was mentioned by J. Kozyrakis and C. Mulliner in their talk/blog post though

Regarding the values of the registers, the Google anti-abuse team put a lot of effort to protect the content of the VM registers, and more generally, the data flow of the VM. In particular, all the registers' values are encoded such as when accessing `regs[0×12]`, we actually get an **encoded** representation of the original value:

```
regs[0×12] := enc(original_value)
```

In addition to registers values encoding, DroidGuard encodes the content of the string buffers (`std::string`) with a key derived from the register index and from the VM key. The buffers are only decoded when the VM needs to access the original content.
It is worth mentioning that the data flow of the VM is critical enough to have a dedicated library function that aims at transferring an encoded buffer from a register into another without clearly decoding the original content of the source buffer. More information about the data flow obfuscation are given in the next section (3.3 Data Flow Obfuscation).

The `enum` mapping of the register's types is changing for **each new update** of DroidGuard. It means, for instance, that the `long` type can be associated with the integer **2** for a given version of DroidGuard and **5** in another version.

During the setup of the VM, some of these registers are initialized with contextual values. The Table 1 lists some of these values that are set by a function closes to the `DroidGuardVM` constructor.

Register	Initial Value
`r[03]`	Extra parameters
`r[04]`	Flow (e.g. `attest`)
`r[08]`	JNI ref on `DroidGuardChimeraService`
`r[0a]`	Syscall function
`r[0d]`	Bytecode buffer address
`r[0e]`	Error code?
`r[10]`	JNIEnv*
…	…

Table 1: Some Initial Registers Values

Identifying the initial register values can be helpful while reverse-engineering the VM handlers. For instance, the syscall helper function is accessed when doing a function call:

```
void DroidGuardVM::make_call() {
 /* 0×03c75c: */ this→read_byte_vector(key: 0×9849e8d9ba42ccdc): {0×09, 0×e4, 0×09};

 // ============================================================

 /* 0×03c824: */ this→get_pointer(reg: 0×a): &vm_syscall_helper

 // ============================================================

 /* 0×03c8a0: */ this→prepare_params(in_reg: {0×09, 0×e4, 0×09},
                                     out_str: {/data/user/0/com.google.android.gms/cache/.xfhrfg}):
                 {&vm_syscall_helper, /* openat */ 0×38, 0×0, /* file.c_str() */ 0×7c23bdda50, 0×0}
 /* 0×008598: */ openat("/data/user/0/com.google.android.gms/cache/.xfhrfg"): -1
 /* 0×03c930: */ this→set_register(0×29, REG_TYPES::INT, -1)
 …
}
```

These registers are used by the **VM handlers** which provide low-level primitives for the dedicated bytecode. As in most of the VM-based obfuscation schemes, we find handlers for arithmetical operations (xor, addition, subtraction), conditional branches, comparison, and we also find more specialized handlers such as:

- Calling a Java function through the JNI
- Doing a syscall
- Dynamically resolving a symbol (through `dlsym`)

- Accessing a Java field
- Doing `sha256_init`, `sha256_update` and `sha256_final`
- …

From a reverse-engineering's point of view, the VM handlers seem to be member functions of the `DroidGuardVM` object. In particular, they do not take any parameters and they do not return a value. The handlers only update the internal state of the VM which includes the register values.

From a memory point of view, the VM handlers are indexed right after the register frames:

```
class DroidGuardVM {
  private:
  registers_t*                                registers;
  std::vector<registers_t*>                   frames;
  std::array<void(DroidGuardVM::*)(), 0x200> handlers;
  ...
};
```

For each new version of `libdroidguard.so`, the position of the handlers in the `handlers` array attribute is shuffled. For instance, `DroidGuardVM→handlers[4]` can point to the VM handler associated with a JNI call and, in a new release of DroidGuard it could then be associated with the SHA-256 processing handler.

This randomization is likely used to prevent fingerprinting and automation from past reverse-engineering.

There are no universal methods to reverse the different VM handlers, but most of them share the following schema, which could be used as a VM's handler footprint template:

```
void DroidGuardVM::handler() {
  decode_operands();

  perform_handler_operation();

  write_register_results();
}
```

To concretely understand these operations, let's consider the VM handler that aims at comparing two registers:

```
void DroidGuardVM::cmp_equal()
```

First of all, the handler starts by decoding the instruction's operands:

```
static constexpr uint8_t PC_REG_IDX = 0x12;

uint32_t pc;
uint8_t tmp;

// == Read the first operand ==
pc = read_register(PC_REG_IDX);
pc = decode(&tmp, pc, sizeof(tmp));
set_register(PC_REG_IDX, pc);

uint8_t OP_DST_IDX = MBA1_DECODE(tmp);

// == Read the second operand ==
pc = read_register(PC_REG_IDX);
pc = decode(&tmp, pc, sizeof(tmp));
set_register(PC_REG_IDX, pc);

uint8_t OP_LHS_IDX = MBA2_DECODE(tmp);

// == Read the third operand ==
pc = read_register(PC_REG_IDX);
pc = decode(&tmp, pc, sizeof(tmp));
set_register(PC_REG_IDX, pc);

uint8_t OP_RHS_IDX = MBA3_DECODE(tmp);
```

The `cmp_equal` handler uses three operands:

`OP_DST_IDX`: The destination register for the comparison result.

OP_LHS_IDX: The left-hand side register to compare with.

OP_RHS_IDX: The right-hand side register to compare with.

