AID(L)ing the Fuzzer

Fuzzing Harness Auto-Generation
for Android AIDL Services

Emanuel Mairoll

hexhive

EPFL

Android API Fuzzing is (Still) Hard

Android system services run with high privileges - prime bug bounty land.

Exposed to clients via Binder IPC.

Google's **`binder_parcel`** fuzzer treat transactions as random "byte soup".

However, IPC validation is strictly enforced - tons of early aborts.

Input instability: small fuzzing changes ruin the entire parcel.

Leads to:

- Low valid input rates.
- Wasted effort re-guessing interface logic.

The Thesis in One Slide

Mine the AIDL spec to autogenerate a structure-aware libFuzzer harness.

Custom mutators keep mutations legal yet still exploratory.

Goal:

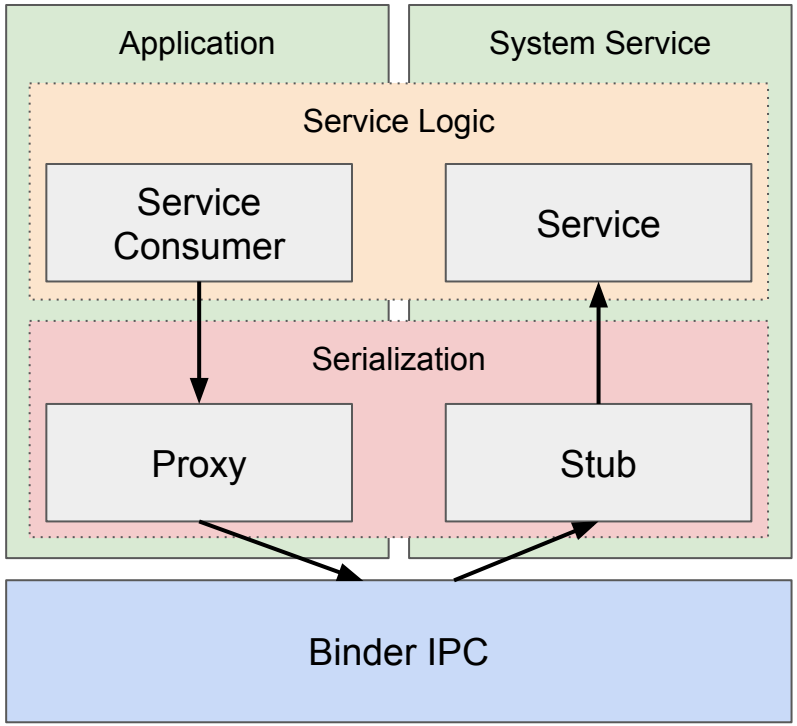
- maximize valid calls
- reach service logic immediately

Binder IPC Recap

Android's primary IPC mechanism consists of:

- **Binder Kernel driver**: handles IPC between processes
- **libbinder library**: structured byte buffers ("Parcels") and serialization logic.

Strict serialization: Any mismatch or misalignment immediately aborts.

