usenix'20

FuzzGen: Automatic Fuzzer Generation

Kyriakos K. Ispoglou (*Google Inc.*)

Daniel Austin (*Google Inc.*)

Vishwath Mohan (*Google Inc.*)

Mathias Payer (*EPFL*)

Presenter: Sirui Mu

Motivation

- Fuzzing libraries are challenging.
  - Libraries cannot be executed directly, thus hard-coded fuzzer stubs are required.
  - Triggering bugs deep in the library is hard since certain sequence of library function calls is require to build up the program state.
- A mechanism that *automatically* constructs arbitrarily complex fuzzer stubs with complex API interactions and library
- state allows sufficient testing of complex API functions.

Intuition

- Existing code on the system utilizes the library in diverse aspects.
  - The library's unit tests
  - Programs depending on the library
- The key contribution of FuzzGen is an *automatic* way to generate libFuzzer stubs, allowing *broad and deep* library fuzzing.

Example: libmpeg2

- By observing a module that utilizes libmpeg2, fuzzer could observe the dependencies between API calls and the order of initialization API calls.
- Dependencies come in 2 forms:
  - Control flow dependencies;
  - Shared arguments.

```c
/* 1. Obtain available number of memory records */
iv_num_mem_rec_ip_t num_mr_ip = { ... };
iv_num_mem_rec_op_t num_mr_op = { ... };
impeg2d_api_function(NULL, &num_mr_ip, &num_mr_op);

/* 2. Allocate memory & fill memory records */
nmemrecs = num_mr_op.u4_num_mem_rec;
memrec   = malloc(nmemrecs * sizeof(iv_mem_rec_t));

for (i=0; i<nmemrecs; ++i)
    memrec[i].u4_size = sizeof(iv_mem_rec_t);

impeg2d_fill_mem_rec_ip_t fill_mr_ip = { ... };
impeg2d_fill_mem_rec_op_t fill_mr_op = { ... };
impeg2d_api_function(NULL, &fill_mr_ip, &fill_mr_op);

nmemrecs = fill_mr_op.s_ivd_fill_mem_rec_op_t
                     .u4_num_mem_rec_filled;

for (i=0; i<nmemrecs; ++i)
    memrec[i].pv_base = memalign(memrec[i].u4_mem_alignment,
  memrec[i].u4_mem_size);

/* 3. Initalize decoder object */
iv_obj_t *iv_obj = memrec[0].pv_base;
iv_obj->pv_fxns  = impeg2d_api_function;
iv_obj->u4_size  = sizeof(iv_obj_t);

impeg2d_init_ip_t init_ip = { ... };
impeg2d_init_op_t init_op = { ... };
impeg2d_api_function(iv_obj, &init_ip, &init_op);

/* 4. Decoder is ready to decode headers/frames */
```

Figure 2: Source code that initializes an MPEG2 decoder object. Low level details such as struct field initializations, variable declarations, or casts are omitted for brevity.

Design

- API inference
- AADG construction
- Fuzzer stub synthesis

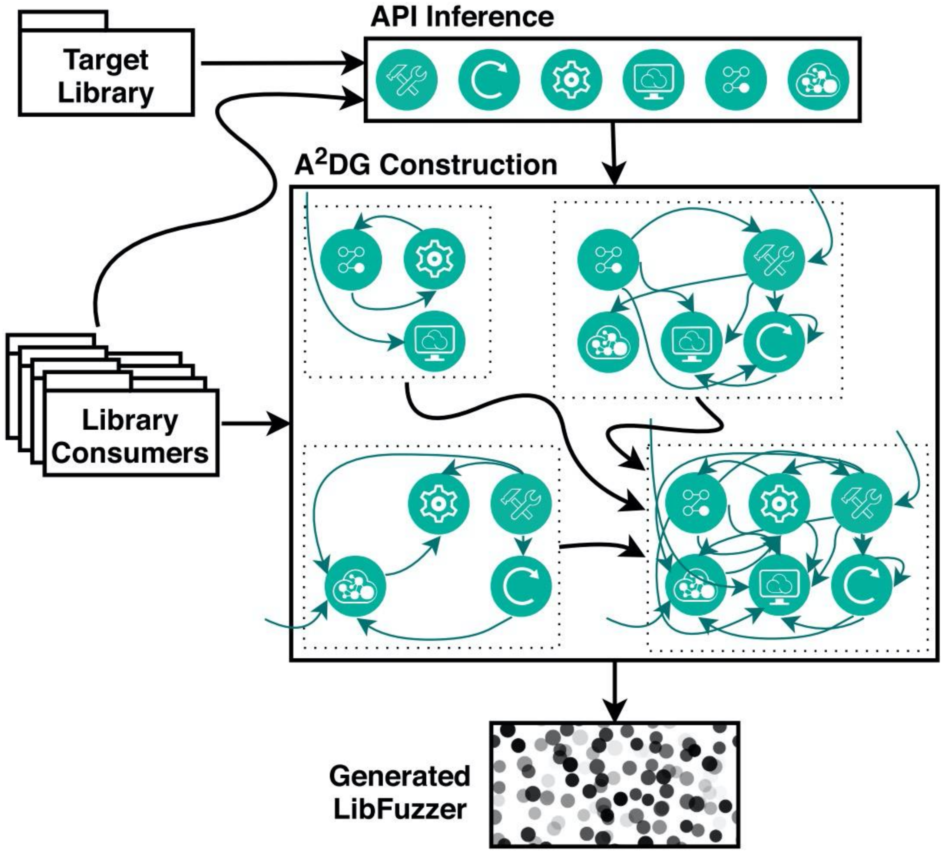
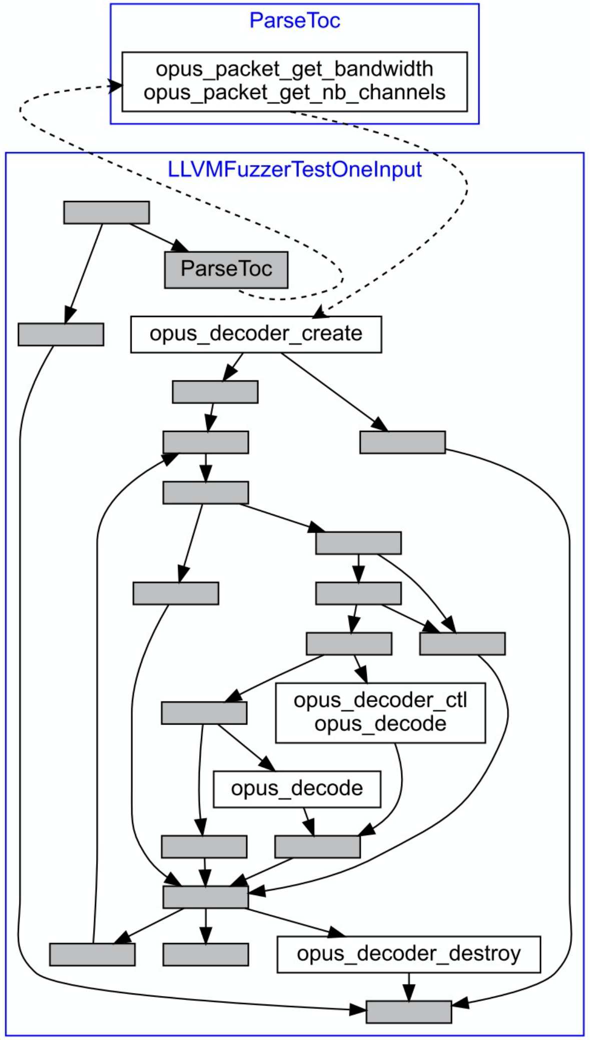


Figure 1: The main intuition behind FuzzGen. To synthesize a fuzzer, FuzzGen performs a whole system analysis to extract all valid API interactions.

