Effective file format fuzzing

Thoughts, techniques and results

Mateusz “j00ru” Jurczyk

Black Hat Europe 2016, London

PS> whoami

- Project Zero @ Google
  - Part time developer and frequent user of the fuzzing infrastructure.
- Dragon Sector CTF team vice captain.
- Low-level security researcher with interest in all sorts of vulnerability research and software exploitation.
- http://j00ru.vexillium.org/
- @j00ru

Agenda

- What constitutes real-life offensive fuzzing (techniques and mindset).
- How each of the stages is typically implemented and how to improve them for maximized effectiveness.
  - Tips & tricks on the examples of software I've fuzzed during the past few years: Adobe Reader, Adobe Flash, Windows Kernel, Oracle Java, Hex-Rays IDA Pro, FreeType2, FFmpeg, pdfium, Wireshark, ...

Fuzzing

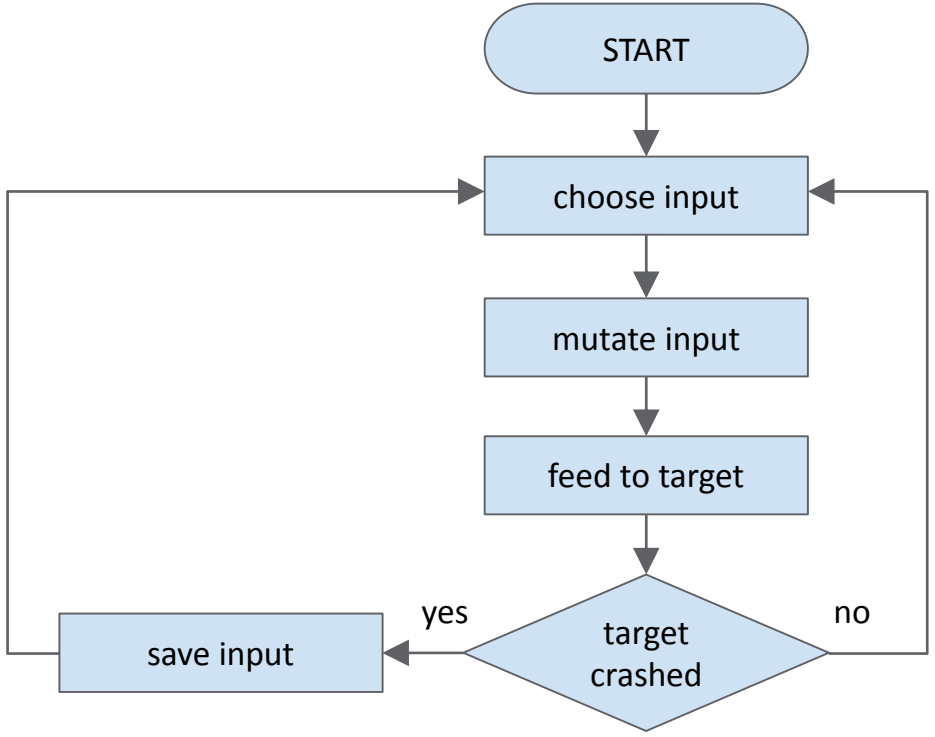
Fuzz testing *or* ***fuzzing*** *is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program.*

http://en.wikipedia.org/wiki/Fuzz_testing

In my (and this talk's) case

- **Software** = commonly used programs and libraries, both open and closed-source, written in native languages (C/C++ etc.), which may be used as targets for memory corruption-style 0-day attacks.
- **Inputs** = files of different (un)documented formats processed by the target software (e.g. websites, applets, images, videos, documents etc.).

On a scheme


START
choose input
mutate input
feed to target
target crashed
yes
no
save input

Easy to learn, hard to master.

Key questions

- How do we choose the fuzzing target in the first place?
- How are the inputs generated?
- What is the base set of the input samples? Where do we get it from?
- How do we mutate the inputs?
- How do we detect software failures / crashes?
- Do we make any decisions in future fuzzing based on the software's behavior in the past?
- How do we minimize the interesting inputs / mutations?
- How do we recognize *unique* bugs?
- What if the software requires user interaction and/or displays windows?
- What if the application keeps crashing at a single location due to an easily reachable bug?
- What if the fuzzed file format includes checksums, other consistency checks, compression or encryption?

Let’s get technical.

Gathering an initial corpus of input files

- A desired step in a majority of cases:
  - Makes it possible to reach some code paths and program states immediately after starting the fuzzing.
  - May contain complex data structures which would be difficult or impossible to generate *organically* using just code coverage information, e.g. magic values, correct headers, compression trees etc.
  - Even if the same inputs could be constructed during fuzzing with an empty seed, having them right at the beginning saves a lot of CPU time.
  - Corpora containing files in specific formats may be frequently reused to fuzz various software projects which handle them.

Gathering inputs: the standard methods

- Open-source projects often include extensive sets of input data for testing, which can be freely reused as a fuzzing starting point.
  - Example: FFmpeg FATE, samples.ffmpeg.org. Lots of formats there, which would be otherwise very difficult to obtain in the wild.
  - Sometimes they're not publicly available for everyone, but the developers have them and will share with someone willing to report bugs in return.
- Many of them also include converters from format X to their own format Y. With a diverse set of files in format X and/or diverse conversion options, this can also generate a decent corpus.
  - Example: `cwebp`, a converter from PNG/JPEG/TIFF to WEBP images.

Gathering inputs: Internet crawling

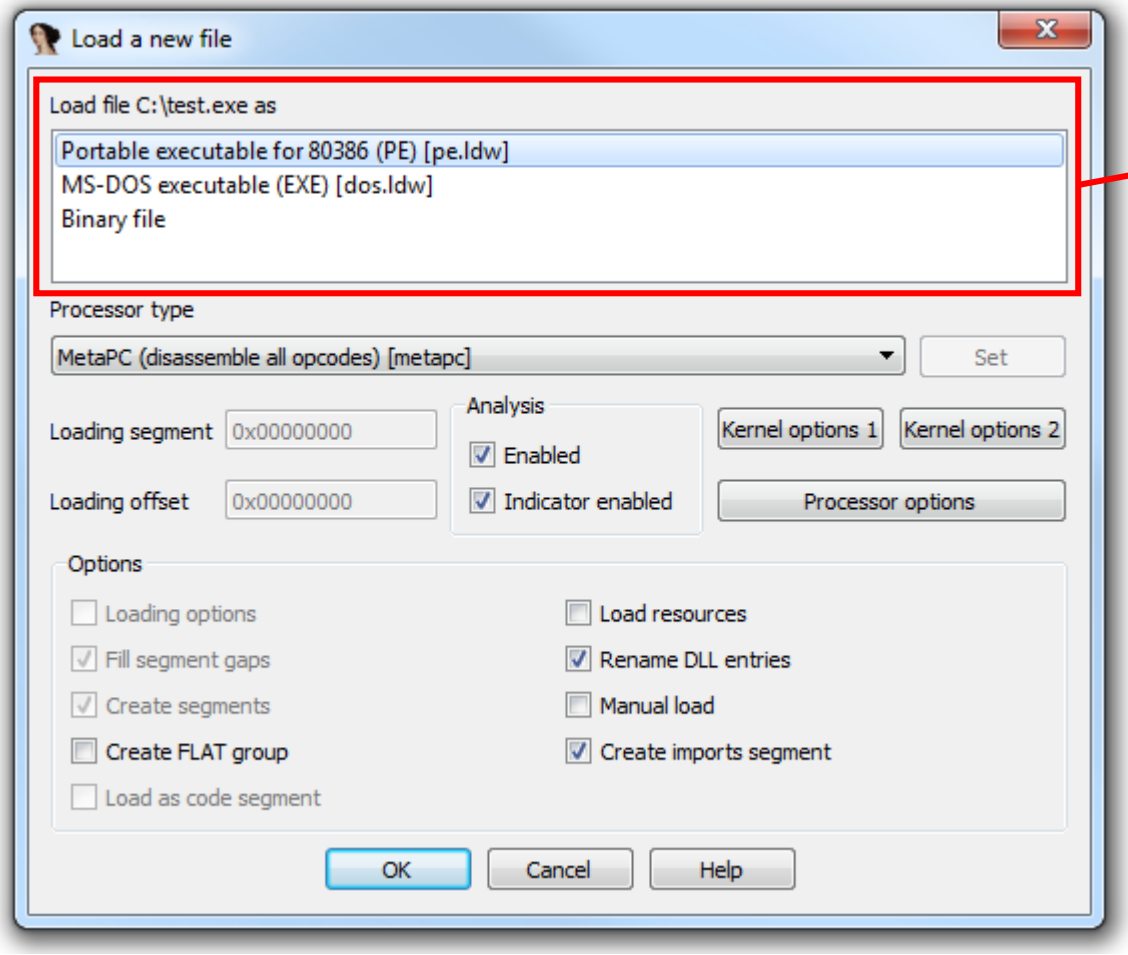
- Depending on the popularity of the fuzzed file format, Internet crawling is the most intuitive approach.
  - Download files with a specific file extension.
  - Download files with specific magic bytes or other signatures.
- If the format is indeed popular (e.g. DOC, PDF, SWF etc.), you may end up with many terabytes of data on your disk.
  - Not a huge problem, since storage is cheap, and the corpus can be later minimized to consume less space while providing equivalent code coverage.

You may also ask what the program thinks

- Things can get a bit dire if you plan to fuzz a program which supports dozens of different formats.
  - Code coverage analysis is of course a good idea, but it tends to slow down the process considerably (esp. for closed-source software).
  - In some cases, you can use the target itself to tell you if a given file can be handled by it or not.
- Case study: IDA Pro.

IDA Pro supported formats (partial list)

MS DOS, EXE File, MS DOS COM File, MS DOS Driver, New Executable (NE), Linear Executable (LX), Linear Executable (LE), Portable Executable (PE) (x86, x64, ARM), Windows CE PE (ARM, SH-3, SH-4, MIPS), MachO for OS X and iOS (x86, x64, ARM and PPC), Dalvik Executable (DEX), EPOC (Symbian OS executable), Windows Crash Dump (DMP), XBOX Executable (XBE), Intel Hex Object File, MOS Technology Hex Object File, Netware Loadable Module (NLM), Common Object File Format (COFF), Binary File, Object Module Format (OMF), OMF library, S-record format, ZIP archive, JAR archive, Executable and Linkable Format (ELF), Watcom DOS32 Extender (W32RUN), Linux a.out (AOUT), PalmPilot program file, AIX ar library (AIAFF), PEF (Mac OS or Be OS executable), QNX 16 and 32-bits, Nintendo (N64), SNES ROM file (SMC), Motorola DSP56000 .LOD, Sony Playstation PSX executable files, object (psyq) files, library (psyq) files



How does it work?

IDA Pro loader architecture

- Modular design, with each loader (also disassembler) residing in a separate module, exporting two functions: `accept_file` and `load_file`.
  - One file for the 32-bit version of IDA (.llx on Linux) and one file for 64-bit (.llx64).

```
$ ls loaders
aif64.llx64       coff64.llx64   epoc.llx          javaldr64.llx64  nlm64.llx64     pilot.llx        snes_spc.llx
aif.llx           coff.llx       exploa64.llx64    javaldr.llx      nlm.llx         psx64.llx64      uimage.py
amiga64.llx64     dex64.llx64    exploa.llx        lx64.llx64       omf64.llx64     psx.llx          w32run64.llx64
amiga.llx         dex.llx        geos64.llx64      lx.llx           omf.llx         qnx64.llx64      w32run.llx
aof64.llx64       dos64.llx64    geos.llx          macho64.llx64    os964.llx64     qnx.llx          wince.py
aof.llx           dos.llx        hex64.llx64       macho.llx        os9.llx         rt1164.llx64     xbe64.llx64
aout64.llx64      dsp_lod.py     hex.llx           mas64.llx64      pdfldr.py       rt11.llx         xbe.llx
aout.llx          dump64.llx64   hppacore.idc      mas.llx          pe64.llx64      sbn64.llx64
bfltldr.py        dump.llx       hpsom64.llx64     n6464.llx64      pef64.llx64     sbn.llx
bios_image.py     elf64.llx64    hpsom.llx         n64.llx          pef.llx         snes64.llx64
bochsrc64.llx64   elf.llx        intelomf64.llx64  ne64.llx64       pe.llx          snes.llx
bochsrc.llx       epoc64.llx64   intelomf.llx      ne.llx           pilot64.llx64   snes_spc64.llx64
```

IDA Pro loader architecture

```
int (idaapi* accept_file)(linput_t *li,
                          char fileformatname[MAX_FILE_FORMAT_NAME],
                          int n);

void (idaapi* load_file)(linput_t *li,
                         ushort neflags,
                         const char *fileformatname);
```

- The `accept_file` function performs preliminary processing and returns 0 or 1 depending on whether the given module thinks it can handle the input file as N^{th} of its supported formats.
  - If so, returns the name of the format in the `fileformatname` argument.
- `load_file` performs the regular processing of the file.
- Both functions (and many more required to interact with IDA) are documented in the IDA SDK.

