FUDGE: Fuzz Driver Generation at Scale

Domagoj Babić*
Stefan Bucur*
dbabic@google.com
sbucur@google.com
Google
USA

Yaohui Chen*
yaohway@ccs.neu.edu
Northeastern University
USA

Franjo Ivančić*
Tim King*
Markus Kusano*
ivancic@google.com
taking@google.com
kusano@google.com
Google, USA

Caroline Lemieux*
clemieux@cs.berkeley.edu
UC Berkeley
USA

László Szekeres*
Wei Wang*
lszekeres@google.com
wwweiwang@google.com
Google
USA

ABSTRACT

At Google we have found tens of thousands of security and robustness bugs by fuzzing C and C++ libraries. To fuzz a library, a fuzzer requires a *fuzz driver*—which exercises some library code—to which it can pass inputs. Unfortunately, writing fuzz drivers remains a primarily manual exercise, a major hindrance to the widespread adoption of fuzzing. In this paper, we address this major hindrance by introducing the Fudge system for automated fuzz driver generation. Fudge automatically generates fuzz driver candidates for libraries based on existing client code. We have used Fudge to generate thousands of new drivers for a wide variety of libraries. Each generated driver includes a synthesized C/C++ program and a corresponding build script, and is automatically analyzed for quality. Developers have integrated over 200 of these generated drivers into continuous fuzzing services and have committed to address reported security bugs. Further, several of these fuzz drivers have been upstreamed to open source projects and integrated into the OSS-Fuzz fuzzing infrastructure. Running these fuzz drivers has resulted in over 150 bug fixes, including the elimination of numerous exploitable security vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**; *Software evolution*; Automated static analysis.

KEYWORDS

software security, testing, fuzzing, fuzz testing, automated test generation, program slicing, code synthesis

ACM Reference Format:
Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3340456

*Authors listed in alphabetical order.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08.
https://doi.org/10.1145/3338906.3340456

1 INTRODUCTION

Fuzzing has emerged as one of the most effective testing techniques for discovering security vulnerabilities and reliability issues in software. The idea behind fuzzing is simple: the fuzzer executes programs with randomly generated inputs, and monitors their behavior for invalid operations, such as memory corruption issues. Recent advancements in fuzzing technologies, such as coverage-guided fuzzing [22, 32, 42], have enabled fuzzing to reach even deeper program paths and uncover significantly more bugs.

The success of fuzzing has led to significant adoption in the industry, and the emergence of services providing continuous fuzzing for open source and commercial software. For example, Google has developed continuous fuzzing infrastructures to test the security of C/C++ libraries, both for its internal software and externally for open-source code. Google's ClusterFuzz project, through its OSS-Fuzz [1, 3] instance, has alone filed tens of thousands of bugs to developers by fuzzing over 200 open source projects.

C/C++ code is a primary target for fuzzing, due to unsafe language features, such as explicit memory management, that makes it prone to bugs and vulnerabilities. To detect such bugs, the fuzzed programs are usually instrumented with checks (e.g., ASAN [33]) that can expose memory corruption issues and other undefined behavior in C/C++ code. Several classes of these bugs, such as buffer overflows, use-after-frees, integer overflows, and uninitialized memory, are often exploitable security vulnerabilities.

Despite the increasing adoption, most C/C++ codebase still cannot be fuzz tested due to the absence of testing harnesses, known as *fuzz drivers*, which exercise the library code.[1] In this paper, we argue that we can significantly accelerate the adoption of fuzzing by automatically generating these fuzz drivers.

There are two main challenges in writing a fuzz driver for a given library. The driver author needs to understand (a) the codebase under test and (b) how the fuzzer engine operates. Listing 1 shows an example of a fuzz driver for the OpenCV library. Fuzzing infrastructures have converged to use the shown `LLVMFuzzerTestOneInput` interface between fuzzers and fuzz drivers, which was introduced by libFuzzer [32]. This interface is called a *fuzz target* [31], and is the de facto standard interface of fuzz drivers—we will use the terms fuzz target and fuzz driver interchangeably. Fuzz targets receive an input buffer generated by a fuzzer through the `data` argument and use it to invoke some relevant functionality of the targeted code, e.g., in Listing 1, feed it into an OpenCV datatype (`cv::Mat`).

[1] An alternative to library-based fuzzing is fuzzing an entire binary, provided the functionality is available as a stand-alone tool. This paper does not cover this setting.

The fuzz target can be linked with various fuzzer engines (e.g., libFuzzer [32], AFL [42], Hongfuzz [36], SBF [37]) to produce a fuzzer test executable. The fuzzer engines execute the fuzz target function *in-process*, generating and executing tens of thousands of test inputs per second. Moreover, these engines gain direct access to real-time coverage information through instrumentation callbacks, used to guide the test generation.

Listing 1 FUDGE-generated fuzz target for OpenCV.

```
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  std::vector<uint8_t> arr = {data, data + size};
  cv::Mat row = cv::Mat(1, arr.size(), CV_8UC1, arr.data());
  try {
    cv::Mat decoded = cv::imdecode(row, CV_LOAD_IMAGE_UNCHANGED);
  } catch (cv::Exception e) { }
  return 0;
}
```

Writing effective targets can be time-consuming and requires knowledge of the target codebase. Developers need to understand what the interface of the target library is and how to use it correctly. For example, fuzzing an image format library requires knowing how to specify the image contents and how to trigger the format parsing operations. Moreover, the library calls must be configured such that crashes only happen in the case of implementation bugs. Crashes caused by violating a function's preconditions are unhelpful distractions (e.g., trying to read from a file that was not opened first). Further, fuzz targets should be deterministic and free of side effects, such that crashes can be reproduced reliably.

In this paper, we present FUDGE — a system for accelerating fuzz target creation. FUDGE helps developers create fuzz targets for libraries quickly and easily by providing automatically synthesized candidate fuzz targets. These automatically-generated targets typically only need cosmetic modifications and small fix-ups before adoption. For example, the target in Listing 1 was generated by FUDGE.

The key insight behind FUDGE is that fuzz targets which exercise library functions in a valid and useful manner can be synthesized from existing uses of the library in a codebase. To this end, the FUDGE technique benefits from Google's monolithic source code repository [27], which includes third-party open source code [17].

FUDGE generates fuzz target candidates via a backend pipeline that processes the entire Google codebase. The pipeline scans the codebase for usages of target libraries, extracts interesting code snippets, and automatically synthesizes runnable fuzz target candidates and build instructions. It then runs the synthesized fuzz target candidates with libFuzzer at scale to collect runtime feedback. The runtime feedback contains relevant features such as code coverage, size of generated tests, observed crashes, etc. Finally, FUDGE collects the generated candidates and execution statistics so that they can be browsed through the frontend by developers. Developers are encouraged to modify fuzz targets (e.g., to increase generality, improve efficiency, or remove potential side effects) and adopt them to enable continuous fuzzing going forward. FUDGE is built on top of ClangMR [41] in order to handle the analysis and code synthesis at a large scale.

Generating a fuzz driver from scratch, using only the target library code, is a challenging program synthesis problem. FUDGE leverages existing usage patterns in client code to produce fuzz targets that discover valid crashes instead of API misuses. Moreover, by observing the runtime behavior of the generated target, FUDGE is able to filter out uninteresting behavior. FUDGE surfaces drivers with interesting behaviors via a user interface. Users can then choose to integrate the synthesized drivers into continuous fuzzing services. The human in the loop provides an important check on the correctness of the drivers before developers are asked to fix any reported bugs.