API Inference

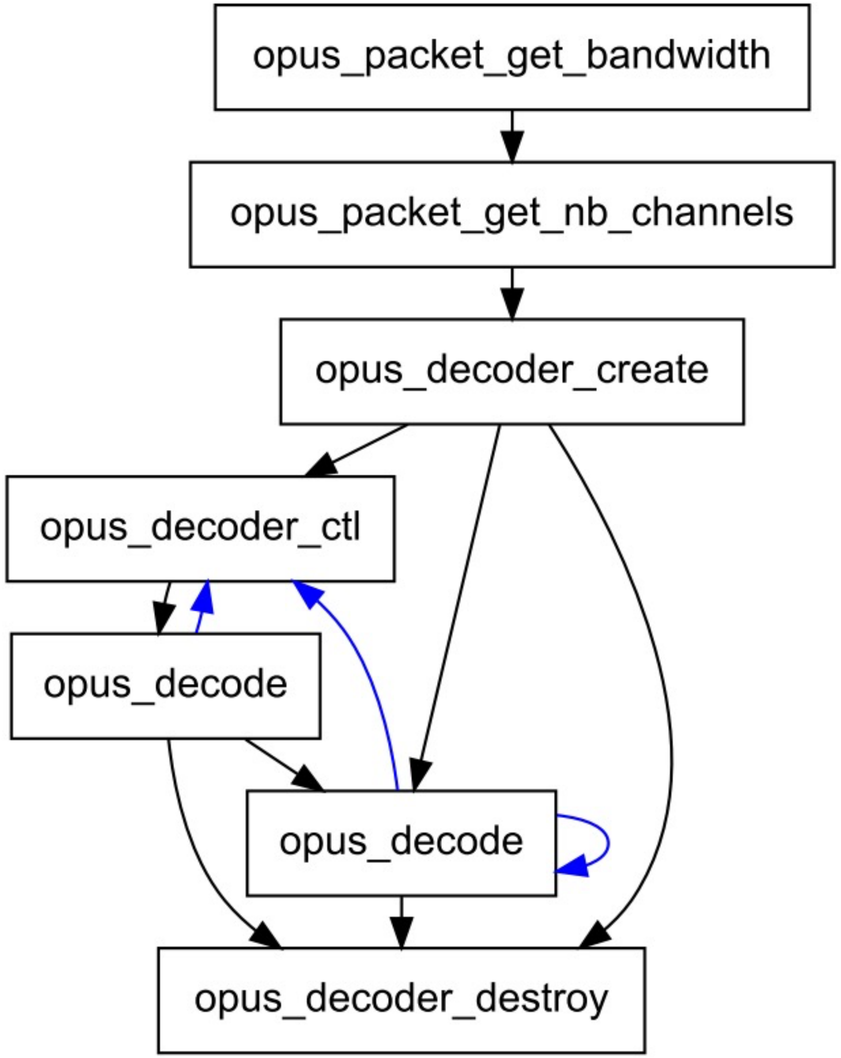
- All functions declared in target library $\mathcal{F}_{lib}$
- All functions declared in all headers included by all co $\mathcal{F}_{incl}$ ners →
- The set of API functions of the target library can be calc $\mathcal{F}_{API} \leftarrow \mathcal{F}_{lib} \cap \mathcal{F}_{incl}$
- To prevent over-approximation, each API function is linked against the target library alone. If link fails, then the callee function does not belong to the target library.

AADG Construction



Control Flow Graph

AADG Construction



Abstract API Dependency Graph

AADG Construction

- AADG construction is a two-step process.
    - A set of AADG is constructed, one for each root function in each consumer.
    - All AADGs are coalesced into a single AADG.

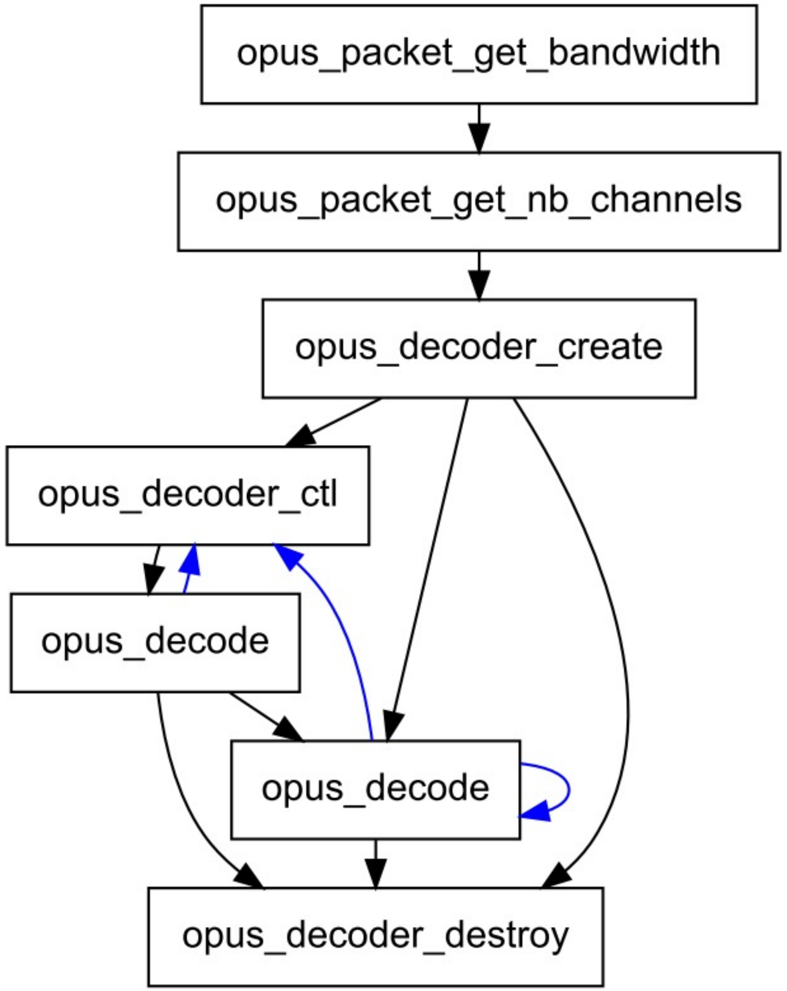
Construct a Basic AADG

- Root function(s): the main function in an executable or the exported functions in a library.
- An individual analysis starts from every root function and explores the full consumer.
- A consumer may produce multiple AADGs.