Easy to write an IDA loader enumerator

```
$ ./accept_file accept_file
[+] 35 loaders found.
[-]          os9.llx: format not recognized.
[-]          mas.llx: format not recognized.
[-]           pe.llx: format not recognized.
[-]     intelomf.llx: format not recognized.
[-]        macho.llx: format not recognized.
[-]           ne.llx: format not recognized.
[-]         epoc.llx: format not recognized.
[-]          pef.llx: format not recognized.
[-]          qnx.llx: format not recognized.
…
[-]        amiga.llx: format not recognized.
[-]        pilot.llx: format not recognized.
[-]          aof.llx: format not recognized.
[-]      javaldr.llx: format not recognized.
[-]          n64.llx: format not recognized.
[-]          aif.llx: format not recognized.
[-]         coff.llx: format not recognized.
[+]          elf.llx: accept_file recognized as "ELF for Intel 386 (Executable)"
```

Asking the program for feedback

- Thanks to the design, we can determine if a file can be loaded in IDA:
  - with a very high degree of confidence.
  - exactly by which loader, and treated as which file format.
  - without ever starting IDA, or even requiring any of its files other than the loaders.
  - without using any instrumentation, which together with the previous point speeds things up significantly.
- Similar techniques could be used for any software which makes it possible to run some preliminary validation instead of fully fledged processing.

Corpus distillation

- In fuzzing, it is important to get rid of most of the redundancy in the input corpus.
  - Both the base one and the *living* one evolving during fuzzing.
  - In the context of a single test case, the following should be maximized:

$$\frac{|program\ states\ explored|}{input\ size}$$

which strives for the highest byte-to-program-feature ratio: each portion of a file should exercise a new functionality, instead of repeating constructs found elsewhere in the sample.

Corpus distillation

- Likewise, in the whole corpus, the following should be generally maximized:

$$\frac{|program\ states\ explored|}{|input\ samples|}$$

This ensures that there aren't too many samples which all exercise the same functionality (enforces program state diversity while keeping the corpus size relatively low).

Format specific corpus minimization

- If there is too much data to thoroughly process, and the format is easy to parse and recognize (non-)interesting parts, you can do some cursory filtering to extract unusual samples or remove dull ones.
  - Many formats are structured into chunks with unique identifiers: SWF, PDF, PNG, JPEG, TTF, OTF etc.
  - Such generic parsing may already reveal if a file will be a promising fuzzing candidate or not.
  - The deeper into the specs, the more work is required. It’s usually not cost-effective to go beyond the general file structure, given other (better) methods of corpus distillation.
  - Be careful not to reduce out interesting samples which only appear to be boring at first glance.

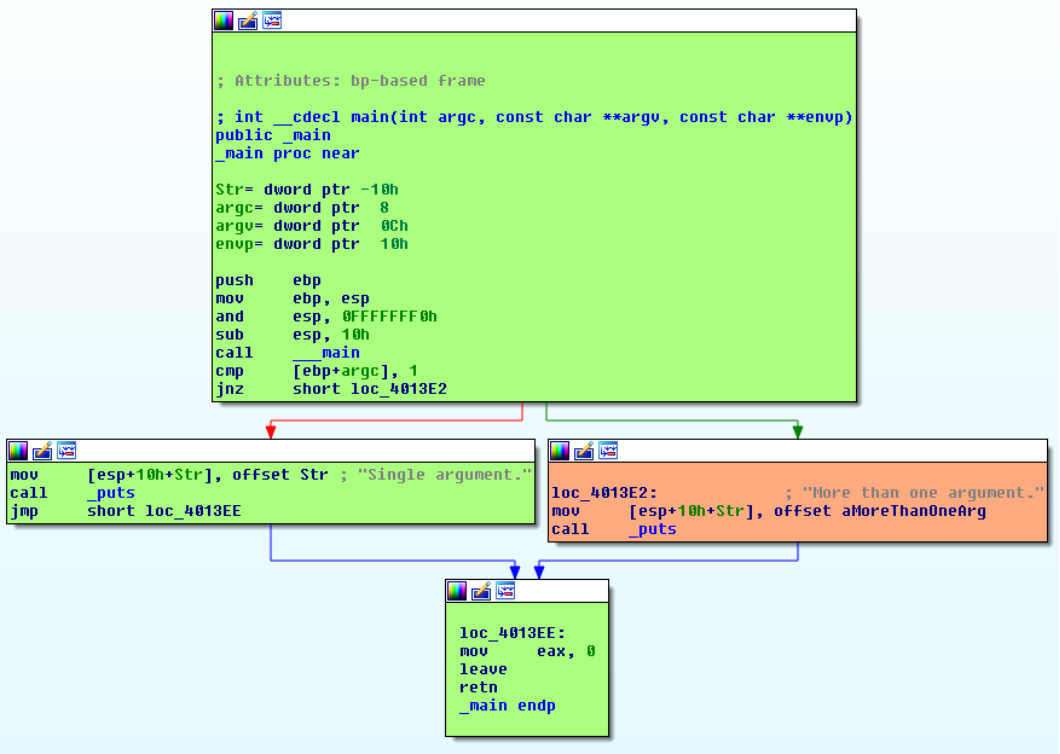
How to define a *program state*?

- File sizes and cardinality (from the previous expressions) are trivial to measure.
- There doesn't exist such a simple metric for *program states*, especially with the following characteristics:
  - their number should stay within a sane range, e.g. counting all combinations of every bit in memory cleared/set is not an option.
  - they should be meaningful in the context of memory safety.
  - they should be easily/quickly determined during process run time.

$Code\ coverage \cong program\ states$

- Most approximations are currently based on measuring code coverage, and not the actual memory state.
  - Pros:
    - Increased code coverage is representative of new program states. In fuzzing, the more tested code is executed, the higher chance for a bug to be found.
    - The sane range requirement is met: code coverage information is typically linear in size in relation to the overall program size.
    - Easily measurable using both compiled-in and external instrumentation.
  - Cons:
    - Constant code coverage does not indicate constant $|program\ states|$. A significant amount of information on distinct states may be lost when only using this metric.

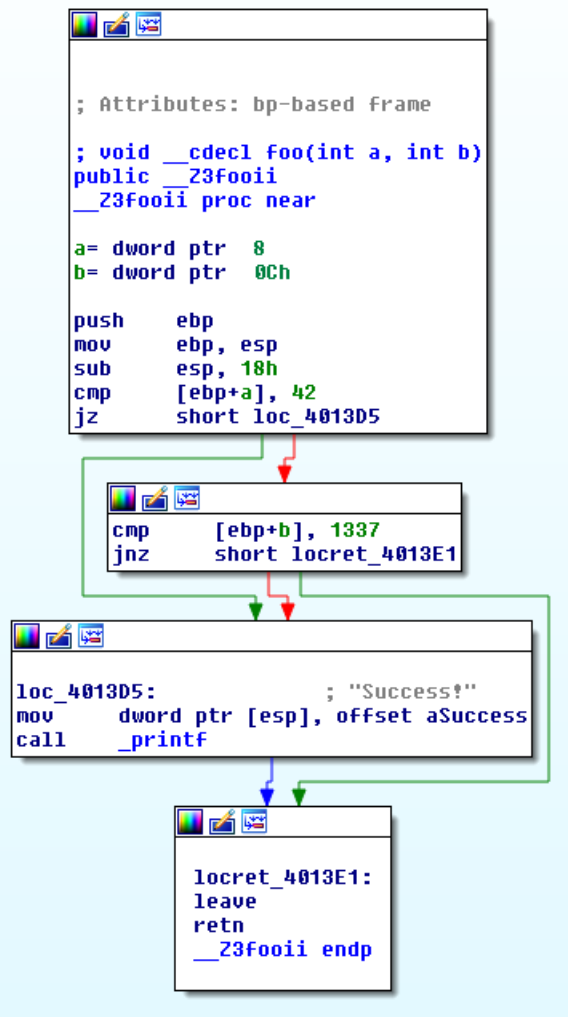
Current state of the art: counting basic blocks



- Basic blocks provide the best granularity.
  - Smallest coherent units of execution.
  - Measuring just functions loses lots of information on what goes on inside.
  - Recording specific instructions is generally redundant, since all of them are guaranteed to execute within the same basic block.
- Supported in both compiler (gcov etc.) and external instrumentations (Intel Pin, DynamoRIO).
- Identified by the address of the first instruction.

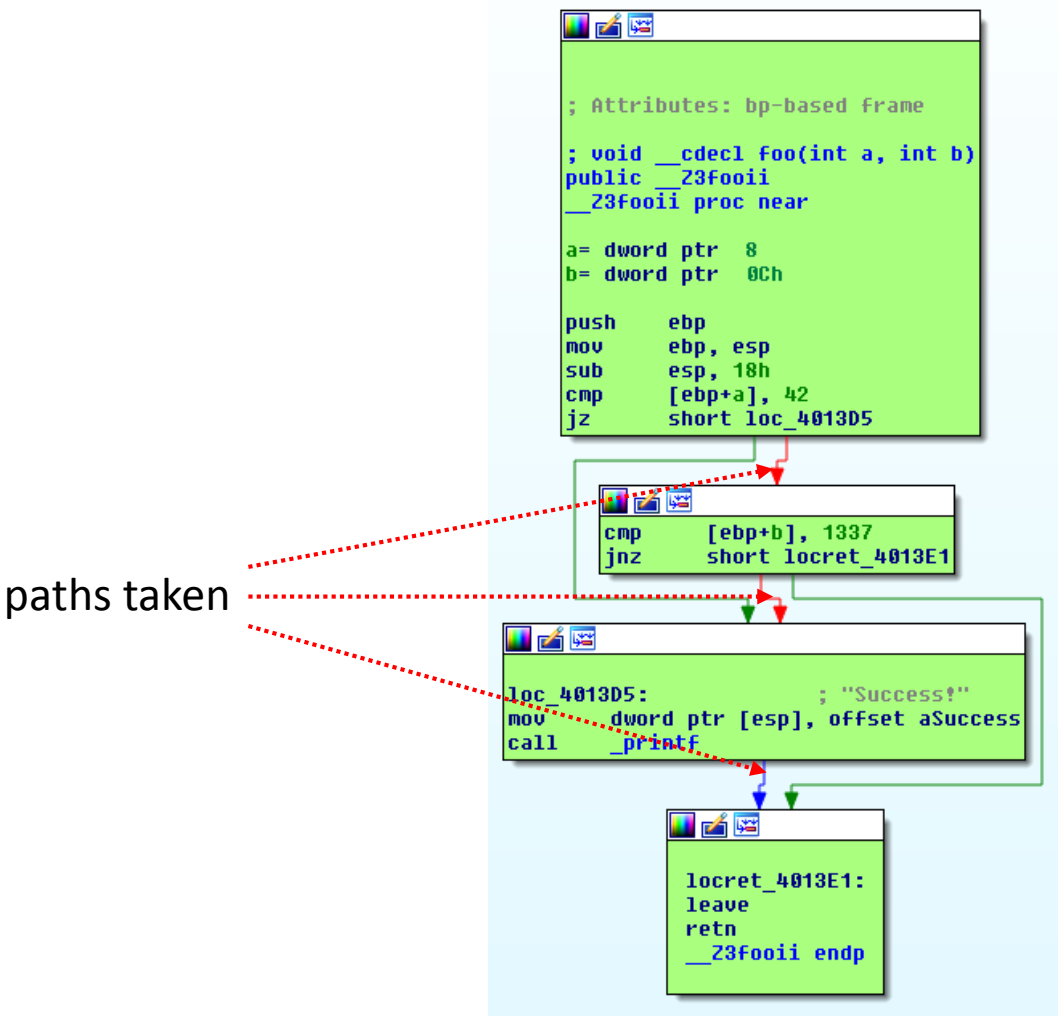
Basic blocks: incomplete information

```
void foo(int a, int b) {
  if (a == 42 || b == 1337) {
    printf("Success!");
  }
}

void bar() {
  foo(0, 1337);
  foo(42, 0);
  foo(0, 0);
}
```



