AID(L)ing the Fuzzer: Fuzzing Harness Auto-Generation for Android AIDL Services

Emanuel Mairoll

June 26, 2025

Abstract

Modern Android system services are exposed through Binder IPC interfaces that are formally specified in the Android Interface Definition Language (AIDL). While coverage-guided fuzzing is widely used to harden these services, the current "binder-parcel" fuzzer deployed by Google treats every transaction as an unstructured byte stream, forcing the fuzzer to rediscover method codes, argument layouts and parcelable boundaries by chance. This wastes a substantial fraction of fuzzing effort on overcoming the very input validation that the autogenerated AIDL stubs already enforce.

Here, we introduce **AID(L)ing the Fuzzer**, a compiler-driven pipeline that automatically generates structure-aware fuzzing harnesses straight from AIDL files. A new "fuzz-harness" backend in the upstream AIDL compiler emits C++ code with type-safe argument packing, custom `libFuzzer` mutators and deterministic handling of Binder objects and file descriptors.

Across six real AOSP services the autogenerated fuzzers consistently achieved > 85 % and often > 99 % successful transactions, reaching service logic almost immediately, whereas the baseline parcel fuzzer managed only 20 - 60 %. Overall code coverage and executions per second are still lower in some cases, but the approach reliably exercises deeper, previously untested paths and shows clear potential.

Contents

Abstract **1**

1 Introduction **4**

2 Background **6**
2.1 Binder IPC . . . 6
2.2 AIDL and Code Generation . . . 6
2.3 Binder Parcel Fuzzer . . . 8

3 Design of Autogenerated Fuzzer **11**
3.1 Using adapted FuzzedDataProvider . . . 11
3.2 Using Custom Mutators . . . 12

4 Implementation **14**
4.1 AIDL Compiler Internals . . . 14
4.2 Custom Language Backend for Fuzz Harness Generation . . . 18
4.3 Feature Set and Supported Data Types . . . 18
4.4 Fuzzer Harness Structure and Generation . . . 19
4.5 Primitive Data Types . . . 20
4.6 Variable-Length Data Types . . . 23
4.7 Parcelables . . . 24
4.8 Binders and File Descriptors . . . 26
4.8.1 Challenges of Dynamic Data Types . . . 26
4.8.2 Dynamic Seed Section . . . 27
4.8.3 Parcel Patching . . . 30

5 Evaluation **33**
5.0.1 Integration in the Build System . . . 33
5.1 Experimental Setup . . . 34
5.1.1 Services Under Test . . . 34
5.1.2 Instrumentation and Metrics . . . 35
5.1.3 Campaign Execution . . . 35

5.2 Results 36
5.2.1 Service Coverage and Call Success Rate 36
5.2.2 Interpretation of Results 36
5.2.3 On Coverage Metrics and Harness Complexity 41

6 Discussion **42**
6.1 Future Work 42
6.2 Related Work 42

7 Conclusion **44**

Bibliography **45**

Chapter 1

Introduction

Androids system architecture relies heavily on a large collection of services that handle everything from device management and network communication to media and security. These services, implemented in a variety of programming languages, interact almost exclusively through Binder, Androids core in-kernel IPC mechanism. To help manage the complexity of these interactions, developers use the Android Interface Definition Language (AIDL) to formally specify service interfaces. This not only streamlines development but also automates much of the code needed to safely pass structured data between processes.

Security is a critical concern in this ecosystem. Binder services are often highly privileged, making them attractive targets for attackers. As a result, fuzzing - systematically testing software by feeding it large volumes of random or semi-random inputs - has become a central part of Androids vulnerability discovery strategy. Despite this, the way most fuzzers are set up for Binder services still leaves a lot on the table. The standard “binder parcel” fuzzers simply treat every transaction as an opaque byte stream, relying on chance to hit valid transaction codes and argument layouts. As a consequence, most test inputs fail almost immediately during parsing or validation, and the fuzzer spends much of its effort trying to bypass input checks instead of exercising meaningful service logic.

What stands out here is the disconnect between the information available and the approach used. The AIDL specifications already provide a complete, machine-readable description of every service interface, down to each argument type and transaction code. But current fuzzers largely ignore this resource, instead operating in a black-box manner. This leads to low input stability, shallow coverage, and inefficiency, especially when dealing with complex or dynamic argument types.

This thesis aims to close that gap: *Can we leverage the structure defined in AIDL to automatically generate smarter, structure-aware fuzzing harnesses for Android Binder services?* Rather than

continuing with ad hoc or manual harness development, the goal is to integrate harness generation directly into the AIDL compilation pipeline. This way, the generated fuzzers always reflect the latest interface, benefit from type safety, and can easily keep pace with changes across the platform.

There are two main motivations for pursuing this approach. First, to demonstrate that harnesses generated with full knowledge of the AIDL interface can achieve higher transaction success rates and reach core service logic much faster than traditional, structure-agnostic fuzzers. Second, to understand the trade-offs in practice: does improved input validity and stability necessarily lead to broader code coverage, or does random, less-structured fuzzing still play a role in exploring edge cases and error-handling paths?

To address these questions, this work presents a new backend for the AIDL compiler that emits C++ fuzzing harnesses tailored to the structure of each interface. The system supports a wide range of data types, including primitives, parcelables, arrays, binders, and file descriptors, and fits smoothly into the Android build system, enabling automated fuzzing for any service defined via AIDL.

In evaluation across six real Android services, the autogenerated fuzzers consistently achieved over 85% - and often over 97% - successful transactions, compared to just 20-60% for the standard binder parcel fuzzer. For several services, the autogen harness reached nearly 99% valid calls, demonstrating a dramatic improvement in input validity and early code reach. While total code coverage is still higher for the baseline due to its broader exploration of edge cases, the autogenerated approach reliably exercises service logic and previously untested paths from the very start.

Chapter 2

Background

2.1 Binder IPC

Android is an extensive and complex project comprising numerous system components, which are implemented as separate services. These services, often developed in various programming languages, need a uniform and robust way to communicate with each other. Android achieves this through the Binder Inter-Process Communication (IPC) mechanism. Binder IPC transmits messages, essentially byte buffers, between processes using a specialized kernel driver. It supports not only simple native data types and structured data types (parcelables), but also more complex resources such as file descriptors and live objects, themselves also referred to as “binders” (hereafter referred to as binder objects). Internally, most Android APIs are essentially wrappers around Binder interfaces - from hardware components and network services to logging mechanisms, binders are omnipresent in the Android system.

For the actual IPC mechanism, methods can be directly invoked on binder objects, with parameters serialized and passed through the Binder interface. Reference counting and lifecycle management for these binder objects are handled transparently by the kernel, abstracting substantial complexity from client and service code.

2.2 AIDL and Code Generation

To interact through these binder interfaces, there must exist code that serializes data (like function arguments) into a form understandable by Binder. This involves serializing data into a so called Binder parcel, fundamentally just a byte buffer understood by Binder and forwarded to the receiving service. Historically, these binary interfaces were manually implemented, but Android Interface

Definition Language (AIDL) has entirely automated this task. AIDL provides a language-agnostic method to define interfaces, closely paired with automatic code generation by the AIDL compiler, which is integrated into the Android build system.

Note: Binder services are sometimes confusingly called AIDL services. However, this report distinguishes explicitly between an AIDL *interface* definition and a Binder *service* implementation.

A typical AIDL file defines methods and their arguments, which can use various native data types, lists, parcelables (structured objects), file descriptors, and binder objects itself, which defined by an `interface` definition. These types map directly to language-native types; a detailed lookup table can be found in the android documentation [1]. Using Android's build system, specifying an AIDL target automatically generates headers and the complete serialization logic, ready to be used in by the developer.

Running the `aidl` compiler manually allows for easy introspection of the autogenerated code. For example, the client-side code for a method like `sum(int32_t x, int32_t y)` looks like in the following.

```
::android::binder::Status BpHello::sum(int32_t x, int32_t y, int32_t* _aidl_return) {
  ::android::Parcel _aidl_data;
  _aidl_data.markForBinder(remoteStrong());
  ::android::Parcel _aidl_reply;
  ::android::status_t _aidl_ret_status = ::android::OK;

  _aidl_ret_status = _aidl_data.writeInterfaceToken(getInterfaceDescriptor());
  if (_aidl_ret_status != ::android::OK) { goto _aidl_error; }
  _aidl_ret_status = _aidl_data.writeInt32(x);
  if (_aidl_ret_status != ::android::OK) { goto _aidl_error; }
  _aidl_ret_status = _aidl_data.writeInt32(y);
  if (_aidl_ret_status != ::android::OK) { goto _aidl_error; }

  _aidl_ret_status = remote()->transact(BnHello::TRANSACTION_sum, _aidl_data, &_aidl_reply, 0);
  if (_aidl_ret_status != ::android::OK) { goto _aidl_error; }
  _aidl_ret_status = _aidl_reply.readt32(_aidl_return);
  if (_aidl_ret_status != ::android::OK) { goto _aidl_error; }

_aidl_error:
  ::android::binder::Status _aidl_status;
  _aidl_status.setFromStatusT(_aidl_ret_status);
  return _aidl_status;
}
```

The autogenerated code makes heavy use of the Binder parcel API, sequentially writing arguments into said parcel. The invoked methods are uniquely identified through an integer transaction code starting at 1.

Similarly, the server-side code reads arguments, invokes the actual method, and handles return values:

```
case BnHello::TRANSACTION_sum: {
  int32_t in_x, in_y, _aidl_return;
  if (!_aidl_data.checkInterface(this)) { _aidl_ret_status = ::android::BAD_TYPE; break; }
  _aidl_ret_status = _aidl_data.readt32(&in_x);
  if (_aidl_ret_status != ::android::OK) { break; }
  _aidl_ret_status = _aidl_data.readt32(&in_y);
  if (_aidl_ret_status != ::android::OK) { break; }

  ::android::binder::Status _aidl_status(sum(in_x, in_y, &_aidl_return));
  _aidl_ret_status = _aidl_status.writeToParcel(_aidl_reply);
  if (_aidl_ret_status != ::android::OK) { break; }
  _aidl_ret_status = _aidl_reply->writeInt32(_aidl_return);
  if (_aidl_ret_status != ::android::OK) { break; }
}
```

As becomes very apparent from these code listings, this interface has very strict error checking, aborting if any step of the serialization or deserialization fails. Only if the parcel is in the exact expected format is the method actually called.

