# Dart and Flutter Reverse Engineering Reference

Axelle Apvrille, Fortinet

September 12, 2024

# Contents

# Dart SDK

## Dart language

- Built-in types: **`int`**, **`double`**, `bool`, `List`, `Set`, `String`
- `await` pauses execution of a function until a `Future` is completed. A function using the `await` keyword should be marked `async`.

### Strings

Dart strings are created one of following ways:

1. Using `String`, e.g. `String flag = 'flag{congrats}'`
2. `String` are immutable, so if they need to be manipulated, use the `StringBuffer` class, and, if needed, convert to a `String` with `toString()`.

```
void main() {
     final buffer = StringBuffer('Pico le Croco');
     buffer.write(' has big teeth');
     print(buffer);
     print(buffer.toString());
}
```

3. A string is also a sequence of Unicode UTF-16 code units, which are represented as integers. So, they can also be created from list of integers, or types derived from integers (e.g Uint8List). Note that if UTF-8 conversion is needed, there are `encode()` and `decode()` methods from the `dart:convert` library.

```
// String to bytes
String foo = 'Hello world';
List<int> bytes = foo.codeUnits;
// Bytes to String
String bar = String.fromCharCodes(bytes);
```

Strings can be interpolated with constructions such as:

```
String foo = 'launch args: $args';
```

### No primitive type for *byte* or *char*

Workaround #1: `List<int>`. But this will waste memory because *int* is 64 bits.

```
List <int> core = [7, 34, 49, 55...];
String s = String.fromCharCodes(core);
```

Workaround #2: `Uint8List` (note to import `dart:typed_data`). Best solution n terms of memory waste because it will be the same as **`byte`** `[]` + some small overhead.

```dart
import 'dart:type_data';

Uint8List flag = Uint8List.fromList([98, 101, ...]);
String s = String.fromCharCodes(flag);
```

#### Future, async, await

- "A future (lower case"f") is an instance of the Future (capitalized "F") class. A future represents the result of an asynchronous operation, and can have two states: uncompleted or completed."
- A future can complete with a value of type T. Its class is then `Future<T>`.
- **async** should be put before the body of any synchronous function.
- **await** means to wait for a future to complete.

#### Nullable

- `!` tells the compiler that you are certain that the given expression will never be null at runtime. NB. If somehow this value is null, a `NullThrownError` will be be thrown…

For example, in the following, the code guarantees that `getExternalStorageDirectory()` will not be null using `!` and consequently allow the access to its field `path`:

```dart
String dir = (await getExternalStorageDirectory())!.path;
```

- Reciprocally, `?` tells the compiler that a value may be *null*. It can be used to *declare* that a given type is nullable. Or it can be used as a condition when a variable is not null:

```dart
// Using ? to declare a nullable type
// by default, ints are not nullable
int? nullableInt = null;

// Will print null if mystring is null
String? mystring;
print(mystring?.length);
```

- Use `??` to provide default values if the variable is null:

```dart
// nullable string
String? mystring;
// name will never be null. If mystring is null, name will be Pico le Croco
String name = mystring ?? "Pico le Croco";
```

#### Late

The `late` keyword is used to indicate a variable is initialized "later". The variable must still be initialized before it is accessed, or a runtime error will occur.

```
late final String _data;

  @override
  void initState() {
    super.initState();
    // Initialize _data here
    _data = "Initialized data";
  }
```

### SDK Contents

Dart SDK contains:

- Compiler (`dart compile ...`)
- Profiling tools
- Package manager (`dart pub ...`)
- Standard libraries: I/O, networking…
- Runtime VM

### SDK Commands

- Create a project: `dart create -t console hello`
- Compile: `dart compile FORMAT source.dart`
- Run: `dart run EXE`
- Disable reporting: `dart --disable-analytics`
- Version: `dart --version`
- Dart SDK archive

### Packages

- device_info_plus: Retrieves info from `android.os.Build`, `/etc/os-release`. Not sensitive info.
- package_info_plus. Retuns app name, package name, version and build.
- web_socket_channel. Web Socket Channel API for Dart.

To add a package to a Dart project:

1. Create a `pubspec.yaml` stub:

```
name: "PROJECTNAME"
dependencies:

environment:
  sdk: '^3.4.0'
```

2. `dart pub add PACKAGE` to add the dependency to the given package
3. `dart pub get` to get it
4. Then, compile with `dart compile...`

## Flutter

### Contents

- Widgets
- UI components
- Libraries: camera, geolocator
- Flutter CLI tool

### Install

- Install it manually
- `export PATH="$PATH:`pwd`/flutter/bin"`
- Personal Homedir: `~/softs/flutter`
- Upgrade: `flutter upgrade`.

Check status with `flutter doctor`:

- Complains about **ninja-build**? You may have to install manually and create a link in `/usr/local/bin/ninja`
- Complains about **clang**? `sudo apt install clang`
- Complains about *Unable to find bundled Java version* of Android Studio? In Android Studio dir, create a symlink: `ln -s ./jbr ./jre`

On a Raspberry Pi, install Flutter via **snap**

```
sudo apt install snapd
sudo snap install core
sudo snap install flutter --classic

# I had to do this...
$ /snap/flutter/current/flutter.sh
$ export PATH=$PATH:/home/axelle/snap/flutter/common/flutter/bin/
```

Finally, check the install with `flutter --version` and `flutter doctor -v`.

- Disable analytics: `flutter config --no-analytics`

### App Creation

| What | Android Studio | Command Line |
|---|---|---|
| Create project | Create Flutter App<br><br>use Java, select platform iOS, Android and Linux | `flutter create projectname` |
| Download new dependancies | | Modify `pubspec.yaml` and run `flutter pub get` |
| Build Release | Build > Build APK | `flutter build apk` |
| Obfuscate | | `flutter build apk --obfuscate --split-debug-info=debug-info --extra-gen-snapshot-options=--save-obfuscation-map=obfuscation-map` |
| Run on Linux | | `flutter run` |

With obfuscation (and using the default settings), all Dart files are created with a dummy name (e.g `Axc.dart`) and flat, i.e in the same directory (no package structure) (FatalSec 2024).

### Implementation

- `build()`: don't put anything blocking in there! Can be called multiple times.

### Platform Channels

Communication between the native layer and Flutter is performed through *Platform Channels*.