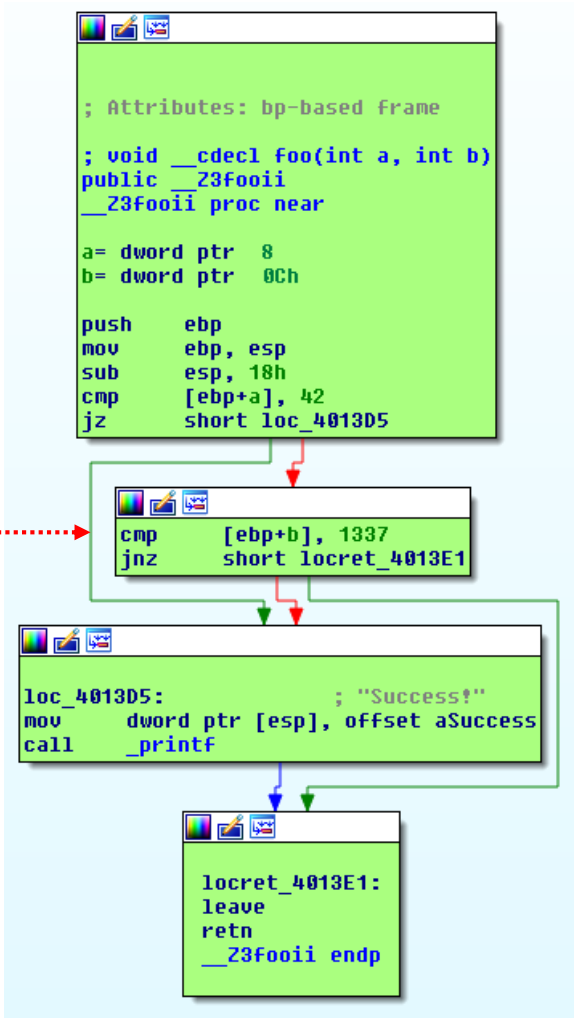
Basic blocks: incomplete information

```
void foo(int a, int b) {
  if (a == 42 || b == 1337) {
    printf("Success!");
  }
}

void bar() {
  foo(0, 1337);
  foo(42, 0);
  foo(0, 0);
}
```



Basic blocks: incomplete information

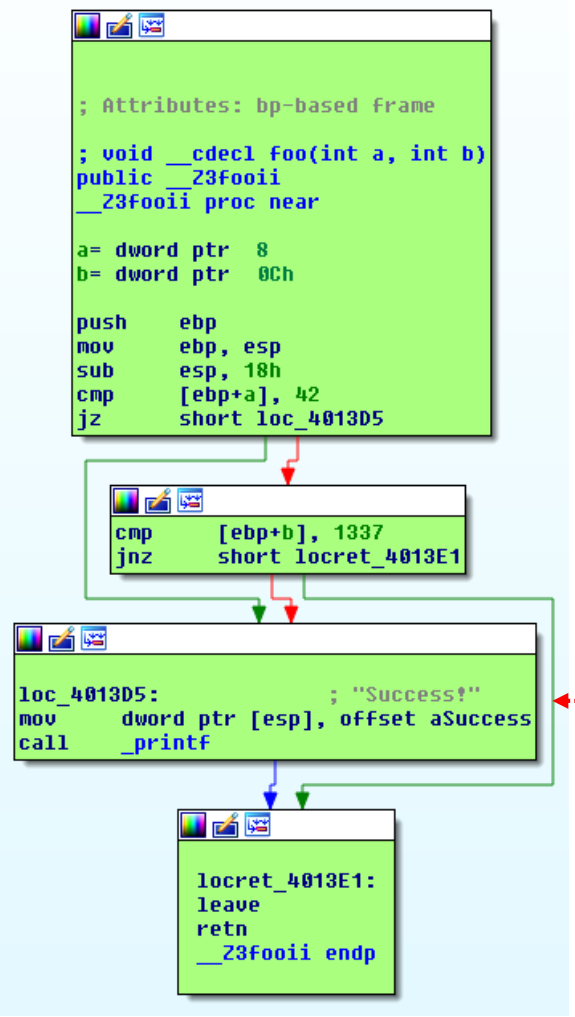
```
void foo(int a, int b) {
  if (a == 42 || b == 1337) {
    printf("Success!");
  }
}

void bar() {
  foo(0, 1337);
  foo(42, 0);
  foo(0, 0);
}
```



new path

Basic blocks: incomplete information

```
void foo(int a, int b) {
  if (a == 42 || b == 1337) {
    printf("Success!");
  }
}

void bar() {
  foo(0, 1337);
  foo(42, 0);
  foo(0, 0);
}
```



Basic blocks: incomplete information

- Even though the two latter `foo()` calls take different paths in the code, this information is not recorded and lost in a simple BB granularity system.
  - Arguably they constitute new *program states* which could be useful in fuzzing.
- Another idea – program interpreted as a graph.
  - vertices = basic blocks
  - edges = transition paths between the basic blocks
  - Let’s record edges rather then vertices to obtain more detailed information on the control flow!

AFL the first to introduce and ship this at large

- From lcamtuf's technical whitepaper:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- Implemented in the fuzzer's own custom instrumentation.

Extending the idea even further

- In a more abstract sense, recording edges is recording the current block + one previous.
  - What if we recorded N previous blocks instead of just 1?
  - Provides even more context on the program state at a given time, and how execution arrived at that point.
  - Another variation would be to record the function call stacks at each basic block.
- In my experience $N = 1$ (direct edges) has worked very well, but more experimentation is required and encouraged.

Counters and bitsets

- Let's abandon the "basic block" term and use "trace" for a single unit of code coverage we are capturing (functions, basic blocks, edges, etc.).
- In the simplest model, each trace only has a Boolean value assigned in a coverage log: REACHED or NOTREACHED.
- More useful information can be found in the specific, or at least more precise number of times it has been hit.
  - Especially useful in case of loops, which the fuzzer could progress through by taking into account the number of iterations.
  - Implemented in AFL, as shown in the previous slide.
  - Still not perfect, but allows some more granular information related to $|program\ states|$ to be extracted and used for guiding.

Extracting all this information

- For closed-source programs, all aforementioned data can be extracted by some simple logic implemented on top of Intel Pin or DynamoRIO.
  - AFL makes use of modified qemu-user to obtain the necessary data.
- For open-source, the gcc and clang compilers offer some limited support for code coverage measurement.
  - Look up gcov and llvm-cov.
  - I had trouble getting them to work correctly in the past, and quickly moved to another solution…
- … SanitizerCoverage!

Enter the SanitizerCoverage

- Anyone remotely interested in open-source fuzzing must be familiar with the mighty AddressSanitizer.
  - Fast, reliable C/C++ instrumentation for detecting memory safety issues for clang and gcc (mostly clang).
  - Also a ton of other run time sanitizers by the same authors: MemorySanitizer (use of uninitialized memory), ThreadSanitizer (race conditions), UndefinedBehaviorSanitizer, LeakSanitizer (memory leaks).
- A definite must-use tool, compile your targets with it whenever you can.

Enter the SanitizerCoverage

- ASAN, MSAN and LSAN together with SanitizerCoverage can now also record and dump code coverage at a very small overhead, in all the different modes mentioned before.
  - Thanks to the combination of a sanitizer and coverage recorder, you can have both error detection and coverage guidance in your fuzzing session at the same time.
- LibFuzzer, Kostya's own fuzzer, also uses SanitizerCoverage (via the in-process programmatic API).

SanitizerCoverage usage

```
% cat -n cov.cc
     1  #include <stdio.h>
     2  __attribute__((noinline))
     3  void foo() { printf("foo\n"); }
     4
     5  int main(int argc, char **argv) {
     6    if (argc == 2)
     7      foo();
     8    printf("main\n");
     9  }

% clang++ -g cov.cc -fsanitize=address -fsanitize-coverage=func

% ASAN_OPTIONS=coverage=1 ./a.out; ls -l *sancov
main
-rw-r----- 1 kcc eng 4 Nov 27 12:21 a.out.22673.sancov

% ASAN_OPTIONS=coverage=1 ./a.out foo ; ls -l *sancov
foo
main
-rw-r----- 1 kcc eng 4 Nov 27 12:21 a.out.22673.sancov
-rw-r----- 1 kcc eng 8 Nov 27 12:21 a.out.22679.sancov
```

So, we can measure coverage easily.

- Just measuring code coverage isn't a silver bullet by itself (sadly).
  - But still extremely useful, even the simplest implementation is better then no coverage guidance.
- There are still many code constructs which are impossible to cross with a dumb mutation-based fuzzing.
  - One-instruction comparisons of types larger than a byte (uint32 etc.), especially with magic values.
  - Many-byte comparisons performed in loops, e.g. `memcmp()`, `strcmp()` calls etc.

Hard code constructs: examples

```
uint32_t value = load_from_input();
if (value == 0xDEADBEEF) {
  // Special branch.
}
```

Comparison with a 32-bit constant value

```
char buffer[32];
load_from_input(buffer, sizeof(buffer));

if (!strcmp(buffer, "Some long expected string")) {
  // Special branch.
}
```

Comparison with a long fixed string

The problems are somewhat approachable

- Constant values and strings being compared against may be hard in a completely context-free fuzzing scenario, but are easy to defeat when some program/format-specific knowledge is considered.
  - Both AFL and LibFuzzer support “dictionaries”.
  - A dictionary may be created manually by feeding all known format signatures, etc.
    - Can be then easily reused for fuzzing another implementation of the same format.
  - Can also be generated automatically, e.g. by disassembling the target program and recording all constants used in instructions such as:

```
cmp r/m32, imm32
```

Compiler flags may come helpful… or not

- A somewhat intuitive approach to building the target would be to disable all code optimizations.
  - Fewer hacky expressions in assembly, compressed code constructs, folded basic blocks, complicated RISC-style x86 instructions etc. → more granular coverage information to analyze.
  - On the contrary, lcamtuf discovered that using `-O3 -funroll-loops` may result in unrolling short fixed-string comparisons such as `strcmp(buf, "foo")` to:

```
cmpb   $0x66,0x200c32(%rip)        # 'f'
jne    4004b6
cmpb   $0x6f,0x200c2a(%rip)        # 'o'
jne    4004b6
cmpb   $0x6f,0x200c22(%rip)        # 'o'
jne    4004b6
cmpb   $0x0,0x200c1a(%rip)         # NUL
jne    4004b6
```

- It is quite unclear which compilation flags are most optimal for coverage-guided fuzzing.
  - Probably depends heavily on the nature of the tested software, requiring case-by-case adjustments.