2.3 Binder Parcel Fuzzer

Fuzzing-based testing has become an integral part of the Android Open Source Project. Many libraries are tested using fuzzers - typically `libFuzzer`, which is directly integrated into the Android build system. You can simply define a `cc_fuzz` target in the `Android.bp` file, and the compilation and linking of the fuzzer is automatically handled for you. For services, the build system actually enforces the definition of a corresponding fuzzer; otherwise it throws an error.

Implementing a fuzzer for a Binder service is very straightforward:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    auto binder = sp<IHelloService>::make();
    fuzzService(binder, FuzzedDataProvider(data, size));
    return 0;
}
```

Despite its simplicity, this fuzzer has inherent performance drawbacks, which become apparent when looking into its implementation.

Its primary strategy is generating entirely random inputs, essentially expecting the fuzzer to

eventually “discover” valid transaction codes and arguments. The main loop of the fuzzer can be summarized as follows:

```
void fuzzService(const std::vector<sp<IBinder>>& binders, FuzzedDataProvider&& provider) {
    ...
    while (provider.remaining_bytes() > 0) {
        uint32_t code = ... provider.ConsumeIntegral<uint32_t>()
        uint32_t flags = provider.ConsumeIntegral<uint32_t>();
        Parcel data;
        sp<IBinder> target = options.extraBinders.at(
                provider.ConsumeIntegralInRange<size_t>(0, options.extraBinders.size() - 1));
        ...
        std::vector<uint8_t> subData = provider.ConsumeBytes<uint8_t>(
                provider.ConsumeIntegralInRange<size_t>(0, provider.remaining_bytes()));
        fillRandomParcel(&data, FuzzedDataProvider(subData.data(), subData.size()), &options);
        ...
        ::android::status_t _aidl_ret_status = target->transact(code, data, &reply, flags);
    ...
}
```

Here, the fuzzer randomly chooses binder targets, transaction codes, and arguments. The function `fillRandomParcel` fills the parcel randomly, selecting among writing raw data, file descriptors, or binder objects:

```
void fillRandomParcel(Parcel* p, FuzzedDataProvider&& provider, RandomParcelOptions* options) {
    ...
    while (provider.remaining_bytes() > 0) {
        auto fillFunc = provider.PickValueInArray<const std::function<void()>>({
                [&]() { // write data
                    ...
                    std::vector<uint8_t> data = provider.ConsumeBytes<uint8_t>(toWrite);
                    p->write(data.data(), data.size());
                    ...
                },
                [&]() { // write FD
                        ...
                        std::vector<unique_fd> fds = getRandomFds(&provider);
                        p->writeFileDescriptor(fds.begin()->release(), true);
                        ...
                    }
                },
                [&]() { // write binder
                    ...
                    binder = getRandomBinder(&provider);
                    (void) p->writeStrongBinder(binder);
                },
        });
```

```
        fillFunc();
    }
}
```

Broadly, this approach guarantees wide interface coverage and is straightforward to implement, since it requires no additional information beyond the existence of a service. However, several drawbacks become apparent when reviewing its effectiveness in practice. First, the fuzzer spends a disproportionate amount of time attempting to "discover" valid transaction codes and method argument formats, as every input is treated as an unstructured byte stream. This effectively shifts the majority of fuzzing effort from exercising real service logic to simply parsing the input format - a process which is already strictly enforced by the autogenerated AIDL code. Second, the prevalence of variable-length fields (such as strings, arrays, or vectors) means that small mutations in input can drastically alter the interpretation of the remaining data, leading to significant input instability and making it difficult for coverage-guided fuzzers to make incremental progress. Finally, because FuzzedDataProvider consumes input data sequentially, even minor changes near the start of the buffer can cascade into unpredictable and unrelated changes in the parsed arguments, further complicating guided exploration. As a result, the effectiveness of this approach is fundamentally limited by its lack of structural awareness.

However, given that virtually all binder services have associated AIDL files explicitly defining their interfaces, it naturally raises the question: *If interface information is available, why not leverage it directly to guide fuzzing?*

Because AIDL files provide a machine-readable, language-agnostic definition of all transaction codes, method signatures, argument types, and even parcelable field layouts, they encapsulate all the protocol information necessary to synthesize valid binder transactions without guesswork. This means that, instead of attempting to reverse-engineer or brute-force the correct structure (as in traditional black-box fuzzing), the fuzzer can be systematically generated to follow the precise interface contract, yielding dramatically higher initial coverage and near-perfect input validity.

This motivates the objective of this project - a fuzzer system that automatically utilizes AIDL interfaces for generating precise and structured fuzzing inputs. The specific design choices made to achieve this goal are detailed in the following chapter; the intricicies of integrating this logic into the AIDL compiler and the code generation process are discussed in the subsequent implementation chapter.

Chapter 3

Design of Autogenerated Fuzzer

The initial step of this project was to determine what kind of fuzzer design would best make use of the structure information provided by AIDL interfaces - without necessarily yet concerning ourselves with how we would get such a fuzzer "automagically".

3.1 Using adapted FuzzedDataProvider

A natural starting point is a fuzzer that directly maps transaction codes and method signatures defined in the AIDL specification. Such a harness might look as follows:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    FuzzedDataProvider provider(data, size);
    sp<IHelloService> helloService = sp<IHelloService>::make();

    while (provider.remaining_bytes() > 0) {
        Parcel dataParcel;
        uint32_t txnCode = IBinder::FIRST_CALL_TRANSACTION +
            provider.ConsumeIntegralInRange<uint8_t>(0, TXN_COUNT);
        switch (txnCode) {
            case 1: {
                // hello(): no extra parameters
                helloService->transact(txnCode, dataParcel, &reply, 0);
                break;
            }
            case 2: {
                // sum(int32_t x, int32_t y)
                int32_t x = provider.ConsumeIntegral<int32_t>();
                int32_t y = provider.ConsumeIntegral<int32_t>();
                dataParcel.writeInt32(x);
```

```
                dataParcel.writeInt32(y);
                helloService->transact(txnCode, dataParcel, &reply, 0);
                break;
            }
            // ...
        }
    }
    return 0;
}
```

This direct implementation closely mirrors the intended use of the binder interface and eliminates most of the guesswork regarding transaction codes and method argument formats. However, while this method resolves most parsing errors, it does not fully address issues of input stability - an important concern given the prevalence of variable-length data types like Strings in Android system services.

3.2 Using Custom Mutators

A more robust solution involves using libFuzzer's custom mutator functionality. Proper structure-aware fuzzing, as outlined in the libFuzzer documentation [7], allows developers to constrain mutation strategies to those that respect the expected input structure.

By defining `LLVMFuzzerCustomMutator`, developers can ensure that libFuzzer mutates only the portions of the input explicitly passed to `LLVMFuzzerMutate`. This is particularly useful in cases where the fuzzing input must follow a specific format, such as in our situation, when the data is first deserialized before being consumed by the service.

Here, the custom mutator explicitly manipulates the input seed for every transaction to meet its specific data requirements, thus ensuring both stability and precision in mutating the fuzzing input:

```
extern "C" size_t LLVMFuzzerCustomMutator(uint8_t *Data, size_t Size,
                                          size_t MaxSize, unsigned int Seed) {
    uint32_t *txnCode = reinterpret_cast<uint32_t*>(Data);
    if (Size < 4) {
        *txnCode = android::IBinder::FIRST_CALL_TRANSACTION + (Seed % TXN_COUNT);
        return 4;
    }
    switch(*txnCode) {
        case 1: {
            // hello(): no extra parameters
            return 4;
        }
        case 2: {
```

```
            // sum(int32 x, int32 y)
            uint32_t *px = reinterpret_cast<uint32_t*>(Data+4);
            uint32_t *py = reinterpret_cast<uint32_t*>(Data+8);
            const size_t canonical_size = 4 + sizeof(int32_t) + sizeof(int32_t);
            // Initialize fields if needed
            if (Size < canonical_size) {
                *px = 0;
                *py = 0;
                return canonical_size;
            }
            // Extract and mutate arguments
            int32_t x_fuzz = *px;
            int32_t y_fuzz = *py;
            LLVMFuzzerMutate(reinterpret_cast<uint8_t*>(&x_fuzz),
                             sizeof(int32_t), sizeof(int32_t));
            LLVMFuzzerMutate(reinterpret_cast<uint8_t*>(&y_fuzz),
                             sizeof(int32_t), sizeof(int32_t));
            *px = x_fuzz;
            *py = y_fuzz;
            return canonical_size;
        }
        // ...
    }
}
```

In the custom mutator, we are "aware" of where in the seed the actual arguments are stored, load them into temporary fuzz variables, pass every one of them into `LLVMFuzzerMutate` and then store it back again. This approach may appear very “lengthy” at first, but the seemingly over-verbose SSA style was chosen because for more complex data type "de-flattening" (so that they can be passed into `LLVMFuzzerMutate`) is more complicated, so we default to always loading it as a variable.

With this approach, in addition to avoiding parsing errors, we can guarantee fuzzing stability, since every field is fuzzed independently. Also, another common problem we are bypassing here is the fuzzer input running out, since we control the length of the output seed using the return value (so we can "extend" seed as needed).

It is of course true that this approach is more complicated than the naive FuzzedDataProvider-based implementation. However, due to the strict constraints that we enforce onto the mutation of the seed in the `LLVMFuzzerMutate`, the `LLVMFuzzerTestOneInput` function is comparatively simple - we can just take the seed and send it as a parcel.

Due to the benefits outlined above, we ultimately decided to use the second approach for AIDL fuzzing. The very obvious question, however, is how to now automatically create/generate a fuzzer following this paradigm. This will be outlined in the following section.