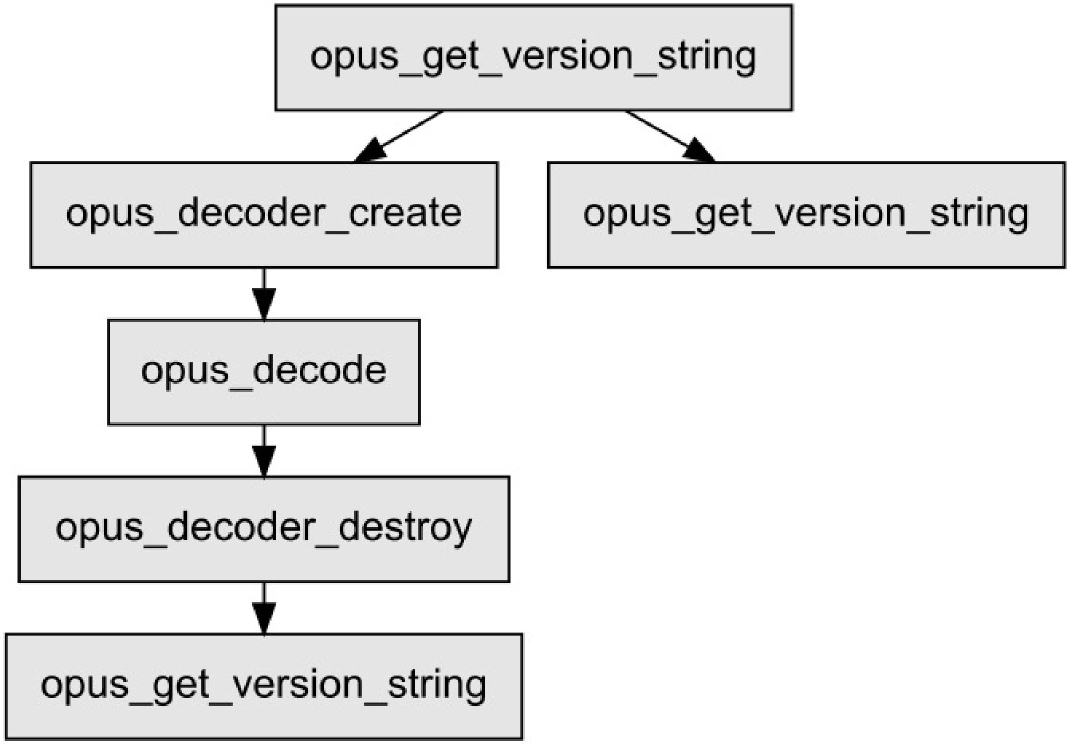
Construct a Basic AADG: the Algorithm

```
function CreateAADG :: Function -> Set Function -> AADG
function CreateAADG(f, vis):
  if f in vis:
    return EmptyAADG()
  vis.insert(f)
  let aadg = GetCFG(f)
  for bb in aadg:
    let callees = GetCallees(bb)
    for callee in callees:
      let calleeNode = SplitNode(aadg, bb, callee)
      if not IsApiFunction(callee):
        ReplaceNode(aadg, calleeNode, CreateAADG(callee, vis))
    RemoveNode(aadg, bb)
  vis.erase(f)
  return aadg
```