After decoding the instruction's operands, the decoded operands are processed through inlined Mixed Boolean-Arithmetic (MBA) functions.
These MBA are specific to each handler and they change for each new version of DroidGuard. As a result, if somehow we were able to disassemble or lift the bytecode from DroidGuard, we would have to extract or reverse all the MBA associated with the VM handlers. More details about the MBA are given in the section 3.3 Data Flow Obfuscation.

After the instruction decoding, we find the handler's *payload* which consists in the handler's logic. In our example, this logic consists in checking if two registers are equal according to their types:

```
bool are_equal = false;

reg_t& RHS = get_reg(OP_RHS_IDX); reg_t& LHS = get_reg(OP_LHS_IDX);

if (RHS.type == LHS.type) {
  switch (RHS.type) {
    case JNI:
      are_equal = this->env_->IsSameObject(decode(RHS.value), decode(LHS.value)); break;
    case LONG:
      are_equal = decode(RHS.value) == decode(LHS.value); break;
    case INT:
      are_equal = (int)decode(RHS.value) == (int)decode(LHS.value); break;
    case DOUBLE:
      are_equal = (double)decode(RHS.value) == (double)decode(LHS.value); break;
    case STR:
      // byte-per-byte comparison
    case NONE:
    default:
      are_equal = false;
  }
}
```

Finally, the result of the comparison is stored in the register read during the instruction decoding:

```
this->set_register(OP_DST_IDX, are_equal);
```

Among the 98 VM handlers implemented in the reverse-engineered version of DroidGuard, I managed to recover the functionalities of 66 of them. To *bypass* the `basicIntegrity` checks, only 5 of them are worth identifying.

The reverse-engineering of the VM handlers helps to understand the layout of the `DroidGuardVM` object and vice versa. In the end, we get the class layout for the `DroidGuardVM` object listed in the Annex 5. We can notice that this layout embeds an array of `HMAC_CTX`. Actually, it seems that DroidGuard is able to protect the integrity and the authenticity of the VM registers. During the analysis of the VM, I noticed that only one register leverages this functionality which is the register that contains the `DroidGuardResult` token. The DroidGuard's HMAC-SHA256 is based on the BoringSSL's functions[4] which are triggered when appending data to the register that holds the `DroidGuardResult`

```
VMH_concat_buffer() {
  // [...]
  if (!init) {
    HMAC_Init(this->hmac[REG_RES_IDX], key, key_len, EVP_sha256());
  }
  HMAC_Update(this->hmac[REG_RES_IDX], data, len);
  // [...]
}
```

The HMAC's secret key is embedded and decoded (xor between two integer) by the bytecode and stored as a long integer in a VM's register. By hooking[5] `HMAC_Init()` we can easily observe the HMAC's secret key.

[4] stripped in the library
[5] Hooking functions in DroidGuard requires to bypass internal internal integrity checks which is out of scope of this article

Key Points

- DroidGuard is a virtual machine with 256 typed registers and about 100 VM handlers.
- The VM's bytecode is dynamically downloaded from the Google's servers and **unique** per attestation.
- The VM operands are protected with MBA which hinders running the bytecode outside the DroidGuard library.
- The VM implements registers integrity based on BoringSSL's HMAC-SHA-256. This integrity seems only used for the register that contains the `DroidGuardResult` though.

3.3 Data Flow Obfuscation

As mentioned in the previous section, the data flow of DroidGuard is protected with different techniques.

We define the data flow of DroidGuard as the registers values, the internal buffers, the strings and the instructions operands.

One of these protections is to apply Mixed Boolean-Arithmetic expressions (MBA) on the instructions' operands[6]. There are many MBA expressions across `libdroidguard.so` and the Table 2 references a few of them.

MBA Expressions
$((X \wedge 1) \ll 1 + X \oplus 1$
$((-1 \oplus X \ll 1) + X) \oplus$ 0xffffffff
$(X \mid \sim Y) + (Y \mid \sim X) - (X \oplus \sim Y) \times 2$

Table 2: Examples of MBA Expressions Found in DroidGuard

The MBA expressions change for each new version of the DroidGuard VM but most of them can be automatically *simplified* by combining Miasm and msynth[8]. The Table 3 lists the simplification of the previous MBA expressions.

MBA Expressions	Resolution
$((X \wedge 1) \ll 1 + X \oplus 1$	$X+1$
$((-1 \oplus X \ll 1) + X) \oplus$ 0xffffffff	$X \times 3$
$(X \mid \sim Y) + (Y \mid \sim X) - (X \oplus \sim Y) \times 2$	$X \oplus Y$

Table 3: MBA Expressions Resolved with msynth

The second mechanism widely used to protect the data flow is the encoding of the string buffers. Basically, the string buffers associated with registers holding this kind of value are xored with a keystream derived from the register index and an encoding key bound to the `libdroidguard.so` (i.e. a static value). The Figure 2 summarizes the decoding process.

[6] Actually the MBA are not limited to the instruction's operands and are widely used in DroidGuard.