Chapter 4

Implementation

4.1 AIDL Compiler Internals

The AIDL compiler is a central component of the Android Open Source Project. Rather than a classical compiler, it is more accurately described as a code generator for multiple languages and/or purposes. Its source can be found in the AOSP under `system/tools/aidl`, and it is implemented in C++. The codebase itself is quite extensive, at over 100,000 lines of code, of which roughly 70,000 lines are devoted to tests, highlighting how essential both stability and correctness are for the generated code.

The AIDL compiler is conceptually organized into three major components:

1. **Argument parsing and CLI options**, which handle tasks such as validating input files and selecting the desired language backend.
2. **AIDL parsing logic**, which processes AIDL files, resolves imports, and transforms them into an internal tree of definitions (not a classic syntax tree, but a data structure describing interfaces, methods, constants, etc.).
3. **Language-specific backends**, which are implemented as pairs of helper (`aidl_to_<lang>.cpp`) and code emission (`generate_<lang>.cpp`) files. The existing back-ends target C++, Java, NDK, and Rust.

The entry point for all this logic is the `aidl.cpp` file, which handles the global logic for command-line invocation.

```
int aidl_entry(const Options& options, const IoDelegate& io_delegate) {
  ...
```

```
  if (options.Ok()) {
    switch (options.GetTask()) {
      ...
      case Options::Task::COMPILE:
        success = android::aidl::compile_aidl(options, io_delegate);
        break;
      case Options::Task::PREPROCESS:
        success = android::aidl::Preprocess(options, io_delegate);
        break;
      case Options::Task::DUMP_API:
        success = android::aidl::dump_api(options, io_delegate);
        break;
  ...
}
```

One can already see here that the AIDL binary has many additional features besides just compiling code, and even "compiling" in this context is already a rather complex process.

The the entry point to code generation is in the `compile_aidl` function. For every input file and defined type, the relevant backend is dispatched. Below is a fragment of how this works:

```
bool compile_aidl(const Options& options, const IoDelegate& io_delegate) {
  const Options::Language lang = options.TargetLanguage();
  ...
  for (const string& input_file : options.InputFiles()) {
    AidlError aidl_err = internals::load_and_validate_aidl(input_file, options, io_delegate,
                                                           &typenames, &imported_files);
    ...
    for (const auto& defined_type : typenames.MainDocument().DefinedTypes()) {
      ...
      if (lang == Options::Language::CPP) {
        cpp::GenerateCpp(output_file_name, options, typenames, *defined_type, io_delegate);
      } else if (lang == Options::Language::NDK) {
        ndk::GenerateNdk(output_file_name, options, typenames, *defined_type, io_delegate);
      } else if (lang == Options::Language::JAVA) {
        java::GenerateJava(output_file_name, options, typenames, *defined_type, io_delegate);
      }
      ...
    }
  }
  ...
}
```

What is important to note here is that even the NDK is simply treated as another "language" backend. This design makes it very straightforward and maintainable to integrate additional outputs (like an auto-generated fuzzer) exactly the same way as C++, Java, or NDK. The AIDL tool itself

does not care what the output is; it just hands the type information and parsed AST to the chosen generator.

It's alos important to highlight that the internal logic for parsing and validating AIDL files (`load_and_validate_aidl`) is both mature and well-tested, and we do not need to worry about it for our use-case. Our focus is entirely on code generation once the type system is resolved.

Code generation in the AIDL compiler is built around a visitor pattern. For every top-level feature in the interface (constants, methods, types), there is a dedicated `Generate<Feature>` function that emits the required code as strings into the output file.

As a practical example, the code for generating C++ constant definitions looks as follows:

```
void GenerateConstantDefinitions(CodeWriter& out, const AidlDefinedType& type,
                                 const AidlTypenames& typenames, const string& template_decl,
                                 const string& q_name) {
  for (const auto& constant : type.GetConstantDeclarations()) {
    const AidlConstantValue& value = constant->GetValue();
    if (value.GetType() != AidlConstantValue::Type::STRING) continue;

    std::string cpp_type = CppNameOf(constant->GetType(), typenames);
    out << template_decl;
    out << "const " << cpp_type << "& " << q_name << "::" << constant->GetName() << "() {\n";
    out << "  static const " << cpp_type << " value("
        << constant->ValueString(ConstantValueDecorator) << ");\n";
    out << "  return value;\n";
    out << "}\n";
  }
}
```

The logic here is simple: for each constant defined in the interface, generate a getter that returns the value as a static reference. Most of the code generation is string emission, templated for each target language and type.

If we look further we can also find the code generator for server transaction (discussed in the background chapter), which follows a similar pattern. It emits variable declarations for all method parameters, emits code for input deserialization, performs type and status checks, calls the real method, and writes results or exceptions back to the parcel. In practice, all "real" work is handled by emitting formatted C++ statements into the output buffer, following the AIDL method and argument structure.

```
void GenerateServerTransaction(CodeWriter& out, const AidlInterface& interface,
                               const AidlMethod& method, const AidlTypenames& typenames,
                               const Options& options) {
  ...
```

```
  for (const unique_ptr<AidlArgument>& a : method.GetArguments()) {
    out.Write("%s %s;\n", CppNameOf(a->GetType(), typenames).c_str(), BuildVarName(*a).c_str());
  }
  // Declare a variable to hold the return value.
  if (method.GetType().GetName() != "void") {
    out.Write("%s %s;\n", CppNameOf(method.GetType(), typenames).c_str(), kReturnVarName);
  }
  // Check that the client is calling the correct interface.
  out.Write("if (!(%s.checkInterface(this))) {\n", kDataVarName);
  out.Write("  %s = ::android::BAD_TYPE;\n", kAndroidStatusVarName);
  out.Write("  break;\n");
  out.Write("}\n");
  ...
  // Deserialize each "in" parameter to the transaction.
  for (const auto& a: method.GetArguments()) {
    // Deserialization looks roughly like:
    //     _aidl_ret_status = _aidl_data.Readt32(&in_param_name);
    //     if (_aidl_ret_status != ::android::OK) { break; }
    const string& var_name = "&" + BuildVarName(*a);
    if (a->IsIn()) {
      out.Write("%s = %s.%s(%s);\n", kAndroidStatusVarName, kDataVarName,
                ParcelReadMethodOf(a->GetType(), typenames).c_str(),
                ParcelReadCastOf(a->GetType(), typenames, var_name).c_str());
      GenerateBreakOnStatusNotOk(out);
    }
    ...
  }
  ...
  // Call the actual method.  This is implemented by the subclass.
  out.Write("%s %s(%s(%s));\n", kBinderStatusLiteral, kStatusVarName, method.GetName().c_str(),
            GenerateArgList(typenames, method, /*for_declaration=*/false, /*type_name_only=*/false)
                .c_str());
  ...
  // Write exceptions during transaction handling to parcel.
  if (!method.IsOneway()) {
    out.Write("%s = %s.writeToParcel(%s);\n", kAndroidStatusVarName, kStatusVarName, kReplyVarName);
    GenerateBreakOnStatusNotOk(out);
    out.Write("if (!%s.isOk()) {\n", kStatusVarName);
    out.Write("  break;\n");
    out.Write("}\n");
  }
  // If we have a return value, write it first.
  if (method.GetType().GetName() != "void") {
    out.Write("%s = %s->%s(%s);\n", kAndroidStatusVarName, kReplyVarName,
              ParcelWriteMethodOf(method.GetType(), typenames).c_str(),
              ParcelWriteCastOf(method.GetType(), typenames, kReturnVarName).c_str());
    GenerateBreakOnStatusNotOk(out);
  }
  ...
}
```

Overall, this design is surprisingly simple and effective, and after considering various alternatives, it became clear that integrating fuzz harness generation directly into the AIDL compiler as a backend is the most maintainable and robust approach.

4.2 Custom Language Backend for Fuzz Harness Generation

The first concrete implementation step was to add support for a new "language" to the AIDL compiler: `FUZZ_HARNESS`. This involved updating the language selection logic to dispatch to the custom harness backend:

```
...
} else if (lang == Options::Language::NDK) {
  ndk::GenerateNdk(output_file_name, options, typenames, *defined_type, io_delegate);
} else if (lang == Options::Language::FUZZ_HARNESS) {
  harness::GenerateFuzzHarness(
      output_file_name, options, typenames, *defined_type, io_delegate);
}
...
```

A first, minimal implementation for the backend might simply emit a hello world:

```
bool GenerateFuzzHarness(const std::string &file, const Options &,
                         const AidlTypenames &types, const AidlDefinedType &def,
                         const IoDelegate &io) {
  auto cw = io.GetCodeWriter(file);
  if (!cw) {
    AIDL_ERROR(def) << "Failed to open " << file;
    return false;
  }

  cw.Write("Hello World!");
  return true;
}
```

4.3 Feature Set and Supported Data Types

AIDL is a very expressive language with numerous features. Early in the design process, it was necessary to decide which subset of features would be supported in the initial version of the harness generator. The following types and features are supported:

- All native fixed-length datatypes: `int`, `long`, `char`, `enum`, etc.

- Strings
- Arrays and lists (with variable length, and randomized per fuzzing pass)
- Binders and file descriptors (these receive special treatment, discussed in a later subsection)

The feature set has gone through several iterations, but these categories cover the vast majority of real-world AIDL usage and are sufficient for evaluating the approach.

4.4 Fuzzer Harness Structure and Generation

With the extension point in place, the next step was to design the actual structure of the generated fuzz harness. One concrete insight from the design chapter was to generate, for each transaction, a dedicated mutator function, and to then modularize the emission of different code components (load, mutate, store, initialize) into buffers that are then composed into the output file.