On Flutter side, create a method channel (`MethodChannel`). Name must be unique in the app. Then, invoke a method of the native side using `invokeMethod`. Note the communication is asynchronous.

```
static const platform = MethodChannel('samples.flutter.dev/battery');
final result = await platform.invokeMethod<int>('getBatteryLevel');
```

On the Android side, also create a method channel (`MethodChannel`) and set a `MethodCallHandler()` to specify what should happen when the method gets called.

```
new MethodChannel(flutterEngine.getDartExecutor().getBinaryMessenger(),
    CHANNEL).setMethodCallHandler(
```

```
            (call, result) -> {
            if (call.method.equals("getBatteryLevel")) {
                ...
                // on success, return result.success()
                // or result.error()
            } else {
                // result.notImplemented();
            }
            });
```

For example, this is a decompiled code:

```
private final Object func(MethodCall methodCall0, Result
    methodChannel$Result0) {
     SharedPreferences.Editor sharedPreferences$Editor0;
     if(l8.a6(methodCall0.method, "setConfig")) {
         String s = (String)methodCall0.argument("smsCount");
         if(s == null) {
             return null;
         }

     ...
     }
```

### Sentinel

The assembly code often uses a *Sentinel*.

A *sentinel* is a special value used to signify the end of a data structure, or the completion of a process. It acts as a signal or marker that indicates when a certain condition has been met, or when the data structure has reached its end. Sentinels are a general concept (not specific to Dart).

Example: in a linked list, a sentinel node can be used to indicate the end of the list. NB. Sentinels do not necessarily mark an end. They can mark *anything*: expired, free…

In Dart, sentinels are used for:

- Object ID 0. Indicates target of a reference has been omitted from the snapshot.
- Class ID 0.

The following assembly code shows the use of a sentinel to indicate whether static fields have been initialized or not:

```
0x001b0980      403740f9       ldr x0, [x26, 0x68]          ; 0xf4 ; "THR::
    field_table_values"
0x001b0984      00a445f9       ldr x0, [x0, 0xb48]          ; 0xda
0x001b0988      702340f9       ldr x16, [x27, 0x40]         ; 0xf5 ; "Load
    Sentinel from PP"
0x001b098c      1f00106b       cmp w0, w16
0x001b0990      81000054       b.ne 0x1b09a0                ; "compare
    sentinel and field_table_values"
0x001b0994      62234091       add x2, x27, 8, lsl 12       ; "case where
    sentinel == field_table_value: we need to init the field"
```

```
0x001b0998      42e840f9       ldr x2, [x2, 0x1d0]         ; 0xdc ; "
    computing 0x81d0 offset from PP: anyIPv4 field"
0x001b099c      62f90494       bl fcn.InitLateStaticFieldStub_2eef24
0x001b09a0      e00100f9       str x0, [x15]               ; "case where
    sentinel != field_table_value: field already initialized"
0x001b09a4      24000094       bl "fcn.serversocket_bind"
```

and corresponds to part of the following dart code: `final server = await ServerSocket.bind(InternetAddress.anyIPv4, 8080);`

The corresponding Dart SDK source code is in runtime/vm/field_table.cc where a sentinel is placed at the top of the field table.

```
field.set_field_id(top_);
table_[top_] = Object::sentinel().ptr();
```

Note this is different from the public Sentinel class.

## Versions

There are **Dart SDK versions** and **Flutter versions**.

| Approximative Date | Dart SDK version | Flutter version |
|---|---|---|
| May 2023 | 3.0.1 | |
| Feb 2024 | 3.3.0 | 3.19.1 |
| March 2024 | 3.3.3 | 3.19.5 |
| July 2024 | 3.4.4 | |

## Dart output formats

Output formats

- Source code: it can be directly run using Dart VM's JIT compiler
- Kernel snapshot: Intermediate representation of Dart source code. Used for Flutter *debug* builds.
- JIT snapshot: JIT snapshots are an optimized intermediate representation of *bytecode*. Bytecode can be seen as intermediate machine code. The bytecode is compiled by Dart VM's JIT compiler. The bytecode is not portable, because it is specific to Dart VM's execution environment. JIT snapshots are typically used during development for example because they allow *Hot Reload* (make changes and see results without restarting the entire app). They are not used for production because slower than AOT snapshots.

- AOT snapshot: pre-compiled native machine code. The initial steps between JIT compilation and AOT compilation are shared, the end is different. The code requires a Dart runtime to run. Used for Flutter *release* builds. The command `dartaotruntime` contains the runtime.
- Self contained executable: This is the only executable format which can be run on systems without the Dart SDK installed. It embeds the Dart VM.

Compilation:

- Self contained exe: `dart compile exe hello.dart`
- AOT snapshot: `dart compile aot-snapshot hello.dart` (non stripped), `dart compile aot-snapshot -S ./debuginfo filename.dart` (stripped)
- JIT snapshot: `dart compile jit-snapshot hello.dart`
- Kernel snapshot: `dart compile kernel hello.dart`

Run:

- Source code: `dart run hello.dart`
- Self contained exe: `./hello.exe`
- AOT snapshot: `FLUTTER_DIR/flutter/bin/cache/dart-sdk/bin/dartaotruntime hello.aot`
- JIT snapshot: `dart run hello.jit`
- Kernel snapshot: `dart run hello.dill`

| Dart formats | Portable | Requires an external Dart Runtime VM to run |
|---|---|---|
| Source code | Yes | Yes |
| Self contained executable | No | No |
| AOT snapshot | No | Yes |
| JIT snapshot | No | Yes |
| Kernel snapshot | Yes | Yes |

| Dart output formats | Size | Exec time | Description |
|---|---|---|---|
| hello.dart | 266 bytes | 0m0,320s (40x) | Source code |
| hello.exe | 5.8 M | 0m0,008s | Self contained executable |
| hello.aot | 863 K (14%) | 0m0,008s | AOT snapshot |
| hello.jit | 4.7 M (81%) | 0m0,242s (30x) | JIT snapshot |
| hello.dill | 936 bytes (0.01%) | 0m0,245s (30x) | Kernel snapshot |

## Isolate

An *isolate* is an independent unit of execution that runs concurrently with other isolates within the same Dart process. Each isolate has it own memory heap, stack and event loop - contrary to OS threads which share the same memory space.