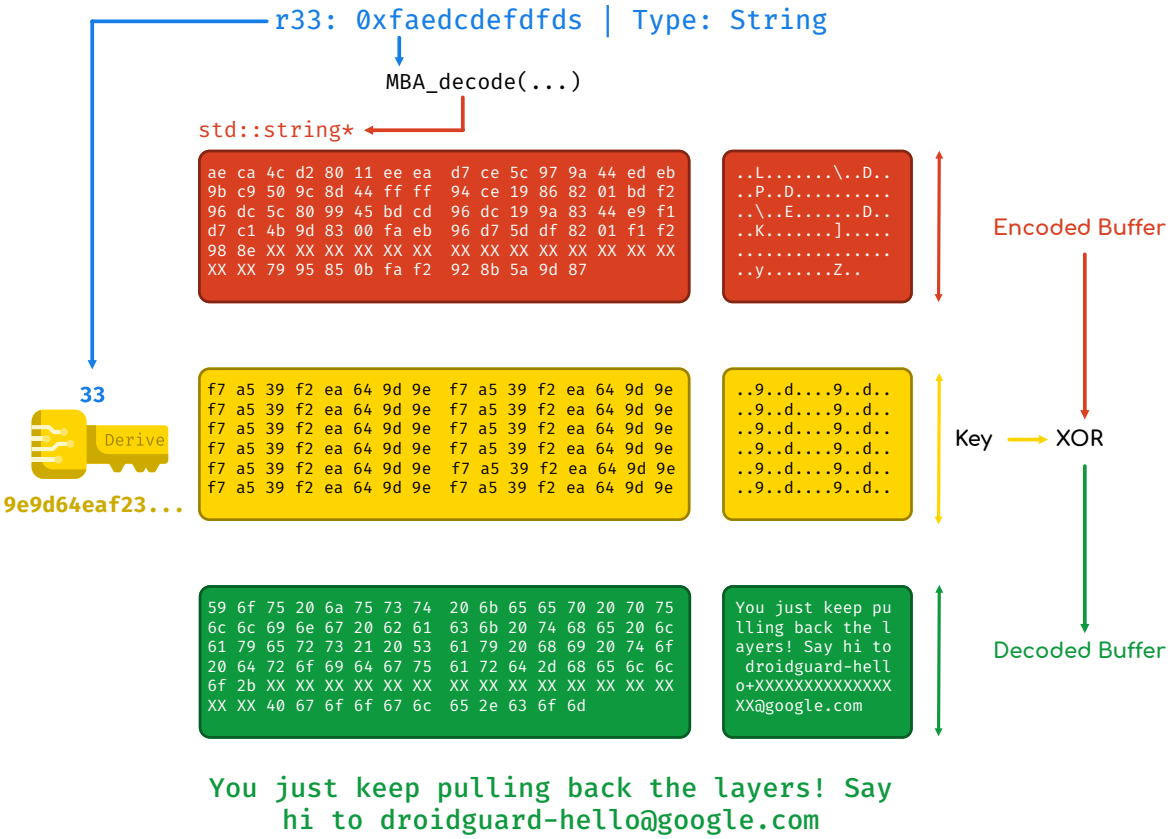

Figure 2: Registers Content Protection

The buffers are decoded **only** when there are used. It means that their clear content cannot be naively observed while iterating over the VM's registers. It can also be mentioned that when transferring an encoded buffer into another register (e.g. `reg[0×cd]` → `reg[0×ab]`), DroidGuard takes care of not leaking the content in a temporary variable. Because of the xor encoding operation, the encoded buffer can be transferred with this relationship:

$$\texttt{regs[0×ab].buffer[i]} = \texttt{regs[0×cd].buffer[i]} \oplus key_{ab}\texttt{[i]} \oplus key_{cd}\texttt{[i]}$$

Even if the MBA can be — to some extent — resolved, and the buffers encoding algorithm reversed, the trade-off between the time it takes and the outcome was not advantageous. Moreover, the anti-abuse team could completely get rid of these protections and use functions more resilient against reverse-engineering. After trial and error, I decided to address all the encoding layers of DroidGuard through code lifting. With this technique, I did not have to reverse and deeply understand the MBA nor the inner mechanisms of the key derivation for the new versions of the VM. I based the code lifting approach with open-source tools like QBDL [13] and Unicorn [14].

Code lifting enables to run the instructions that are relevant to meet our needs, regardless of the global complexity of the function. The main reverse-engineering task was *only* identifying these relevant instructions.

Key Points

- DroidGuard mainly protects its data flow with Mixed Boolean-Arithmetic Expressions and buffer encodings.
- The MBA are small enough to be simplified with msynth[8].
- More generally, the encodings and the MBA can be attacked with code lifting which scales with the regular updates.

3.4 Code Integrity

In addition to data-flow obfuscation, DroidGuard implements anti-hooks mechanisms through code integrity checks. If the integrity of the (in-memory) `.text` section is not correct, at some point the bytecode takes a branch that performs irrelevant operations.

Hooking is a strong reverse-engineering primitive and bypassing these integrity checks greatly simplifies the analysis of `libdroidguard.so`.

During the initialization of the VM, DroidGuard stores the address of `JNI_OnLoad` in a virtual register:

```
reg_move_info_t minfo;
minfo.reg_idx = 0×13;
minfo.keys    = this→keys_;
minfo.reg_ptr = &this→registers[0×13]
minfo.env     = this→env_;
this→set_register(&minfo, TYPES_POINTER, (uintptr_t)&JNI_OnLoad ^ 0×cf7dba8687cac60e)
```

Later on during the execution of the bytecode, this register is accessed and the address of `JNI_OnLoad` is used to compute the range of addresses to verify.

```
this→set_pointer() {
  r[20] := 0×cf7dba8687cac60e;
}
this→sha256_init();
this→xor() {
  r[21] := r[20] ^ r[13]; // r[21] := &JNI_OnLoad
}
this→add() // Compute the .text addresses range to check
...
this→sha256_update() {
  sha256_update(.text!0×123, 0×456);
}
this→sha256_final()
```

By hooking the BoringSSL SHA-256 functions[7] and by checking if the input buffer encloses the `.text` section, we can change the pointer to an address that points to a **copy** of the original `.text` section. As a result, the checksum is computed on the copied `.text` section and its value is consistent with the value compared for the integrity check. The DroidGuard native library is pretty small such as a copy of the `.text` section has a small memory footprint.