Coalesce AACGs

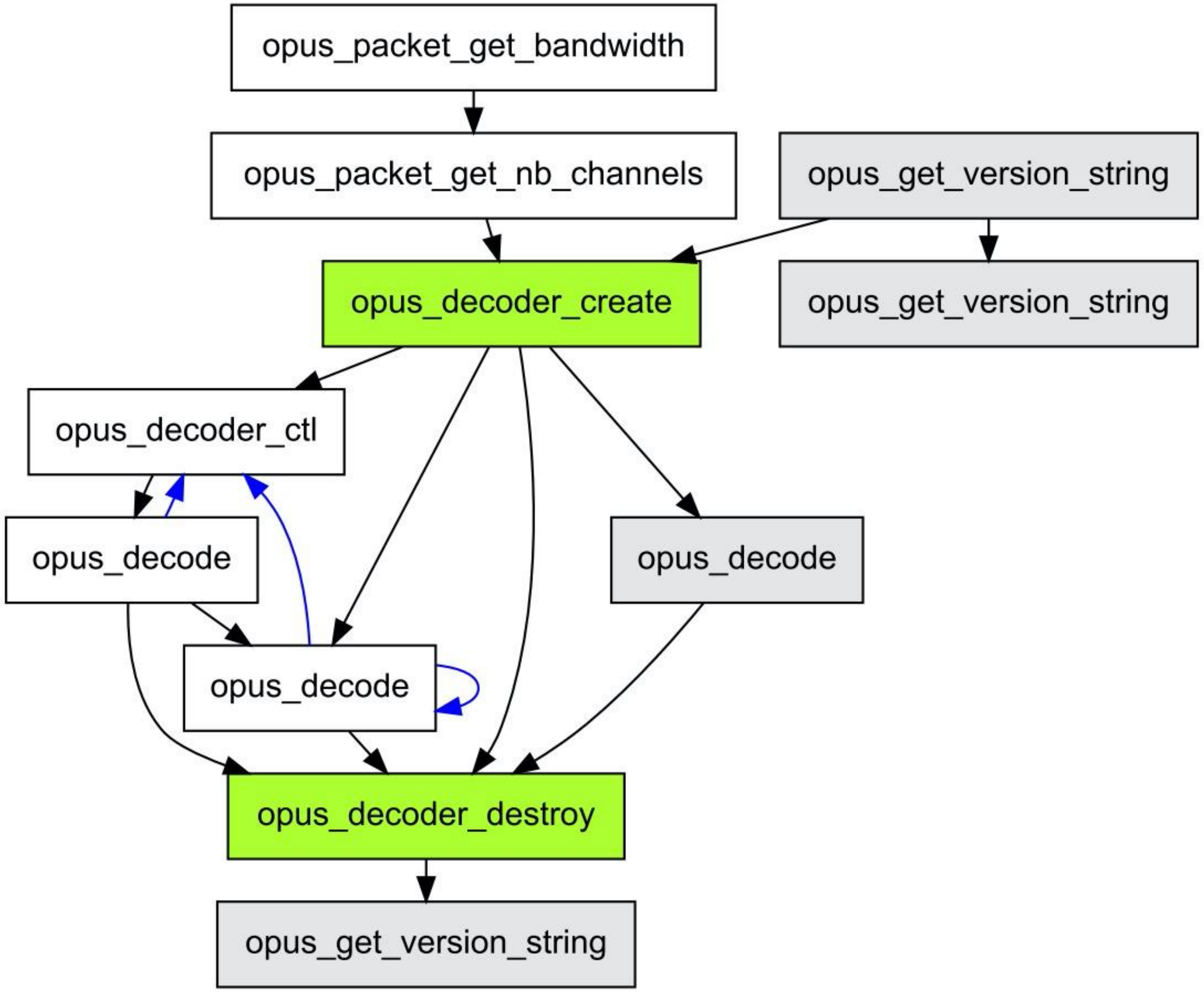


+

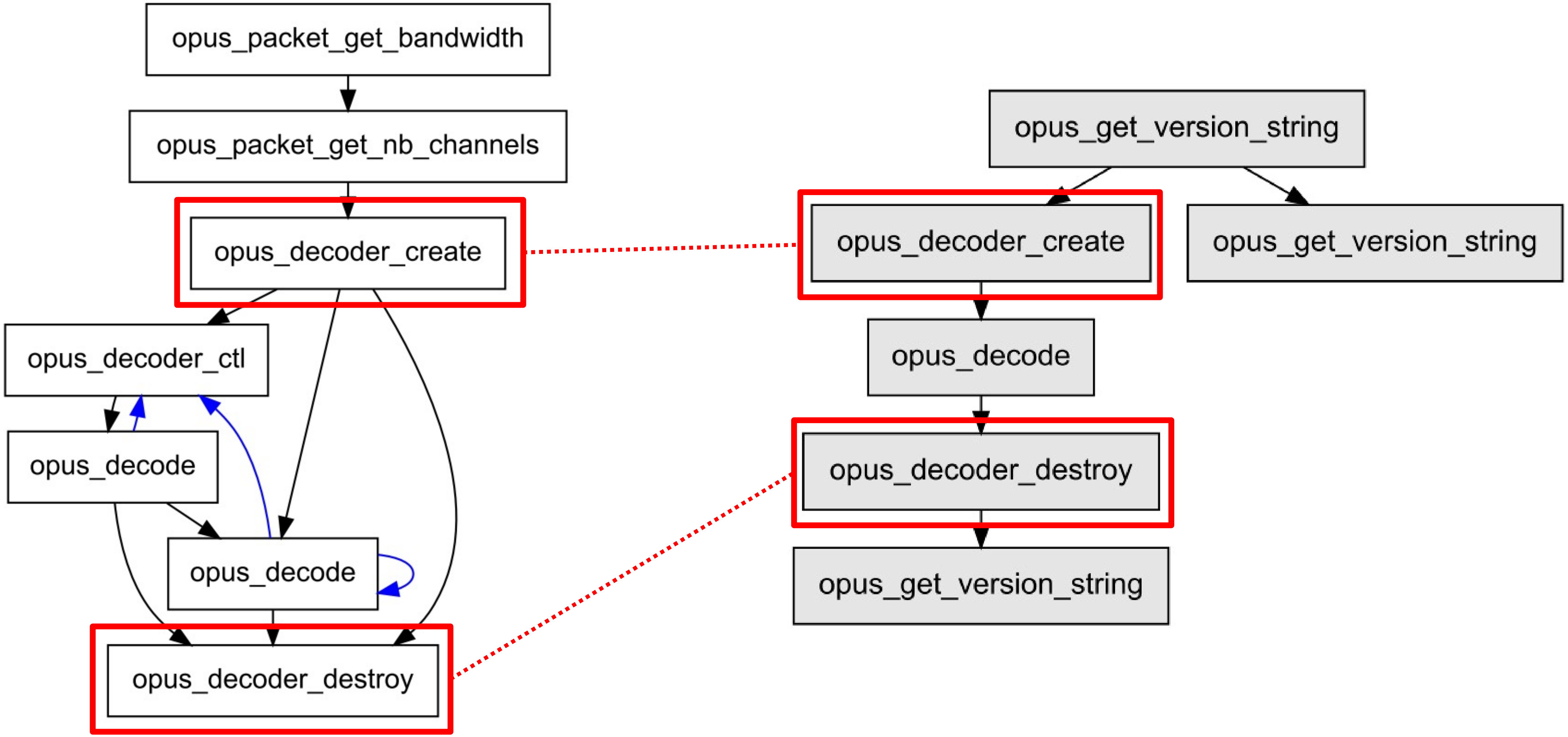


Coalesce AACGs

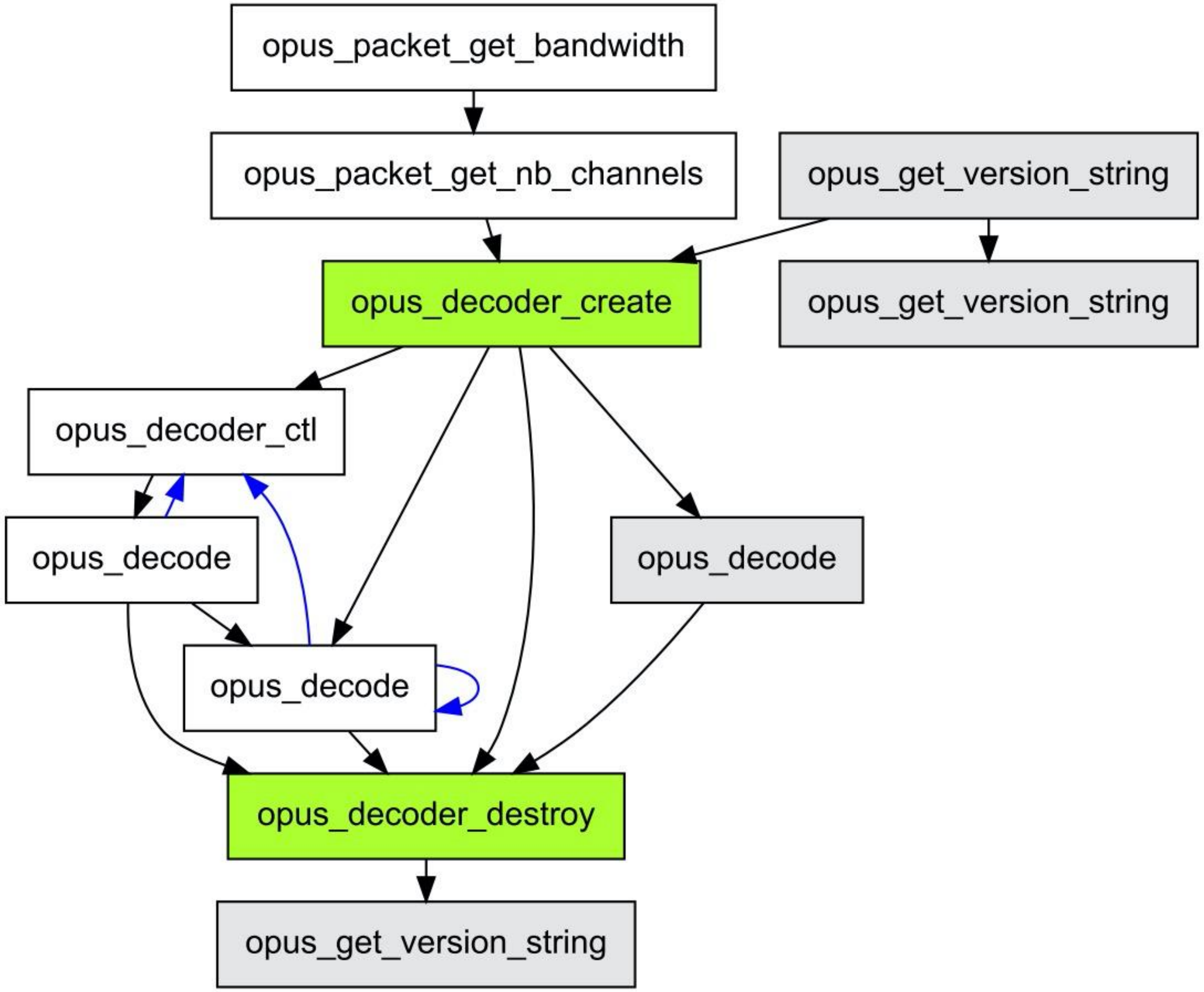
- After generating AACGs for each consumer, FuzzGen tries to coalesce these AACGs into a single AACG.



Coalesce AACGs



Coalesce AACGs



Argument Flow Analysis

- To create effective fuzzers, AACG requires both control and data dependencies.
- To construct *data dependencies* between API calls, FuzzGen leverages two analyses
  - Argument value-set inference (how to generate values)
  - Argument dependency analysis (how individual values are reused)

Argument Value-Set Inference

- Argument value-set inference answers two questions
  - Which arguments to fuzz
  - How to fuzz these arguments
- FuzzGen performs a data-flow analysis in the target library for every argument to get the possible values that an argument can get.