The overall skeleton for the generated fuzzer harness is as follows:

```
bool GenHarness(android::aidl::CodeWriter &out, const Aidlterface &iface,
                const AidlTypenames &types) {
  out << R"(
#include <binder/IBinder.h>
#include <binder/Binder.h>
#include <binder/Parcel.h>
#include <cstring>
...
  )";

  // selection of global constants
  const size_t txnCount = iface.GetMethods().size();
  out.Write("const int TXN_COUNT        = %zu;\n", txnCount);
  out.Write("const int SCRATCH_SIZE     = %zu;\n", kDefaultScratchSize);

  out << "extern \"C\" size_t LLVMFuzzerMutate(uint8_t*,size_t,size_t);\n\n";
  ...
  out << R"(extern int LLVMFuzzerTestOneInput(
      android::sp<android::IBinder> service, const uint8_t* data,size_t size){
  if(size<8) return 0;
  android::Parcel dataParcel;
  dataParcel.setEnforceNoDataAvail(false);
  dataParcel.setServiceFuzzing();

  if(service) dataParcel.writeInterfaceToken(service->getInterfaceDescriptor());
  ...
  android::Parcel reply;
```

```
  reply.setEnforceNoDataAvail(false);
  reply.setServiceFuzzing();

  service->transact(txn,dataParcel,&reply,flags);
  return 0;
}
)";
  out << txnHelpers << "\n";
  out << R"(extern "C" size_t LLVMFuzzerCustomMutator(
      uint8_t* Data,size_t Size,size_t MaxSize,unsigned int Seed){
  if(Size<8){
    if(MaxSize<8) return 0;
    uint32_t txn = android::IBinder::FIRST_CALL_TRANSACTION + (Seed+53391899)%TXN_COUNT;
    memcpy(Data,&txn,4);
    uint32_t flags=0; memcpy(Data+4,&flags,4);
    return 8;
  }
  LLVMFuzzerMutate(Data+4,sizeof(uint32_t),sizeof(uint32_t));
  uint32_t mid=*reinterpret_cast<uint32_t*>(Data)-android::IBinder::FIRST_CALL_TRANSACTION;
  switch(mid){
)";

  for (size_t i = 0; i < txnCount; ++i)
    out.Write("    case %zu: return 8 + "
              "AIDLFuzzerMutateTxn%zu(Data+8,Size-8,MaxSize-8,Seed);\n",
              i, i + android::IBinder::FIRST_CALL_TRANSACTION);

  out << R"(    default: return Size;
  }
}
)";
  return true;
}
```

This approach organizes the code emission into several buffers, one for each logical component (e.g., transaction helpers), which are then assembled at the end.

4.5 Primitive Data Types

With the scaffolding in place, the next step is generating code for supporting primitives data types. Here, the previously discussed SSA (static single assignment) style is applied. Here, we depart from a simple one-pass, one-buffer approach used for the C++ backend, and instead use distinct code buffers for loading, mutating, storing, and initializing data for each transaction. This is realized by visiting each transaction's arguments and emitting the necessary code into these buffers, which are later combined.

```
bool GenTxn(android::aidl::CodeWriter &out, size_t idx,
            const std::unique_ptr<AidlMethod> &method,
            const AidlTypenames &types, std::string &globals, DynCtx &ctx) {
  out.Write("size_t AIDLFuzzerMutateTxn%zu(uint8_t* Data, size_t Size, "
            "size_t MaxSize, unsigned int /* Seed */){\n",
            idx);
  ...
  std::string L, M, S, I; // LOAD; MUTATE; STORE; INIT
  ...
  for (const auto &m : method->GetArguments()) {
    // essentially a large switch statement
    EmitType(m->GetName(), m->GetType(), idx, min, globals, L, M, S, I, types, ok);
  }
  ...
  out << "// load\n"
      << L << "// mutate\n"
      << M << "// store\n"
      << S
      << "if (out.dataSize() > MaxSize) return 0;\n"
         "memcpy(Data, out.data(), out.dataSize());\n"
         "return out.dataSize();\n";
  ...
  return ok;
}
```

A macro is used for native data types:

```
#define FIXED_PARCEL(Cpp, Name, Read, Write)                                  \
  int EmitFuzz##Name(std::string &, std::string &L, std::string &M,           \
                     std::string &S, std::string &I, const std::string &n,    \
                     size_t &min) {                                           \
    Fmt(L, #Cpp " %s;", n.c_str());                                           \
    Fmt(L, "in.read%s(&%s);", #Read, n.c_str());                              \
    Fmt(M,                                                                    \
        "LLVMFuzzerMutate((uint8_t*)&%s,sizeof(" #Cpp "),sizeof(" #Cpp "));", \
        n.c_str());                                                           \
    Fmt(S, "out.write%s(%s);", #Write, n.c_str());                            \
    Fmt(I, "out.write%s((" #Cpp ")0);", #Write);                              \
    min += sizeof(Cpp);                                                       \
    return sizeof(Cpp);                                                       \
  }

FIXED_PARCEL(int8_t, Byte, Byte, Byte)
FIXED_PARCEL(int32_t, Int, Int32, Int32)
FIXED_PARCEL(int64_t, Long, Int64, Int64)
FIXED_PARCEL(float, Float, Float, Float)
FIXED_PARCEL(double, Double, Double, Double)
FIXED_PARCEL(bool, Bool, Bool, Bool)
#undef FIXED_PARCEL
```

For example, the autogenerated transaction mutator for the previously discussed `int32_t sum(int32_t x, int32_t y)` would look like:

```
size_t AIDLFuzzerMutateTxn2(uint8_t* Data, size_t Size, size_t MaxSize){
  android::Parcel in, out;
  in.setData(reinterpret_cast<const uint8_t*>(Data), Size);
  if (Size < 8) {
    out.writeInt32((int32_t)0);
    out.writeInt32((int32_t)0);
    if (out.dataSize() > MaxSize) return 0;
    memcpy(Data, out.data(), out.dataSize());
    return out.dataSize();
  }
  // load
  int32_t x;
  in.readt32(&x);
  int32_t y;
  in.readt32(&y);
  // mutate
  LLVMFuzzerMutate((uint8_t*)&x,sizeof(int32_t),sizeof(int32_t));
  LLVMFuzzerMutate((uint8_t*)&y,sizeof(int32_t),sizeof(int32_t));
  // store
  out.writeInt32(x);
  out.writeInt32(y);
  if (out.dataSize() > MaxSize) return 0;
  memcpy(Data, out.data(), out.dataSize());
  return out.dataSize();
}
```

One key design decision was to avoid performing mutations directly in the seed buffer itself. While the initial implementation tried an in-place approach - mutating values within the buffer passed to the mutator, and resorting to `memcpy` when needed - it quickly became clear that Android Binder parcels are more complicated than they first appear. One of the main problems is alignment: all fields in a binder parcel must be 32-bit aligned, even if the type itself (like a char or a bool) is smaller. In addition, the parcel format internally handles some types in special ways, sometimes reinterpreting or casting values during (de)serialization. Attempting to replicate this logic by hand would be error-prone and very likely to break if any detail changed in the Android framework.

Because of this, the implementation switched to using the `android::Parcel` API for both loading and storing data - even for primitive types. This means that, instead of directly mutating bytes in the input, we always deserialize from the buffer into local variables using `Parcel` methods, mutate those variables, and then serialize them back into a new parcel before copying the bytes out. The downside is a minor performance hit due to extra copying, but the advantages are clear: the implementation is correct, robust against changes in the parcel format, and - critically - can also handle more tricky types (like binder objects), which simply cannot be safely flattened/unflattened without the official APIs.

4.6 Variable-Length Data Types

Next, we implement simple variable-length data types: `String16`, as well as arrays and vectors. In the Android binder parcel format, such types are encoded as a 32-bit length field, followed by the actual data. For example, a string is stored as `[int32_t length] [char16_t data...]` and vectors follow a similar pattern. Although one could attempt to manually implement this encoding and decoding, the `android::Parcel` API provides built-in methods that handle these types consistently and correctly, including proper alignment and error checking. For these reasons, the generated harness uses the API for both reading and writing, as this avoids bugs and ensures forward compatibility.

For example, the code generator function for strings looks like the following:

```
int EmitFuzzString(std::string & /*g*/, std::string &L, std::string &M,
                   std::string &S, std::string &I, const std::string &n,
                   size_t &min) {
  Fmt(L, "android::String16 %s;", n.c_str());
  Fmt(L, "in.readString16(&%s);", n.c_str());
  Fmt(M, "FuzzString16(%s);", n.c_str());
  Fmt(S, "out.writeString16(%s);", n.c_str());
  Raw(I, "out.writeString16(android::String16());");
  min += 4;
  return 0;
}
```

This pattern is almost identical for arrays and vectors: always read, mutate, and write using the parcel API, swapping in the correct type and method names. All actual mutation logic is pushed into a helper, so the generator just emits calls.

When traversing the AIDL types, the generator continually checks for any variable-length data types. When encountering such a type, the corresponding helper function is emitted into the `globals` section of the harness, ensuring that each mutation helper appears only once per output file.

The helper for mutating `String16` values is shown below:

```
thread_local uint8_t gScratch[SCRATCH_SIZE];

static void FuzzString16(android::String16& s) {
    size_t bytes = s.size() * sizeof(char16_t);
    ...
    memcpy(gScratch, s.c_str(), bytes);
    ...
    size_t nb = LLVMFuzzerMutate(gScratch, bytes, SCRATCH_SIZE);
```

```
    s = android::String16(reinterpret_cast<const char16_t*>(gScratch),
                          nb / sizeof(char16_t));
}
```

Arrays and vectors use basically the same pattern, treating them just as blobs of data after length extraction.