Key Points

- DroidGuard implements code integrity checks on the `.text` section.
- This integrity checks can prevent hooking and software breakpoints.
- The checksum used to verify the `.text` section is SHA-256 from BoringSSL.
- It can be bypassed by changing the input of the SHA-256 to a **copy** of the `.text` section.

4 The Device Integrity Checks

With a good overall understanding of the VM internals and the VM handlers, we can target specific handlers to highlight the SafetyNet checks and/or disable some of them.

Regardless of the obfuscation scheme used to protect the control flow and the data flow of the detection algorithm, at some point the detection logic needs to interact with the operating system through public API or syscalls. This is the critical point where we can observe the functions and their parameters with clear values.

In the case of DroidGuard, we just have to target 5 VM handlers to get a good overview of the devices integrity checks.

[7] which were stripped and reverse-engineered

Magisk & Root Detections As in most of the root detections techniques, the bytecode running through DroidGuard checks predefined `su` paths listed in the Annex 8.1

To enhance the predefined `su` path checks, DroidGuard iterates over the content of some directories (like `/sdcard`) to verify if they contain files that match keywords like `giefroot` or `sbin_orig`. In that case, DroidGuard performs extended checks on these directories and files. By looking at the list of these keywords, we can notice entries like `pegasus.apk` or `coldboot_init` which suggests that DroidGuard is able to identify devices that would have been compromised by Pegasus. The list of the identified keywords is given in the Annex 8.5.

In addition to the file checks, DroidGuard tries to detect Magisk by looking for system properties like `init.svc.magisk_pfsd` and by inspecting the current mounting points (`/proc/self/mounts`). The list of the system properties is given in the Annex 8.1.1.

DroidGuard also covers legacy rooting tools like KingRoot[11]. The detection of this tool seems to be performed through an analysis of the environment variables (cf. Annex 8.1.2)

Hooking Frameworks As mentioned in the API documentation[1], SafetyNet aims at detecting *"Signs of other active attacks, such as API hooking"*. Basically, this detection is done through:

- Inspecting the modules loaded in `/proc/self/maps`
- Iterating over the loaded modules with `dl_iterate_phdr`
- Inspecting `/system/bin/app_process` in the case of the Xposed detection

The Annex 8.2 contains the list of the modules monitored by SafetyNet.

The design of SafetyNet is such that these checks are **exclusively** achieved in the context of the process `com.google.android.gms.unstable`. In particular, it means that these checks do not enable SafetyNet to detect Frida in the application that requested the SafetyNet attestation.

Emulators Since DroidGuard is also involved in the detection of bots and protocol emulating script, it aims at detecting emulators.

The logic behind this detection mostly relies on the system properties (cf. Annex 8.3.1), but also on the device hardware characteristics like the memory, the battery, and the device's screen (cf. Annex 8.3.2).

Bootloader Verification The information about the bootloader are decisive in the determination of the `ctsProfileMatch` value. SafetyNet collects the status of the bootloader from different sources:

1. **System Properties:** `ro.boot.flash.locked`, `ro.boot.vbmeta.device_state` (cf. Annex 8.4.1)
2. **Java API:** `getSystemService("persistent_data_block").getFlashLockState()`
3. **Hardware Attestation:** `KeyStore.getCertificateChain()`

The values of the system properties and the Java API can be easily modified but the result of the certification chain is a bit more tricky to circumvent.

The SafetyNet's hardware attestation relies on the Android public API which is described in the Android developers website[12]. To perform this attestation, the bytecode running through DroidGuard uses VM's handlers dedicated to JNI calls. As it is listed in the Listing 3, it creates and instantiates all the Java objects required to build the attestation. This part of the VM execution has been translated in Java in the Annex 8.4.3.

```
VMH_read_buffer()
VMH_read_buffer()
VMH_JNI_CallMethod() {
  CallObjectMethodA("KeyGenParameterSpec$Builder.build()"): KeyGenParameterSpec@ca74d81
}
VMH_read_buffer()
VMH_read_buffer()
VMH_read_buffer()
VMH_read_string_at_offset()
VMH_JNI_GetStaticField() {
  GetStaticObjectField("KeyProperties.KEY_ALGORITHM_EC"): "EC"
}
VMH_read_buffer()
VMH_read_buffer()
VMH_set_uint32()
VMH_read_buffer()
VMH_read_buffer()
VMH_JNI_FindClass()
VMH_read_buffer()
VMH_read_string_at_offset()
VMH_read_buffer()
VMH_JNI_CallStaticObjectMethod() {
  NewStringUTF("AndroidKeyStore"): 0×41
  CallStaticObjectMethodA("KeyPairGenerator", "KeyPairGenerator.getInstance",
                          "EC", "AndroidKeyStore"): "KeyPairGenerator$Delegate@f9f7e26"
}
```

Listing 3: VM Trace Associated With the Hardware Attestation

At the end of the execution, DroidGuard calls `getCertificateChain()` which contains the certificates chain as described in the documentation[12]. It is worth mentioning that the root certificate is signed by a Google's hardware-backed private key and this chain contains a certificate which embeds hardware-signed information among which the status of the bootloader (c.f Annex 8.4.3).

DroidGuard does not read and does not take any integrity decision regarding this certificate chain. The whole chain is sent in the DroidGuardResult & Telemetry Data that defers the integrity decision by the Google's backend.

As a result of this analysis, it has been possible to bypass most of these checks and get a `basicIntegrity` value at `true`[10]. Bypassing the `ctsProfileMatch` flag is another challenge that shifts the attack to finding an boot chain or a low-level vulnerability.