Dart programs have at least one isolate, to run the main "thread", and possibly more. For instance, the developer may decide to create more isolate to handle decompression of a large file.

## Dart AOT Snapshot Format

### ELF shared object

1. VM snapshot: contains base functionality of Dart VM + common libraries.
2. 1 or more **Isolate** snapshots (1 per isolate): freezes the status of the Dart VM before main() is called.

ELF segments of a snapshot:

1. Instructions. Code to be executed, contained in a `.text` segment
2. Data. Initial state of Dart heap, contained in a `.rodata` segment

- How to display dynamic symbols: `objdump -T snapshot`

### AOT snapshot

```
+---------------------------- +
+    Dart AOT Header           +
+ ---------------------------- +
+ Cluster Information          +
+ ---------------------------- +
+ Serialized Cluster 1         +
+ ---------------------------- +
+ Serialized Cluster 2         +
+ ---------------------------- +
+ Serialized Cluster 3         +
+ ---------------------------- +
+            ...               +
+ ---------------------------- +
```

1. **Header**

- Magic number `f5f5dcdc`, 4 bytes
- Size, 8 bytes
- Snapshot kind, 8 bytes
- Version hash, 32 bytes
- Features: Null terminated string

2. **Cluster Info**

- Base Object Count. DLEB128. Base objects are self-explanatory objects (e.g. *null*, *empty array*, *void*, *True*, *False*…). To my understanding, all these objects are included in *VM* snapshots, there are none in *Isolate* snapshots. For isolate snapshots, the count indicates the number of base objects *available to the snapshot*.
- Object Count. DLEB128. Number of objects in the snapshot.
- Cluster Count. DLEB128. Number of clusters in the snapshot. This can also be seen as the number of types.
- Code order length. DLEB128. *To be explained*

LEB128 is a *variable length encoding of integers* where each byte has its most significant bit set, except the last byte of the sequence. For example, in a sequence `0xE5 0x8E 0x26`, 0xE5 and 0x8E have their most significant bit set so we know there are more bytes to process. But 0x26 has its most significant bit to 0, so we know it is the last one. Then, to decode the sequence, we reverse order of bytes, strip each most significant bit and read the value:

- Reverse order: 0x26 0x8E 0xE5
- In binary, this is: 00100110 10001110 11100101
- Strip the most significant bit: 0100110 0001110 1100101
- Read the value for 0b01001100001110110010 1: **624485**

Dart uses a **custom version of LEB128** where its the opposite: only the last byte has its most significant bit set. Let's call this version *DLEB128* (for Dart LEB128).

3. **Cluster Serialization**

Clusters of the snapshot are serialized one by one. The serialization of a cluster consists in 3 steps:

1. Trace. (`Trace`)
2. Alloc. (`WriteAlloc`) In this stage, we parse all objects of the cluster and attribute reference identifiers to each of them (`AssignRef`). Then, basic serialization of some objects occur. For example, the serialization of Mint (medium integers) and SMI (small integers) occur at this stage.
3. Fill. (`WriteFill`). Completes the serialization of each object.

The code which handles the serialization of a snapshot is located in `runtime/vm/app_snapshot.cc` of Dart's SDK.

| Type | Class / Link | Cid |
|---|---|---|
| Mint | MintSerializationCluster | `kMintCid` |
| Code | CodeSerializationCluster | `kCodeCid` |
| Object Pool | ObjectPoolSerializationCluster | `kObjectPoolCid` |

| Name | Value |
|---|---|
| `kIllegalCid` | 0 |
| `kClassCid` | 5 |
| `kFunctionCid` | 7 |
| `kCodeCid` | 18 |
| `kObjectPoolCid` | 22 |
| `kMintCid` | 60 |
| `kStringCid` | 92 |
| `kOneByteStringCid` | 93 |
| `kTwoByteStringCid` | 94 |

Note that when a *custom cluster* (new type) needs to be serialized, Dart assigns a CID to that cluster from a CID which isn't used in the snapshot.

## Registers

### Dedicated registers for Dart

- **PP** (Pool Pointer). Pointer on the beginning of the Object Pool.
- **THR**. Pointer on the running VM thread (`dart::Thread` object). With this pointer, you get relative offsets to several functions/concepts such as stack limit.
- Register for Stack Pointer is dedicated in Dart Aarch64 to *x15*

```
+ ------------ + ----- + ----- + ----- +
|              |  PP   |  THR  |   SP  |
+ ------------ + ----- + ----- + ----- +
| x86-64       | r15   |  r14  | rsp   |
| Aarch32      | r5    |  r10  | r13   |
| Aarch64      | x27   |  x26  | x15   |
+ ------------ + ----- + ----- + ----- +
```

### Object Pool (PP)

The Object Pool is a table which stores and references frequently used objects, immediates and constants within a Dart program.

At compilation time, all frequently uses objects, immediates and constants are replaced by an indirect access to the Object Pool. For example, code such as `String hello = "Dart"` would be replaced by `String hello = objectpool[77]`.

The Object Pool is stored on the Heap. It is serialized in Dart AOT snapshots.

Example of x86-64 assembly code loading a string from the object pool and printing it:

```
mov r11, qword [r15 + 0x168f]
mov qword [rsp], r11
call sym.printToConsole
```

- For Aarch32: `LDR R1, [R5, #433h]`
- For Aarch64: `LDR X16, [X27, #433h]`
- For x86_64: `mov rbx, qword ptr ds:[r15+433h]`

### THR offsets

- stack limit: used to check for stack overflow, and also for interrupts
- `field_table_value`: array with values of static fields of the current isolate
- `top`: allocation top of TLAB (thread local allocation buffer)
- null object

In `runtime/vm/compiler/runtime_offsets_extracted.h`:

```
#if defined(PRODUCT) && defined(TARGET_ARCH_X64) &&
                                    \
    defined(DART_COMPRESSED_POINTERS)
...
static constexpr dart::compiler::target::word Thread_stack_limit_offset = 0
    x38;
static constexpr dart::compiler::target::word
    Thread_write_barrier_mask_offset =  0x40;
static constexpr dart::compiler::target::word Thread_heap_base_offset = 0x48
    ;
static constexpr dart::compiler::target::word Thread_top_offset = 0x50;
static constexpr dart::compiler::target::word Thread_end_offset = 0x58;
static constexpr dart::compiler::target::word
    Thread_dispatch_table_array_offset = 0x60;
static constexpr dart::compiler::target::word
    Thread_field_table_values_offset = 0x68;
...
```

NB. Blutter's `ida_dart_struct.h` outputs the Dart Thread structure.