Contributions. This paper makes the following contributions:

- We describe the FUDGE system for automatic fuzz driver generation. FUDGE statically analyzes the client code to learn valid usages of a library of interest, adapts these usages using code mutations, and emits buildable fuzz targets.
- We present case studies from our experience with using FUDGE. FUDGE has created thousands of fuzz targets, over 200 of which have been vetted by developers and accepted for continuous fuzzing. Several of these targets were generated for open-source libraries, and are continuously running on the OSS-Fuzz infrastructure. They have found numerous exploitable security vulnerabilities in widely used open-source projects.
- We distill a number of lessons that we have learned in the process of developing FUDGE and handling fuzz target synthesis at a large scale.

Overview. Section 2 provides a high-level overview of the FUDGE system, walking through the generation of an example fuzz target; Section 3 provides more technical details. We demonstrate the use of FUDGE through three open-source case studies in Section 4. Section 5 discusses the lessons that we have learned in using FUDGE. Section 6 presents related work, and potential future directions are described in Section 7. Section 8 concludes the paper.

2 OVERVIEW

Figure 1 shows the high-level architecture of the FUDGE system. It consists of two main components: a backend pipeline generating fuzz target candidates, and a frontend UI exposing these to the user.

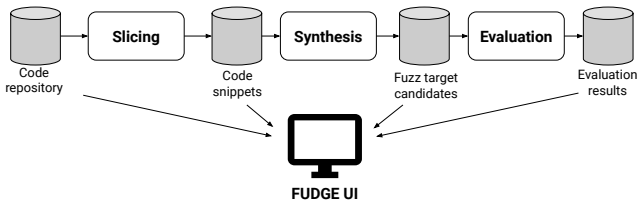

Figure 1: FUDGE high-level overview. The Slicing module extracts code snippets of library usages. These code snippets are mutated and transformed into fuzz targets by the Synthesis module. The Evaluation module builds and runs the candidate fuzz targets. The results are post-processed and presented to the user on the FUDGE User Interface.

The pipeline consists of the Slicing, Synthesis, and Evaluation modules. This is a distributed pipeline that runs daily, processing the entire Google code repository. It attempts to generate fuzz targets

for thousands of packages, and stores the resulting candidates and related metadata in a database. The results can be conveniently browsed using the FUDGE UI frontend, where developers can take candidate fuzz targets, modify them if needed, and adopt them. We describe each phase of the pipeline by walking through the life of an example fuzz target.

Slicing phase. We will consider the FreeImage library as our target library in our running example. The Slicer module scans the whole code repository looking for usages of this library, to extract code snippets. It processes each C++ source file in the codebase through a custom Clang frontend action, an extension mechanism of the LLVM compiler framework [21]. Suppose the Slicer module encounters the function shown in Listing 2. This is a function inside the Ogre 3D graphics engine library [25], slightly modified for brevity, that calls into the FreeImage library.

Listing 2 Example use site of FreeImage in Ogre.

```
Codec::DecodeResult FreeImageCodec::decode(DataStreamPtr &input) {
  FreeImage_SetOutputMessage(FreeImageLoadErrorHandler);
  MemoryDataStream memStream(input, true);
  FIMEMORY *fiMem = FreeImage_OpenMemory(
     memStream.getPtr() , memStream.size() );
  FIBITMAP *fiBitmap =
    FreeImage_LoadFromMemory((FREE_IMAGE_FORMAT) mImageType , fiMem);
  if (!fiBitmap) {
    OGRE_EXCEPT(Exception::ERR_INTERNAL_ERROR,
                "Error decoding image", "FreeImageCodec::decode");
  }
  ImageData *imgData = OGRE_NEW ImageData();
  MemoryDataStreamPtr output;
   imgData->width = FreeImage_GetWidth(fiBitmap);
   imgData->height = FreeImage_GetHeight(fiBitmap);
  FREE_IMAGE_TYPE imageType = FreeImage_GetImageType(fiBitmap);
  if (imageType == FIT_BITMAP) {
    FIBITMAP *newBitmap = FreeImage_ConvertToGreyscale(fiBitmap);
    FreeImage_Unload(fiBitmap);
    fiBitmap = newBitmap;
  }
  // ...
  FreeImage_Unload(fiBitmap);
  FreeImage_CloseMemory(fiMem);
  DecodeResult ret;
  ret.first = output;
  ret.second = CodecDataPtr(imgData);
  return ret;
}
```

The slicer analyzes the abstract syntax tree (AST) of this function to determine the relevant statements that should be extracted. A function is considered for slicing only if there is at least one call to the target library that might be a parsing API, e.g., the API receives a byte buffer argument. The `FreeImage_OpenMemory` call has the signature (`uint8_t* data, uint32_t size_in_bytes`), so the slicer processes this function. The slicer algorithm first selects all FreeImage calls as statements of interest. Then it collects dependent statements by following control and data-flow dependencies. In our example, it ends up collecting the statements highlighted in Listing 2 for inclusion in the output code snippet.

Some expressions inside the selected statements are not sliced out (see gaps in highlighted lines). This happens when an AST expression contains symbols that are defined outside of the function or have types defined outside of the target library. E.g., the `memStream.getPtr()` expression is left out from the extracted code snippet, because `MemoryDataStream` is a type defined in the Ogre library, and not in the targeted FreeImage library. This expression is replaced with a placeholder variable of type `uint8_t*`, which is the type of `memStream.getPtr()`. Similarly, the `mImageType` variable reference is replaced with a placeholder, because—although it has type `int`—it is a member variable of the `FreeImageCodec` Ogre class, and out of scope of the extracted code snippet.

Synthesis phase. The Synthesis module receives the extracted code snippet and completes it by concretizing the introduced placeholder variables, and sending the fuzzer input to the function call arguments. There are many possible ways to fill in these placeholders. For example, we could replace the placeholder of the original `mImageType` with an `int` constant (e.g., `0`, `1`), or we can also choose to fuzz this argument and feed the first few bytes of the fuzzer-provided input into it. We do not know in advance what the best way is to finalize the snippet. Thus, FUDGE synthesizes multiple versions of the code, using different replacement options and their combinations. This results in multiple candidate fuzz target variants for each sliced code snippet. Listing 3 shows one of the candidates, which feeds the fuzzer input into `FreeImage_OpenMemory`, and replaces the original `mImageType` reference with `0`.

Listing 3 One of the candidate fuzz targets output by Synthesis.

```
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  FIMEMORY *fiMem = FreeImage_OpenMemory((uint8_t*)data, size);
  FIBITMAP *fiBitmap =
    FreeImage_LoadFromMemory((FREE_IMAGE_FORMAT)0, fiMem);
  if (!fiBitmap) return 0;
  FreeImage_GetWidth(fiBitmap);
  FreeImage_GetHeight(fiBitmap);
  FREE_IMAGE_TYPE imageType = FreeImage_GetImageType(fiBitmap);
  if (imageType == FIT_BITMAP) {
    FIBITMAP *newBitmap = FreeImage_ConvertToGreyscale(fiBitmap);
    FreeImage_Unload(fiBitmap);
    fiBitmap = newBitmap;
  }
  FreeImage_Unload(fiBitmap);
  FreeImage_CloseMemory(fiMem);
  return 0;
}
```

Evaluation phase. Finally, each of these synthesized candidate targets is evaluated by building and running it with a preconfigured time bound. This evaluation step allows FUDGE to quickly weed out drivers with obvious API misuse issues that cause immediate, uninteresting, crashes. For example, generating a fuzz target that attempts to write to a file before having first opened said file leads to immediate crashes of low value to the user. The candidates that build and run successfully are saved to a temporary code repository, and their evaluation results are stored in a database.