4.7 Parcelables

Handling `parcelable` types is more involved. Unlike primitive or variable-length types, there is no direct API to read or write a parcelable structure as a single operation (despite the name). Instead, its the "other way around": By comforming to the `Parcelable` protocol, one guarantees that the object implements specific serialization functions, and usually the AIDL compiler generates these code for you. Parcelables are defined in AIDL as their own types, with a list of fields:

When processed by the AIDL compiler, each parcelable results in generated methods such as `readFromParcel()` and `writeToParcel()` which implement field-by-field (de)serialization. The typical structure of such a generated method is as follows:

```
::android::status_t MyStruct::writeToParcel(::android::Parcel* _aidl_parcel) const {
  _aidl_parcel->writeInt32(0); // placeholder for size
  ...
  _aidl_ret_status = _aidl_parcel->writeInt32(data);
  if (_aidl_ret_status != ::android::OK) { return _aidl_ret_status; }
  _aidl_ret_status = _aidl_parcel->writeFloat(majorVersion);
  if (_aidl_ret_status != ::android::OK) { return _aidl_ret_status; }
  _aidl_ret_status = _aidl_parcel->writeString16(greatString);
  if (_aidl_ret_status != ::android::OK) { return _aidl_ret_status; }
  ...
  size_t _aidl_end_pos = _aidl_parcel->dataPosition();
  _aidl_parcel->setDataPosition(_aidl_start_pos);
  _aidl_parcel->writeInt32(static_cast<int32_t>(_aidl_end_pos - _aidl_start_pos));
  _aidl_parcel->setDataPosition(_aidl_end_pos);
  return _aidl_ret_status;
}
```

An important observation is that each parcelable starts with a size prefix - the total number of bytes occupied by the structure in the parcel. To correctly serialize a parcelable, one must write a placeholder for the size, serialize all fields, compute the resulting size, and patch the placeholder value. This is exactly the process followed by the AIDL compiler, and the same logic must be implemented in the fuzzer harness.

However, reusing the C++ code generator for parcelables is not really practical in our context. First, since we are emitting C++ code from our custom "language", we would need to call into the original code generation, which is wrapped in a lot of other logic. More importantly, to be able to handle special cases (e.g., binders, file descriptors) during serialization, it is more maintainable to directly reimplement parcelable (de)serialization within the harness generator.

This results in the following logic for emitting code for parcelable types:

```
AidlTypeSpecifier &t, size_t txn,
              size_t &min, std::string &globals, std::string &L, std::string &M,
              std::string &S, std::string &I, const AidlTypenames &types,
              bool &ok) {
    ...
    // size placeholder
    Fmt(L, "int32_t %s_size;", name.c_str());
    Fmt(L, "in.readt32(&%s_size);", name.c_str());
    Fmt(S, "size_t %s_start_pos = out.dataPosition();", name.c_str());
    Raw(S, "out.writeInt32(0); // size");
    Fmt(I, "size_t %s_start_pos = out.dataPosition();", name.c_str());
    Raw(I, "out.writeInt32(0); // size");
    min += sizeof(int32_t);
    for (const auto &f : types.GetParcelable(t)->GetFields())
      EmitType(name + "_" + f->GetName(), f->GetType(), txn, min, globals, L, M,
               S, I, types, ok);
    // patch size
    Fmt(S, "size_t %s_end_pos = out.dataPosition();", name.c_str());
    Fmt(S, "out.setDataPosition(%s_start_pos);", name.c_str());
    Fmt(S, "out.writeInt32(static_cast<int32_t>(%s_end_pos - %s_start_pos));",
        name.c_str(), name.c_str());
    Fmt(S, "out.setDataPosition(%s_end_pos);", name.c_str());
    // same for init
    ...
}
```

The process works similar as in the previously discussed C++ backend: a placeholder for the size is written, all fields are serialized recursively, and the final size is patched in at the correct offset. This approach allows parcelables to contain other complex types, including other parcelables, binders, or file descriptors, which will be discussed in the next subsection.

4.8 Binders and File Descriptors

4.8.1 Challenges of Dynamic Data Types

For the final set of features, we wanted to support binders, and then file descriptors (FDs), which are conceptually very similar, at least from the perspective of resource management and parcel serialization. However, supporting dynamic types proved considerably more complex than expected. To clarify why, it helps to revisit some core properties of Android binder objects:

- The binder driver in the kernel maintains a reference count for every binder passed as an argument, ensuring the object remains alive as long as references exist.
- In order to allow said binder driver to keep track of ownership, it expects the objects to be flattened in a very particular way by the `writeStrongBinder()` method. Reimplementing this logic in a stable way is almost impossible.
- Transactions on binder objects are dispatched via the `onTransact` method; the callee is expected to reply with a valid parcel (at minimum, `Android::OK`).
- Every binder carries an interface descriptor (e.g., `"com.example.myservice"`), which is commonly checked before any method invocation.

What this means in practice: If we want to pass a binder as an argument during fuzzing, we *have* to actually instantiate a binder object - we cannot just write random fuzzing data and hope for the best. Additionally, we must carefully manage the lifecycle of this object - neglecting proper management quickly leads to memory leaks and can exhaust system resources (not hypothetical: this will crash your fuzzer).

The next problem: where do you create these binders? A naive approach might be to instantiate binders at random during mutation. However, this is fundamentally at odds with reliable fuzzing. If a crash is found, it must be reproducible with just the seed input - any “out-of-band” binder instantiation during mutation breaks this, making it impossible to triage or minimize issues.

Further, there’s an ordering problem: The binder object must be instanciated when the argument is written into the parcel (so during mutation), but the actual instanciation can only really happen during `LLVMFuzzerTestOneInput`, where we typically have only the “flattened” blob and not the objects themselves.

The same applies to file descriptors, which require a real file to back them, correct lifecycle management and proper handling during serialization.

In summary, we need a mechanism that lets us:

- Encode the creation and configuration of dynamic objects directly in the seed,
- Mutate and manipulate these objects during fuzzing,
- Patch their handles into the argument parcel at the correct positions,
- Track all resources and avoid leaks or crashes.

4.8.2 Dynamic Seed Section

To address this, we extend the fuzzing input with a dynamic section - a region at the beginning of the seed encoding all dynamic handles (binders and FDs) before the main argument parcel. For every dynamic object required by the transaction, we encode its configuration in this section. The layout looks like this:

```
  int64_t txncode | int64_t flags | int32_t dyn_pool (eg. 1) | String16
itf_desc1 | uint8[] dyn_data1 | ... | int32_t dyn_terminator (always 0)|
                          uint8[] parcel_data
```

For every dynamic argument present in a service method, we maintain a pool of reusable FuzzHandlers - so, for each argument of dynamic type (e.g., binder or file descriptor), there is a dedicated pool of `FuzzBinder` or `FuzzFd` instances. During `LLVMFuzzerTestOneInput`, the harness first parses and maps all dynamic objects into their corresponding pool. As the dynamic section is parsed, the current object for that pool is configured with its encoded descriptor and fuzzed data, and a reference is pushed onto a queue in the order arguments are expected for the current transaction. The fuzzer will later consume these in order as it patches them into the argument parcel.

```
static android::sp<android::FuzzBinder> pool_some_binder[pool_some_binder_SIZE] = {
  android::sp<android::FuzzBinder>(new android::FuzzBinder()),
  android::sp<android::FuzzBinder>(new android::FuzzBinder()),
  ...
};
static int pool_some_binder_idx = 0;

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data,size_t size){
  if(size<8) return 0;
  // read txn header
  android::Parcel cur; cur.setData(data,size); cur.setDataPosition(0);
  uint32_t txn=0, flags=0; cur.readUint32(&txn); cur.readUint32(&flags);
  // dynamic records
  std::deque<android::sp<IFuzzHandle>> dyn_queue;
  while(true){
```

```
    int32_t pool=0; cur.readt32(&pool);
    if(pool==DYN_TERMINATOR) break;
    android::String16 name;        cur.readString16(&name);
    std::vector<uint8_t> blob;     cur.readByteVector(&blob);
    switch(pool){
      case 1:{
      auto& obj=pool_some_binder[ pool_some_binder_idx++ % pool_some_binder_SIZE ];
      obj->configure(name,std::move(blob));
      dyn_queue.push_back(obj);
      }break;
      ...
    }
  }
  ...
}
```

Correspondlingly for the custom mutator, we perform a pre-pass over all arguments and emit the following for every dynamic-typed one: A non-fuzzed int32_t (the pool index), a fuzzed String16 (for the interface descriptor), a fuzzed byte array (serving as seed data for behavior / file content), and finally the required `DYN_TERMINATOR` after the full dynamic secton. All logic for creating and managing these objects is therefore reproducible, encoded entirely within the fuzzing seed.

The actual implementation if these `FuzzHandle` objects is deliberately simple. For binder arguments, the implementation uses a lightweight FuzzBinder class, inheriting from BBinder and implementing the IFuzzHandle interface. The interesting part is the onTransact() method: Instead of trying to synthesize protocol-specific responses, the default implementation just calls `fillRandomParcel()` on the reply, reusing the logic from the original binder parcel fuzzer. This is mostly good enough, since for most binder-based services, the replies are not really validated in depth.

```
cclass IFuzzHandle : public virtual RefBase {
public:
    virtual void configure(const String16&, std::vector<uint8_t>&&) = 0;
    virtual void writeToParcel(Parcel&) const                       = 0;
};

class FuzzBinder : public BBinder, public IFuzzHandle {
public:
    ...
    void configure(const String16& desc,
                   std::vector<uint8_t>&& bytes) override {
        mDescriptor = desc;
        mBytes      = std::move(bytes);
        ...
    }
    void writeToParcel(Parcel& p) const override {
```

```
        p.writeStrongBinder(const_cast<FuzzBinder*>(this));
    }
    const String16& getInterfaceDescriptor() const override {
        return mDescriptor;
    }
    status_t onTransact(uint32_t, const Parcel&, Parcel* reply,
                        uint32_t) override {
        ...
        fillRandomParcel(
            reply,
            FuzzedDataProvider(sub.data(), sub.size()),
            &opts);
        return OK;
    }
    ...
};
```

A similar pattern is used for file descriptor arguments. The FuzzFd class maintains a memfd for in-memory file handling. Upon each configuration, it rewinds the file and writes the fuzz data into it. As with binders, only a fixed-length pool of reusable objects per argument position is kept alive, which ensures resource usage remains bounded.

```
class FuzzFd : public IFuzzHandle {
public:
    ...
    void configure(const String16& /*unused*/, std::vector<uint8_t>&& bytes) override {
        ...
        if (mFd.get() == -1) {
            int fd = memfd_create("fuzz_fd", MFD_CLOEXEC);
            mFd.reset(fd);
        ...
        }
        ...
        android::base::WriteFully(mFd.get(), bytes.data(), bytes.size());
    }

    void writeToParcel(Parcel& p) const override {
        p.writeFileDescriptor(mFd.get(), false);
    }
...
};
```

As with the helpers for fuzzing Strings and Vectors, these classes are only emitted into the harness if a corresponding dynamic type is found in the service interface.