### x86-64 assembly using null object

```
mov r11, qword [r14 + 0x68] ; store null object in r11
mov qword [rsp], r11        ; push r11 on the stack
call sym.new_Random         ; call constructor for Random()
```

### Stack overflow

At the beginning of each function, after allocation on the stack, there is a stack overflow check. This has a dual purpose: (1) check for stack overflow and (2) serve as interruption point (e.g. if the VM wants to cleanly interrupt a thread) (Egorov 2021).

Below is the assembly code for Aarch64 where a function prologue checks the stack limit:

```
STP       X29, X30, [X15, #FFFFFFF0h]!
MOV       X29, X15
SUB       X15, X15, #10h
; X26 + 0x38 is the stack limit of the current thread
LDR       X16, [X26, #38h]
CMP       X15, X16
B.LS      loc_3D75DC
```

### Pointer decompression

To keep smaller pointers, Dart often uses *compressed pointers* where only the lower 32 bits of a pointer are kept in memory. Before use, the pointer must be *decompressed* :

1. Get the lower 32-bit address
2. Add the upper bits

On Aarch64, we have the following example:

```
ldur w1, [x0, 0xf]       ; contains the lower 32-bit address
add x1, x1, x28, lsl 32  ; the upper bits are in X28 (reserved HEAP register
    bits for Aarch64)
```

Thus, the decompressed pointer is `Addr[x0 + 0xf] + x28 << 32`

### Recap of important registers

| Architecture | Register | Use |
|---|---|---|
| arm7eabi | r5 | **Object Pool** |
| | r10 | Pointer to running VM thread |
| | r11 | Frame Pointer **FP** |
| | r13 | Stack Pointer **SP** |
| | r14 | Link Register **LR** |
| | r15 | Program Counter **PC** |

| Architecture | Register | Use |
|---|---|---|
| arm64 | X15 | **Custom Stack Pointer**. **SP** |
| | X26 | Pointer to running VM thread. **THR** |
| | X27 | **Object Pool**. **PP** |
| | X28 | Address of the **HEAP** (e.g. useful for Pointer Decompression). |
| | X29 | Frame Pointer. **FP**. |
| | X30 | Link Register. **LR**. |
| x86_64 | r10 | Arguments descriptor register |
| | r12 | Code register |
| | r14 | Pointer to running VM thread |
| | r15 | **Object Pool** |

## Encoding of Small Integers (SMI)

Dart represents integers differently depending on their size:

- **Small Integers (SMI)**. Those are integers which can fit on 31 bits (for 32-bit architectures) or 63 bits (for 64-bit architecture). They are represented with their least significant bit set to 0. The value is encoded on the remaining bits.
- **Medium Integers (Mint)**. Those which need more bits than 31/63.

```
+ ------------------------------ + - +
| 31 30 39 ..................... 1 | 0 |
+ ------------------------------ + - +
| Value                          | I |
+ ------------------------------ + - +
```

Note that not all small integers are represented as *SMI*. To my understanding, small integer which use the built-in `int` type are represented "normally". Only those which trigger the creation of an object, such as *list of integers*, are held as an SMI.

| Source code | Representation in assembly |
|---|---|
| `int i = 2` | standard: `mov rax, #2` |
| `List <int> tab = [ 1, 2 ]` | SMI |

### x86-64 example

Assembly code for a byte array:

```
 ; size of array = 0x1c / 2 = 14
mov       r10d, 1Ch
call      stub _iso_stub_AllocateArrayStub
...
mov       r11d, A0h                       ; P
mov       qword ptr ds:[rax+17h], r11
mov       r11d, D2h                       ; i
mov       qword ptr ds:[rax+1Fh], r11
mov       r11d, C6h                       ; c
mov       qword ptr ds:[rax+27h], r11
mov       r11d, DEh                       ; o
```

### x86-64 control for SMI/Mint case

In some cases, the compiler has some extra work: it does not know if the XOR result fits in a small or a medium integer. Consequently, it writes code for both cases. It tests if the result fits in a SMI by doubling it and checking if there's an overflow. If there's no overflow, this is a SMI. If it overflow, it must be stored in a Mint.

```
; rdx contains XOR result: core[i] ^ 0x43
mov       rax, rdx
; compute rax * 2
add       rax, rax
; no overflow: SMI case, overflow: Mint case.
jno       no_overflow
; Mint case: create Mint containing XOR result value
call      stub _iso_stub_AllocateMintSharedWithoutFPURegsStub
mov       qword ptr ds:[rax+7], rdx
...
no_overflow:
mov       rdx, rcx
; get address of core[i]
lea       r13, qword ptr ds:[rdx+8*rdi+17h]
; store XOR result in core[i]
mov       qword ptr ds:[r13], rax
```

## Calling convention (ABI)

### Dart SDK >= 3.4.0

Since Dart SDK 3.4.0, Dart uses a standard calling convention where the first few arguments are passed in specific registers, and on the stack if there are more.

| | arg 1 | arg 2 | arg 3 | arg 4 | arg 5 | arg 6 | … |
|---|---|---|---|---|---|---|---|
| x86-64 | RDI | RSI | RDX | RBX | R8 | R9 | rsp |
| aarch64 | R1 | R2 | R3 | R5 | R6 | R7 | r15 |
| aarch32 | R1 | R2 | R3 | R8 | r13 (SP) | … | … |

In aarch64, below **`this`** pointer is passed in x1, second argument in x2 and third argument in x3 for `nextGame`.

```
0x00212470      01f040b8        ldur w1, [x0, 0xf]
0x00212474      21801c8b        add x1, x1, x28, lsl 32
0x00212478      62a37df9        ldr x2, [x27, 0x7b40]
0x0021247c      63a77df9        ldr x3, [x27, 0x7b48]
0x00212480      08000094        bl method.simon_game_main.ColorMemoryState.
    nextGame
```