User interface and workflow. From the user's perspective, checking in a new target works as follows. The user goes to the web-based FUDGE UI where they can browse, filter, and rank the evaluated candidates among all packages, or find all the FUDGE-generated candidates for a particular package or API. For the FreeImage library, the UI shows that the candidate in Listing 3 was built and run successfully, created 21 new test inputs during the evaluation, and covered the most new lines among the candidates for FreeImage. It also shows that the target crashed with an out-of-memory error after a while.

The user pulls the source and build file of this candidate fuzz target into their local working directory by clicking a button on the code browser that saves the necessary copy command to their clipboard. The fuzz target can be tested locally immediately, e.g., with libFuzzer, as the build file is available. The log of libFuzzer reveals that the OOM is due to leaking `fiMem`. The user spends a few minutes fixing the leak and doing minor cleanups and is ready to check in the finalized target into the main repository. This is the process followed by the authors to check in the example target into the FreeImage directory of the OSS-Fuzz GitHub repository (https://git.io/fjZNb).

This illustrates how Fudge increases the productivity of users that are either not familiar with fuzzing or not familiar with the target project: It pre-generates a set of potentially interesting candidate fuzz targets and provides execution statistics to the user. Contrast this few minutes of work to the manual process of writing a fuzz target, which requires a significant time investment into discovering library code and finding the relevant APIs to fuzz.

3 FUDGE DESIGN

This section describes each component of the system in more detail, starting with the backend modules, then the frontend. All modules of the backend are massively parallelized pipelines, built on top of Google's MapReduce technology [8]; in particular, the more recent C++ implementation of Flume [6].

3.1 AST Slicing

The goal of the slicing module is to extract code snippets from existing client code of the target library. The snippet will serve as the basis of the generated fuzz targets. The slicing pipeline processes the entire Google codebase in a few hours using ClangMR [41]. This massively parallelized process parses every source file in the repository using the Clang compiler frontend and runs the code extraction method shown in Algorithm 1 on each function definition.

Algorithm 1 The slicing algorithm of Fudge.

Input: abstract syntax tree (AST) of function $func$.
Output: selected set of statements to extract S.

1: $S \leftarrow$ SelectTargetLibraryCalls($func$)
2: $S_{prev} \leftarrow \emptyset$
3: **while** $S \neq S_{prev}$ **do** ▷ fixed point is not reached
4: $S_{prev} \leftarrow S$
5: $S \leftarrow S \cup$ DataFlowDependenciesOf(S)
6: $S \leftarrow S \cup$ ControlFlowDependenciesOf(S)
7: **return** S

We use an algorithm similar to standard program slicing [38, 40] on the AST of a given function, starting from a set of seed statements and following both backward and forward dependencies. If a function contains at least one target library call that looks fuzzable (e.g., takes a character buffer like argument), then we select all the calls to the target library as seed statements. If there are no fuzzable calls, we select no seed statements. If we have seed statements, we propagate their dependencies until a fixed point is reached. Specifically, we iteratively find and mark the set of transitively relevant statements according to both data flow and control flow dependencies. For efficiency purposes, Fudge utilizes *syntactic slicing* only, i.e., we do not perform alias analysis.

In Algorithm 1, data flow dependencies of statements in S are (1) variable definition statements of variables used in S, and (2) control statements that use variables defined by statements in S. Control flow dependencies of S are (1) control statements that dominate a statement in S and (2) control flow terminator statements (e.g., `return`, `break`). We add dependent statements until a fixed point is reached.

Listing 4 Code snippet sliced out from the example use site.

```
uint8_t* UnknownA; uint32_t UnknownB; int UnknownC;
unsigned UnknownD; unsigned UnknownE;

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  FIMEMORY *fiMem = FreeImage_OpenMemory(UnknownA, UnknownB);
  FIBITMAP *fiBitmap =
    FreeImage_LoadFromMemory((FREE_IMAGE_FORMAT)UnknownC, fiMem);
  if (!fiBitmap)
    return 0;
  UnknownD = FreeImage_GetWidth(fiBitmap);
  UnknownE = FreeImage_GetHeight(fiBitmap);
  FREE_IMAGE_TYPE imageType = FreeImage_GetImageType(fiBitmap);
  if (imageType == FIT_BITMAP) {
    FIBITMAP *newBitmap = FreeImage_ConvertToGreyscale(fiBitmap);
    FreeImage_Unload(fiBitmap);
    fiBitmap = newBitmap;
  }
  FreeImage_Unload(fiBitmap);
  FreeImage_CloseMemory(fiMem);
  return 0;
}
```

While propagating dependencies, we also remove expressions with *unknown* symbols or types from newly added statements. An *unknown symbol* can be a reference to a variable defined outside of the sliced function, e.g., a function parameter, or a member variable (e.g., `mImageType` in Listing 2). It can also be a reference to a global variable or function defined outside of the translation unit (e.g., reference to the `FreeImageLoadErrorHandler` function on the first line of the example). Note that if a referenced function is defined in the same translation unit, we can slice it interprocedurally. An *unknown type* is a non-primitive data type defined outside of the target library, e.g., the type of `memStream` is the Ogre library defined `MemoryDataStream` in Listing 2. These expressions are replaced with `UnknownX` placeholder variables as shown in the code snippet extracted from the example function in Listing 4. Doing this prevents the algorithm from slicing out client library specific code, as we do not follow the dependencies of these out-of-scope expressions. Note that the introduction of placeholder variables keeps the code syntactically correct and compilable.

Once we selected the statements to extract from the source function, we rebuild the code in a new AST context, inside an `LLVMFuzzerTestOneInput` function. While rebuilding, we lazily import any global variable definition or constant that is referenced in the sliced code (and defined in the same source file). We support slicing out called functions recursively from the same compilation unit: If the sliced function references another function defined in the same source file, we perform another round of intra-procedural slicing on the referenced function (seeded with target library calls) and import it to the new context. Finally, we replace all return

statements with `return 0;` as the fuzz target interface expects the target function to return zero for successful executions.

In addition to extracting the code snippet from the client code, we extract how the client code includes and builds against the target library. In Google's monolithic repository, all build dependency information is directly available in standard Bazel [15] BUILD rules. From the Clang AST we can establish the set of header files that need to be included, and from the build dependencies of the client library we can find the target libraries to link against. The headers and build instructions are added as metadata to the snippets and sent to the synthesis module.

3.2 Target Synthesis

The goal of the code synthesis module is to complete the code snippets extracted by the slicer — i.e., ensure that (1) our target API call is fed with the fuzzer input and (2) there are no `UnknownX` placeholders left in the code. An incomplete code snippet can be made complete through a series of rewrites. Table 1 lists some of these rewrites.