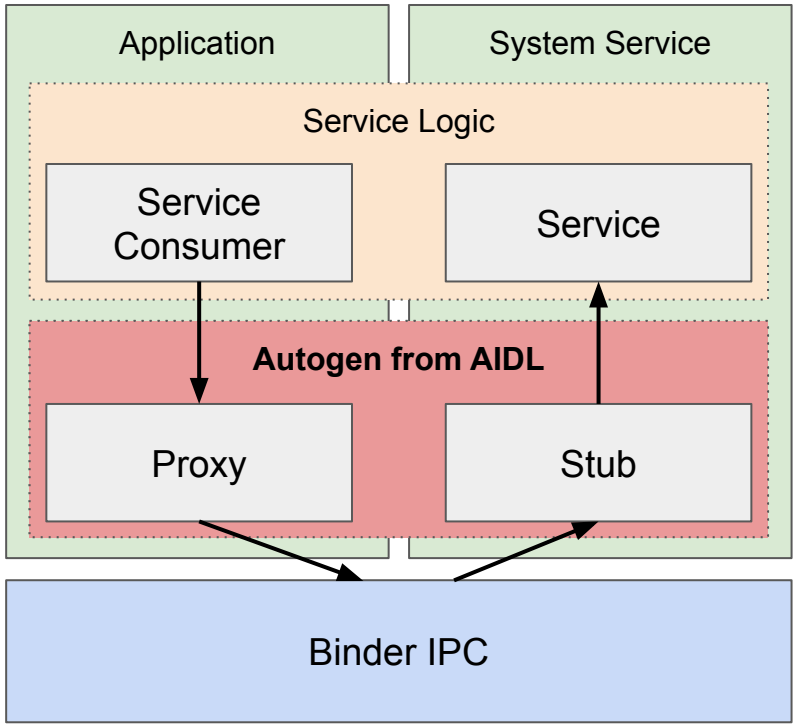


Binder IPC Recap

Android's primary IPC mechanism consists of:

- **Binder Kernel driver**: handles IPC between processes
- **libbinder library**: structured byte buffers ("Parcels") and serialization logic.

Strict serialization: Any mismatch or misalignment immediately aborts.



AIDL 101

“Android Interface Definition Language”

Defines IPC interfaces in a language-agnostic way.

Android build system generates client/server stubs.

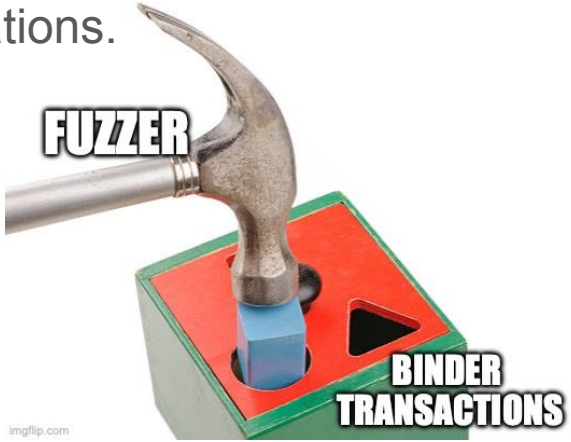
Types: primitives, parcelables, arrays, binders, file descriptors…

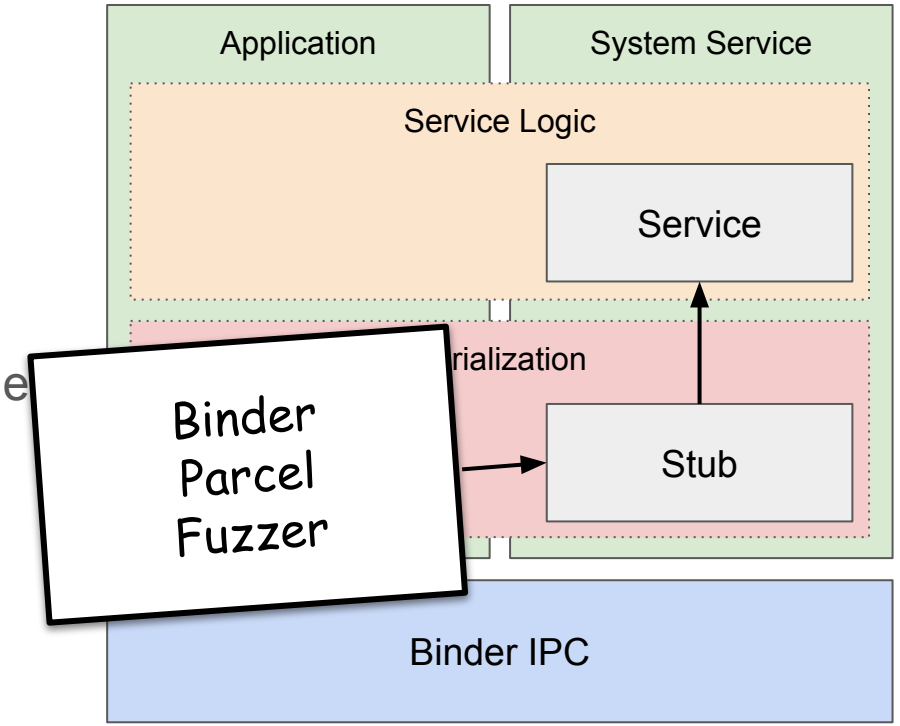
Language targets: Java, C++, NDK, Rust.

```
interface IHello {
    void hello();
    int sum(int x, int y);
    void printStruct(in MyStruct data);
    void callback(MyBinder callback);
}
```