Key Points

- The device's integrity checks associated with the `basicIntegrity` flag rely on external API (libc's function, syscalls, Java API) that are possible to monitor and modify.
- Some strings suggest that the SafetyNet's bytecode embeds heuristics to identify if Pegasus is present on the device.
- All the checks are done in a process different from the app that requests the SafetyNet attestation. This limits the detection of hooking frameworks like Frida.
- The hardware-backed attestation relies on `KeyStore.getCertificateChain()`.

5 DroidGuardResult & Telemetry Data

All the DroidGuard reverse-engineering was motivated by the understanding of the content behind the `DroidGuardResult` value referenced in the Listing 2.

After analysis, it turns out that this value is actually a Protobuf message that wraps *telemetry* data. These data are collected throughout the execution of the bytecode and they are stored in a dedicated register.

Compared to classical string encoding (cf. 3.3 Data Flow Obfuscation), the buffer that contains the telemetry data is protected with another layer of encoding that involves an HMAC secret key and other encoding keys. Nevertheless, it is still possible to access the underlying content through hooking and code lifting.

Without really reversing the encoding layers, we can extract the telemetry data listed in the Annex 8.7.

6 Reverse-Engineering C++: What We Should Be Aware Of?

As mentioned in the previous sections, DroidGuard is written in C++ and manages a class that we called DroidGuardVM. For better or for worse, the ABI and the STL optimizations can be tricky. This section deals with non-trivial C++ ABI features.

6.1 The Pointer this

Non-static member functions of C++ classes always start with this as the **first** parameter of the function. While reverse-engineering, and more precisely, while trying to understand the layout of the DroidGuard VM, this property can help to identify the VM class member functions that aims at interacting with class data from those that are *helpers*.

Nevertheless, it exists an exception where non-static class member functions do not have this as the first parameter.

6.2 Copy Elision

To avoid a copy of a C++ object returned by a function, the C++ standard[8] requires to reference the returned object in the parameter of the function such as it can be constructed in-place by the function.

Concretely, if the consider the following DroidGuardVM function:

```
std::vector<uint8_t> DroidGuardVM::read_byte_vector(size_t enc_size)
```

The *real* prototype of the function is actually:

```
void read_byte_vector(std::vector<uint8_t>* out, DroidGuardVM* vm, size_t enc_size)
```

So from a reverse-engineering point of view, we observe 3 parameters but in fact, two of them are ABI specific.

6.3 std::string Optimization

When we need to store *small* bytes, the std::string container can be more interesting over a std::vector<uint8_t> as most of the C++ Standard Template Library (STL) implements an optimization for small strings[6].

From a reverse-engineering point of view, this optimization can be tricky to spot in particular when the std::string functions are inlined.

```
void* x1;
uint64_t x2;

uint64_t x3 = *x1
if ((x3 & 1) != 0) {
  x2 = *(x1 + 8)
} else {
  x2 = x1 >> 1;
}
```

```
std::string* x1;
uint64_t x2;

uint64_t cap = x1->cap
if ((cap & 1) != 0) {
  x2 = x1->size();
} else {
  x2 = x1 >> 1; // Small string optimization
}
```

Listing 4: Generated Code when accessing std::string::size

If we consider the function std::string.size(), on the left-hand side of the Listing 4 we have the code generated by the compiler which does not contain hints about the type of the variable x1. The main

[8] since C++ 11

reverse-engineering effort is to understand the type of this variable for which the memory dereference and the shift makes sense on the right-hand side according the `std::string` optimization.

During the analysis of DroidGuard, we can encounter this kind of optimization in different places and it was frequent that the condition `(cap & 1) =/= 0` was re-written with the following MBA:

$$X \oplus \texttt{0xfffffffffffffffe} + 1 \neq (X \mid 1) \oplus \texttt{0xfffffffffffffffe}$$

7 Conclusion

Generally speaking, DroidGuard/SafetyNet successfully achieves its purpose: provide a reliable and efficient solution to detect *compromised/tampered* devices. Regardless of the low-level protections like the Mixed Boolean-Arithmetic expressions or the buffers encoding, its global design with regular updates, a unique bytecode per-request or the dedicated *unstable* process, makes its analysis difficult.

7.1 The Reverse-Engineering Cost

Evaluating the robustness of a solution is a difficult exercise as it depends on the motivation and the experience of the reverser as well as its tools. This research has been done during week-ends and early in the morning which could represent about five straight weeks. In addition, it had to develop dedicated tools to inspect the VM (like dumping all the registers), to trace the VM's handlers and to ease the reverse-engineering of a new version of the VM based on the previous version.

This part adds two straight weeks of work. After having the suitable toolset, I could handle a new version of DroidGuard in a couple of hours.

7.2 The Limits of the DroidGuard/SafetyNet Design

As it has been discussed in the section 2 SafetyNet Workflow, DroidGuard and its integrity checks are performed in a dedicated process[9] and not in the process of the application that triggered the SafetyNet request. It means that DroidGuard is able to identify system-wide tampering (like Magisk or a bootloader unlocked) but it is not able to detect local applications tampering like Frida gadget or native hooking.

There are no integrity checks in the memory space of the application that perfomed the request.

Therefore, RASP[10] solutions are relevant to ensure the application is not locally tampered. In addition, SafetyNet aims at running on a large number of Android devices with disparate brands, versions, hardware etc. On the top of that, SafetyNet must take care of not raising false-positives alerts as it could have a direct impact on the app's developer business. This universality and this attention to avoid false-positives (or not) can be a weakness. For instance, one solution to get rid of the hardware attestation is to make believe that the device does not support hardware attestation[11]. On the other hand, the documentation about the trustworthy of the `ctsProfileMatch` value is also clear:

[... false for] *"Genuine but uncertified device, such as when the manufacturer doesn't apply for certification"*

This is not an issue for applications that aim at running on a specific range of devices, like SoftPOS[12] solutions but it can be a critical issue in the video games industry if the gamers cannot play the game because of a non-Google certified device.

