Dart and Flutter Reverse Engineering Reference

Axelle Apvrille, Fortinet

March 29, 2024

Contents

Dart SDK **3**
Dart language . . . 3
Strings . . . 3
No type to represent *bytes*, nor *characters* . . . 3
SDK Contents . . . 4
SDK Commands . . . 4

Flutter **4**
Contents . . . 4
Install . . . 4
App Creation . . . 5
Platform Channels . . . 5
Classes . . . 5

Versions **5**

Dart output formats **6**

Isolate **7**

Dart AOT Snapshot Format **7**
ELF shared object . . . 7
AOT snapshot . . . 7

Registers **9**
Dedicated registers for Dart . . . 9
Object Pool (PP) . . . 10
THR offsets . . . 10
x86-64 assembly using null object . . . 11
Aarch32 assembly checking for stack overflow . . . 11
Aarch64 assembly checking for stack overflow . . . 11
x86-64 assembly checking for stack overflow . . . 11
Recap of important registers . . . 11

Encoding of Small Integers (SMI) **12**
x86-64 example . . . 13
x86-64 control for SMI/Mint case . . . 13

Calling convention (ABI) **14**

Function prologue **14**

Dart SDK source code ref **15**

Assembly memento **16**
Aarch64 Memento 16
Aarch32 Memento 16
x86-64 Memento 16

Tools **17**
Unix / Bash commands 17
GDB 17
Disassembler Memento 17
reFlutter example 18
Blutter example 18

References **19**

Dart SDK

Dart language

- Built-in types: **`int`**, **`double`**, `bool`, `List`, `Set`, `String`

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);
```

No type to represent *bytes*, nor *characters*

Workaround #1: `List<int>`

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

Workaround #2: `Uint8List` (note to import `dart:typed_data`)

```
import 'dart:type_data';

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

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

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**? I had 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`

App Creation

- With Android Studio, create a Flutter app, or command line `flutter create projectname`
- Create a Flutter project app using Java, and for iOS, Android and Linux.
- To build RELEASE app, Android Studio > Build > Build APK or to build in command line `flutter build apk`
- To run an App in Linux, you can also just use `flutter run`

Platform Channels

Communication between the Dalvik layer and Flutter is perform through *Platform Channels*. Platform channels are implemented by Flutter and are the standard way to communication between a Flutter client application and its host (Android in our case). In particular, Flutter provides a class named `MethodChannel` to help *Dart* code call *Java or Kotlin* code.

```
1     private final Object func(MethodCall methodCall0, Result
          methodChannel$Result0) {
2         SharedPreferences.Editor sharedPreferences$Editor0;
3         if(l8.a6(methodCall0.method, "setConfig")) {
4             String s = (String)methodCall0.argument("smsCount");
5             if(s == null) {
6                 return null;
7             }
8
9             ...
10        }
```

Classes

- The `Sentinel` class is used to “indicate the normal response is not available”.

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

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 formats	Portable	Requires an external Dart Runtime VM to run

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 seperately UI rendering, or network requests etc.

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 0b01001100011101100101: **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.

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

```
1 mov r11, qword [r15 + 0x168f]
2 mov qword [rsp], r11
3 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`:

```
1 static constexpr dart::compiler::target::word Thread_top_offset = 0x24;
2 static constexpr dart::compiler::target::word
     Thread_field_table_values_offset = 0x30;
3 static constexpr dart::compiler::target::word Thread_stack_limit_offset = 0
     x38;
4 static constexpr dart::compiler::target::word Thread_bool_true_offset = 0x70
     ;
5 static constexpr dart::compiler::target::word Thread_bool_false_offset = 0
     x78;
6 static constexpr dart::compiler::target::word
     Thread_call_to_runtime_entry_point_offset = 0xfc;
7 static constexpr dart::compiler::target::word Thread_isolate_group_offset =
       0x338;
8 static constexpr dart::compiler::target::word Thread_vm_tag_offset = 0x6d8;
9 static constexpr dart::compiler::target::word
     Thread_saved_stack_limit_offset = 0x6f0;
```

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()
```

Aarch32 assembly checking for stack overflow

```
; 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
```

Aarch64 assembly checking for stack overflow

```
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
```

x86-64 assembly checking for stack overflow

```
; push base pointer on the stack
push rbp
; the new value for the base pointer is the stack pointer
mov rbp, rsp
; allocate 16 bytes
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
```

Recap of important registers

Architecture	Register	Use
arm7eabi	r5	**Object Pool**
	r10	Pointer to running VM thread
	r11	Frame Pointer
	r13	Stack Pointer
	r14	Link Register
	r15	Program Counter
arm64	X15	**Custom Stack Pointer**. *SP*
	X26	Pointer to running VM thread. **THR**
	X27	**Object Pool**. **PP**
	X28	**HEAP_BITS**.
	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)

In Dart, all arguments are 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	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
BL        _StringBase.+                   ; concatenate both strings: '
    stage2: ph0wn{'
```

Function prologue

Example in x86-64:

```
; push base pointer on the stack
push rbp
; the new value for the base pointer is the stack pointer
mov rbp, rsp
; allocate 16 bytes
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:

```
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
```

Dart SDK source code ref

	URL
ClassId enumeration	`sdk/runtime/vm/class_id.h`
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
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
Class Smi	`runtime/vm/object.h`

	URL
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

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`
- 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 example

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
```

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 2024)
- Blutter
- 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.

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.

———. 2024. “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.

“Dart.” n.d. https://dart.dev.

Egorov, Vyacheslav. 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.

“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.