Past encounters

- In 2009, Tavis Ormandy also presented some ways to improve the effectiveness of coverage guidance by challenging complex logic hidden in single x86 instructions.
  - "Deep Cover Analysis", using sub-instruction profiling to calculate a score depending on how far the instruction progressed into its logic (e.g. how many bytes `repz cmpb` has successfully compared, or how many most significant bits in a `cmp r/m32, imm32` comparison match).
  - Implemented as an external DBI in Intel PIN, working on compiled programs.
  - Shown to be sufficiently effective to reconstruct correct crc32 checksums required by PNG decoders with zero knowledge of the actual algorithm.

Ideal future

- From a fuzzing perspective, it would be perfect to have a dedicated compiler emitting code with the following properties:
  - Assembly being maximally simplified (in terms of logic), with just CISC-style instructions and as many code branches (corresponding to branches in actual code) as possible.
  - Only enabled optimizations being the fuzzing-friendly ones, such as loop unrolling.
  - Every comparison on a type larger than a byte being split to byte-granular operations.
    - Similarly to today's JIT mitigations.

Ideal future

```
cmp dword [ebp+variable], 0xaabbccdd
jne not_equal
```

```
cmp byte [ebp+variable], 0xdd
jne not_equal
cmp byte [ebp+variable+1], 0xcc
jne not_equal
cmp byte [ebp+variable+2], 0xbb
jne not_equal
cmp byte [ebp+variable+3], 0xaa
jne not_equal
```

Circumventing Fuzzing Roadblocks with Compiler Transformations

Posted on August 15, 2016 by lafintel

TL;DR: We build some LLVM passes which 'deoptimize' code generated by LLVM to increase code coverage with AFL (and potentially other feedback driven fuzzers, e.g. libFuzzer). Get the code here.

Ideal future

- Standard comparison functions (`strcmp`, `memcmp` etc.) are annoying, as they hide away all the meaningful state information.
- Potential compiler-based solution:
  - Use extremely unrolled implementations of these functions, with a separate branch for every N up to e.g. 4096.
  - Compile in a separate instance of them for each call site.
    - would require making sure that no generic wrappers exist which hide the real caller.
    - still not perfect against functions which just compare memory passed by their callers by design, but a good step forward nevertheless.

Unsolvable problems

- There are still some simple constructs which cannot be crossed by a simple coverage-guided fuzzer:

```
uint32_t value = load_from_input();
if (value * value == 0x3a883f11) {
  // Special branch.
}
```

- Previously discussed *deoptimizations* would be ineffective, since all bytes are dependent on each other (you can't brute-force them one by one).
- That's basically where SMT solving comes into play, but this talk is about dumb fuzzing.

We have lots of input files, compiled target and ability to measure code coverage.

What now?

Corpus management system

- One could want a coverage-guided corpus management system, which would be used before fuzzing:
  - to minimize an initial corpus of potentially gigantic sizes to a smaller, yet equally valuable one.
    - Input = N input files (for unlimited N)
    - Output = M input files and information about their coverage (for a reasonably small M)
    - Should be scalable.

Corpus management system

- And during fuzzing:
  - to decide if a mutated sample should be added to the corpus, and recalculate it if needed:
    - Input = current corpus and its coverage, candidate samples and its coverage.
    - Output = new corpus and its coverage (unmodified, or modified to include the candidate sample).
  - to merge two corpora into a single optimal one.

Prior work

- Corpus distillation resembles the *Set cover problem*, if we wanted to find the smallest sub-collection of samples with coverage equal to that of the entire set.
  - The exact problem is NP-hard, so calculating the optimal solution is beyond possible for the data we operate on.
  - But we don't really need to find the optimal solution. In fact, it's probably better if we don't.
  - There are polynomial greedy algorithms for finding $\log_n$ approximates.

Prior work

Example of a simple greedy algorithm:

1. At each point in time, store the current corpus and coverage.
2. For each new sample X, check if it adds at least one new trace to the coverage. If so, include it in the corpus.
3. (Optional) Periodically check if some samples are redundant and the total coverage doesn’t change without them; remove them if so.

Prior work – drawbacks

- Doesn’t scale at all – samples need to be processed sequentially.
- The size and form of the corpus depends on the order in which inputs are processed.
  - We may end up with some unnecessarily large files in the final set, which is suboptimal.
- Very little control over the volume–redundancy trade-off in the output corpus.

My proposed design

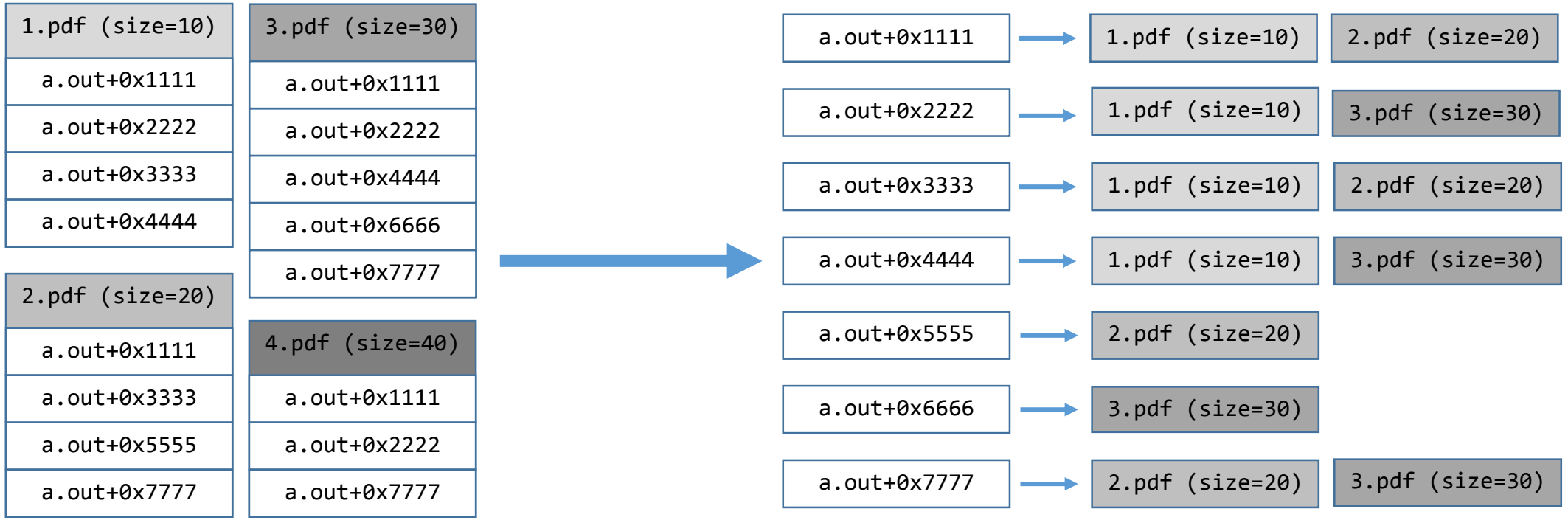
For each execution trace we know, store N smallest samples which reach that trace.

The corpus consists of all files present in the structure.

In other words, we maintain a `map<string, set<pair<string, int>>>` object:

$$trace\ idi \rightarrow \{(sample\ id_1, size_1), (sample\ id_2, size_2), \ldots, (sample\ idN, sizeN)\}$$

Proposed design illustrated (N=2)



Key advantages

1. Can be trivially parallelized and run with any number of machines using the MapReduce model.
2. The extent of redundancy (and thus corpus size) can be directly controlled via the N parameter.
3. During fuzzing, the corpus will evolve to gradually minimize the average sample size by design.
4. There are at least N samples which trigger each trace, which results in a much more uniform coverage distribution across the entire set, as compared to other simple minimization algorithms.
5. The upper limit for the number of inputs in the corpus is $|coverage\ traces| * N$, but in practice most common traces will be covered by just a few tiny samples. For example, all program initialization traces will be covered by the single smallest file in the entire set (typically with size=0).

Some potential shortcomings

- Due to the fact that each trace has its smallest samples in the corpus, we will most likely end up with some redundant, short files which don't exercise any interesting functionality, e.g. for libpng:

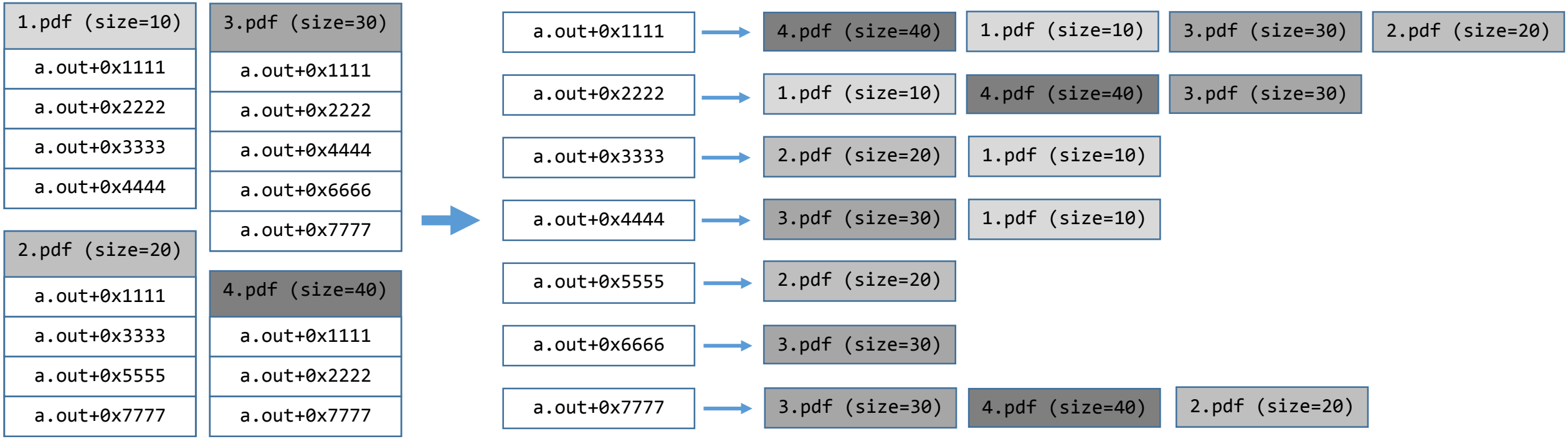
`89504E470D0A1A0A`	`.PNG....`	(just the header)
`89504E470D0A1A02`	`.PNG....`	(invalid header)
`89504E470D0A1A0A0000001A0A`	`.PNG.........`	(corrupt chunk header)
`89504E470D0A1A0A0000A4ED69545874`	`.PNG........iTXt`	(corrupt chunk with a valid tag)
`88504E470D0A1A0A002A000D7343414C`	`.PNG.....*..sCAL`	(corrupt chunk with another tag)

- This is considered an acceptable trade-off, especially given that having such short inputs may enable us to discover unexpected behavior in parsing file headers (e.g. undocumented but supported file formats, new chunk types in the original format, etc.).