Table 1: Examples of matched type patterns of `UnknownX` expressions and the corresponding replacement carried out by the synthesizer.

Matched UnknownX type	Replacement expression
`(char*, int)`	`(FuzzerInputPtr(), FuzzerInputSize())`
`std::string`	`FuzzerInputAsString()`
`std::string`	`FuzzerInputAsTempFilePath()`
`int`	`0`
`int`	`1`
`int`	`temp_var_a`
`int`	`FuzzerInputAsInt32()`
`int*`	`&temp_var_a`
`int&`	`temp_var_a`
`int&`	`FuzzerInputAsInt32()`

For instance, in our working example snippet in Listing 4, the call `FreeImage_OpenMemory(UnknownA, UnknownB)` has arguments of type `uint8_t*` and `uint32_t`. This function call argument pair matches one of our rewrite patterns: the one that replaces the two matched `UnknownX` expressions with `FuzzerInputPtr()` and `FuzzerInputSize()` (first line of Table 1). We similarly match expressions with string-like types as potential fuzz inputs. Another rewriting possibility for string arguments is to pass a file path. This works for APIs such as `FreeImage_Load` that loads an image from a file. In this case, we extend the final fuzz target code with some code that saves the fuzzer input in a temporary file (`TempFile f(data, size);`) and pass the file path to the function (`f.path()`); see Listing 7.

There are a few other ways of rewriting `UnknownX` expressions of different types, beyond feeding the fuzzer input. Consider arguments, such as the `mImageType` in our working example (see Listing 2), that are initialized outside of the sliced function and therefore replaced with `UnknownX` placeholders. These arguments can be either input or output variables (passed by pointer or reference). One rewrite option that often works, is to create a local variable for them and default initialize it. This means that we rewrite the expression to a symbol like `temp_var_a` and add a line `SomeType temp_var_a {};` preceding it.

Listing 5 Pseudo-code of the synthesis algorithm of FUDGE.

```
def FuzzTargetSynthesis(extracted_ast):
  complete_asts = []
  incomplete_asts = [extracted_ast]
  seen_before = Set([extracted_ast])
  while incomplete_asts is not []:
    ast = incomplete_asts.pop()
    if HasUnknownExpressions(ast):
      for rewrite in MatchingRewrites(ast):
        new_ast = Apply(rewrite, ast)
        if new_ast not in seen_before:
          incomplete_asts.append(new_ast)
          seen_before.add(new_ast)
    else:
      complete_asts.append(Finalize(ast))
  return complete_asts
```

To summarize, the synthesis module is searching in the space of all possible completions of extracted code snippets. The algorithm enumerating all completions is shown in Listing 5. We maintain and process a work list of ASTs that are not fully complete yet. For each AST, we identify all the locations (AST nodes) where each rewrite could be applied, and apply it. We also maintain a hash set of ASTs that we have seen before, to avoid generating duplicates or rediscovering the same sub-tree of the search tree multiple times. The shown algorithm returns all possibilities, but if the resulting set is too large, we randomly sample a given maximum number from them to keep for evaluation.

The `Finalize` function is necessary to clean up and concretize the final code. For instance, if we selected multiple arguments as FuzzerInput(), then this finalization step adds the necessary code that splits up the input buffer into multiple slices and feeds them into the different fuzzable arguments. We also add the necessary `#include` directives. For example, if the finalization pass introduced a `std::string` type we include `<string>`. Finally, we generate a Bazel BUILD file that contains the rule for building the synthesized fuzz target that depends on the target library. The generated fuzz target source files and their corresponding BUILD files are saved and passed to the evaluation module.

3.3 Target Evaluation

The goal of the evaluation phase is to weed out non-functional fuzz target candidates, and rank the rest by fitness. The FUDGE pipeline resorts to heuristics for extracting and transforming code snippets from library client code. These heuristics do not guarantee that the resulting code is a suitable fuzz driver. For example, the APIs exercised may be irrelevant (e.g., code that does not perform any interesting logic), or the API may not be used in a way suitable for fuzzing, such as performing too strict error checking. To maximize the chance of generating a valid driver, we resort to generating a large and diverse set of candidates for each library API, and ranking them when presented to developers.

Deciding whether a fuzz driver candidate is suitable for the library is ultimately a holistic process. It involves answering whether (a) the driver fuzzes the right API for the library, and (b) the APIs are used correctly for the purpose of fuzzing. Fully automating this process is imprecise and complex. Instead, our approach is to resort mainly to human judgment for answering the first question and provide signals computed automatically to help answer the second

question. By looking at the fuzz drivers shortlisted and ranked according to the correctness signals, developers need only to filter out the candidates that do not fuzz the correct APIs.

The evaluation module computes the signals indicating whether the APIs are used correctly. We currently use the following signals to filter and rank the candidates:

(1) The candidate should build successfully.
(2) It should run successfully without generating a crashing input for at least a few seconds.
(3) The size of the minimized corpus of the target should be larger than some lower threshold.
(4) The larger the number of lines of the library covered, the better. We measure both absolute coverage and increase in coverage relative to the existing fuzz drivers for the library.

In practice, we found that these metrics provide good precision: those fuzz targets that build successfully and generate a non-trivial amount of coverage with a non-trivial corpus size are likely to be valid candidates. We filter out fuzz targets which find a crashing input in very few seconds. While this may miss some targets that expose very shallow bugs in the target library, the reduction in false positives due to API misuse in the target is more important for Fudge's usability.

We compute the evaluation signals using a pipeline whose input is the set of candidate fuzz targets generated. The first stage of the pipeline builds each target and records the build status and the build logs.

For those targets that build successfully, the next stage consists of performing fuzzer runs and collecting any crashes or generated tests. Given that fuzzing is random by nature, we replicate the fuzzer run N times for each driver (in our implementation, $N = 10$) and collect the artifacts from all runs. We use libFuzzer as the fuzzer engine; its in-process fuzzing approach has the performance needed to make a fuzz target fitness judgment in a limited time. Each fuzzer run takes 5 minutes, which we found to provide a good trade-off between resource consumption and sensitivity to fuzzer saturation. More specifically, by experimenting with multiple fuzzing times, we found that for 95% of the targets, fuzzing for 5 minutes attains 99% of the coverage attained by doubling the fuzzing time.

The next evaluation stage collects all the generated tests for each fuzz driver and performs corpus minimization. The resulting corpus is used to perform a coverage run. Finally, the collected line coverage information is aggregated and compared to baseline coverage achieved by all other pre-existing fuzz drivers.

Each fuzzer run performed in the pipeline is sandboxed. This is required not only for security reasons, but also for robustness: a synthesized fuzz driver may behave in arbitrary ways, including attempting I/O or leaking file descriptors and other resources. Without proper sandboxing, these drivers may starve and eventually lead to the crash of the pipeline nodes.

The pipeline execution is massively parallel, each stage being performed on all the fuzz drivers at once. The parallelism enables the pipeline to finish all the stages within hours. The results of the evaluation are saved in a database.

3.4 User Interface

Users interact with the system through a web-based UI, which shows the information produced by the backend candidate generation pipeline. Recall that the goal of Fudge is to minimize the time of adding a new fuzz target for a library, even for users that are not intimately familiar with the target libraries (packages). To do so, the Fudge UI provides different views, presenting the available packages, their APIs, the extracted code snippets, and the generated candidates.