Link to the patch

### Calling convention of Dart SDK < 3.4.0

In earlier versions of Dart, all arguments used to be pushed on the stack (`push r11`).

```
mov r11, qword [r15 + 0x1d3f]
push r11
mov r11, qword [r15 + 0x1d47]
push r11
call fcn.string_concat
```

| | arg 1 | arg 2 | arg 3 | arg 4 | … |
|---|---|---|---|---|---|
| Standard calling convention x86-64 | rdi | rsi | rdx | r8 | … |
| Dart calling convention | push on the stack | | | | |

Aarch32:

```
ldr lr, [r5, 0xe9f] ; "stage2: "
ldr sb, [r5, 0xea3] ; "ph0wn{"
stm sp, {sb, lr}    ; push them on the stack
bl fcn.concat       ; concatenate strings
```

Aarch64 (see use of X15 as stack pointer):

```
LDR       X16, [X27, #1C90h]              ; "stage2: "
LDR       X30, [X27, #1C98h]              ; ph0wn{
STP       X30, X16, [X15]                 ; we push them on the stack
```

```
4 BL        _StringBase.+                    ; concatenate both strings: '
      stage2: ph0wn{'
```

## Global Dispatch Table (GDT)

The methods of each cluster are accessed through a Global Dispatch Table. The GDT should be imagined as one-dimension array with references to all methods of class A, then all methods of class B etc.

See the example from the README of the SDK:

```
1  class A {
2    void foo() { }
3  }
4
5  class B extends A {
6    void foo() { }
7  }
8
9  class C {
10   void bar() { }
11 }
12
13 class D extends C {
14   void bar() { }
15 }
```

has the following GDT. This works because `C.foo` does not exist.

```
1 +-------+-------+-------+-------+
2 | A.foo | B.foo | C.bar | D.bar |
3 +-------+-------+-------+-------+
```

The corresponding assembly will be:

```
1 movzx cid, word ptr [obj + 15] ; load receiver's class id
2 call [GDT + cid * 8 + (selectorOffset - 16) * 8]
```

The compiler will use results of the global Type Flow Analysis to de-virtualize as many call endpoints as it can, and will resort to dispatch through GDT if it can't.

### Aarch64 assembly calling a method from the GDT

```
1 ; x0 is an object. lr = x0 - 0xffc
2 0x1b30d0: sub             lr, x0, #0xffc
3 ; lr = x21+ (lr << 3), i.e x21 + lr * 8
4 0x1b30d4: ldr             lr, [x21, lr, lsl #3]
5 ; call the corresponding method of object x0
6 0x1b30d8: blr             lr ; then call
```

- TODO: I don't know why x21 represents the class id of x0 object…

- If we combine the first 2 instructions, we have: lr = x21 + (x0 - 0xffc) * 8
- 0xffc is the selector offset for the method

## Dart instructions

| Dart instruction | Corresponding Aarch64 assembly | Source code and Comments |
|---|---|---|
| BoxInt64 | | |
| BoxInt64Instr | sbfiz x0, x2, #1, #0x1f; cmp x2, x0, asr #1; b.eq #0x2131b8; bl #0x2b67e4; stur x2, [x0, #7] | Convert int to char |
| CheckNull | cmp x1, null; b.eq xxx | |
| CheckStackOverflow | ldr x16, [thr, #64]; cmp sp, x16; b.ls xxx | |
| CheckStackOverflowSlowPath | Bl Stub::StackOverflowSharedWithoutFPURegsStub | code |
| Enter Frame | stp fp, lr, [sp, #-16]! and mov fp, sp | Prologue x86_64 aarch64 |
| LoadInt32Instr | sbfx x3, x2, #1, #0x1f, tbz w2, #0, #0x2131a0, ldur x3, [x2, #7] | Convert char to int |
| Leave Frame | | aarch64 |
| Parallel Move | | |
| Push Argument | ldr x16, [pp, #5160]; stp x0, x16, [sp, #-16]! | Push x0 and x16 on the stack |
| Return | mov sp, fp; ldp fp, lr, [sp], #16; ret | |
| UnboxedConstant | | |
| UnboxInt64 | | |

## Typical assembly snippets

### Function prologue

Example in x86-64:

```
1 ; push base pointer on the stack
2 push rbp
3 ; the new value for the base pointer is the stack pointer
4 mov rbp, rsp
5 ; allocate 16 bytes
6 sub rsp, 10h
```

```
; r14 holds the current Dart VM thread pointer
cmp rsp, qword [r14 + 0x38]
; if stack pointer is <= [r14 + 0x38 ]: jump stack overflow error
jbe 0x9e850
```

For Aarch32:

```
; push frame pointer (r11) and link register on the stack
PUSH      {R11, LR}
; move frame pointer to the bottom of the stack
ADD       R11, SP, #0
SUB       SP, SP, #8
MOV       R0, #2Ch
; check stack overflow
; r10 holds the current VM thread pointer
LDR       R12, [R10, #1Ch]
CMP       SP, R12
BLLS      sub_32FCF4
```

For Aarch64:

```
; EnterFrame X29=FP, X30=LR, X15=SP, #FFFFFFF0h=-0x10
STP       X29, X30, [X15, #FFFFFFF0h]!
MOV       X29, X15
; Allocate space on the stack (X15=SP)
SUB       X15, X15, #10h
; Check stack overflow
; X26 + 0x38 is the stack limit of the current thread
LDR       X16, [X26, #38h]
CMP       X15, X16
B.LS      loc_3D75DC
```

### Function epilogue

For Aarch64:

```
; EPILOGUE
; Restore the value of Frame Pointer into Stack Pointer
mov             SP, fp
; ldp = load pair registers
; we load the content of [SP] in FP
; and [SP]+0x10 to LR
; LR = Link Register - holds the return address
ldp             fp, lr, [SP], #0x10
ret
```

This restores the registers to their original value at the beginning of the call, and matches a prologue such as

```
; PROLOGUE
stp             fp, lr, [SP, #-0x10]!
mov             fp, SP
```

### Stack overflow case

At the beginning of each function, the assembly checks the stack hasn't gone beyond its authorized limit (THR::stack_limit). If this happens, the program branches to an error case.

```
bl              #0x2f0b24  ; StackOverflowSharedWithoutFPURegsStub
b               #0x240128
```