This article intentionally eludes the technicals details to bypass SafetyNet security measures or to perform the attacks, as it is a cat-and-mouse game that everyone can enjoy playing.

[9] `com.google.android.gms.unstable`

[10] **R**untime **A**pplication **S**elf **P**rotection

[11] `https://github.com/kdrag0n/safetynet-fix/blob/57b726c260bb40b838c5d942965282a5a482bdbe/java/app/src/main/java/dev/kdrag0n/safetynetfix/proxy/ProxyKeyStoreSpi.kt#L45`

[12] **S**oftware **P**oint **O**f **S**ale is a solution that allows merchants to accept contactless payments on their smartphones

8 Annexes

8.1 Root Checks

- `/data/local/tmp/su`
- `/sbin/su`
- `/data/local/xbin/su`
- `/bin/su`
- `/data/local/bin/su`
- `/product/bin/su`
- `/system/bin/.ext/su`
- `/system_ext/bin/su`
- `/system/bin/su`
- `/system/xbin/su`
- `/odm/bin/su`
- `/vendor/bin/su`
- `/vendor/xbin/su`

8.1.1 Magisk Detection

System Properties:

- `init.svc.magisk_service`
- `persist.magisk.hide`
- `init.svc.magisk_pfs`
- `init.svc.magisk_pfsd`
- `magisk.version`
- `ro.magisk.disable`

Files Content:

- `/proc/self/mounts`
- `/proc/self/mountinfo`

8.1.2 KingRoot Checks

- `TANGBOX`
- `REDIRECT_SRC1`
- `REDIRECT_DST1`
- `FORBID_SRC1`
- `WHITELIST_SRC1`

8.2 Dynamic Instrumentation Checks

- `libarthook_native.so`
- `libsandhook.edxp.so`
- `libsandhook-native.so`

- libsandhook.so
- libxposed_art.so
- libfrida-gadget.so
- libmemtrack_real.so
- frida-agent-64.so
- libva++.so
- librfbinder-cpp.so
- frida-agent-32.so
- libva-native.so
- libwhale.edxp.so
- libriru_edxp.so
- libriru_snet-tweak-riru.so
- libriru_edxposed.so

8.2.1 Xposed Detection

It checks the content of /system/bin/app_process

8.3 Emulator Checks

8.3.1 System Properties

- init.svc.droid4x
- init.svc.noxd
- init.svc.qemud
- init.svc.goldfish-setup
- init.svc.goldfish-logcat
- vmos.browser.home
- init.svc.ttVM_x86-setup
- ro.trd_yehuo_searchbox
- qemu.sf.fake_camera
- vmos.camera.enable
- init.svc.microvirtd
- init.svc.vbox86-setup
- ro.rf.vmname

8.3.2 Devices Features

Memory:

- getSystemService(Context.ACTIVITY_SERVICE).getMemoryInfo().totalMem
- ApplicationPackageManager.hasSystemFeature(PackageManager.FEATURE_RAM_NORMAL)

Screen Information from getSystemService(Context.WINDOW_SERVICE).getDefaultDisplay()

- getMetrics()

    - DisplayMetrics.widthPixels
    - DisplayMetrics.heightPixels
    - DisplayMetrics.density
    - DisplayMetrics.xdpi
    - DisplayMetrics.ydpi
- getRealMetrics()
    - android.util.DisplayMetrics.widthPixels
    - android.util.DisplayMetrics.heightPixels

Battery Information

`getSystemService(Context.POWER_SERVICE).isInteractive()`

```
BatteryManager.EXTRA_LEVEL
BatteryManager.EXTRA_SCALE
BatteryManager.EXTRA_STATUS
BatteryManager.EXTRA_PLUGGED
```

8.4 Bootloader Status

8.4.1 System Properties

- ro.boot.flash.locked
- ro.boot.vbmeta.device_state
- ro.boot.verifiedbootstate
- ro.boot.vbmeta.digest

8.4.2 Java API

- getSystemService("persistent_data_block").getFlashLockState()
- ApplicationPackageManager.hasSystemFeature(PackageManager.FEATURE_VERIFIED_BOOT)

8.4.3 Hardware Attestation

```
KeyStore ks = KeyStore.getInstance("AndroidKeyStore")
ks.load(null);
ks.aliases(); // Iterate and check the aliases

long rndLong = (new Random()).nextLong()
String alias = "unstable.<hash>." + rndLong.toString()

KeyGenParameterSpec spec = new KeyGenParameterSpec.Builder(alias, KeyProperties.PURPOSE_SIGN)
  .setAlgorithmParameterSpec(new ECGenParameterSpec("secp256r1"))
  .setDigests(KeyProperties.DIGEST_SHA512)
  .setAttestationChallenge(<unique number>)
  .build();

KeyGenerator keyGenerator = KeyPairGenerator.getInstance("EC", "AndroidKeyStore")
keyGenerator.initialize(spec);
keyGenerator.generateKeyPair();

// The first certificate extends an ASN.1 structure described here
// https://developer.android.com/training/articles/security-key-attestation#certificate_schema_keydescription
// Among the information, it contains the bootloader status
Certificate certificates[] = keyStore.getCertificateChain(alias);
```