Argument Dependency Analysis

- Data-flow dependencies are as important as control-flow dependencies.
- Data-flow dependencies to be encoded in an AADG can be *intra-procedural* and *inter-procedural*.
  - Intra-procedural analysis: static alias analysis in consumers tracking arguments and return values
  - Inter-procedural analysis: for each edge in AADG, FuzzGen performs another data-flow analysis for each pair of arguments and return values to determine whether they are dependent from each other.

Fuzzer Stub Synthesis

- For each AADG, FuzzGen creates a single stub that leverages the fuzzer's entropy to traverse the AADG.
- The first few bits of the fuzzer input encodes a path in the AADG.
- The rest of the input bits are interpreted as API arguments.

Implementation

- The FuzzGen prototype is written in about 19,000 lines of C++ code, consisting of LLVM passes that implements the analyses and code to synthesis fuzzer stubs.

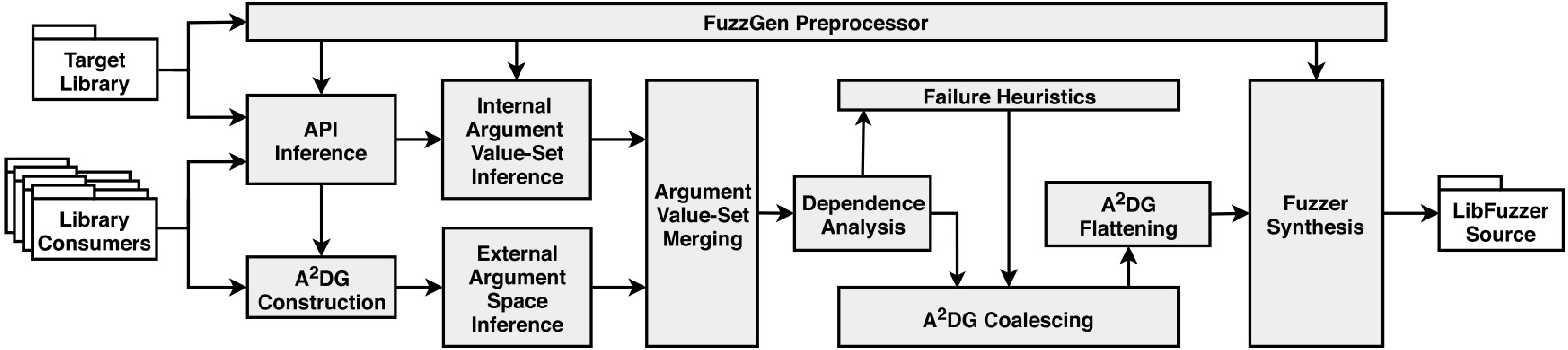


Figure 4: FuzzGen implementation overview.

Internal Argument Value-Set Inference

- Possible values and their types for function arguments are calculated through a per-function data-flow analysis.
- FuzzGen assigns different attributes to each argument.

Attribute	Description
dead	Argument is not used
invariant	Argument is not modified
predefined	Argument takes a constant value from a set
random	Argument takes any (random) value
array	Argument is an array (pointers only)
array size	Argument represents an array size
output	Argument holds output (destination buffer)
by value	Argument is passed by value
NULL	Argument is a NULL pointer
function pointer	Argument is a function pointer
dependent	Argument is dependent on another argument

Table 1: Set of possible attributes inferred during the argument value-set analysis.

External Argument Value-Set Inference

- Performs a backward slice from each API call through all consumers.
- Assign the same attributes to the arguments, using the same rules.

Argument Value-Set Merging

- Generally, FuzzGen's analysis is more accurate with external arguments because these arguments tend to provide real use-cases of the function.
- Value-Set merging is based on heuristics and may be adjusted in future work.

Dependency Analysis

- Knowing the possible values of each argument is not enough. FuzzGen must also know when to reuse the same argument across multiple function calls.
- FuzzGen performs a per-consumer data-flow analysis using precise intra-procedural and coarse-grained inter-procedural tracking to connect multiple API calls.

AADG Coalescing

- AADG coalescing may result in state inconsistency.

```
/* consumer #1 */
sd = socket(...);
connect(...);

// send only sock
shutdown(sd,
         SHUT_RD);
write(sd, ...);

close(sd);
```

(a)

```
/* consumer #2 */
sd = socket(...);
connect(...);

// send & recv
write(sd, ...);

read(sd, ...);

close(sd);
```

(b)

```
/* coalesced */
sd = socket(...);
connect(...);

shutdown(sd,
         SHUT_RD);
write(sd, ...);
read(sd, ...);

close(sd);
```

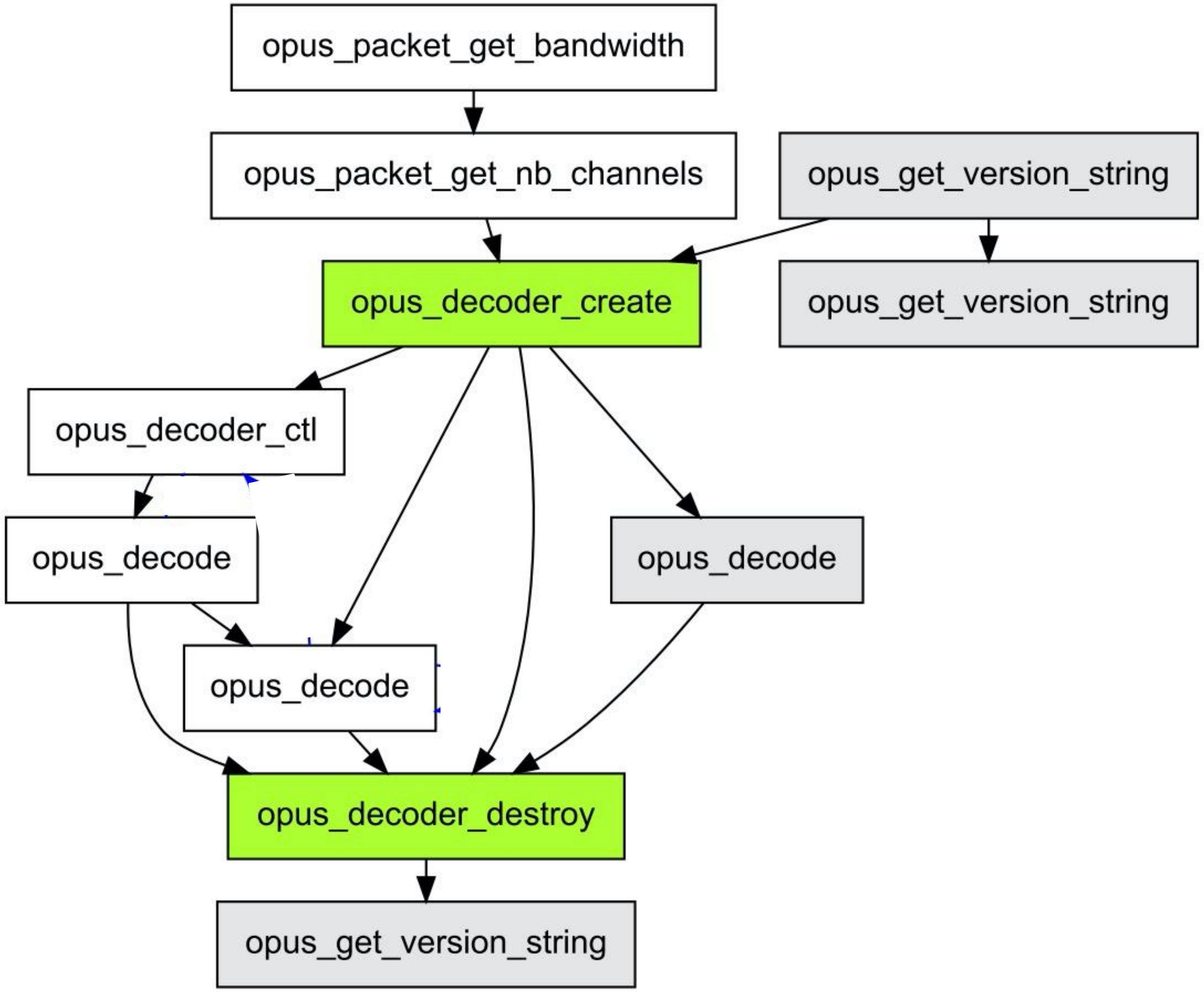
(c)