Candidate targets view. The candidates view lets users browse the list of promising candidates that were built and executed successfully. Targets are presented similarly as shown in Figure 2.

Package Name	Fuzz Target	Called APIs	New Lines Covered	Corpus Size	Use Site
free_image ↪	view ↪	FreeImage_OpenMemory ↦ FreeImage_GetFileType ↦ FreeImage_CloseMemory	318 ↪	71 ↪	view ↪
libxml ↪	view ↪	xmlURIEscapeStr ↦ xmlParseURIReference ↦ xmlFreeURI	43↪	23 ↪	view ↪
...	...	...	...	...	...

Figure 2: Candidates view of UI.

The ↪ in the table represents links. E.g., clicking the package name takes the user to the package directory in the main repository. The UI is integrated with Google's internal code search engine [29] and browser. The target *view* leads the user to the fuzz target source code with its `BUILD` file. The *Called API*s column shows the sequence of the target library functions that the candidate calls. The API names that are exercised by existing fuzz targets are color-coded with green; the ones that are not covered yet are shown in red.

We also indicate the number of new lines the candidate covered during its evaluation relative to the coverage of existing fuzz targets. We link to the coverage report showing the coverage information overlayed on the code in the code browser. The *Corpus Size* column shows the number of test cases generated during the evaluation run. Additional evaluation results are also presented, such as the number of crashes found so far. Finally, for each candidate, we link to the original use site and the code snippet sliced out from it (see *Use sites view* below).

The results can be searched, sorted, and filtered by package or by APIs exercised. That is, the user can choose to only see candidates for `libxml` or only candidates that call `FreeImage_OpenMemory`, sorted by new lines covered relative to existing fuzz targets, etc. This lets users pick interesting candidates based on their actual coverage and performance measured during the evaluation. Once the user finds a candidate worthy of checking in, they can easily pull it into their local working directory; the necessary command can be copied from the code browser.

Packages view. The packages view, depicted in Figure 3, focuses on packages that may require further fuzzing. It lists the available packages (i.e., different third-party libraries) in the code repository with their relevant information, such as the number of existing fuzz targets for the package and their aggregated coverage. This

provides a high-level overview of the fuzzing progress across the whole codebase and enables engineers to prioritize and decide which packages to add more fuzzers to.

Package Name	User Packages	Existing Targets	Fuzz Coverage	Candidate Targets	Public APIs
free_image ↪	5 ↪	1 ↪	4% ↪	63 ↪	9 ↪
libxml ↪	43 ↪	5 ↪	39% ↪	82 ↪	12 ↪
...	...	...	...	...	...

Figure 3: Packages view of UI.

One signal that helps determine a package's importance is the number of other packages depending on it. This number is shown in the *User Packages* column with a link (↪) to the view that lists all client libraries and their details. The next column shows the number of existing targets with a link to a view listing the existing targets of the given package (described later). *Fuzz Coverage* is the percentage of code covered by the checked in fuzz targets, with a link to a detailed report. Clicking the number of candidate targets links to the candidate view filtered down for the given package. Once users pick their package of interests, they typically want to see the interfaces of the package in order to find fuzzable APIs. This is done by following the link in the number of *Public API*s column, leading to the APIs view.

APIs view. The APIs of a package are shown in a table like that in Figure 4. The primary purpose of this view is to identify APIs that should be fuzzed. The interfaces that are most useful to fuzz are the ones that do non-trivial parsing or processing on user input. We use heuristics to filter these APIs, based on their names and type signatures. For example, we look for words like "parse", "load", or "open" in the function names, and look for character-buffer-like input argument types. Using these heuristics, we narrow down a potentially very long list of functions for the user to select from. By default, we also rank the APIs by their number of use sites. The names and the canonicalized types are shown on the interface. Clicking on the API name takes the user to the function definition.

API Name	Function Type Signature	Use Sites	Candidate Targets	Existing Targets
FreeImage_ OpenMemory ↪	(uint8_t* data, uint32_t size_in_bytes) ↳ struct FIBITMAP	31 ↪	22 ↪	0 ↪
FreeImage_Load ↪	(enum FREE_IMAGE_FORMAT fif, const char* filename, int flags) ↳ struct FIBITMAP *	12 ↪	0 ↪	0 ↪
FreeImage_ LoadFromMemory ↪	(enum FREE_IMAGE_FORMAT fif, struct FIMEMORY* stream, int flags) ↳ struct FIBITMAP*	9 ↪	6 ↪	1 ↪
...	...	...	...	...

Figure 4: APIs view of UI.

By default, we only show the APIs that are called from other packages to focus on the ones that are actually used. The *Existing Targets* column indicates whether the API is covered by an existing fuzz target, while the *Candidate Targets* column shows the number of candidate targets calling the API. Already covered API names are also marked green. Existing targets can be viewed by clicking the existing target number. Yellow APIs are not covered yet, but working candidates are available. The candidates can be viewed similarly, by following the provided link. Red means that the API is neither covered nor are there any new candidates yet. When APIs are neither covered nor available, this could highlight current limitations of Fudge that we try to address in the future.

Use sites view. Given a package or an API, the use sites view lists all caller functions in a table form, similarly to the previously described views. Each row shows the caller function name, the sequence of API calls at the call site, e.g., `FreeImage_OpenMemory` ↦ `FreeImage_GetFileTypeFromMemory` ↦ `FreeImage_CloseMemory`, a link to the call site location shown in the code browser, and a link to the extracted code snippet (such as the one on Listing 4). We also link to the candidates generated from the given call site. In cases where the system failed to generate a working candidate target from a call site, users can use intermediate code snippets extracted from it and complete it manually.

Existing targets view. Finally, the existing targets view of the UI lists all fuzz targets, potentially filtered down for a particular package or API. For each target, we show the APIs it calls, its coverage, the bugs it found, and other statistics.

The UI also provides views showing the overall state and progress made in terms of checked in fuzz targets, their coverage, the bugs found, and bugs fixed. This helps users keep track of the impact of fuzzing and fuzz target generation.

4 CASE STUDIES

We have been using and continue to use Fudge to increase the fuzz coverage of both internal and external third-party C/C++ libraries. Fudge has synthesized thousands of fuzz targets across hundreds of security-sensitive open source projects used by Google. The most promising targets are surfaced in the Fudge UI and vetted by engineers interested in applying fuzzing to their project or third party projects. The engineers choose whether to integrate a candidate into our continuous fuzzing services. Each integration indicates that an engineer is willing to triage and fix bugs reported on the fuzzing targets. Integration thus gives us a high degree of certainty that the generated target is exercising the library correctly and the resulting bugs are expected to be useful (up to human judgment). 200+ Fudge-generated targets have been integrated to date and are being continuously fuzzed. Over 600 bugs have been discovered by fuzzing these targets and more than 150 likely exploitable security bugs (e.g., buffer overflows) have already been fixed.