Binder-Parcel Fuzzer (The Baseline)

Random target, code and **parcel!**
Writes arbitrary file descriptors, binders, byte blobs -> “Chaos Fuzzing”
Transaction codes and arguments discovered by random chance.
Low valid call rate (~20-60%) and unstable mutations.





Fuzzing: Why not just use AIDL?

Interfaces already formally specified in AIDL.

Method names, transaction codes, argument layouts - fully known!

Structure-aware fuzzing could directly use these specs.

Generated fuzzer would match exact interface structure.

Enables fuzzing actual service logic

rather than just serialization.

Our Approach

Add new language to AIDL compiler: **`--lang fuzz-harness`**.

Completely isolated from parsing / AST validation logic.

Output = C++ fuzz harness with full method knowledge.

- Custom per-transaction mutators.
- Type-safe argument construction.
- Special handling for “live” data-types.

Supports primitives, strings, arrays,

parcelables, binders, file descriptors

How It Works: Basics

AIDL compiler uses visitor pattern to walk method signatures.

For each method: build load / mutate / store / init code blocks.

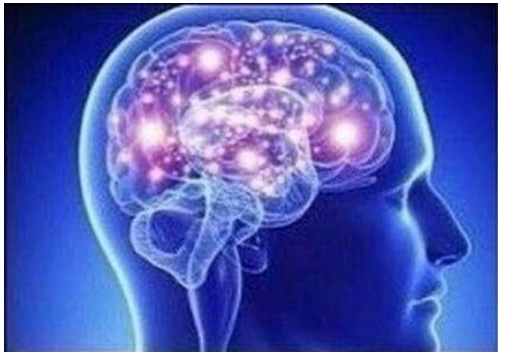
Always round-trip via **`android::Parcel`** for alignment.

`LLVMFuzzerMutate` called per argument and mutation step.

Assembled into **`LLVMFuzzerCustomMutator`** + per transaction helpers.

```
int32_t x; in.readt32(&x);
int32_t y; in.readt32(&y);
LLVMFuzzerMutate((uint8_t*)&x,sizeof(int32_t),sizeof(int32_t));
LLVMFuzzerMutate((uint8_t*)&y,sizeof(int32_t),sizeof(int32_t));
out.writeInt32(x);
out.writeInt32(y);
```

Handling More Complex Types



Variable Length Types (Strings, Primitive Arrays + Lists):

- Prefixed with length.
- Fuzzed via scratch buffer, treated as plain bytes.

Parcelables:

- Prefixed with flattened size.
- Autogen writes placeholder 0, serializes fields one by one, back-patches size.
- Works recursively - supports nested parcelables.

Binders & FDs - The Hard Stuff

Can’t just write raw bytes - must instantiate real objects.