AADG Flattening

- AADG contains complex control flow and loops. To create simple fuzzers, FuzzGen flattens the AADG before synthesising any fuzzers.
- Traverse the AADG and visit each node at least once. Remove any back edges, turning the AADG into a DAG.
- Perform a topological sort on the DAG to find a total order among all API functions.
- Functions that do not have a total order are grouped into a set. These functions can be called in arbitrary order.

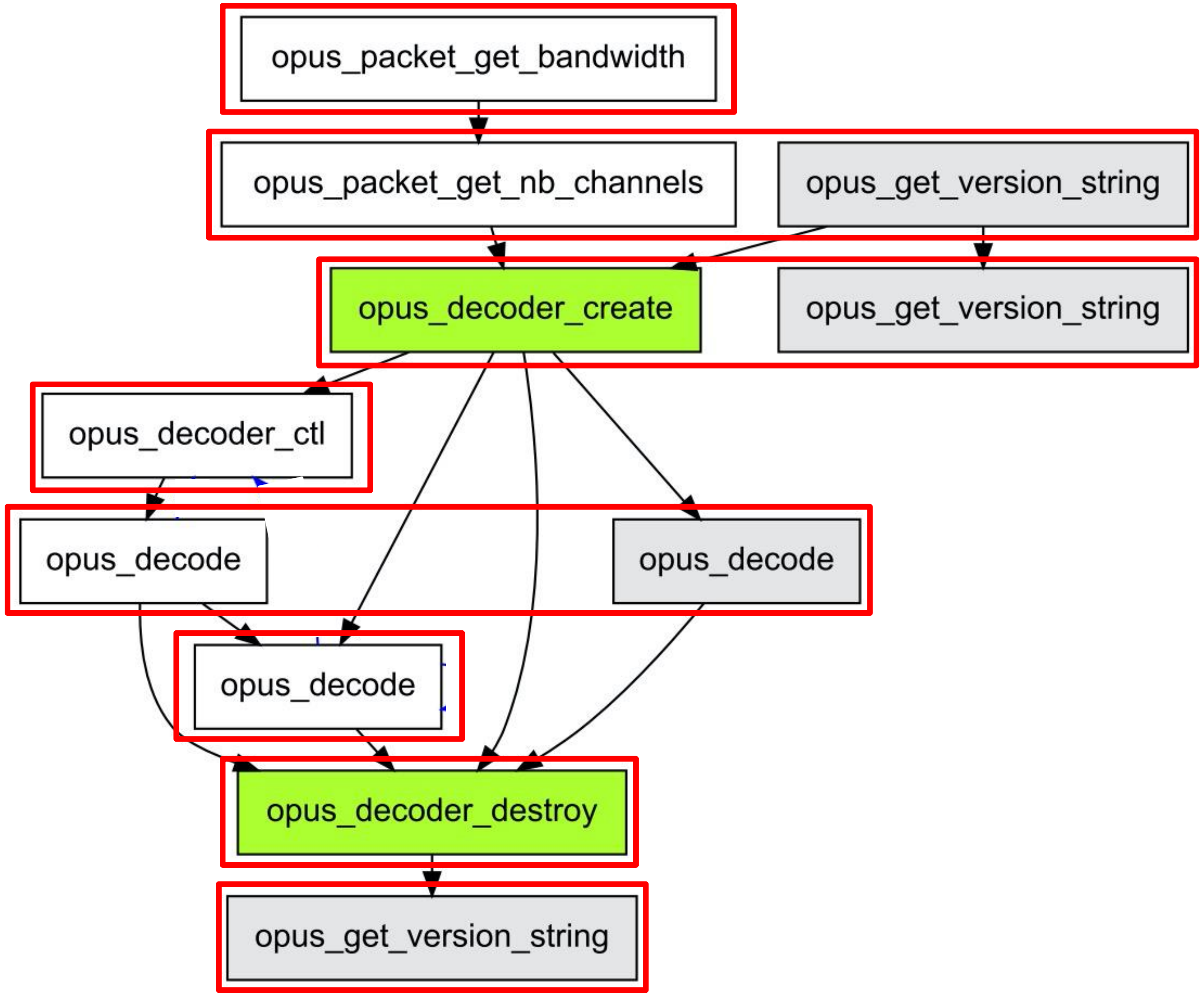
AADG Flattening

1. Remove back edges



AADG Flattening

1. Remove back edges
2. Topological sort
3. Group functions without total order



Evaluation

- FuzzGen was evaluated on AOSP and Debian.
- FuzzGen was compared against manually written libFuzzer stubs.
- Test artifacts are 7 widely deployed codec libraries.
- All tests were repeated for 5 times and fuzzing timeout was set to 24 hours.

Consumer Selection

- Fuzzers based on more consumers tend to include more functionality and more complexity.
- Merging too many consumers increases AADG complexity without introducing more interesting paths.
- Which set of consumers provide a representative set of API calls?

Consumer Selection

Consumers	API		A^2DG		
	Used	Found	Total	Nodes	Edges
0	0	0	1	0	0
1	6	34	1	7	12
2	6	34	1	9	14
3	10	34	1	16	22
4	12	34	1	24	30
5	25	51	1	142	289
6	31	51	2	148	303
7	33	65	2	181	438
8	44	65	1	540	1377
9	47	65	2	551	1393
10	50	65	2	611	1473
11	51	65	2	613	1475
12	53	65	2	697	1587
13	56	65	2	883	1773
14	56	65	2	885	1778
15	56	65	2	885	1778

Table 4: Complexity increase for the `libopus` library. **Consumers**: Total number of consumers used. **API**: **Used**: Total number of distinct API calls used in the final fuzzer. **Found**: Total number of distict API calls identified in headers. A^2DG: **Total**: Total number of A^2DG graphs produced (if coalescing is not possible there are more than one graphs). **Nodes** & **edges**: The total number of nodes and edges across all A^2DGs.