In November 2018, the Google Security Team announced an effort to "admit as many OSS projects as possible [into OSS-Fuzz] and ensure that they are continuously fuzzed" [28]. As a part of this effort, we have integrated more than 10 new open source projects with OSS-Fuzz using Fudge targets. These targets are fully open sourced and the bugs that they find are eventually publicly disclosed [10]. The already public bugs can be seen in the OSS-Fuzz issue tracker [16]. We present case studies on two projects that have been integrated into OSS-Fuzz using Fudge-generated fuzzing targets. We also present a case study of sending a Fudge-derived target directly to a project.

4.1 Leptonica

Leptonica [4] is an open source, general-purpose image processing library. Users of Leptonica include the popular Tesseract [34] OCR (Optical Character Recognition) library (26k stars on GitHub). Leptonica supports I/O operations on a large number of underlying image formats (PDF, SVG, JPEG, webp, ...), and provides convenience utilities for handling many of these types in a uniform fashion. Dan Bloomberg, the main author of Leptonica, had previously fuzzed Leptonica "with random number generators modifying input, and had found some issues, and thought the code was well-protected."

Listing 6 FUDGE generated fuzz target for Leptonica.

```
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  const int16_t angle = ReadInt16(&data, &size);
  const int16_t x_center = ReadInt16(&data, &size);
  const int16_t y_center = ReadInt16(&data, &size);

  Pix* pix = pixReadMem(
      reinterpret_cast<const unsigned char*>(data), size);
  if (pix == nullptr) { return 0; }
  Pix* pix_rotated = pixRotateShear(
      pix, x_center, y_center, (M_PI / 180.) * angle, L_BRING_IN_WHITE);
  if (pix_rotated) { pixDestroy(&pix_rotated); }
  pixDestroy(&pix);
  return 0;
}
```

FUDGE synthesized 14 fuzz targets for Leptonica, all of which are now continuously fuzzed. One of the targets synthesized by FUDGE is given in Fig. 6.[2] At its core, the target attempts to read a byte array as an image and rotates the image around a center point by an angle. The generated target has many common characteristics of a hand written fuzz target: it begins by parsing fuzzed data (`pixReadMem`), it does a simple operation (`pixRotateShear`), it has minimal error checking, and it ends with cleaning up memory (`pixDestroy`). Generating the fuzz target required extracting: library constants (`L_BRING_IN_WHITE`), sub-expressions (`M_PI / 180`), and error handling (`if (pix == nullptr)`). Generation further required replacing `UnknownX` values for the `int16_t` variables, and adding conversion expressions between types. However, observe that the final synthesized program is not much more complicated than a unit test of the same library code.

The FUDGE-generated fuzz targets have been highly effective at discovering bugs in Leptonica. Developers have already fixed 73 reported bugs found by the generated fuzz targets. This includes 22 security sensitive bugs (e.g., heap-out-of-bounds memory reads).[3] After working with us for several months, Dan Bloomberg had this to say about FUDGE: "*I believe that this is a very important and innovative project, one that can potentially have a major impact on the stability and security of open source libraries.*"

4.2 OpenCV

OpenCV [7] contains over 2,500 C++ implementations of computer vision algorithms. It is used within the industry by companies such as Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, and Toyota[4]. OpenCV uses unit tests for both logical and performance regressions. There are currently 353 logical test files and 110 performance test files.

We upstreamed two fuzz targets[5], that were automatically generated by FUDGE, which are currently fuzzed continuously within OSS-Fuzz. These fuzz targets, both just a few lines of code, led to 39 bugs[6] being fixed in OpenCV so far, 12 of which were deemed security relevant (e.g., heap-buffer overflows)[7].

One of these targets, shown in Listing 1,[8] consists only of loading the fuzzed input into a matrix[9] and reading the matrix as an image. Notice the `try`-block around the decoding API call. This control structure was extracted from the original client code of the library and included in the target. Catching exceptions thrown by `cv::imdecode` is important to avoid crashing when the fuzzed library legitimately detects that the input is malformed. Otherwise, the fuzzer would not be productive.

This example demonstrates the benefit of synthesizing numerous candidate targets and ranking them by performance characteristics. The FUDGE pipeline generated many candidates fuzzing the same API without the `try-catch` block. Those candidates were deemed less effective per the collected fuzzing performance statistics.

4.3 HTSlib

HTSlib[10] (380 GitHub stars) is a C library for handling genome sequencing data in different formats. It is a core component of Samtools [35] (755 GitHub stars), which provides utilities for post-processing gene sequence alignments. The authors directly upstreamed a fuzz target (pull request #796)[11] to the HTSlib maintainers who have used it to identify and fix over 36 bugs (PR#805)[12].

Listing 7 FUDGE generated fuzzing target for HTSlib.

```
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  const TempFile file(data, size);
  htsFile* ht_file = hts_open(file.path().c_str(), "r");
  if (ht_file == nullptr) { return 0; }
  hts_close(ht_file);
  return 0;
}
```

The upstreamed fuzz target was *derived* from the FUDGE generated fuzz target in Listing 7. The generated target starts by writing `data` to a temporary file using a fuzz target utility class `TempFile`. This first line was added by the synthesis module described in Section 3.2: the first argument of the sliced out `hts_open` call was rewritten to `FuzzerInputAsTempFilePath()` (see Table 1), which was finalized to this code in the last step of the algorithm in Listing 5. Again, this was just one of the variants that we synthesized from the sliced out code snippet (which was only the `hts_open` call with `nullptr` checking and with `hts_close`). In this variant the library function interprets the fuzzed argument as a filename. This version was ranked on top of other variants, as its coverage was the highest.

[2] The target was modified for clarity before open-sourcing. Full source at git.io/fjsOv
[3] Example patches to github.com/DanBloomberg/leptonica: git.io/fjsOU, git.io/fjsOI.
[4] https://opencv.org/about
[5] https://github.com/google/oss-fuzz/pull/2034
[6] Public bug reports at https://bugs.chromium.org/p/oss-fuzz?q=proj:opencv.
[7] E.g., bug fixes at https://github.com/opencv/opencv/pulls/{14193,14201,14268}.
[8] Headers are omitted. See full source at https://git.io/fjsz7.
[9] The `vector arr` is there to provide `cv::Mat` a mutable copy of the fuzzed input.
[10] https://github.com/samtools/htslib
[11] https://github.com/samtools/htslib/pull/796
[12] https://github.com/samtools/htslib/pull/805

As the candidate fuzz target stood out it was selected to be manually vetted. Once it was clear it was plausible and finding likely bugs, it was manually written to more fully process different formats in C (HTSlib's language). The modified target was sent to the maintainers of HTSlib, who gave advice for how to avoid writing and reading temporary files. The resulting target was the one upstreamed in PR#796.

While the Fudge-generated target was rewritten manually before checking it in, it did: (1) identify a function of interest, (2) find concrete evidence that this needs attention, (3) focus the user attention, and (4) accelerate the creation of the first fuzz target for the library. The issues reported by this target helped the HTSlib developers fix 36 bugs.

5 LESSONS LEARNED

This section describes a number of lessons that we learned while designing Fudge and utilizing it to generate fuzz targets.

Lesson 1: Choosing a suitable fuzz target (still) requires a human. During the early phases of the Fudge design, we envisioned the system as providing a one-click experience for the developers: they would be presented with a generated fuzz target and its performance parameters, and a single button click would check that target into the repository. It turns out that execution feedback, as discussed in Section 3.3, provides valuable signals to deduce when a candidate target is clearly misusing an API. This information is used to remove the vast majority of candidate fuzz targets from consideration. However, we discovered that a fully automated workflow to pick a final fuzz target was not yet attainable, both for technical reasons and due to human factors.