Corpus distillation – “Map” phase

```
Map(sample_id, data):
  Get code coverage provided by "data"

  for each trace_id:
    Output(trace_id, (sample_id, data.size()))
```

Corpus distillation – “Map” phase

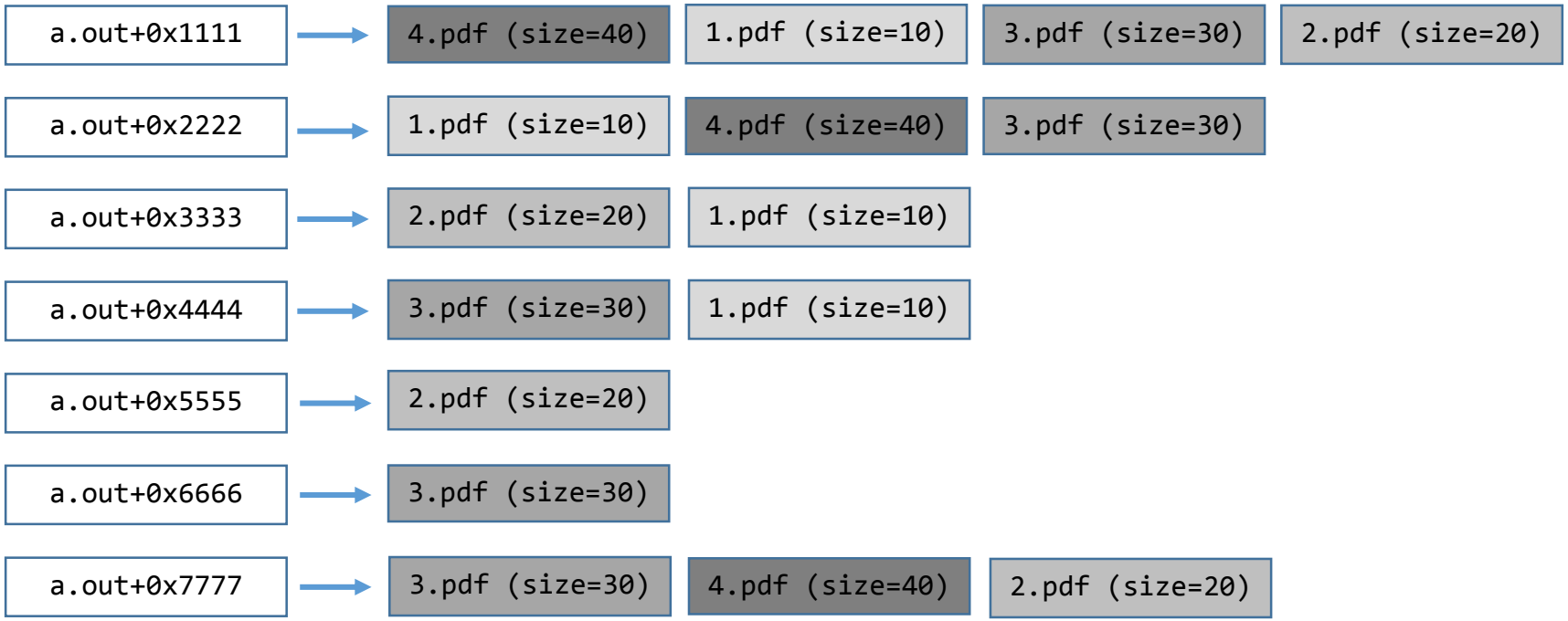


Corpus distillation – “Reduce” phase

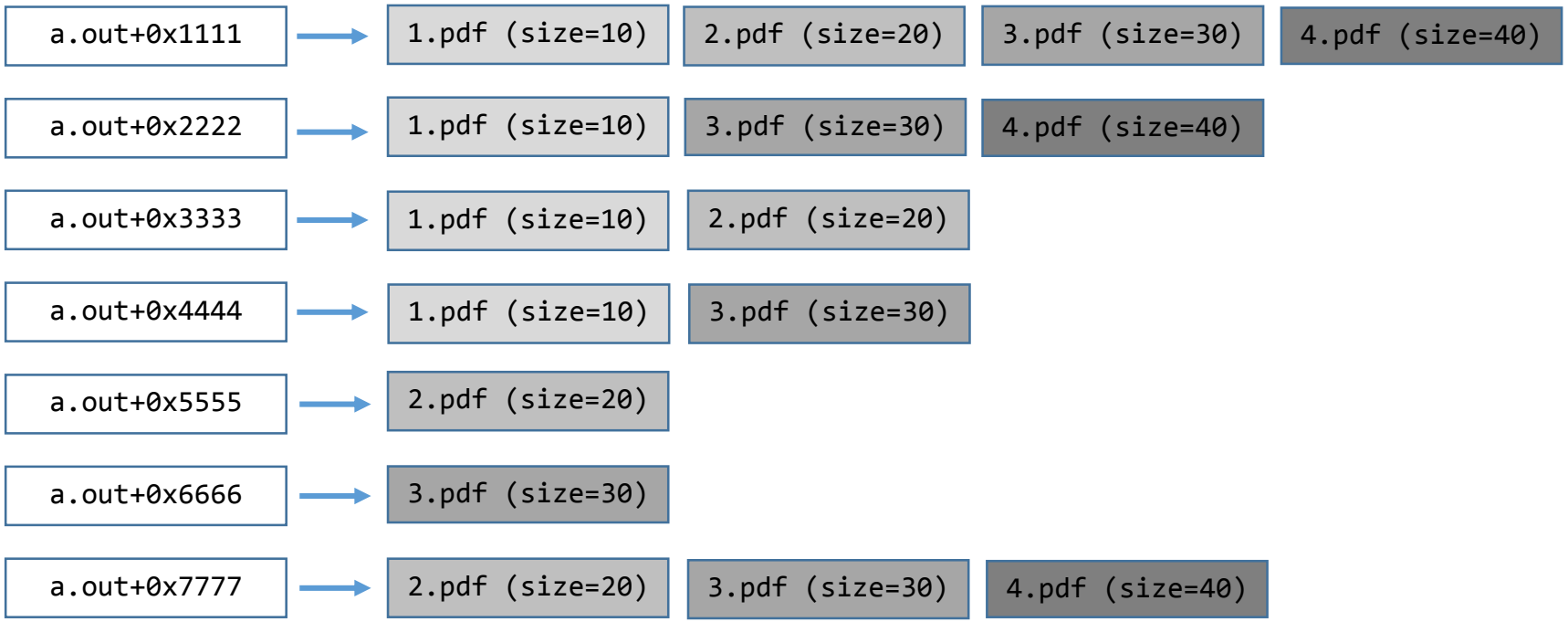
Reduce(trace_id, S = $\{(sample\_id_1, size_1), \ldots, (sample\_id_N, sizeN)\}$:

```
  Sort set S by sample size (ascending)

  for (i < N) && (i < S.size()):
    Output(sample_id_i)
```

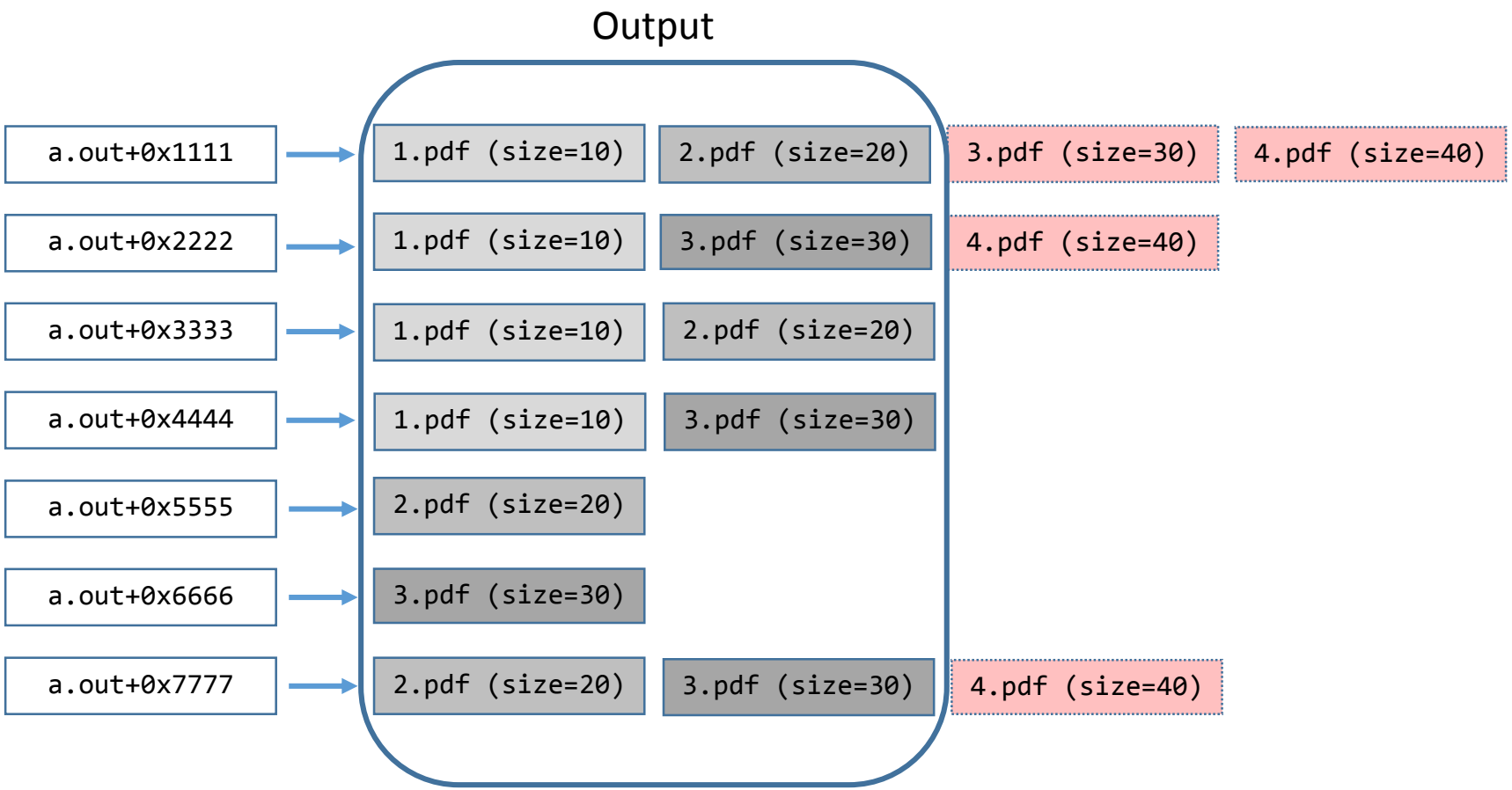
Corpus distillation – “Reduce” phase



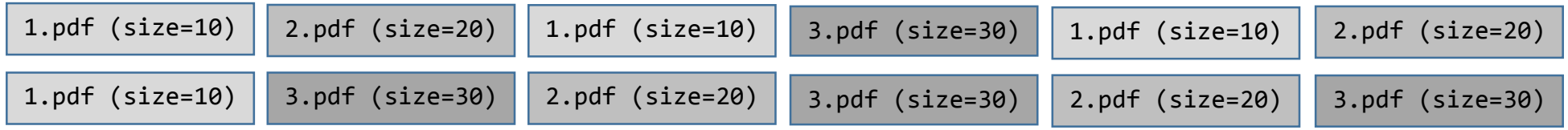
Corpus distillation – “Reduce” phase



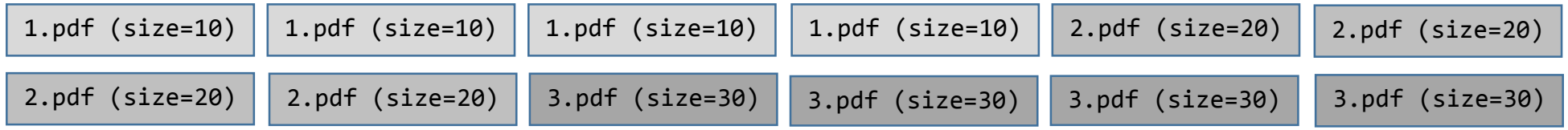
Corpus distillation – “Reduce” phase



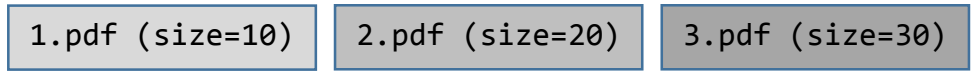
Corpus distillation – local postprocessing




```
$ cat corpus.txt | sort
```




```
$ cat corpus.txt | sort | uniq
```