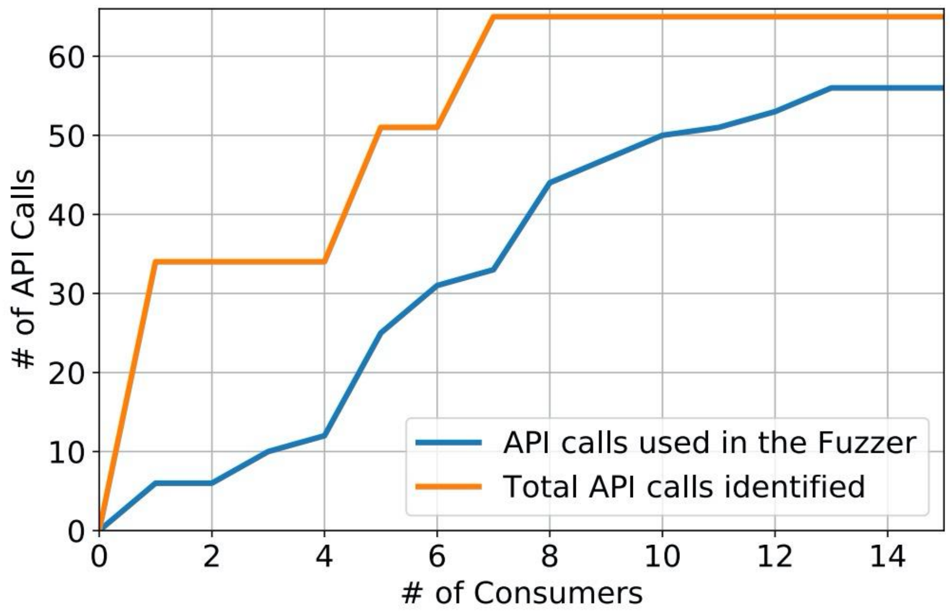


Figure 7: Consumer tail off for distinct API calls for `libopus` library.

Consumer Ranking

- Intuition: the number of API calls per LoC correlates to a relatively high usage of the target API.
- The heuristics for ranking consumers is called consumer density and is defined as follows:

$$\mathcal{D}_c \leftarrow \frac{\#\, distinct\ API\ calls}{Total\ lines\ of\ real\ code}$$

- Use 4 consumers demonstrates all features of FuzzGen and results in small fuzzers that are easy to verify.

Test Artifacts

	Library Information						Consumer Information					Final A^2DG			
	Name	**Type**	**Src Files**	**Total LoC**	**Funcs**	**API**	**Total**	**Used**	**Total LoC**	**Avg D_c**	**UAPI**	**Graphs**	**Coal.**	**Nodes**	**Edges**
Android	libhevc	video	303	113049	314	1	2	2	3880	0.002	1	10	5	29	58
	libavc	video	190	83942	581	1	2	2	4064	0.002	1	9	4	29	53
	libmpeg2	video	118	19828	179	1	2	2	4230	0.001	1	9	5	30	56
	libopus	audio	315	50983	276	65	23	4	1079	0.074	12	4	4	24	30
	libgsm	speech	41	6145	31	8	9	4	396	0.060	7	4	4	57	88
Deb	libvpx	video	1003	352691	1210	130	40	4	594	0.075	13	4	4	29	46
	libaom	video	955	399645	4232	86	39	4	491	0.106	17	4	4	40	51

Table 2: Codec libraries and consumers used in our evaluation. **Library Information**: **Src Files** = Number of source files, **Total LoC** = Total lines of code (without comments and blank lines), **Funcs** = Number of functions found in the library, **API** = Number of API functions. **Consumer Information**: **Total** = Total number of library consumers on the system, **Used** = Library consumers included in the evaluation, **Total LoC** = Total lines of code of all library consumers (without comments and blank lines), **Avg D_c** = Average consumer density, **UAPI** = Number of API functions used in the consumers. **Final** A^2DG: **Graphs** = Total number of A^2DGs, **Coalesced** = Number of nodes coalesced (same as the number of A^2DGs merges, since our algorithm uses a single node for merging), **Nodes**, **Edges** = Total number of nodes and edges (respectively) in the final A^2DG.

Measuring Code Coverage

- Use SanitizerCoverage to instrument test artifacts.

Evaluation Results

Library	Manual fuzzer information								FuzzGen fuzzer information								Difference		
	Total LoC	Edge Coverage (%)				Bugs Found		exec/sec	Total LoC	Edge Coverage (%)				Bugs Found		exec/sec	p	Cov	Bugs
		Max	Avg	Min	Std	T	U			Max	Avg	Min	Std	T	U				
libhevc	308	56.15	55.70	55.32	0.32	2493	23	83	1170	74.50	74.16	74.01	0.18	404	7	29	0.012	+18.46	-16
libavc	306	54.91	50.30	44.71	4.28	283	1	*8	1155	70.62	65.98	64.65	2.33	0	0	151	0.008	+15.68	-1
libmpeg2	457	51.39	49.59	45.42	2.14	1509	3	20	1204	56.95	56.60	56.26	0.26	6753	3	47	0.012	+7.01	0
libopus	125	15.85	15.71	15.16	0.27	0	0	174	624	39.99	35.22	32.63	3.08	110	3	218	0.012	+19.51	+3
libgsm	121	75.55	75.55	75.31	0.00	0	0	5966	490	69.40	68.20	67.40	0.77	229	1	4682	0.012	-7.35	+1
libvpx	122	54.79	54.13	53.61	0.49	0	0	63	481	52.17	50.99	48.05	1.52	464652	1	2060	0.012	-3.14	+1
libaom	69	44.54	35.03	30.40	5.12	57	2	111	1132	41.10	33.43	25.96	5.87	75	2	166	0.674	-1.60	0

Table 3: Results from fuzzer evaluation on codec libraries. We run each fuzzer 5 times. **Total LoC** = Total lines of fuzzer code, **Edge Coverage** % = edge coverage (**max**: maximum covarage from best run, **avg**: average coverage across all runs, **min**: maximum coverage from the worst run, **std**: standard deviation of the coverage), **Bugs found** = Number of total (**T**) and unique (**U**) bugs found, **exec/sec** = Average executions per second (from all runs), **Difference** = The difference between FuzzGen and manual fuzzers (p value from Mann-Whitney U test, unique bugs and maximum edge coverage). *The executions per second in this case are low because all 283 discovered bugs are timeouts.