### String interpolation

String interpolation calls `StringBase::_interpolate`. The following assembly code corresponds to :

```
String foo = 'launch args: $args';
```

1. Get the string "launch args:" from the Object Pool
2. Store it in the address of x0 + 0xf (STore Unsigned Register)
3. Load the contents of FP - 0x10, which contains the function's arguments, in x1
4. Store it in the address of x0 + 0x13
5. Put x0 on the stack i.e pass it as argument to the following function we are calling below
6. Call function at 0x408be4, which has been found to contain StringBase::interpolate. So, we are passing to interpolate an object that contains the "format" string and the content strings.

```
ldr             x17, [PP, #0x2b20]  ; [pp+0x2b20] "launch args: "
stur            w17, [x0, #0xf]
ldur            x1, [fp, #-0x10]
stur            w1, [x0, #0x13]
str             x0, [SP]
bl              #0x408be4  ; [dart:core] _StringBase::_interpolate
```

### Await

The following dart code `appDocumentDir = await getApplicationDocumentsDirectory()` ; gets assembled as the following:

```
bl              #0x781a48  ; [package:path_provider/path_provider.dart] ::
    getApplicationDocumentsDirectory
mov             x1, x0
stur            x1, [fp, #-0x10]
bl              #0x35ad14  ; AwaitStub
```

1. Call getApplicationDocumentsDirectory()
2. The result is in x0. Copy it to x1
3. Store the result (x1) somewhere (every case is different)
4. Call the AwaitStub.

### Late

The initialization of a late field consists in (1) getting it from the Object Pool, and (2) calling a function such as `InitLateInstanceField`.

```
mov    rdx,QWORD PTR [r15+0x176f]
call   8dd14 <stub _iso_stub_InitLateInstanceFieldStub>
```

| Late initialization function | Use | Example |
|---|---|---|
| `InitLateInstanceFieldStub` | init of a late field of an object | `class Foo { late int bar; }` |
| `InitLateFinalStaticFieldStub` | init of a late final global variable | `final func = print` |
| `InitLateStaticFieldStub` | init of a late global variable, not final | `var func = print` |

**Function pointers** are typically assembled as *late fields*. A typical example for this is the use of `debugPrint()` in Flutter applications. Actually, `debugPrint` is a callback to `debugPrintThrottled` (see print.dart). As such, it is assembled in two steps: (1) initialization of the late field, and (2) calling `debugPrintThrottled`.

```
0x1a634c: ldr             x2, [PP, #0x2378]  ; [pp+0x2378] Field <::.
    debugPrint>: static late (offset: 0x710)
0x1a6350: bl              #0x2c0c84          ; InitLateStaticFieldStub
...
0x1a6398: bl              #0x129b98  ; [dart:core] _StringBase::_interpolate
0x1a639c: str             NULL, [SP]
0x1a63a0: mov             x1, x0
0x1a63a4: ldr             x4, [PP, #0x2388]  ; [pp+0x2388] List(7) [0, 0x2,
    0x1, 0x1, "wrapWidth", 0x1, Null]
0x1a63a8: bl              #0x1550f0  ; [package:flutter/src/foundation/print
    .dart] ::debugPrintThrottled
```

### Closure

Calling a closure is done in 3 steps:

1. Retrieving a function object for the closure, from the Object Pool
2. Allocating the closure stub. The returned closure object points to the function + contains a pointer to the actual instructions (`UntaggedClosure::entry_point_`)
3. Calling the closure. For that, use the address of the instructions at `entry_point_`.

```
mov rbx, qword [PP + 0x1977] ; "the function object"
mov rdx, qword [THR + 0x68]
```

```
call sym.stub__iso_stub_AllocateClosureStub
mov qword [rsp + 0x10], rax ; store the closure object on the stack
mov r11, qword [PP + 0x197f] ; "get c1 from PP"
mov qword [rsp + 8], r11    ; "push c1 on the stack"
mov r11, qword [PP + 0x1987] ; "get c2 from PP"
mov qword [rsp], r11        ; "push c2 on the stack"
mov r10, qword [PP + 0x4c7]
mov rcx, qword [rax + 0x37] ; this is the address of the closure's
   instructions
call rcx                    ; "call the closure"
```

## Dart SDK source code ref

| | URL |
|---|---|
| Calling convention | important patch - see `ComputeCallingConvention()` in runtime/vm/compiler/backend/dart_calling_conventions.cc |
| ClassId enumeration | `sdk/runtime/vm/class_id.h` |
| `debugPrint` | print.dart |
| Heap snapshot info | See heap_snapshot.md |
| ObjectPool class | runtime/vm/object.h |
| ObjectPool serialization | runtime/vm/app_snapshot.cc see `ObjectPoolSerializationCluster` |
| Offsets to THR for various functions | runtime/vm/compiler/runtime_offsets_extracted.h |
| Register enumeration | runtime/vm/constants_arm.h, runtime/vm/constants_arm64.h, runtime/vm/constants_x64.h |
| Registers for arguments | see `kCpuRegistersForArgs[]` in [runtime/vm/constants_x64.h](https://github.com/dart-lang/sdk/blob/main/runtime/vm/constants_x64.h#L683] etc |
| Snapshot class | runtime/vm/snapshot.h |
| Snapshot serialization | `sdk/runtime/vm/app_snapshot.cc` in `SerializationCluster` |
| Snapshot Kind | `sdk/runtime/vm/snapshot.h` |
| Serialization of integers | runtime/vm/app_snapshot.cc |

| | URL |
|---|---|
| Stub compiler code | runtime/vm/compiler/stub_code_compiler.cc |
| Class Smi | `runtime/vm/object.h` |
| Cluster Info serialization | runtime/vm/app_snapshot.cc |
| Read/Write Uint | `runtime/vm/kernel_binary.h` |
| Read/Write LEB128 | `runtime/vm/datastream.h` L173 |

## Assembly memento

### Aarch64 Memento

- Store Unsigned Register: `STUR src, [destination]`
- Signed BitField Insert Zeroes: e.g `SBFIZ X0, X5, #1, #1Fh` copies the lower 31 bits of X5 at position 1 in X0 (=> x2)
- Load Unsigned Register: `LDUR dst, [value]`
- Sign Extended BitField Extract: e.g `SBFX X1, X0, #1, #31` extracts bits 1 to 31 with sign extension and copies to X1 (/2)
- EOR can only be done on a register, not on an immediate value:

```
1 MOVZ      X16, #37h    ; load XOR Key 0x43 in register X16
2 EOR       X5, X1, X16  ; XOR byte with register X16
```