Certificate Extension Data Schema

```
KeyDescription ::= SEQUENCE {
    attestationVersion        3,
    attestationSecurityLevel  SecurityLevel,
    keymasterVersion          INTEGER,
    keymasterSecurityLevel    SecurityLevel,
    attestationChallenge      OCTET_STRING,
    uniqueId                  OCTET_STRING,
    softwareEnforced          AuthorizationList,
    teeEnforced               AuthorizationList,
}
SecurityLevel ::= ENUMERATED {
    Software            (0),
    TrustedEnvironment  (1),
    StrongBox           (2),
}
AuthorizationList ::= SEQUENCE {
    purpose                     [1]   EXPLICIT SET OF INTEGER OPTIONAL,
    algorithm                   [2]   EXPLICIT INTEGER OPTIONAL,
    keySize                     [3]   EXPLICIT INTEGER OPTIONAL,
    digest                      [5]   EXPLICIT SET OF INTEGER OPTIONAL,
    padding                     [6]   EXPLICIT SET OF INTEGER OPTIONAL,
    ecCurve                     [10]  EXPLICIT INTEGER OPTIONAL,
    rsaPublicExponent           [200] EXPLICIT INTEGER OPTIONAL,
    rollbackResistance          [303] EXPLICIT NULL OPTIONAL,
    activeDateTime              [400] EXPLICIT INTEGER OPTIONAL,
    originationExpireDateTime   [401] EXPLICIT INTEGER OPTIONAL,
    usageExpireDateTime         [402] EXPLICIT INTEGER OPTIONAL,
    noAuthRequired              [503] EXPLICIT NULL OPTIONAL,
    userAuthType                [504] EXPLICIT INTEGER OPTIONAL,
    authTimeout                 [505] EXPLICIT INTEGER OPTIONAL,
    allowWhileOnBody            [506] EXPLICIT NULL OPTIONAL,
    trustedUserPresenceRequired [507] EXPLICIT NULL OPTIONAL,
    trustedConfirmationRequired [508] EXPLICIT NULL OPTIONAL,
    unlockedDeviceRequired      [509] EXPLICIT NULL OPTIONAL,
    allApplications             [600] EXPLICIT NULL OPTIONAL,
    applicationId               [601] EXPLICIT OCTET_STRING OPTIONAL,
    creationDateTime            [701] EXPLICIT INTEGER OPTIONAL,
    origin                      [702] EXPLICIT INTEGER OPTIONAL,
    rootOfTrust                 [704] EXPLICIT RootOfTrust OPTIONAL,
    osVersion                   [705] EXPLICIT INTEGER OPTIONAL,
    osPatchLevel                [706] EXPLICIT INTEGER OPTIONAL,
    attestationApplicationId    [709] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdBrand          [710] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdDevice         [711] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdProduct        [712] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdSerial         [713] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdImei           [714] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdMeid           [715] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdManufacturer   [716] EXPLICIT OCTET_STRING OPTIONAL,
    attestationIdModel          [717] EXPLICIT OCTET_STRING OPTIONAL,
    vendorPatchLevel            [718] EXPLICIT INTEGER OPTIONAL,
    bootPatchLevel              [719] EXPLICIT INTEGER OPTIONAL,
}
RootOfTrust ::= SEQUENCE {
    verifiedBootKey    OCTET_STRING,
    deviceLocked       BOOLEAN,
    verifiedBootState  VerifiedBootState,
    verifiedBootHash   OCTET_STRING,
}
VerifiedBootState ::= ENUMERATED {
    Verified    (0),
    SelfSigned  (1),
    Unverified  (2),
    Failed      (3),
}
```

8.5 Conditional Checks on Files Matching Specific Keywords

- `daemonsu`

- pegasus.apk
- androVM-prop
- busybox
- mu
- .coldboot_init (related to Pegasus: [7] page 29)
- su
- temp_su
- init.magisk.rc
- baservice
- badamon
- droid4x-prop
- ttVM-prop
- igpi
- qemu_props
- giefroot
- microvirt-prop
- smsdamon
- waw
- smsservice
- libimcrc_64.so
- wland
- microvirtd
- libinjector.so
- nox-prop
- su
- su2
- sbin_orig
- magisk
- supersu
- .author

8.6 `DroidGuardVM` Layout

```
static constexpr size_t NB_REGISTERS = 0x100;

enum REG_TYPES : uint8_t {
  STRING,                                          // |
  INT,                                             // | The enum is randomized for each new release
  LONG,                                            // | of DroidGuard
  DOUBLE,                                          // |
  JOBJ,                                            // |
  POINTER,                                         // |

  NONE = 6,                                        // | "6" does not change across the versions
};                                                 // |
struct reg_t {
  REG_TYPES type;                                  // |
  uintptr_t value;                                 // |
};
struct registers_t {
  DroidGuardVM* vm;                                // |
  uintptr_t _;                                     // |
  std::array<reg_t, NB_REGISTERS> r;               // |
};
class DroidGuardVM {
  private:
  registers_t*                  registers;         // |
  std::vector<registers_t*>     frames;            // |
  std::array<uintptr_t, 0x200>  handlers;          // |
  uint32_t                      counter;           // |
  uint32_t                      pc;                // |
  std::array<HMAC_CTX**, 0x100> hmac;              // | BoringSSL HMAC_CTX for the VM registers
  __uint128_t                   enc_key;           // | Key involved in pointer and buffer encodings
  int32_t                       enc_register;      // |
  std::string                   bytecode;          // | The current running bytecode
                                                   // |
  uintptr_t                     crypto_key_1;      // |
  uintptr_t                     crypto_key_2;      // |
  int32_t                       count;             // |
  std::array<uint64_t, 0x42>    constants;         // |
                                                   // |
  pthread_t thread;                                // |
  uintptr_t tagged_buffer;                         // |
                                                   // |
  std::array<uint8_t, 0x400> scratch_buffer_1;     // | Mostly used by syscalls when reading content
  std::array<uint8_t, 0x410> scratch_buffer_2;     // |
                                                   // |
  JavaVM  jvm;                                     // |
  JNIEnv* env;                                     // |
  jobject mDroidGuard;                             // |
  jobject mDroidGuardChimeraService;               // |
  jobject jobj1;                                   // |
  jobject jobj2;                                   // |
  jobject mRuntimeAPI;                             // |
  jobject mJavaLangString;                         // |
  std::string flow;                                // |
  jobject mExtra;                                  // |
  bool    has_error;                               // |
};
```