4.8.3 Parcel Patching

After parsing and configuring all dynamic arguments, we have a queue of handles (e.g., `FuzzBinder`, `FuzzFd`) in the order they are meant to appear in the parcel. The challenge now is: how do we actually write these handles into the correct locations inside the transaction parcel, given that the precise offset of each handle is only known during mutation, but the object can only be constructed and inserted at runtime during `LLVMFuzzerTestOneInput`?

The Binder serialization logic is highly stateful and tightly bound to memory addresses - the `writeStrongBinder` call, for instance, produces a `flat_binder_object` structure that directly embeds the pointer of the binder instance being written. This means that the only correct way to serialize such handles is to call the real API with a live object, at exactly the right spot in the parcel data buffer.

The obvious problem: During mutation, in the transaction helper, we do not have access to the actual handles. All we have is the raw seed buffer, which is mutated independently of the dynamic object pool used during input execution. Conversely, in `LLVMFuzzerTestOneInput`, we have access to the pool of handles, but the only thing we receive is the fully flattened seed parcel, with no explicit structure or annotations indicating where to insert the objects.

The solution is to use explicit "placeholders" during mutation: Instead of trying to encode a live object, we simply reserve space in the parcel for the object by writing a unique `DYN_MAGIC_VALUE` (an `int64_t`) at each required location. The precise byte size of each placeholder is chosen to match the real object's serialized size. For binders, this is `sizeof(flat_binder_object) + sizeof(int32_t)`, for file descriptors it is `sizeof(int32_t) + sizeof(int32_t)`. These sizes are always constant (at least for a concrete target platform), allowing us to later patch the parcel in-place.

To identify these locations during execution, we search the parcel buffer for occurrences of `DYN_MAGIC_VALUE`. Collisions with real input bytes are vanishingly rare, but as a quick "napkin calculation": Suppose we have a method where the argument seed size is a very large 100,000 bytes, and the fuzzer runs at 100,000 executions per second. The probability that a random 8-byte window in the seed coincides with a given 64-bit magic value is 2^{-64}. There are roughly 100,000 possible offsets per input, so in one execution the probability of a collision is $100{,}000 \times 2^{-64} \approx 5.4 \times 10^{-15}$. At 100,000 executions per second, that is about 5.4×10^{-10} collisions per second, or roughly one collision every 60 years of continuous fuzzing. Thus, if a collision is ever detected, simply discarding that input is entirely safe in practice.

As example, the following are excerpt of the store section of an auto-generated transaction helper. This is for a method with a binder argument:

```
// dyn section 1
out.writeInt32(dyn1_pool);
out.writeString16(dyn1_name);
```

```
out.writeByteVector(dyn1_data);
out.writeInt32(DYN_TERMINATOR);
// all args before binder
...
flat_binder_object some_binder_obj;
*reinterpret_cast<uint64_t *>(&some_binder_obj) = DYN_MAGIC_VALUE;
// write placeholder
out.writeObject(some_binder_obj, false); // writeObject has no side effects
out.writeInt32(0); // from finishFlattenBinder
```

And, for a method with a file descriptor argument:

```
// dyn section 2
out.writeInt32(dyn2_pool);
out.writeString16(dyn2_name);
out.writeByteVector(dyn2_data);
out.writeInt32(DYN_TERMINATOR);
// all args before fd
...
// write placeholder
out.writeInt64(DYN_MAGIC_VALUE);
```

Within the `LLVMFuzzerTestOneInput`, after deserializing the dynamic section and configuring all handle pools, we scan the serialized parcel buffer for each `DYN_MAGIC_VALUE`. For every match, we pop the next configured handle from the dynamic queue, seek to the exact offset in the parcel, and invoke `writeToParcel` for the corresponding object - thereby patching the real handle into the parcel in-place. This is accomplished with a simple loop:

```
for(size_t off=0; off+sizeof(uint64_t)<=argParcelSize; ){
  const uint64_t* v = reinterpret_cast<const uint64_t*>(arg+off);
  if(*v!=DYN_MAGIC_VALUE){ ++off; continue; }
  ...
  // patch the magic value
  dataParcel.setDataPosition(payload_start + off);
  auto handle = dyn_queue.front(); dyn_queue.pop_front();
  handle->writeToParcel(dataParcel);
  // continue after the newly written data
  off = dataParcel.dataPosition() - payload_start;
}
// if too many handles supplied (likely a collision), reject seed
if(!dyn_queue.empty()) return -1;
```

After this surgical precision work, we can finally just call `transact` on the service with our parcel, and find that the service can successfully receive binder objects and file descriptors.

While this approach seems cumbersome and arguably overcomplicated at first, it has a few clear benefits: It guarantees that binders or file descriptors passed into the service are always valid,

and – just as importantly – that every fuzzing seed is fully replayable, making debugging and triage practical.

Chapter 5

Evaluation

During development, the autogen fuzzer was continually tested against a dedicated demonstration service, `IHelloService`, which was based on the `com.yuandaima.IHelloService` Demo Service [8] and extended as new features were implemented in the fuzzer generator. This iterative approach ensured that every newly added feature could be directly validated in a controlled environment. The service itself is stateless and simply prints the method arguments it receives, making it straightforward to determine whether method invocations and argument decoding succeeded.

The final `IHelloService` interface and its related AIDL files are shown below. These encompass a range of argument types, including primitive values, parcelables, binders, file descriptors, and arrays:

5.0.1 Integration in the Build System

Building and running the fuzzer within the AOSP build system required only minor adjustments to existing workflows. First, the AOSP environment is set up, then the AIDL compiler is built:

```
cd ~/AOSP
source build/envsetup.sh
lunch aosp_cf_x86_64_phone-trunk_staging-eng
cd ~/AOSP/system/tools/aidl
mm -j64
```

Generating the fuzz harness involves invoking the AIDL tool with the custom language:

```
cd ~/AOSP/AIDLCppDemo/aidl
aidl -I . --lang fuzz-harness com/yuandaima/IHello.aidl --out ../gen --header_out ../gen/include/
```

The generated fuzzer can then be compiled and integrated much like an ordinary binder parcel fuzzer, using a simple entrypoint. For example:

```
#include <fuzzbinder/libbinder_driver.h>
#include "HelloService.h"
using android::fuzzServiceAutogen;
using android::sp;
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    auto binder = sp<HelloService>::make();
    fuzzServiceAutogen(binder, data, size);
    return 0;
}
```

In the build system, this is captured with a `cc_fuzz` target:

```
cc_fuzz {
    name: "AutogenFuzzer",
    defaults: [
        "service_fuzzer_defaults",
    ],
    ...
    srcs: ["src/aidl_autogenerated.cpp", "src/FuzzHelloService.cpp"],
}
```

5.1 Experimental Setup

5.1.1 Services Under Test

For comprehensive evaluation, the autogen fuzzer was tested not only on the demonstration service but also on a set of six representative system services from the Android Open Source Project. Each of these services features a unique combination of interface complexity, data types, and logical position within the overall system:

- `android/net/IDnsResolver` – Service responsible for DNS resolution and network configuration, with a wide variety of primitive and structured arguments, as well as frequent asynchronous callbacks.
- `android/hardware/nfc/INfc` – The main NFC stack interface, dealing with device management and state transitions; has binders and parcelables.
- `android/os/IStatsd` – Androids telemetry and statistics daemon, which receives complex event structures, often with nested parcelables and binder callbacks.

- `android/hardware/threadnetwork/IThreadChip` – Interface for low-level Thread networking used in IoT and smart home environments. Other than one callback, more or less simple interface.
- `android/os/IInstalld` – Handles APK installation; very complex interface with many methods and arguments.
- `android/hardware/automotive/occupant_awareness/IOccupantAwareness.cpp` – Automotive occupant detection service, including parcelables and binders.

To ensure a fair comparison, a key feature - "refuzzing" of returned binder objects - was explicitly disabled in the reference binder parcel fuzzer, since the autogen fuzzer does not yet implement this.

5.1.2 Instrumentation and Metrics

Both fuzzers were instrumented to track successful calls by logging every transaction and recording the number of times a transaction returned an OK status code:

```
static long transactions = 0;
static long ok = 0;
::android::status_t _aidl_ret_status = target->transact(code, data, &reply, flags);
transactions++;
if (_aidl_ret_status == ::android::OK) {
    ok++;
}
if (transactions % 1000 == 0) {
    ALOGI("Binder transactions: %ld, OK: %ld, Ratio: %.2f%%\n", transactions, ok,
          (static_cast<double>(ok) / transactions) * 100);
}
```

In addition, the built-in libFuzzer coverage statistic (COV line) was parsed to provide a simple code coverage metric, timestamped for analysis.

5.1.3 Campaign Execution

For each of the eight services (the six listed above plus the demonstration `IHelloService`), fuzzing campaigns were executed for 12 hours per service and fuzzer. The campaign was run on a set of parallel-spawned Android emulators, coordinated by a simple automation script. Logcat was monitored for keeping track of the ratio of successful return codes.

Table 5.1: Final successful-call ratio and basic-block coverage after 12 h of fuzzing. Best result per metric **bold**.