### Aarch32 Memento

- LSL: Logical Shift Legt
- `TST R0, #1`: tests R0 & 1
- ASR: Arithmetic Shift Right
- `PUSH {R11, LR}`: push both frame pointer and link register on the stack
- `stm sp, {sb, lr}`: same?
- EOR

### x86-64 Memento

- LEA: Load Effective Address, works on addresses (no access to memory)
- SAR: Shift Arithmetic Right

- `XOR register, immediate`
- jno: Jump No Overflow

## Tools

| | Blutter | Darter | Doldrum | Flutter Spy | JEB | reFlutter |
|---|---|---|---|---|---|---|
| Supported versions | Android ARM64 | ? Old | <= 2.12 (a few forks for 2.13) | | | |
| Dumps the Object Pool | Yes | Yes | No | No | Only strings | No |
| Retrieves Function Names and offsets | Yes | Yes | Yes | No | Yes | Yes |

### Unix / Bash commands

- `ldd FILE.aot`
- `readelf -h FILE.aot | grep Entry`
- `strings FILE.aot | grep xxx`
- `bgrep -t hex 'deadbeef'file` bgrep
- `binwalk -R '\xde\xad\xbe\xef'file`

### GDB

```
$ gdb ./caesar.aot
Reading symbols from ./caesar.aot...
(gdb) info file
...
warning: Cannot find section for the entry point of caesar.aot
```

### Disassembler Memento

Godbolt is able to compile Dart and display the assembly. Handle for simple tests.

JEB:

- Customize default relocation address in Options/Backend properties/ root/parsers/native/disas/*
- View opcodes: Edit > Rendering Options > Show bytes count (6)

Radare:

- Search for a given instruction: `/x OPCODE`, or `/ad eor~0x37`
- Search for a pattern on several instructions with `/ad/`: use `.*` as a joker and `;` to separate instructions: `"/ad/ add.*, x27, 8, lsl 12; 0x1d0]"`

```
add x2, x27, 8, lsl 12
ldr x2, [x2, 0x1d0]
```

- Entry point: `ie`
- Locate main (only if non-stripped): `iM`
- Modify instruction delimiter for search: `e asm.cmt.token=X`
- Define a function: `af`

### reFlutter example

- Install reFlutter Python package
- Source Python environment
- `reflutter w0rdle.apk`
- Select option “Display absolute code offset for functions”
- Get Uber-APK-Signer
- Sign the apk: `java -jar uber-apk-signer-1.3.0.jar --apk release.RE.apk`
- `adb install release.RE-aligned-debugSigned.apk`
- Run it
- Retrieve the dump in `/data/data/com.ph0wnctf.wordle/dump.dart`

```
Library:'package:flutterdle/game.dart' Class: Flutterdle extends Object {
    // missing dump
}

// successful dump of address if Stats.fromJson in domain.dart
Library:'package:flutterdle/domain.dart' Class: Stats extends Object {
  Function 'toJson':. (Stats) => Map<String, dynamic> {
                Code Offset: _kDartIsolateSnapshotInstructions + 0
                    x0000000000109648
  }
}
```

### Blutter

#### Setup

- Fork that accepts APK as input + produces Radare2 script: GitHub
- Using Blutter in a Docker container:

Dockerfile:

```
FROM debian:trixie-slim

RUN DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -yqq python3-pyelftools python3-
    requests git cmake ninja-build build-essential pkg-config libicu-dev
    libcapstone-dev bash git unzip \
     && rm -rf /var/lib/apt/lists/*

RUN git clone https://github.com/cryptax/blutter
ENV TERM=xterm-256color
RUN echo "PS1='\e[92m\u\e[0m@\e[94m\h\e[0m:\e[35m\w\e[0m# '" >> /root/.
    bashrc

RUN mkdir -p /workdir
WORKDIR /workdir
ENTRYPOINT ["/bin/bash"]
```

docker-compose.yml

```
---
services:
  blutter:
    build:
      context: .
      dockerfile: Dockerfile
    image: cryptax/blutter:2024.07
    container_name: blutter
    volumes:
      - /tmp/blutter:/workdir
```

Use:

1. Copy sample in `/tmp/blutter`
2. Build: `docker compose build`
3. Run: `docker compose run blutter`
4. In the container, launch Blutter: `python3 /blutter/blutter.py your.apk ./blutter-out`

#### Example of Object Pool dump

```
pool heap offset: 0x481540
[pp+0x10] Stub: Subtype3TestCache (0x17203c)
[pp+0x18] Stub: Subtype7TestCache (0x171e5c)
[pp+0x20] Stub: AllocateArray (0x174424)
[pp+0x28] Sentinel
[pp+0x30] List(5) [0x1, 0, 0, 0, Null]
[pp+0x38] List(5) [0x1, 0, 0, 0, Null]
...
```

#### Example of assembly output

```
_ _winningMessage(/* No info */) {
  // ** addr: 0x3c71a0, size: 0x454
  // 0x3c71a0: EnterFrame
  //     0x3c71a0: stp             fp, lr, [SP, #-0x10]!
  //     0x3c71a4: mov             fp, SP
  // 0x3c71a8: AllocStack(0x10)
  //     0x3c71a8: sub             SP, SP, #0x10
```

#### Frida hooks

- Blutter provides offsets to various functions. Add the base address of `libapp.so` (or `libflutter.so`).

```
var libapp = Module.findBaseAddress('libapp.so');
    if (libapp == null)
        setTimeout(easterEgg, 500);
    else {
        console.log("Libapp loaded at "+libapp);
        var address = libapp.add(0x21356c);
        ...
    }
```

- Frida will have problems retrieving the arguments of native Flutter functions, because the registers are different (Batteux 2022)
- To modify the output of a function, use `replace()` on the return value of `onLeave` (Beckers 2020):

```
onLeave: function(retval)
    {
      console.log("Retval: " + retval)
      retval.replace(0x1);

    }
```

## References


- ("Flutter," n.d.): Flutter reference website
- ("Dart," n.d.): Dart reference website
- (Lipke 2020)
- (Loura 2020): object serialization
- ("Reverse Engineering a Flutter App by Recompiling Flutter Engine" 2021): using reFlutter
- (Egorov 2022)
- (Software 2022) : JEB support for Dart
- (Apvrille 2022a) : blog post
- (Batteux 2022) : how the Object Pool is serialized in an AOT snapshot.