Listing 5: `DroidGuardVM` Class Layout

8.7 `DroidGuardResult` Protobuf Content

```
classes_info = {
  info = [
    {
      "class":   "com.google.android.gms.droidguard.DroidGuardChimeraService"
      "methods": ["a" "b" "onBind" "onCreate"]
    },
    {
      "class":   "com.google.android.gms.framework.tracing.wrapper.TracingIntentService"
      "methods": ["a" "attachBaseContext" "onHandleIntent"]
    },
    {
      "class":   "com.google.android.chimera.IntentService"
      "methods": ["onBind" "onCreate" "onDestroy" "onHandleIntent"
                  "onStart" "onStartCommand" "setIntentRedelivery"]
    },
    {
      "class":   "com.google.android.chimera.Service"
      "methods": ["dump" "getApplication" "getChimeraImpl" "getContainerService"
                  "getForegroundServiceType" "onBind" "onConfigurationChanged" "onCreate"  ... ]
    },
    {
      "class": "android.content.ContextWrapper"
    }
  ]
}
```

```
ro_zygote    = "zygote64_32"
pointer_info = "7f3669240000-7f3669241000 rw-p 00000000"
cmdline      = "com.google.android.gms.unstable"
env_path     = "/product/bin:/apex/com.android.runtime/bin:/apex/com.android.art/bin:[ ... ]"
cache_dir    = "/data/user/0/com.google.android.gms/cache"

vbmeta_device_state = "locked"
vbmeta_digest       = "5c43a03e2a47d742deefb3a05c2bcdd1afadedb89ddbdba7651f99fdc92438f8"
verifiedbootstate   = "green"
security_patch      = "2021-12-12"
f134                = "com.google.android.gms" # Output of com.google.android.gms.droidguard.loader.RuntimeApi.c()
kernel_info         = "5.4.223-ga45ffa6db-74ceeb #1 SMP PREEMPT Tue Jul 21 01:52:07 UTC 2021"
flow                = "attest"
installer           = "com.android.vending"
proc_self_stat      = "561 (id.gms.unstable) S 949 949 0 0 -1 107832 324 0 0 0 "
```

```
current_class_loaders = """
dalvik.system.PathClassLoader[
    DexPathList[
        [zip file "/data/app/~~**********************==/com.google.android.gms-************-******-*A==/base.apk"]
        nativeLibraryDirectories=[/system/lib64, /system/product/lib64]
    ]
]
"""
f242 = [
  # List of KeyStore.getCertificateChain
]
file_info = [
  {"path": "/data",            "flag": 13},
  {"path": "/data/agents",     "flag": 2},
  {"path": "/data/local",      "flag": 13},
  {"path": "/data/local/tmp",  "flag": 13},
  {"path": "/data/local/bin",  "flag": 2},
  {"path": "/bin",             "flag": 13},
  {"path": "/data/local/xbin", "flag": 2},
  {"path": "/system/bin/.ext", "flag": 2},
   ...
]
```

```
mount_info = [
  "/dev/block/loop22 /apex/com.android.art@1"      "/dev/block/loop22 /apex/com.android.art",
  "/dev/block/loop23 /apex/com.android.i18n@1"     "/dev/block/loop23 /apex/com.android.i18n"
  "/dev/block/loop27 /apex/com.android.vndk.v30@1" "/dev/block/loop27 /apex/com.android.vndk.v30"
]

proc_self_maps_info = [
    "/apex/com.android.art/javalib/bouncycastle.jar",
    "/system/framework/boot-ims-common.vdex",
    "/data/data/com.google.android.gms/app_dg_cache/1FEFB755F7DFAAFB69E71C4B872D96A200EC65BF/the.apk"
    ...
]
```

References

[1] https://developer.android.com/training/safetynet/attestation#potential-integrity-verdicts

[2] https://github.com/microg/RemoteDroidGuard

[3] https://github.com/microg/GmsCore/blob/ad12bd5de4970a6607a18e37707fab9f444593a7/play-services-core-proto/src/main/proto/snet.proto#L15-L25

[4] https://github.com/microg/GmsCore/blob/ad12bd5de4970a6607a18e37707fab9f444593a7/play-services-core-proto/src/main/proto/snet.proto#L27-L30

[5] https://habr.com/en/post/446790/

[6] https://joellaity.com/2020/01/31/string.html

[7] https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-android-technical-analysis.pdf

[8] https://github.com/mrphrazer/msynth

[9] https://chromium.googlesource.com/chromium/src/+/refs/heads/main/components/cronet

[10] https://www.romainthomas.fr/projects-images/safetynet/

[11] https://kingrootapp.net/

[12] https://developer.android.com/training/articles/security-key-attestation

[13] https://www.sstic.org/2021/presentation/qbdl_quarkslab_dynamic_loader/

[14] https://github.com/unicorn-engine/unicorn