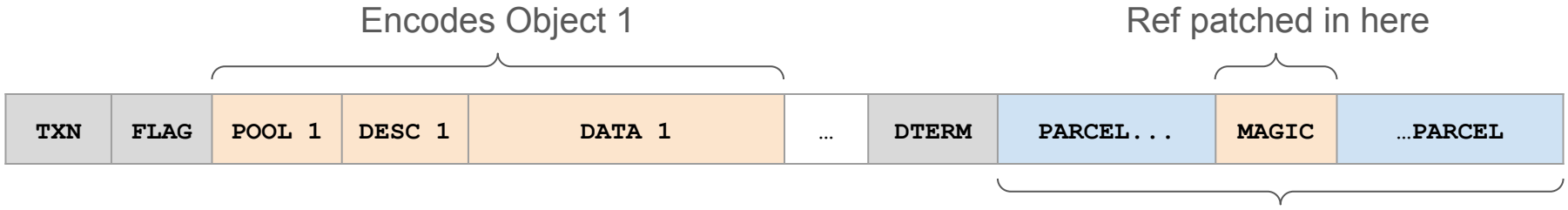
Uses dynamic seed prefix to encode Binder/FD config.

Applied to pool object during fuzz execution.

Seed contains magic placeholders, patched in-place during fuzz run.

Ensures reproducible seeds even for dynamic data.

Guarantees valid objects and avoids memory/resource leaks.



Experimental setup

Targets: 6 real Android services + demo **`HelloService`**.

`IDnsResolver, INfc, IInstalld,`

`IStatsd, IThreadChip, IOccupantAwareness`

2x 12h fuzz runs per service (baseline + autogen).

Baseline: upstream **`binder_parcel`** fuzzer (refuzzing disabled).

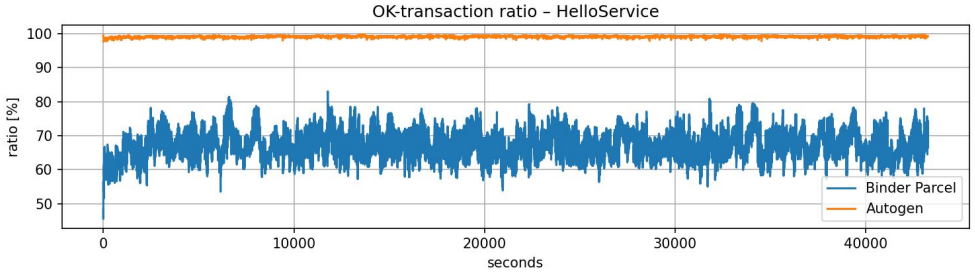
Both fuzzers seeded with empty corpus to avoid bias.

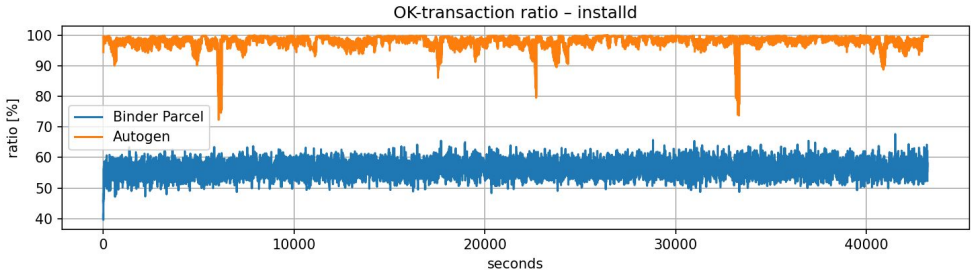
Metrics:

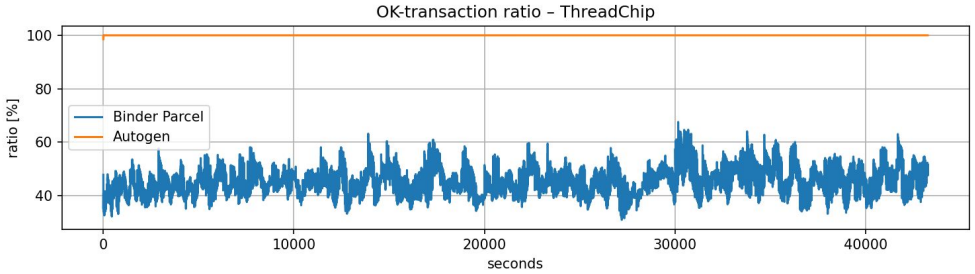
- OK-call ratio (custom instrumentation)
- libFuzzer edge coverage.