Evaluation Results: Coverage

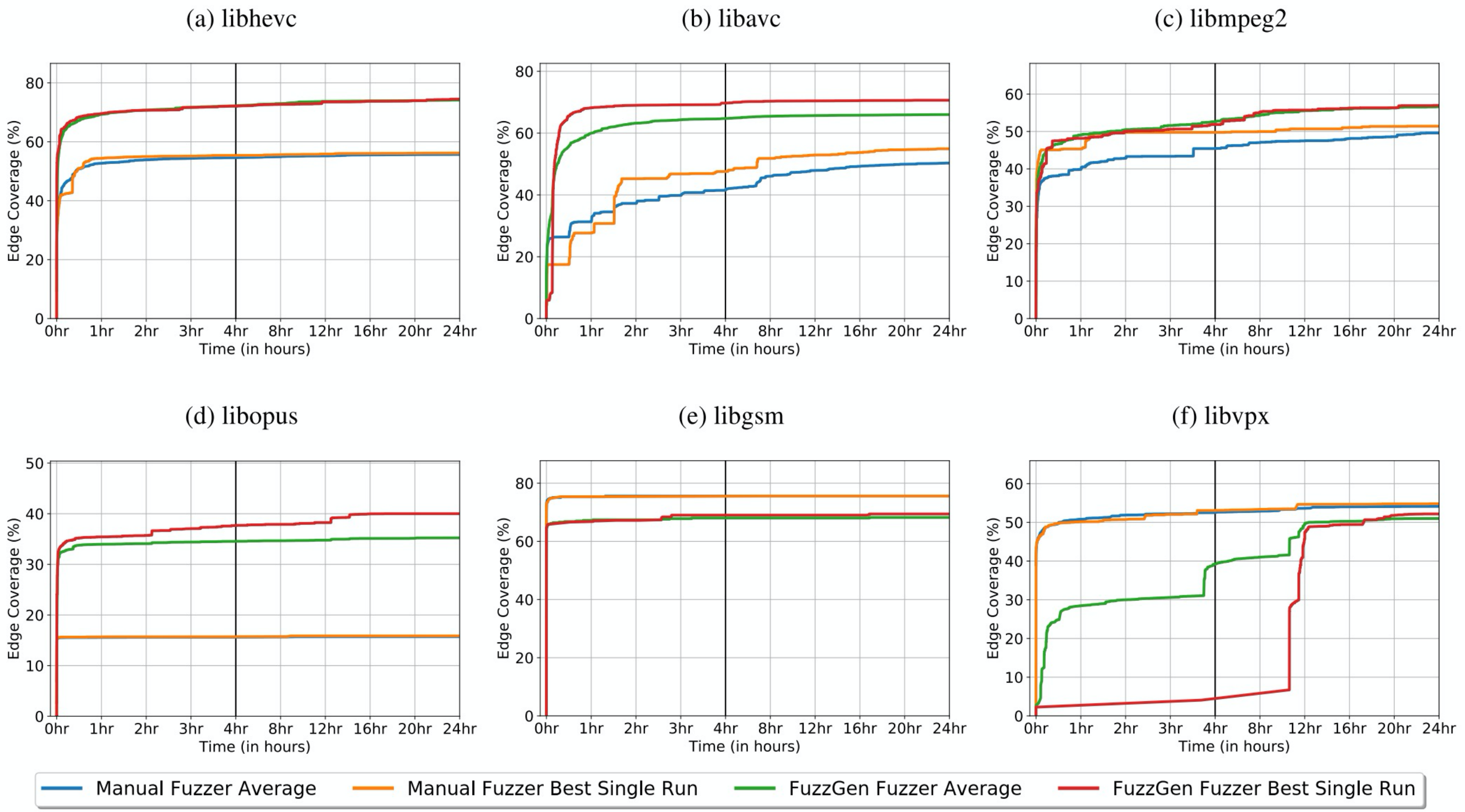


Figure 5: Code coverage (%) over time for each library. The blue line shows the average edge coverage over time for manual fuzzers and the orange line shows the edge coverage for the best single run (among the five) for manual fuzzers. Similarly, the green line shows the average edge coverage for FuzzGen fuzzers, and the red line the edge coverage from best single run for FuzzGen fuzzers.

Evaluation Results: Coverage

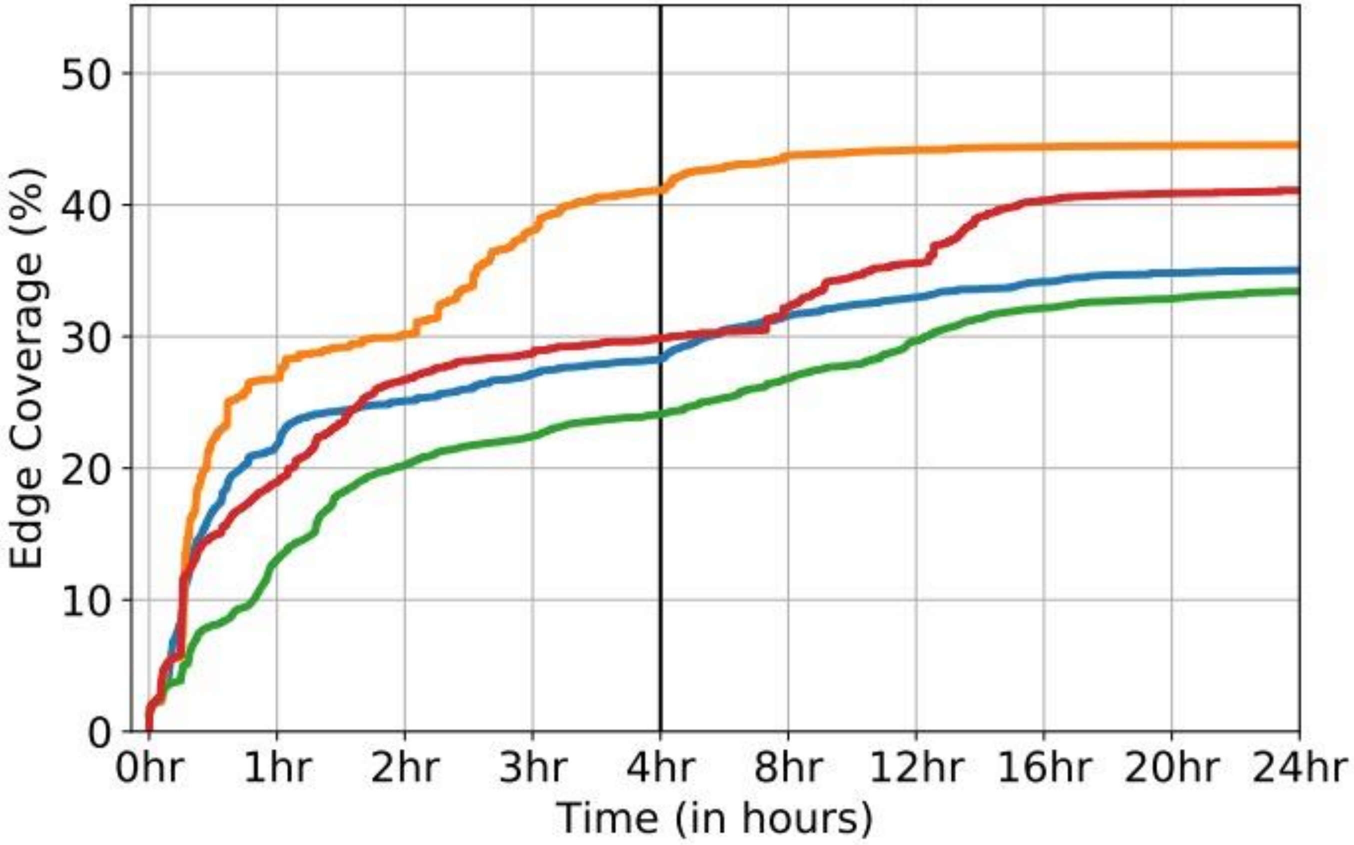


Figure 6: Code coverage over time for libaom.

Evaluation Observation

- Manual fuzzers are smaller in size and more targeted.
- Due to the focus on a single component, manual fuzzers can find more bugs in that component compared to FuzzGen fuzzers.
- FuzzGen fuzzers are broader and can achieve higher code coverage. However, this imposes performance penalty.
- In the long run, FuzzGen fuzzers are able to explore a broader program surface.

Related Works

- FUDGE. Leverages code slicing to automatically generate fuzzer stubs.

Future Works

- Maximum code coverage.
- Single library focus.
- Coalescing dependence graphs into a unifying AADG.
- False positives.

Contribution

- Design of a *whole system analysis* that infers valid API interactions for a given library based on existing programs and libraries that use the target library — abstracting the information into an Abstract API Dependence Graph (AADG);
- Based on the AADG, FuzzGen creates libFuzzer stubs that construct complex program state to expose vulnerabilities in deep library functions was developed — fuzzers are generated without human interaction;
- Evaluation of the prototype on AOSP and Debian demonstrates the effectiveness and the generality of the FuzzGen technique. Generating fuzzers for 7 libraries, FuzzGen discovered 17 bugs. The generated fuzzers achieve 54.94% code coverage on average, compared to 48.00% that fuzzer stubs — written manually by experts — achieve.