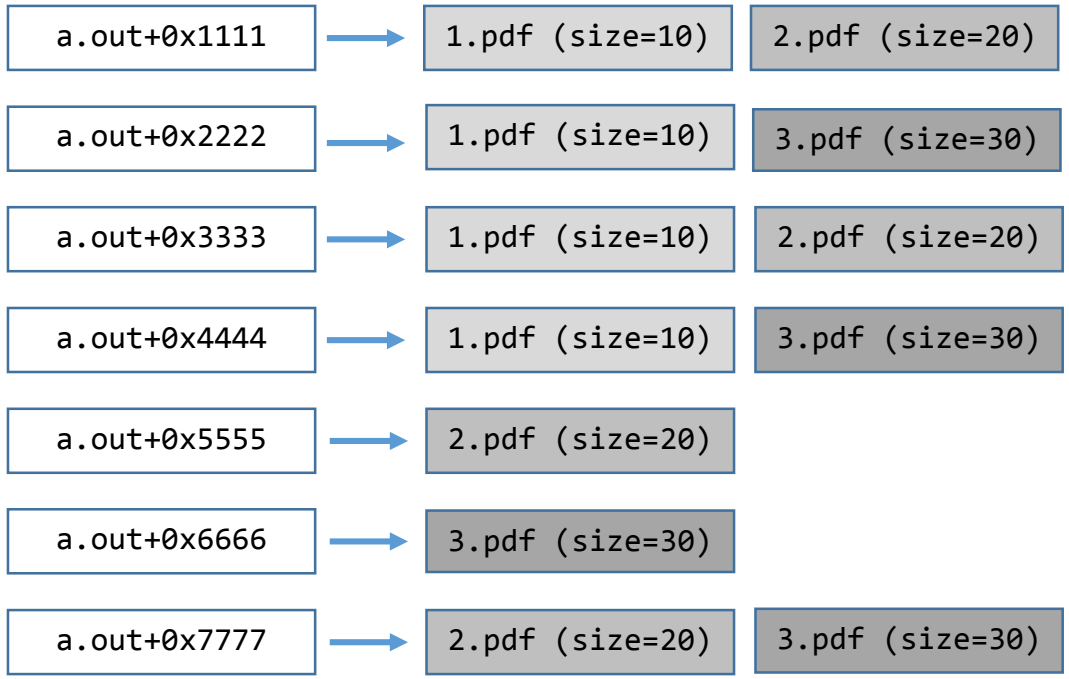
Corpus distillation – track record

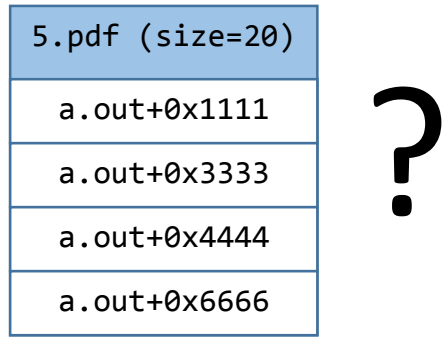
- I've successfully used the algorithm to distill terabytes-large data sets into quality corpora well fit for fuzzing.
- I typically create several corpora with different N, which can be chosen from depending on available system resources etc.
- Examples:
  - PDF format, based on instrumented *pdfium*
    - $N = 1$, 1800 samples, 2.6 GB
    - $N = 10$, 12457 samples, 12 GB
    - $N = 100$, 79912 samples, 81 GB
  - Fonts, based on instrumented *FreeType2*
    - $N = 1$, 608 samples, 53 MB
    - $N = 10$, 4405 samples, 526 MB
    - $N = 100$, 27813 samples, 3.4 GB

Corpus management – new candidate

```
MergeSample(sample, sample_coverage):
  candidate_accepted = False

  for each trace in sample_coverage:
    if (trace not in coverage) || (sample.size() < coverage[trace].back().size()):
      Insert information about sample at the specific trace
      Truncate list of samples for the trace to a maximum of N
      Set candidate_accepted = True

  if candidate_accepted:
    # If candidate was accepted, perform a second pass to insert the sample in
    # traces where its size is not just smaller, but smaller or equal to another
    # sample. This is to reduce the total number of samples in the global corpus.
    for each trace in sample_coverage:
      if (sample.size() <= coverage[trace].back().size())
        Insert information about sample at the specific trace
        Truncate list of samples for the trace to a maximum of N
```

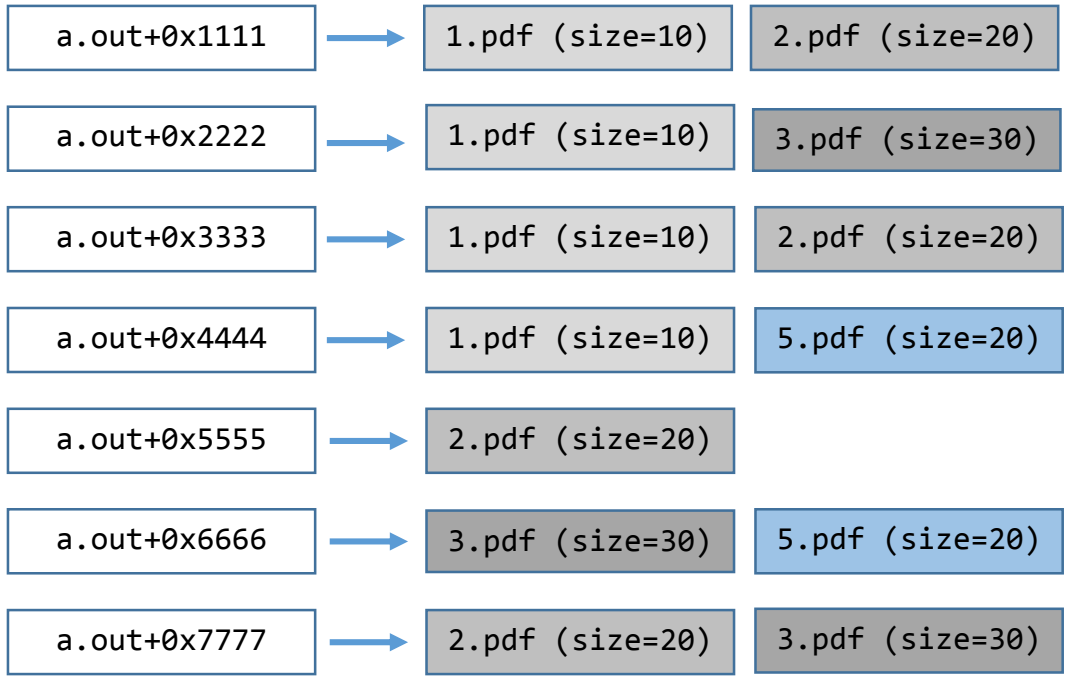
New candidate illustrated (N=2)

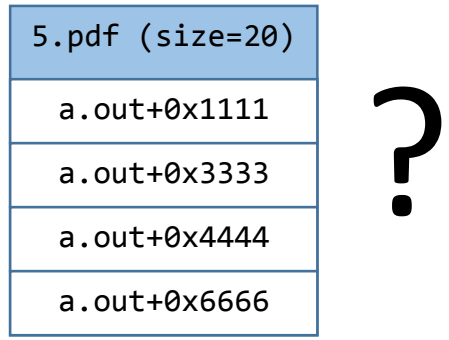




?

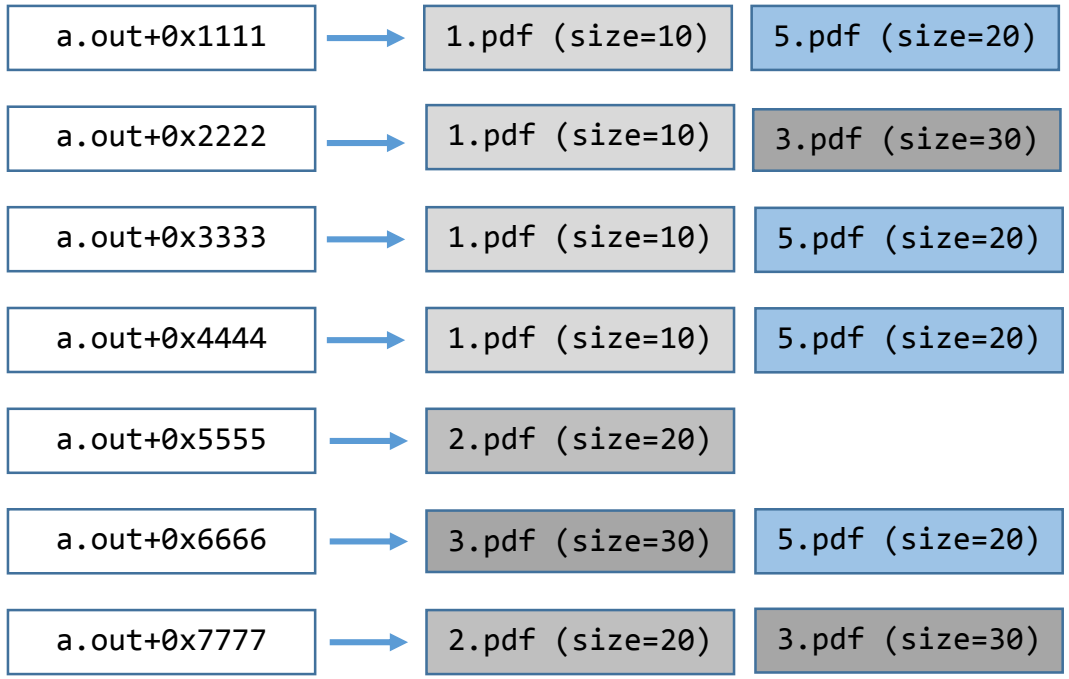
New candidate – first pass

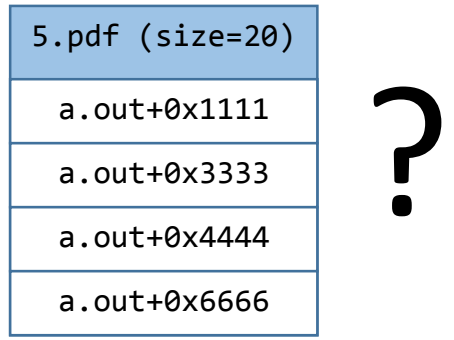




?

New candidate – second pass





?

Corpus management: merging two corpora

Trivial to implement by just including the smallest N samples for each trace from both corpora being merged.

Trophy – Wireshark

- I've been fuzzing Wireshark since November 2015.
  - Command-line *tshark* utility built with ASAN and AsanCoverage.
  - 35 vulnerabilities discovered, reported and fixed so far.
- Initially started with some samples from the project's SampleCaptures page.
  - **297 files**, **233 MB total**, **803 kB average** file size, **9.53 kB median** file size.
- Over several months of coverage-guided fuzzing with the discussed algorithms, the corpus has dramatically evolved.
  - **77373 files**, **355 MB total**, **4.69 kB average** file size, **47 b median** file size.

Trophy – Wireshark

- The nature of the code base makes it extremely well fit for dumb, coverage-guided fuzzing.
  - A vast number of dissectors.
  - Mostly written in C.
  - Operates on structurally simple data (wire protocols), mostly consecutive bytes read from the input stream.
    - Makes it easy to brute-force through the code.
- Generally great test target for your fuzzer, at least the version from a few months back.