Service	OK-ratio [%] ↑		Coverage ↑	
	Autogen	B-Parcel	Autogen	B-Parcel
HelloService	**99.05**	78.26	675	**2060**
IDnsResolver	**97.19**	40.19	2948	**3232**
installd	**100.0**	56.95	979	**4071**
NFC	**100.0**	41.26	678	**1884**
OccupantAwareness	**58.21**	51.97	1875	**1947**
ThreadChip	**100.0**	53.09	720	**1981**
statsd	**84.04**	13.95	1317	**1341**

5.2 Results

5.2.1 Service Coverage and Call Success Rate

A key observation is that the autogen fuzzer is immediately able to exercise valid interface calls, as it has direct access to the AIDL structure. As a result, the percentage of calls that successfully complete (i.e., return an OK code) is very high - around 99% for many and above 85% for most services. In contrast, the reference binder parcel fuzzer, depending on the complexity of the service, achieves between 20% and 60% successful calls.

Coverage, however, tells a different story. While the autogen fuzzer starts strong, reaching new code rapidly due to its structured input, it is quickly overtaken by the binder parcel fuzzer in all evaluated services.

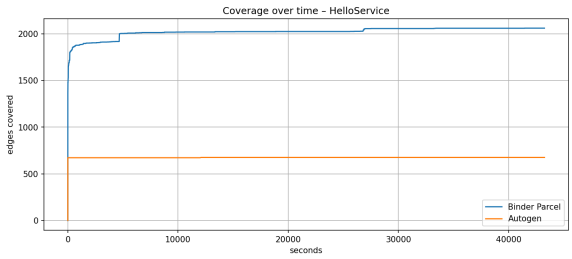
For some services, such as `INfc`, both fuzzers eventually “stall” (i.e., make no further progress in coverage). In the case of the autogen fuzzer, this plateau is reached almost immediately, likely due to external dependencies such as user interaction.

For `statsd`, a storage issue on the fuzzing host caused partial loss of logcat output, so the final OK-ratio for the binder parcel fuzzer is marked as n/a in the below table and the corresponding graph is omitted.

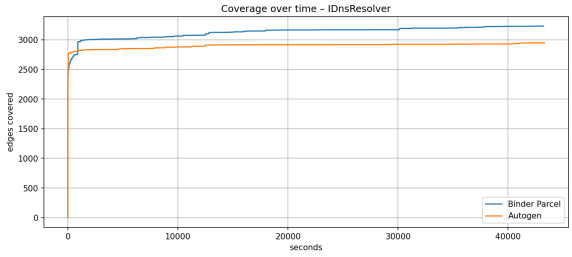
5.2.2 Interpretation of Results

Coverage

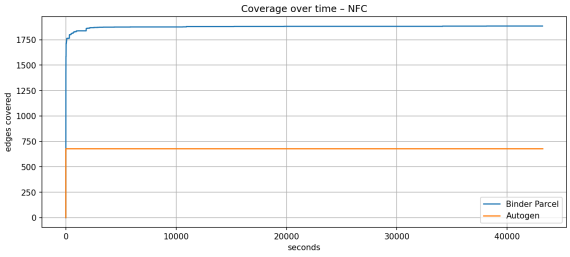
At first glance, the results are surprising: the “worse” (randomized) binder parcel fuzzer ultimately outperforms the autogen fuzzer in terms of coverage. Several factors may contribute to this:



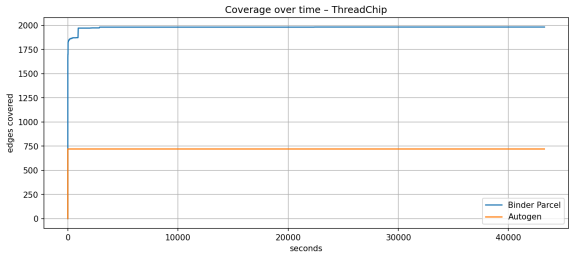
(a) HelloService



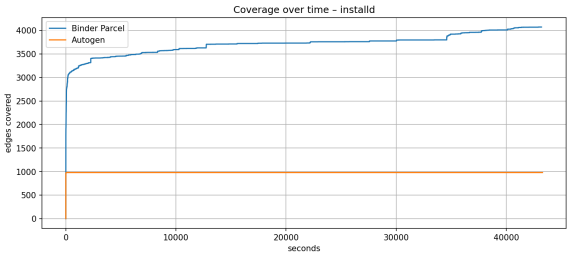
(b) IDnsResolver



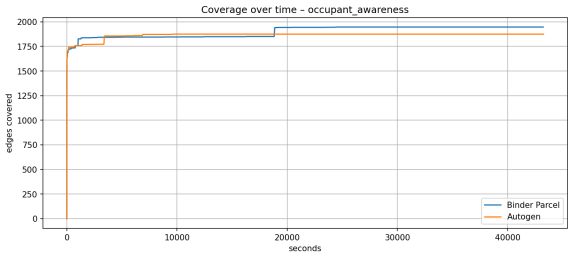
(c) NFC



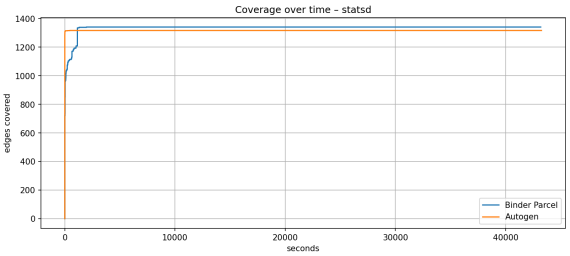
(d) ThreadChip



(e) installd

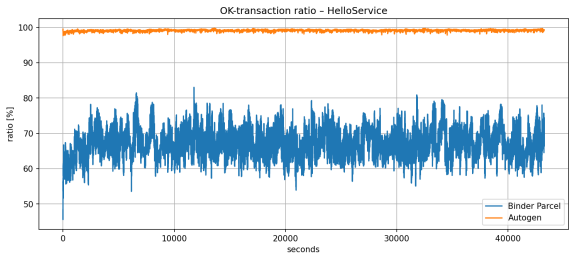


(f) OccupantAwareness

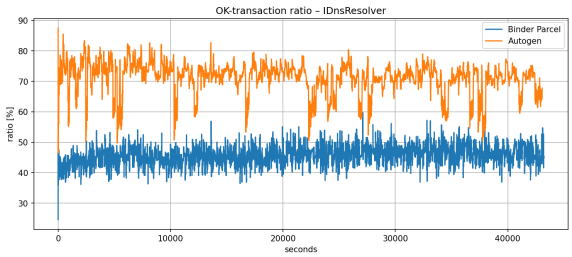


(g) statsd

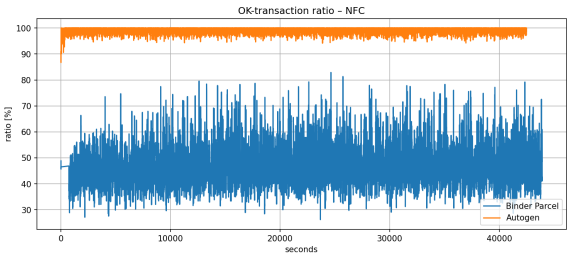
Figure 5.1: Basic-block coverage over 12 h: Autogenerated Harness (○) vs. Binder-Parcel baseline (○). Absolute coverage is still modest for the autogen fuzzer, suggesting head-room for smarter mutators in future work.



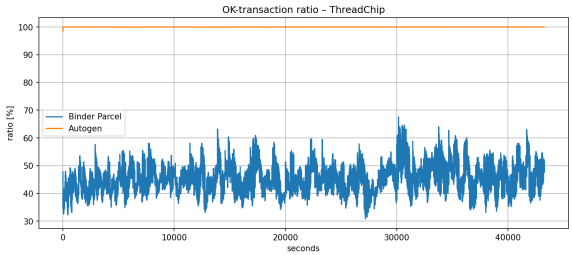
(a) HelloService



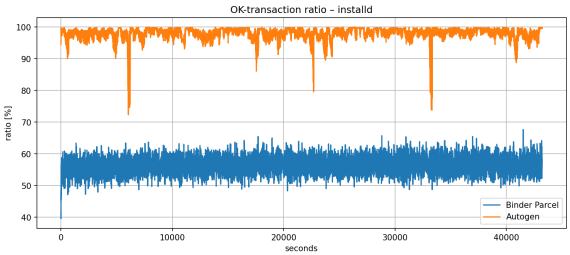
(b) IDnsResolver



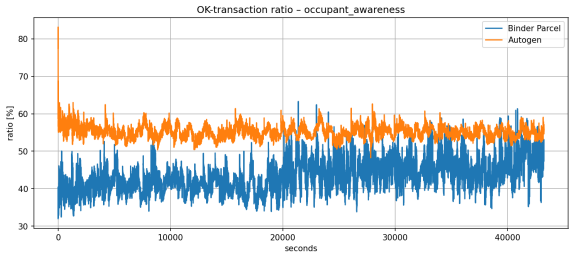
(c) NFC



(d) ThreadChip



(e) installd



(f) OccupantAwareness

Figure 5.2: Fraction of transactions that returned `OK`. The autogen harness consistently outperforms the binder-parcel baseline across all services, often achieving near-perfect ratios (99%) - except for `IDnsResolver` and `OccupantAwareness`, where lower values may indicate implementation bugs or missed interface details, highlighting room for improvement in the autogenerated code.