Results: OK Rate = Autogen WINS

- Autogen fuzzer immediately achieves >85% valid calls.
- Baseline consistently stuck between 20% - 75%.
- Holds across all tested services.
- Real logic gets hit faster and more often.







Results: Coverage is… it’s complicated

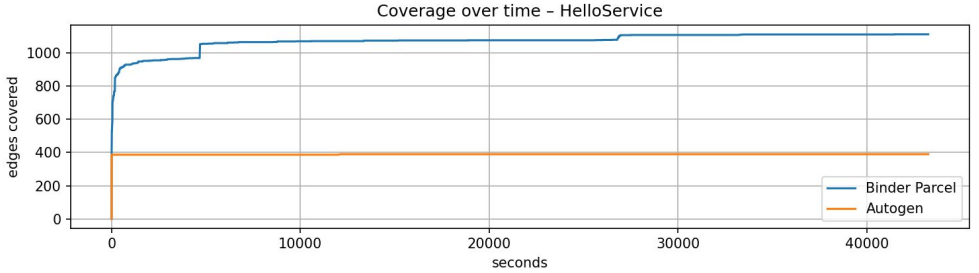
Baseline eventually overtakes Autogen in raw coverage.

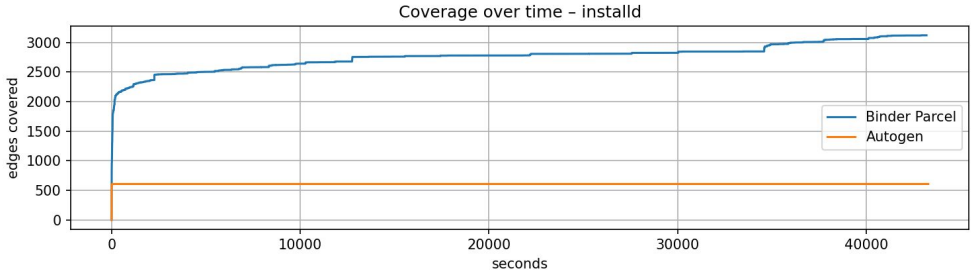
Possible Reasons:

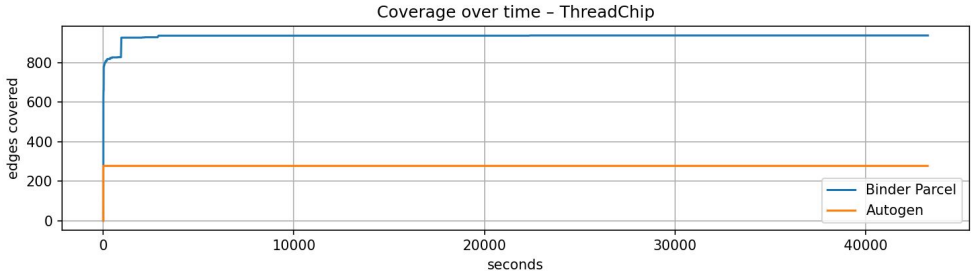
- Cov artificially inflated - measuring harness and error handling code
- Seed mutation overly constrained.
- Mutation strategy suboptimal.
- Bug in the parsing logic

BUT:

Coverage ≠ quality







Future Work

Coverage filters to ignore harness code and focus on service.

Smarter mutation strategies to overcome coverage plateaus.

Better handling of internal service state and method chaining.

Add returned-binder refuzzing for callback-heavy protocols.

Patch into Androids `cc_fuzz` build target.

TL;DR

Binder Service fuzzing is hard because serialization is strict.

But AIDL already defines structure.

We made the AIDL compiler generate structure-aware fuzzers.

Autogen fuzzers = fast, stable, accurate.

Valid input rate went 🚀

Still work to do on full coverage.

Integrates into existing build infra & works today.

Q&A