Trophy – Wireshark

ID ▾	Type ▾	Status ▾	Priority ▾	Milestone ▾	Owner ▾	Summary + Labels ▾
641	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based out-of-bounds read in getRate CCProjectZeroMembers
642	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based buffer overflow in AirPDcapPacketProcess CCProjectZeroMembers
643	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based out-of-bounds read in find_signature CCProjectZeroMembers
644	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based buffer overflow in dissect_diameter_base_framed_ipv6_prefix CCProjectZeroMembers
645	----	Fixed	----	----	mjurczyk@google.com	Wireshark use-after-free in addresses_equal (dissect_rsvp_common) CCProjectZeroMembers
646	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds read in ascend_seek CCProjectZeroMembers
647	----	Fixed	----	----	mjurczyk@google.com	Wireshark heap-based buffer overflow in vwr_read_s2_s3_W_rec CCProjectZeroMembers
648	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds read in dissect_ber_set CCProjectZeroMembers
649	----	Fixed	----	----	mjurczyk@google.com	Wireshark static buffer overflow in my_dgt_tbcd_unpack CCProjectZeroMembers
650	----	Fixed	----	----	mjurczyk@google.com	Wireshark heap-based buffer overflow in iseries_parse_packet CCProjectZeroMembers
651	----	Fixed	----	----	mjurczyk@google.com	Wireshark use-after-free in print_hex_data_buffer / print_packet CCProjectZeroMembers
652	----	Fixed	----	----	mjurczyk@google.com	Wireshark SIGSEGV in dissect_nbap_MACdPDU_Size CCProjectZeroMembers
653	----	Fixed	----	----	mjurczyk@google.com	Wireshark SIGSEGV in memcpy (get_value / dissect_btatt) CCProjectZeroMembers
654	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds read in add_ff_vht_compressed_beamforming_report CCProjectZeroMembers
655	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based buffer overflow in file_read (wtap_read_bytes_or_eof/mp2t_find_next_pcr) CCProjectZeroMembers
656	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds read in dissect_oml_attrs CCProjectZeroMembers
657	----	Fixed	----	----	mjurczyk@google.com	Wireshark heap-based out-of-bounds read in AirPDcapDecryptWPABroadcastKey CCProjectZeroMembers
658	----	Fixed	----	----	mjurczyk@google.com	Wireshark heap-based out-of-bounds read in infer_pkt_encap CCProjectZeroMembers
659	----	Fixed	----	----	mjurczyk@google.com	Wireshark heap-based out-of-bounds read in dissect_ber_constrained_bitstring CCProjectZeroMembers
660	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds read in dissct_rsl_ipaccess_msg CCProjectZeroMembers
661	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds read in dissect_zcl_pwr_prof_pwrprofstatersp CCProjectZeroMembers
662	----	Fixed	----	----	mjurczyk@google.com	Wireshark assertion failure in wmem_alloc CCProjectZeroMembers
663	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based buffer overflow in dissect_tds7_colmetadata_token CCProjectZeroMembers
694	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based out-of-bounds read in nettrace_3gpp_32_423_file_open CCProjectZeroMembers
695	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds read in hiqnet_display_data CCProjectZeroMembers
696	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based buffer overflow in dissect_nhdr_extopt CCProjectZeroMembers
697	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based out-of-bounds read in iseries_check_file_type CCProjectZeroMembers
739	----	Fixed	----	----	mjurczyk@google.com	Wireshark use-after-free in wtap_optionblock_free CCProjectZeroMembers
740	----	Fixed	----	----	mjurczyk@google.com	Wireshark heap-based out-of-bounds read in AirPDcapDecryptWPABroadcastKey CCProjectZeroMembers
750	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds write in dissect_ber_integer CCProjectZeroMembers
754	----	Fixed	----	----	mjurczyk@google.com	Wireshark heap-based out-of-bounds read in dissect_pktc_rekey CCProjectZeroMembers
802	----	Fixed	----	----	mjurczyk@google.com	Wireshark stack-based buffer overflow in dissect_2008_16_security_4 CCProjectZeroMembers
803	----	Fixed	----	----	mjurczyk@google.com	Wireshark SIGSEGV in erf_meta_read_tag CCProjectZeroMembers
804	----	Fixed	----	----	mjurczyk@google.com	Wireshark assertion failure in alloc_address_wmem CCProjectZeroMembers
806	----	Fixed	----	----	mjurczyk@google.com	Wireshark static out-of-bounds reads while accessing ett_zbee_zcl_pwr_prof_enphases CCProjectZeroMembers

Trophy – Adobe Flash

- Have been fuzzing Flash for many years now (hundreds of vulnerabilities reported), but only recently started targeting the ActionScript `Loader()` class.
- Official documentation only mentions JPG, PNG, GIF and SWF as supported input formats:

> The Loader class is used to load SWF files or image (JPG, PNG, or GIF) files.

Trophy – Adobe Flash

- After several hours of fuzzing, I observed two sudden peaks in the number of covered traces.
- The fuzzer discovered the “ATF” and “II” signatures, and started generating valid ATF (Adobe Texture Format for Stage3D) files, later with embedded JXR (JPEG XR)!
  - Two complex file formats whose support is not documented anywhere, as far as I searched.
  - Immediately after, we started to observe tons of interesting crashes to pop up.
- 7 vulnerabilities fixed by Adobe so far.

Corpus post-processing

- If the files in your corpus are stored in a way which makes them difficult to mutate (compression, encryption etc.), some preprocessing may be in order:
  - **SWF applets** are typically stored in LZMA-compressed form (“CWS” signature), but may be uncompressed to original form (“FWS” signature).
  - **PDF documents** typically have most binary streams compressed with Deflate or other algorithms, but may be easily decompressed.

    ```
    pdftk doc.pdf output doc.unc.pdf uncompress
    ```
  - **Type 1 fonts** are always “encrypted” with a simple cipher and a constant key: can be decrypted prior to fuzzing.
  - And so on…

Running the target

Command-line vs graphical applications

- Generally preferred for the target program to be a command-line utility.
  - Quite common on Linux, less so on Windows.
  - Most open-source libraries ship with ready testing tools, which may provide great or poor coverage of the interfaces we are interested in fuzzing.
    - In case of bad or non-existent executable tools, it definitely pays off to write a thorough one on your own.
  - Much *cleaner* in terms of interaction, logging, start-up time, etc.
    - Nothing as annoying as having to develop logic to click through various application prompts, warnings and errors.

Graphical applications on Linux

- On Linux, if you have no other choice but to run the target in graphical mode (most likely closed-source software, otherwise you *do* have a choice), you can use Xvfb.
  - X virtual framebuffer.
  - Trivial to start the server: `$ Xvfb :1`
  - Equally easy to start the client: `$ DISPLAY=:1 /path/to/your/app`
- Pro tip: for some applications, the amount of input data processed depends on the amount of data displayed on the screen.
  - The case of Adobe Reader.
  - In such cases, make your display as large as possible: `$ Xvfb –screen 0 8192x8192x24 :1`.
  - In command line, set the Reader window geometry to match the display resolution: `$ acroread –geometry 500x8000`.

Graphical programs have command-line options, too!

```
$ ./acroread -help
Usage: acroread [options] [list of files]
Run 'acroread -help' to see a full list of available command line options.
------------------------

Options:
        --display=<DISPLAY>                                     This option specifies the host and display to use.
        --screen=<SCREEN>                                       X screen to use. Use this options to override the screen part of the DISPLAY environment
variable.
        --sync                                                  Make X calls synchronous. This slows down the program considerably.
        -geometry [<width>x<height>][{+|-}<x offset>{+|-}<y offset>]    Set the size and/or location of the document windows.
        -help                                                   Prints the common command-line options.
        -iconic                                                 Launches in an iconic state on the desktop.
        -info                                                   Lists out acroread Installation Root, Version number, Language.
        -tempFile                                               Indicates files listed on the command line are temporary files and should not be put in
the recent file list.
        -tempFileTitle <title>                                  Same as -tempFile, except the title is specified.
        -toPostScript                                           Converts the given pdf_files to PostScript.
        -openInNewInstance                                      It launches a new instance of acroread process.
        -openInNewWindow                                        Same as OpenInNewInstance. But it is recommended to use OpenInNewInstance. openInNewWindow
will be deprecated.
        -installCertificate <server-ip> <server-port>           Fetches and installs client-side certificates for authentication to access the server
while creating secured connections.
        -installCertificate [-PEM|-DER] <PathName>              Installs the certificate in the specified format from the given path to the Adobe Reader
Certificate repository.
        -v, -version                                            Print version information and quit.
        /a                                                      Switch used to pass the file open parameters.

…
```

While we're at Adobe Reader…

- We performed lots of Adobe Reader for Linux fuzzing back in 2012 and 2013.
  - Dozens of bugs fixed as a result.
- At one point Adobe discontinued Reader's support for Linux, last version being 9.5.5 released on 5/10/13.
- In 2014, I had a much better PDF corpus and mutation methods than before.
  - But it was still much easier for me to fuzz on Linux…
  - Could I have any hope that crashes from Reader 9.5.5 for Linux would be reproducible on Reader X and XI for Windows / OS X?

766 crashes in total

- 11 of them reproduced in then-latest versions of Adobe Reader for Windows (fixed in APSB14-28, APSB15-10).

- Mateusz Jurczyk of Google Project Zero and Gynvael Coldwind of Google Security Team (CVE-2014-8455, CVE-2014-8456, CVE-2014-8457, CVE-2014-8458, CVE-2014-8459, CVE-2014-8460, CVE-2014-8461, CVE-2014-9158, CVE-2014-9159)

- Mateusz Jurczyk of Google Project Zero and Gynvael Coldwind of Google Security Team (CVE-2014-9160, CVE-2014-9161)

When the program misbehaves…

- There are certain behaviors undesired during fuzzing.
  - Installation of generic exception handlers, which implement their own logic instead of letting the application crash normally.
  - Attempting to establish network connections.
  - Expecting user interaction.
  - Expecting specific files to exist in the file system.
- On Linux, all of the above actions can be easily mitigated with a dedicated `LD_PRELOAD` shared object.

Disabling custom exception handling

```
sighandler_t signal(int signum, sighandler_t handler) {
 return (sighandler_t)0;
}

int sigaction(int signum, const void *act, void *oldact) {
 return 0;
}
```

Disabling network connections

```
int socket(int domain, int type, int protocol) {
  if (domain == AF_INET || domain == AF_INET6) {
    errno = EACCES;
    return -1;
  }

  return org_socket(domain, type, protocol);
}
```

... and so on.

Fuzzing the command line

- Some projects may have multiple command line flags which we might want to flip randomly (but deterministically) during the fuzzing.
- In open-source projects, logic could be added to command-line parsing to seed the options from the input file.
  - Not very elegant.
  - Would have to be maintained and merged with each subsequent fuzzed version.
- Solution: external target launcher
  - Example: hash the first 4096 bytes of the input file, randomize flags based on that seed, call `execve()`.

FFmpeg command line

```
$ ffmpeg -y -i /path/to/input/file -f <output format> /dev/null
```

FFmpeg available formats

```
$ ./ffmpeg -formats

  File formats:
   D. = Demuxing supported
   .E = Muxing supported
   --
   D  3dostr          3DO STR
    E 3g2             3GP2 (3GPP2 file format)
    E 3gp             3GP (3GPP file format)
   D  4xm             4X Technologies
```

<300 lines omitted>

```
   D  xvag            Sony PS3 XVAG
   D  xwma            Microsoft xWMA
   D  yop             Psygnosis YOP
   DE yuv4mpegpipe    YUV4MPEG pipe
```

FFmpeg wrapper logic

```
char * const args[] = {
  ffmpeg_path,
  "-y",
  "-i",
  sample_path,
  "-f",
  encoders[hash % ARRAY_SIZE(encoders)],
  "/dev/null",
  NULL
};

execve(ffmpeg_path, args, envp);
```

Always make sure you're not losing cycles

- FreeType2 has a convenient command-line utility called *ftbench*.
  - Runs provided font through 12 tests, exercising various library API interfaces.
  - As the name implies, it is designed to perform benchmarking.
- When you run it with no special parameters, it takes a while to complete:

```
$ time ftbench /path/to/font
…
real    0m25.071s
user    0m23.513s
sys     0m1.522s
```

Here’s the reason

```
$ ftbench /path/to/font

ftbench results for font `/path/to/font'
--------------------------------------------------------------------------------

family: Family
 style: Regular

number of seconds for each test: 2.000000

...

executing tests:
  Load                      50.617 us/op
  Load_Advances (Normal)    50.733 us/op
  Load_Advances (Fast)      0.248 us/op
  Load_Advances (Unscaled)  0.217 us/op
  Render                    22.751 us/op
  Get_Glyph                 5.413 us/op
  Get_CBox                  1.120 us/op
  Get_Char_Index            0.326 us/op
  Iterate CMap              302.348 us/op
  New_Face                  392.655 us/op
  Embolden                  18.072 us/op
  Get_BBox                  6.832 us/op
```

It didn’t strike me for a long time…

- Each test was running for 2 seconds, regardless of how long a single iteration took.
- The `-c 1` flag to the rescue:

```
number of iterations for each test: at most 1
number of seconds for each test: at most 2.000000
…
real    0m1.748s
user    0m1.522s
sys     0m0.124s
```

- And that’s for a complex font, the speed up for simple ones was 100x and more.
- Still managed to find quite a few bugs with the slow fuzzing.