Table 5.2: OK-ratio [%] progression

	0s	1s	5s	10s	60s	3600s	Final
HelloService							
Binder Parcel	48.3	49.95	54.93	58.48	63.35	61.45	**78.26**
Autogen	99.1	99.74	98.89	98.39	97.85	99.5	**99.05**
IDnsResolver							
Binder Parcel	24.6	50.48	39.53	44.57	39.61	45.22	**44.42**
Autogen	87.5	50.0	83.05	68.5	67.8	71.01	**70.4**
installd							
Binder Parcel	41.4	39.3	48.82	47.97	53.88	55.81	**56.95**
Autogen	94.48	99.45	99.73	99.74	96.5	99.49	**100.0**
NFC							
Binder Parcel	48.82	44.03	44.03	44.03	44.03	47.41	**41.26**
Autogen	90.0	78.0	78.0	100.0	100.0	100.0	**100.0**
OccupantAwareness							
Binder Parcel	32.0	39.51	34.67	31.61	41.67	39.95	**51.97**
Autogen	77.5	67.64	71.2	65.48	57.47	52.2	**58.21**
ThreadChip							
Binder Parcel	47.8	34.7	33.64	35.05	29.76	40.78	**53.09**
Autogen	98.5	99.31	99.98	100.0	100.0	100.0	**100.0**
statsd							
Binder Parcel	0.0	0.0	0.0	0.0	6.67	13.95	n/a
Autogen	92.8	89.7	89.7	81.19	86.93	87.57	**84.04**

- The binder parcel fuzzer, through its randomness, occasionally triggers rare or error-handling code paths, increasing its measured coverage - even though many of its calls fail.
- The autogen fuzzer, by faithfully constructing only valid inputs, may inadvertently "overfit" to the expected use case, avoiding malformed or edge-case arguments that reveal more code.
- Another significant limitation at present is the incomplete support for some complex features - particularly lists of binders and file descriptors with length > 1. The current pool-based instantiation strategy can only create one binder per argument per transaction, limiting the fuzzer's ability to fully exercise interfaces with variable-length binder or FD arrays.
- Currently, the mutator applies `LLVMFuzzerMutate` to every argument on every mutation run, which can result in excessive or unfocused mutations. A more advanced strategy - such as per-argument mutation scheduling or only fuzzing a subset of fields per iteration - could improve both stability and coverage.

Table 5.3: Edge coverage progression

	0s	1s	5s	10s	60s	3600s	Final
HelloService							
Binder Parcel	0	1250	1386	1467	1659	1912	**2060**
Autogen	0	651	672	672	672	672	**675**
IDnsResolver							
Binder Parcel	0	1860	2200	2347	2503	3012	**3232**
Autogen	0	797	2181	2207	2766	2836	**2948**
installd							
Binder Parcel	0	1400	1769	1916	2739	3423	**4071**
Autogen	519	692	968	970	979	979	**979**
NFC							
Binder Parcel	0	0	1305	1570	1726	1873	**1884**
Autogen	657	657	657	677	678	678	**678**
OccupantAwareness							
Binder Parcel	0	1305	1466	1509	1682	1843	**1947**
Autogen	0	1450	1549	1561	1703	1856	**1875**
ThreadChip							
Binder Parcel	0	1594	1690	1699	1840	1980	**1981**
Autogen	0	718	719	719	720	720	**720**
statsd							
Binder Parcel	718	718	718	722	965	1341	**1341**
Autogen	1281	1281	1281	1311	1315	1317	**1317**

OK-calls

The difference in the OK-call percentage is striking: the autogen fuzzer consistently achieves much higher successful-call rates than the baseline - often above 90%, and for many services close to 99%. In contrast, the binder parcel fuzzer achieves much lower ratios, especially on complex interfaces. This demonstrates that the autogen fuzzer is highly effective at reaching and invoking service logic with valid inputs.

It is worth noting, however, that the OK ratio for the autogen fuzzer is not always perfect. In particular, services such as `IDnsResolver` and `OccupantAwareness` show noticeably lower success rates. Unlike the baseline, these failures are not due to random input invalidity, but instead likely stem from remaining bugs or missing edge-case handling in the autogenerated serialization logic. Given the complexity of AIDL interfaces and the many corner cases involved in Binder serialization - especially with nested parcelables, dynamic arrays, or binder/file descriptor arguments - imple-

menting fully correct serialization remains a challenging task, and some failures are expected at this stage.

These results should be interpreted as preliminary. The autogen fuzzer is still a work in progress, with certain features (most importantly “refuzzing” of returned binder objects) not yet implemented, and the current serialization code not yet fully validated across all possible argument patterns. The current statistics therefore serve as an initial demonstration of the approach’s viability, while also highlighting areas where further debugging and refinement are needed.

5.2.3 On Coverage Metrics and Harness Complexity

Another very important factor to consider is where the additional coverage measured by the parcel fuzzer actually originates, particularly for a service as simple as `HelloService`, which contains no complex logic behind deserialization at all. Manual inspection of the logs confirms that all exposed methods of `HelloService` are reached and exercised by both fuzzers, with the autogen fuzzer consistently triggering these methods faster and with fewer failures than the binder parcel fuzzer. Yet, the raw coverage metrics reported for the parcel fuzzer remain significantly higher. This discrepancy raises the question: Where do these extra code edges come from, if not from new paths inside the service logic itself?

A possible answer lies in the complexity of the fuzzer driver and harness code itself. The reference binder parcel fuzzer, as implemented in `libbinder_driver.cpp`, is a large and intricate harness with many conditional branches, edge cases, and fallback behaviors, all of which are subject to coverage tracking. By contrast, the autogenerated fuzzer is intentionally minimal and only contains code required to invoke service methods. As a result, the coverage numbers for the parcel fuzzer include not only the target service but also large portions of the harness logic, while the autogen fuzzer’s coverage is dominated by actual service logic.

This shows that coverage metrics alone are not a sufficient indicator of fuzzing quality or depth of service logic exploration, particularly when comparing fuzzers with substantially different harness complexity. More granular metrics - such as source-based coverage from Clang, or custom instrumentation focusing only on service code - would provide a more accurate picture of the effective code path exploration achieved by different fuzzers.

Chapter 6

Discussion

6.1 Future Work

Immediate directions for improvement are clear:

- Implementation of “refuzzing” for returned binder interfaces, expanding coverage of callback-heavy or multi-stage protocols.
- Support for variable-length lists of binders and file descriptors, allowing the fuzzer to exercise more complex interfaces and argument patterns.
- Broadening mutation strategies and relaxing some constraints to allow exploration of certain edge cases.
- Enhanced logging and statistical tracking, to more precisely identify coverage bottlenecks and understand where the fuzzer is most effective.

6.2 Related Work

Several frameworks have been proposed to automate fuzzing-harness creation for complex targets. **FuzzGen** by Ispoglou *et al.*[5] automatically infers how a library’s API is used by real programs and generates fuzz drivers based on an “Abstract API Dependency Graph” of those interactions. This allows the fuzzer to execute realistic sequences of API calls without the extraneous logic of full applications, achieving deeper coverage than manual stubs.

Google’s large-scale **FUDGE** system[2] tackles library fuzzing at scale. It mines existing client code, applies slicing and lightweight synthesis to create thousands of buildable fuzz drivers, over

two hundred of which have been adopted into continuous fuzzing pipelines, leading to more than 150 upstream fixes. Unlike our compile-time approach, FUDGE operates post-facto and relies on the presence of diverse client applications.

Interface-aware fuzzing has also been explored for operating-system components. Machiry *et al.* developed **DIFUZE**[3], which uses static analysis to create well-structured ioctl requests for kernel drivers.

Binder-level argument-aware fuzzing was first explored by Feng and Shin's **BinderCracker**[4]. BinderCracker records valid transactions, mutates parameters in a *parameter-aware* manner and replays them via low-level ioctl calls, revealing more than one hundred vulnerabilities across Android 4–6. Unlike our compile-time stubs, BinderCracker requires a heavy runtime harness and does not guarantee alignment with evolving AIDL interfaces.

Within Android user space, Liu *et al.* introduced **FANS**[6], which systematically fuzzes Binder services by automatically analysing C++ interface definitions (partly autogenerated from AIDL), inferring transaction codes, argument types and cross-call dependencies and generating harnesses from that.

Our approach, **AID(L)ing the Fuzzer**, builds on these insights but differs in two key ways. First, instead of relying on heavyweight static analysis to reconstruct interfaces, we directly use the "authoritative" AIDL definitions to emit harness code at compile time. This guarantees perfect alignment with current interface contracts and eliminates the risk of stale models. Second, by embedding custom mutators into the auto-generated driver, this approach preserves libFuzzer's seed-to-coverage feedback loop while achieving transaction success rates close to 99%. Similar to FANS's generation-based testing, our method produces compile-time harnesses for each target interface, allowing easy integration with coverage-guided fuzzers. Consequently, our framework offers interface-aware fuzzing in a developer-friendly and maintainable workflow, allowing rigorous fuzz testing of Android system components with minimal manual effort.

Chapter 7

Conclusion

In summary, the auto-generated fuzzer already provides a highly promising foundation. Its structure-aware approach is exceptionally reliable and efficient at exercising valid interface methods, but it is not yet able to match the total coverage of the more mature and less structured binder parcel fuzzer. With continued development and improved support for dynamic features and mutation strategies, it is plausible that an advanced autogen fuzzer could ultimately surpass the current state-of-the-art in both reliability and coverage, which would inevitably lead to more bugs being found and fixed in the Android Open Source Project.

Bibliography

[1] *AIDL backends - Android source documentation*. URL: https://source.android.com/docs/core/architecture/aidl/aidl-backends#types.

[2] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. "FUDGE: fuzz driver generation at scale". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 975–985. ISBN: 9781450355728. DOI: 10.1145/3338906.3340456. URL: https://doi.org/10.1145/3338906.3340456.

[3] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. "DIFUZE: Interface Aware Fuzzing for Kernel Drivers". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2123–2138. ISBN: 9781450349468. DOI: 10.1145/3133956.3134069. URL: https://doi.org/10.1145/3133956.3134069.

[4] Huan Feng and Kang Shin. "BinderCracker: Assessing the Robustness of Android System Services". In: (Apr. 2016). DOI: 10.48550/arXiv.1604.06964.

[5] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. "FuzzGen: Automatic Fuzzer Generation". In: *29th USENIX Security Symposium*. 2020, pp. 2271–2285.

[6] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. "FANS: Fuzzing Android Native System Services via Automated Interface Analysis". In: *29th USENIX Security Symposium*. 2020, pp. 307–322.

[7] *Structure-Aware Fuzzing - libFuzzer Documentation*. July 2024. URL: https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.

[8] yuandaimaahao. *AIDLCppDemo*. July 2023. URL: https://github.com/yuandaimaahao/AIDLCppDemo.