- (Nikiforov 2022)
- (Ortega 2022) : MoneyMonger
- (Apvrille 2022b): Flutter header parser script
- (Apvrille 2023a): Dart AOT snapshot ImHex pattern,
- (Apvrille 2023b): calling convention
- (Apvrille 2023d): Small Integers
- (Apvrille 2023c) : Fluhorse
- (Alexander 2023)
- (Falliere 2023) : list of Dart snapshot version hashes
- (Apvrille 2023e) : presentation at BlackAlps 2023
- (Apvrille 2023f) : download link for CTF challenge stage 3
- (Team 2023): how to recompile Dart SDK and patch it for dynamic analysis
- (Apvrille 2024b)
- (Apvrille 2024a) : using Radare2 to disassemble Dart
- Blutter + (Wangwarunyoo 2023)
- Doldrum
- Darter
- Flutter Spy: Bash tool to extract information from Flutter Android apps.
- ImHex
- reFlutter: instruments `libflutter.so` to dump memory of addresses of objects and re-compile the Flutter application. The patched application is run and dumps information of code it visits.

Thanks to the Dart community (abitofeverything, mraleph, julemand101, kayZ…) for their comments.

Alexander, Felix. 2023. “Flutter Hackers: Uncovering the Dev’s Myopia.” https://infosecwriteups.com/flutter-hackers-uncovering-the-devs-myopia-part-2-598a44942b5e.

Apvrille, Axelle. 2022a. “Reversing an Android Sample Which Uses Flutter.” https://cryptax.medium.com/reversing-an-android-sample-which-uses-flutter-23c3ff04b847.

———. 2022b. “Source Code to Find Flutter Snapshots and Read Their Header.” https://github.com/cryptax/misc-code/blob/master/flutter/flutter-header.py.

———. 2023a. “Dart AOT Snapshot ImHex Pattern.” https://github.com/cryptax/ImHex-Patterns/blob/master/patterns/dartaot.hexpat.

———. 2023b. “Dart’s Custom Calling Convention.” https://cryptax.medium.com/darts-custom-calling-convention-8aa96647dcc6.

———. 2023c. “Fortinet Reverses Flutter-Based Android Malware "Fluhorse".” https://www.fortinet.com/blog/threat-research/fortinet-reverses-flutter-based-android-malware-fluhorse.

———. 2023d. “Reversing Flutter Apps: Dart’s Small Integers.” https://cryptax.medium.com/reversing-flutter-apps-darts-small-integers-b922d7fae7d9.

———. 2023e. “Unraveling the Challenges of Reverse Engineering Flutter Applications.” https://github.com/cryptax/talks/blob/master/BlackAlps-2023/flutter.pdf.

———. 2023f. “W0rdle 3 Challenge.” https://github.com/ph0wn/writeups/blob/master/2022/reverse/w0rdle/w0rdle.apk.

———. 2024a. “Reversing Dart AOT Snapshots.” http://www.phrack.org/issues/71/11.html#article.

———. 2024b. “The Complexity of Reversing Flutter Applications.” https://github.com/cryptax/talks/tree/master/Nullcon-2024.

Batteux, Boris. 2022. “The Current State & Future of Reversing Flutter Apps.” https://www.guardsquare.com/blog/current-state-and-future-of-reversing-flutter-apps.

Beckers, Jeroen. 2020. “Intercepting Flutter Traffic on Android (ARMv8).” https://blog.nviso.eu/2020/05/20/intercepting-flutter-traffic-on-android-x64/.

“Dart.” n.d. https://dart.dev.

Egorov, Vyacheslav. 2021. “Microbenchmarking Dart (Part 1).” https://mrale.ph/blog/2021/01/21/microbenchmarking-dart-part-1.html.

———. 2022. “Introduction to Dart VM.” https://mrale.ph/dartvm/.

Falliere, Nicolas. 2023. “List of Dart Snapshot Version Hash (Internal Version) to Version Tag (Public Git Tag).” https://gist.github.com/nfalliere/84803aef37291ce225e3549f3773681b.

FatalSec. 2024. “Reverse Engineering Obfuscated Flutter App.” https://www.youtube.com/watch?v=0uUSwMg2suk.

“Flutter.” n.d. https://flutter.dev.

Lipke, Andre. 2020. “Reverse Engineering Flutter Apps.” https://blog.tst.sh/reverse-engineering-flutter-apps-part-1/.

Loura, Ricardo. 2020. “Reverse Engineering Flutter for Android.” https://rloura.wordpress.com/2020/12/04/reversing-flutter-for-android-wip/.

Nikiforov, Philip. 2022. “Fork Bomb for Flutter.” https://swarm.ptsecurity.com/fork-bomb-for-flutter/.

Ortega, Fernando. 2022. “MoneyMonger: Predatory Loan Scam Campaigns Move to Flutter.” https://www.zimperium.com/blog/moneymonger-predatory-loan-scam-campaigns-move-to-flutter/.

“Reverse Engineering a Flutter App by Recompiling Flutter Engine.” 2021. https://tinyhack.com/2021/03/07/reversing-a-flutter-app-by-recompiling-flutter-engine/.

Software, PNF. 2022. “Dart AOT Snapshot Helper Plugin to Better Analyze Flutter-Based Apps.” https://www.pnfsoftware.com/blog/dart-aot-snapshot-helper-plugin-to-better-analyze-flutter-based-apps/.

Team, Ostorlab. 2023. “Flutter Reverse Engineering and Security Analysis.” https://blog.ostorlab.co/flutter-reverse-engineering-security-analysis.html.

Wangwarunyoo, Worawit. 2023. “Blutter - Reversing Flutter Application by Using Dart Runtime.” https://conference.hitb.org/hitbsecconf2023hkt/materials/D2%20COMMSEC%20-%20B(l)utter%20%e2%80%93%20Reversing%20Flutter%20Applications%20by%20using%20Dart%20Runtime%20-%20Worawit%20Wangwarunyoo.pdf.