First, the existing API uses of a library do not always capture all the properties desired in a fuzz driver. While execution feedback is very effective at weeding out bad candidates, the corpora sizes or the coverage improvements alone are not enough to decide which fuzz target is the most appropriate. Factors like security relevance, depth of functionality, and parameter choices can only be determined through an understanding of the intended role of an API. In the absence of a formal specification for the library, this is a task best performed by a human. Fortunately, Fudge substantially simplifies this task by presenting the developer only with a short list of choices for investigation.

Second, the role of the source code is not only to specify an algorithm for computer execution, but to communicate intent between developers. Google has a strong engineering culture of code *readability*, underpinned by rigorous code reviews, and the fuzz targets are no exceptions. We found that choosing appropriate variable names, achieving modularity, implementing defensive programming practices, and otherwise implementing style guide recommendations are a task much better suited for humans today.

To support developers, we provide code review recommendations. For example, we ask developers to check whether fuzzing efficiency could be improved, by rewriting a candidate target from potentially relying on a file parsing API with an equivalent in-memory parsing API. Finally, the generated fuzz target is expected to become the maintenance responsibility of developers once it is checked in. Therefore, during code review, developers are expected to thoroughly scrutinize the code for issues at all levels before accepting it in the codebase.

Lesson 2: API call sites present good locality. In theory, the set of APIs called for a library at a given use site could be arbitrarily distributed across functions, classes, and units. This would pose a serious challenge for any static analysis technique trying to extract the control and data flow relations between the APIs. In practice, we have discovered that, for surprisingly many APIs, enough uses exist that exhibit unit- and even function-level locality. Extracting these uses often requires only an intraprocedural analysis. Even when an interprocedural analysis is required, confining it to the same translation unit renders it massively parallelizable.

Lesson 3: Analyzing C++ is challenging due to a long tail of language features. C++ is a complex language, with language features spanning classes, template metaprogramming, and complex scoping rules. At the scale of Google's codebase, we found that it is almost certain that even the most esoteric language features are used somewhere in the code. While we were able to extract a reasonable first set of API use sites by supporting basic language features (C-like free functions), going after the long tail required handling increasingly more types of language syntax. In our Clang-based implementation, this translated into AST visitors handling an increasing number of AST expressions, declarations, and types.

Lesson 4: Randomized algorithms have a good cost-value tradeoff for program synthesis. Randomized algorithms are generally a powerful technique for navigating a search space with complex rules. Fuzzing has been a prime example in the field of test generation, becoming more affordable, yet still more effective than more precise techniques such as symbolic execution. We observed that this rule applies to the fuzz driver generation problem, too. A priori generating a meaningful fuzz driver of an API is a difficult program synthesis problem. However, randomly generating variations of an extracted usage pattern and keeping only the most promising candidates is more tractable. In practice, we have found that we can reach valid API uses using a single generation of mutations – given the scalability of our infrastructure.

Lesson 5: Program synthesis artifacts can be presented as code findings to developers. One of the challenges we tackled in Fudge's design is how to interface the target generation workflow with the development workflow of a Google engineer. We discovered that a *code findings* approach provides minimal disruption to developers, while integrating with the rest of their workflow. The Fudge UI implements this approach by associating each generated fuzz target with the location in the source repository (i.e., library package) where it is relevant. Developers maintaining or working with the code see the associated list of candidate targets and may decide to take further action by inspecting and checking in the targets. In turn, the target generation pipeline lends itself well to a batch computation model, where a periodic pipeline refreshes the list of all candidates available to developers.

6 RELATED WORK

6.1 API Usage Analysis and Code Snippet Extraction

There has been a lot of previous work mining and analyzing API usage patterns that are related to the code snippet extraction of Fudge. The existing tools are primarily created to help developers understand and use a library new to them. Some of these tools only mine function call sequences (without arguments or control-flow) to build some presentable model, other extract code snippets.

In the former category, MAPO [45] was one of the first systems mining sequences of API calls from code search engines, finding and reporting the most frequent sequences. UP-Miner [39] extends MAPO to reduce the redundancy of the mined call sequences. MLUP [30] extended these systems by doing a multi-level analysis of the sequences to find out which methods calls are often used together. PAM [11] uses probabilistic machine learning to mine a less redundant and more representative set of sequences than MAPO or UP-Miner.

Some more advanced tools extract actual source code snippets, similarly to Fudge, typically to serve as the auto-generated documentation of the given API. The tool developed by Buse and Weimer [5] uses a path-sensitive data-flow analysis to generate code snippets for API usage documentation. eXoaDocs [20] also generates usage example snippets for single APIs using program slicing. UsETeC [46] builds on this research but instead of extracting usage examples from client code, it mines them from unit tests. APIMiner [23], MUSE [24] and CLAMS [19] are similar tools. All the above tools work on Java, while Fudge works on C/C++. Fudge uses a somewhat similar slicing algorithm to UsETeC's or to the summarization algorithm of APIMiner and CLAMS, but with a different slicing criteria, with the extension of introducing `UnknownX` expressions, and with support for interprocedural slicing inside the same translation unit.

6.2 Unit-test Generation

While fuzz test generation is a new research area, a closely related area is automated unit-test generation. Randoop [26] is pure random test generation tool for Java. It uses feedback information as guidance to generate random method-calls. Palulu [2] in turn uses a dynamic-random approach. It first infers a call sequence model from a sample execution, then follows that model to create random tests. It has no information about arguments and neither uses the source code to extract additional information statically. RecGen [44], on the other hand, is a static-random approach. It does not have a dynamic analysis phase, it only relies on static analysis to guide the random generation. Because of this, it may fail to create valid sequences of calls for complex interfaces. Finally, Palus [43], a tool developed at Google, improves the above tools, by combining both static and dynamic approaches. All the above tools were developed for Java unit test generation. The synthesis module of Fudge has a significantly simpler task than these tools, as it only needs to complete an extracted code snippet, while the unit-test generators start from scratch.

Another approach to generating unit-tests is to *carve* them from existing tests, e.g., extracting unit tests from execution traces of system tests [9]. Basilisk uses a similar notion to carve *parameterized* unit tests from system tests for C programs [18]. Given a particular system test execution which calls a function to be fuzzed, this method could be utilized to precisely extract a fuzz target for the function. Fudge, having only static information, uses more heuristic approaches to parameterize the extracted code snippets. On the other hand, Fudge has a broader goal: providing a tool to both discover the functions to fuzz as well as expose fuzz targets for those functions.

Some automated testing tools can generate a test driver when the target of testing is a single top level function. For example, the symbolic execution tool DART [13] can extract the interface of a given function and automatically create a driver for it. Micro execution [12] can also execute a function without a user-provided driver, by automatically identifing its I/O interface, allowing randomized test input generation or white-box fuzz testing [14] for the given function. This form of test driver generation focuses on enabling the execution of single function, without having to write set-up or tear-down driver code, while FUDGE focuses on creating drivers exercising typical sequences of library API calls.

7 FUTURE WORK

We continue to apply Fudge to additional target packages of interest. This also includes on-going integrations with OSS-Fuzz for additional OSS packages. In terms of technological improvements to Fudge, we continue to improve the synthesizer to handle additional language features. Furthermore, we are working on generating fuzz targets based on dynamic execution tracing of unit tests that exercise a target library as well. We are also considering leveraging machine learning to improve some of the heuristics or human decisions our system relies on. For example, an ML model could be used to triage candidate fuzz targets based on a mix of static signals (API names, the call graph of the library) and dynamic signals (coverage, corpus size, number of crashes).

8 CONCLUSION

In this paper, we presented Fudge, an automated fuzz driver generating system. Fudge enables developers to apply fuzz testing to their projects faster and easier than ever before. Fudge has already helped significantly increase fuzzing coverage inside Google and in the open source community, and in making fuzz testing more widespread in general. The Fudge-generated targets that were added to various projects helped to find numerous security vulnerabilities and stability issues. Our case studies make us believe that most C/C++ codebase that is not fuzzed yet could see major security and robustness improvements if fuzz testing was applied to them.

ACKNOWLEDGMENTS

We would like to thank the many insightful collaborators, including: Abhishek Arya, Christopher Moon, Dan Bloomberg, Daniel Austin, Daniel Berlin, Felix Gröbert, Fermin Serna, Jonathan Metzman, Jose Duart, Kostya Serebryany, Martin Barbella, Matt Ruhstaller, Max Moroz, Michael Specter, Oliver Chang, Phil Ames, Sam Kerner, Wontae Choi, Yang Yang, and others. Additionally, we would like to appreciate the various OSS project owners that have helped us to get their projects and new fuzzing targets integrated into OSS-Fuzz.

REFERENCES

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. Google Testing Blog. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[2] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. 2006. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS: 1st Workshop on Model-Based Testing and Object-Oriented Systems*. Portland, OR, USA, 27–34.

[3] Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, Jonathan Metzman, and the ClusterFuzz Team. 2019. Open sourcing ClusterFuzz. Google Open Source Blog. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html

[4] Dan Bloomberg. 2001–2018. Leptonica. http://www.leptonica.com.

[5] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 782–792. http://dl.acm.org/citation.cfm?id=2337223.2337316

[6] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 363–375. https://doi.org/10.1145/1806596.1806638

[7] Intel Corporation, Willow Garage, and Itseez. 2019. Open Source Computer Vision Library. https://opencv.org

[8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264

[9] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. 2009. Carving and Replaying Differential Unit Test Cases from System Test Cases. *IEEE Transactions on Software Engineering* 35, 1 (Jan 2009), 29–45. https://doi.org/10.1109/TSE.2008.103

[10] Chris Evans, Ben Hawkes, Heather Adkins, Matt Moore, Michal Zalewski, and Gerhard Eschelbeck. 2015. Feedback and data-driven updates to Google's disclosure policy. https://googleprojectzero.blogspot.com/2015/02/feedback-and-data-driven-updates-to.html.

[11] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free Probabilistic API Mining Across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/2950290.2950319

[12] Patrice Godefroid. 2014. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 539–549. https://doi.org/10.1145/2568225.2568273

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*.

[14] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*.

[15] Google Inc. 2015. Bazel – a fast, scalable, multi-language and extensible build system. http://www.bazel.io

[16] Google Inc. 2018. OSS-Fuzz Issue Tracker. https://bugs.chromium.org/p/oss-fuzz

[17] Google Inc. 2019. Third-Party. Google's open source documentation. https://opensource.google.com/docs/thirdparty

[18] Alexander Kampmann and Andreas Zeller. 2018. Carving Parameterized Unit Tests. *CoRR* abs/1812.07932 (2018). arXiv:1812.07932 http://arxiv.org/abs/1812.07932

[19] Nikolaos Katirtzis, Themistoklis Diamantopoulos, and Charles Sutton. 2018. Summarizing Software API Usage Examples Using Clustering Techniques. In *Fundamental Approaches to Software Engineering*, Alessandra Russo and Andy Schürr (Eds.). Springer International Publishing, Cham, 189–206.

[20] Jinhan Kim, Sanghoon Lee, Seung-Won Hwang, and Sunghun Kim. 2013. Enriching Documents with Examples: A Corpus Mining Approach. *ACM Trans. Inf. Syst.* 31, 1, Article 1 (January 2013), 27 pages. https://doi.org/10.1145/2414782.2414783

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[22] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *CoRR* abs/1812.00140 (2018). arXiv:1812.00140 http://arxiv.org/abs/1812.00140

[23] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. 2013. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 401–408. https://doi.org/10.1109/WCRE.2013.6671315

[24] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 880–890. http://dl.acm.org/citation.cfm?id=2818754.2818860

[25] Ogre Development Team. 2019. OGRE - Open Source 3D Graphics Engine. https://www.ogre3d.org/.

[26] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[27] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (June 2016), 78–87. https://doi.org/10.1145/2854146

[28] Matt Ruhstaller and Oliver Chang. 2018. A New Chapter for OSS-Fuzz. Google Security Blog. https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

[29] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 191–201. https://doi.org/10.1145/2786805.2786855

[30] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. 2015. Mining Multi-level API Usage Patterns. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 23–32. https://doi.org/10.1109/SANER.2015.7081812

[31] Kostya Serebryany. 2015. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html#fuzz-target.

[32] Kostya Serebryany. 2015. Simple guided fuzzing for libraries using LLVM's new libFuzzer. http://blog.llvm.org/2015/04/fuzz-all-clangs.html.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[34] Ray Smith. 2007. An Overview of the Tesseract OCR Engine. In *Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR)*. 629–633.

[35] 1000 Genome Project Data Processing Subgroup, Alec Wysoker, Bob Handsaker, Gabor Marth, Goncalo Abecasis, Heng Li, Jue Ruan, Nils Homer, Richard Durbin, and Tim Fennell. 2009. The Sequence Alignment/Map format and SAMtools. *Bioinformatics* 25, 16 (06 2009), 2078–2079. https://doi.org/10.1093/bioinformatics/btp352 arXiv:http://oup.prod.sis.lan/bioinformatics/article-pdf/25/16/2078/531810/btp352.pdf

[36] Robert Swiecki. 2015. Honggfuzz. http://honggfuzz.com.

[37] László Szekeres. 2017. *Memory Corruption Mitigation via Hardening and Testing*. Ph.D. Dissertation. Stony Brook University.

[38] Frank Tip. 1994. *A Survey of Program Slicing Techniques*. Technical Report. Amsterdam, The Netherlands, The Netherlands.

[39] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 319–328. https://doi.org/10.1109/MSR.2013.6624045

[40] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[41] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. 2013. Large-Scale Automated Refactoring Using ClangMR. In *2013 IEEE International Conference on Software Maintenance*. 548–551. https://doi.org/10.1109/ICSM.2013.93

[42] Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.

[43] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 353–363. https://doi.org/10.1145/2001420.2001463

[44] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. 2010. Random Unit-test Generation with MUT-aware Sequence Recommendation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 293–296. https://doi.org/10.1145/1858996.1859054

[45] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. Springer-Verlag, Berlin, Heidelberg, 318–343. https://doi.org/10.1007/978-3-642-03013-0_15

[46] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang. 2014. Mining API Usage Examples from Test Code. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 301–310. https://doi.org/10.1109/ICSME.2014.52