And when you have a fast target…

- Some fuzzing targets are extremely fast.
  - Typically self-contained, open-source libraries with a simple interface, e.g. regex engines, decompressors, image format implementations etc.
  - Each iteration may take much less than 1ms, potentially enabling huge iterations/s ratios.
- In these cases, the out-of-process mode becomes a major bottleneck, as a process start up may take several milliseconds, resulting in most time spent in `execve()` rather than the tested code itself.

And when you have a fast target…

- Solution #1: the Fork Server, as first introduced by AFL in October 2014, implemented by Jann Horn.
  - `execve()` once to initialize the process address space, then only `fork()` in a tight loop directly before `main()`.
  - Detailed description on lcamtuf's blog: *Fuzzing random programs without execve()*.
- Solution #2: in-process fuzzing.
  - Relatively easy to achieve with AddressSanitizer, SanitizerCoverage and their programmatic API.
  - LibFuzzer is a ready to use in-process, coverage-guided fuzzer developed by the author of the two projects mentioned above.
- One of the two options is extremely encouraged for very fast targets, as they may easily result in a speed up of 2 – 10x and more.

Mutating data

Mutating inputs

- Obviously highly dependent on the nature of the input data.
  - Dedicated mutation algorithms may be better then generic ones, if designed properly.
  - Sometimes even required, if the data is structured in a very peculiar format which gets trivially corrupted by applying random mutations.

Mutating inputs

- In most scenarios, however, universal approaches do very well for most real-world file format parsers.
  - As evidenced by hundreds of vulnerabilities discovered by such fuzzers.
  - If parts of the format are sensitive or provide a skeleton for the rest of the data, it might be easier to exclude them from mutations, or perform post-mutation fixups.
  - Writing a dedicated mutator / protocol specification / etc. also puts us at risk of *the human factor* – we may fail to think of some constraints which could trigger crashes.
    - Generic, dumb mutations will never fail us: they may not hit a specific condition due to probability, but surely not because of our stupidity.

Mutating inputs – algorithms

- There are a few *field-tested* set of mutators which appear to be quite effective for binary blobs, especially in combination with coverage guidance.
  - ***bitflipping*** – flipping between 1 and 4 consecutive bits in a specific byte.
  - ***byteflipping*** – completely replacing a specific byte with a random one.
  - ***special ints*** – insertion of „special integers” of width 2 and 4 (`INT_MIN`, `INT_MAX` etc.) in different endianness.
  - ***add subtract binary*** – binary addition and subtraction of random values at random offsets in the stream.
  - ***chunk spew*** – taking a data chunk from one location in the input and inserting it into another one.
  - ***append*** / ***truncate*** – appending data (random or based on existing input) at the end of the sample, or truncating it to a specific size.

Mutating inputs – algorithms

- It still pays off to have some text-specific mutators in your arsenal, too.
  - ***flip numbers*** – increasing, decreasing, or completely replacing textual numbers found in the input stream.
  - ***attribute mangle*** – automatically detecting the structure of tags and attributes in the input stream, and removing them or shuffling around.
  - Both above algorithms work great e.g. with PDF files (together with regular mutators to flip bits in the embedded binary streams).
- And of course let's not forget about the great Radamsa mutator.

Mutation ratios

- If we don’t make or enforce any assumptions regarding the size of the inputs, performing a fixed number of mutations doesn’t sound like a great idea.
  - Modifying 4 bytes out of 16 obviously has a completely different impact than changing 4 in 1048576 (one megabyte).
- Instead, a percentage amount of data can be malformed, which makes the number of mutations proportional to the input size.
- Having *fixed* ratios is not great either.
  - Various file formats have various structures, data densities, metadata/data ratios etc.
  - Various software have various tolerance towards faults in the processed input.
  - As a result, I believe mutation ratios should be adjusted on a **per-algorithm**, **per-target** and **per-format** basis.

Mutation ratios

- To give the fuzzer even more freedom (and potentially trigger more interesting program states), the ratio doesn’t even have to be fixed for each {algorithm, target, format}.
  - We can just decide on a range of ratios to pick from during each iteration.
- Furthermore, we can allow the chaining of different mutation algorithms, and have a list of all the different chains.
- Let’s call this overall specification „mutation strategy”.

When is a mutation strategy is optimal?

- Based on experimental data and experience, my belief is that a mutation strategy is most optimal if the target succeeds to fully process the mutated data **~50%** of the time, and likewise fails **~50%** of the time.
  - This means that that mutated samples are *on the verge* of being correct, which seems to be the right balance between „always fails” (too aggressive) and „always passes” (too loose).

Automatic mutation evaluation

- With the help of code coverage guidance, the need for manual mutation strategy configuration could be completely avoided.
  - The fuzzer could autonomously determine the most effective algorithms, ratios and their chainings.
  - Together with portions of the input most useful to mutate in the first place, as an extension to lcamtuf's *Automatically inferring file syntax with afl-analyze*.
  - This would massively simplify the fuzzing process for regular users, and also reduce out one more *human factor* from the pipeline.

The Windows Kernel font fuzzing effort

Doing it in Bochs

- As you may have heard, I am a big fan of doing things in Bochs (the software x86 emulator written in C++).
  - *Bochspwn: Identifying 0-days via System-Wide Memory Access Pattern Analysis*
  - *Dragon Sector: SIGINT CTF 2013: Task 0x90 (300 pts)*
- Not very fast (up to ~100 MHz effective frequency), but we can still scale against that.

Doing it in Bochs

- Offers some very useful properties:
  - Can be run on any Linux system, even with no virtualization support available.
  - Provides an extensive, documented instrumentation API to interact with the guest.
    - Guest ⇔ host communication channel.
    - Blue Screen of Death detection.
    - Other virtual machine introspection, if needed.
  - Runs Windows out of the box.
  - Trivial configuration, which I was already familiar with.

Let’s do it!

The input data

- This part was the easiest one – reuse an existing corpus based on instrumented FreeType2 fuzzing.
- Had to extract TrueType and OpenType files, as other (especially the more exotic ones) are not supported by the Windows kernel.

What about .FON and Type1 PostScript fonts?

- We initially also fuzzed .FON bitmap fonts.
  - The only recurring discovery there was a divide-by-zero system crash, which has already been reported to Microsoft long ago. ☹
- There were several reasons not to target Type 1:
  - Windows requires two corresponding files (.PFM and .PFB) to load a single font.
  - Structurally very simple, the most complex part are CharStrings, which have already been manually audited to death by myself.
  - Most of the font handling logic is shared in ATMFD.DLL for both Type1 and OTF formats.

Mutating TTF & OTF

- Design decision: the mutations would be applied in the Bochs instrumentation (on the host).
  - Makes it much faster to mutate at native speed instead of in emulated code.
  - In case of a guest crash, the system automatically knows which sample caused it.

But… how to mutate them properly?

- Both TTF and OTF follow a common *chunk* structure: SFNT.
  - Each file consists of a number of *tables*.

OpenType Tables

Whether TrueType or PostScript outlines are used in an OpenType font, the following tables are required for the font to function correctly:

Required Tables

Tag	*Name*
cmap	Character to glyph mapping
head	Font header
hhea	Horizontal header
hmtx	Horizontal metrics
maxp	Maximum profile
name	Naming table
OS/2	OS/2 and Windows specific metrics
post	PostScript information

SFNT tables

- Some tables are common for both TTF and OTF, others are specific to just one of the formats.
- Some tables are required and must be present, others are optional.
- There are ~50 total in existence (but ~20 actually important ones).
- One thing in common: they are all **different**.
  - Different length, structure, importance, nature of data etc.
  - It only seems reasonable to treat each of them individually rather than equally.

How it’s usually done

The typical scheme I’ve seen in nearly every font fuzzing presentation:

1. Mutate the TTF/OTF file as a whole, not considering its internal structure.
2. Fix up the table checksums in the header, so that Windows doesn’t immediately refuse them.

Mutating TTF & OTF my way

- I’ve gone through my font corpus to discover that on average, there are ~10 tables whose modification affects the success of its loading and displaying.
- Hence, if we want the overall mutated font’s loadability to stay around 50%, the statistical correctness of each table should be:

$$\sqrt[10]{0.5} \approx 0.93$$

Mutating TTF & OTF my way

- I wrote a simple program to determine the desired mutation ratio per each algorithm and table in order to maintain the ~93% correctness, on many samples, and then averaged the results.
- This resulted in the following table.

SFNT table mutation ratios

	Bitflipping	Byteflipping	Chunkspew	Special Ints	Add Sub Binary
hmtx	0.1	0.8	0.8	0.8	0.8
maxp	0.009766	0.078125	0.125	0.056641	0.0625
OS/2	0.1	0.2	0.4	0.2	0.4
post	0.004	0.06	0.2	0.15	0.03
cvt	0.1	0.1	0.1	0.1	0.1
fpgm	0.01	0.01	0.01	0.01	0.01
glyf	0.00008	0.00064	0.008	0.00064	0.00064
prep	0.01	0.01	0.01	0.01	0.01
gasp	0.1	0.1	0.1	0.1	0.1
CFF	0.00005	0.0001	0.001	0.0002	0.0001
EBDT	0.01	0.08	0.2	0.08	0.08
EBLC	0.001	0.001	0.001	0.001	0.001
EBSC	0.01	0.01	0.01	0.01	0.01
GDEF	0.01	0.01	0.01	0.01	0.01
GPOS	0.001	0.008	0.01	0.008	0.008
GSUB	0.01	0.08	0.01	0.08	0.08
hdmx	0.01	0.01	0.01	0.01	0.01
kern	0.01	0.01	0.01	0.01	0.01
LTSH	0.01	0.01	0.01	0.01	0.01
VDMX	0.01	0.01	0.01	0.01	0.01
vhea	0.1	0.1	0.1	0.1	0.1
vmtx	0.1	0.1	0.1	0.1	0.1
mort	0.01	0.01	0.01	0.01	0.01

Mutating TTF & OTF my way

- As mentioned before, I dislike fixed ratios, so I eventually set the range to $\langle 0, 2 \times R \rangle$, R being the calculated ideal ratio, in order to insert more randomness into how much data is actually mutated.
- With a trivial piece of code to disassemble, modify and reassemble SNFT files, I was then able to mutate fonts in a meaningful way.

Generating TTF

- Even with a semi-smart approach to dumb fuzzing, some bugs just cannot be found by mutating existing, valid files.
- One example: TTF programs.
  - Dedicated “virtual machine” with its own operand stack, complex font-specific state structure, and 100+ instructions.
  - Quite fragile: expects some basic program consistency (e.g. sufficient arguments on the stack for each instruction), otherwise the interpreter exits early.
  - Frequent source of bugs in the past, but only due to trivial errors in the handlers.
  - Issues triggered by more complex constructs impossible to trigger with bitflipping.

The solution: TTF program generator!

Steps taken:

- read the *The TrueType Instruction Set* docs to get familiar with all instructions.
- spent several hours writing the program generator in ~500 lines of Python.
- converted all files in the corpus from TTF/OTF to TTX (XML font representation generated by the fonttools project).
- coupled the generator with an `ElementTree` XML parser, in order to add the generated instruction stream between all `<assembly></assembly>` tags (and some other minor processing).
- added logic to the Bochs instrumentation to run the generated followed by font compiler.

Found one extra bug with the generator.

Host – guest communication channel

- Communication between environments implemented via instrumented `LFENCE` instruction.
  - `void bx_instr_after_execution(unsigned cpu, bxInstruction_c *i);`
  - Operation code in EAX, input/output via ECX, EDX.
- Supported operations: `REQUEST_DATA`, `SEND_STATUS`, `DBG_PRINT`.
- Had to be careful to have all memory regions passed via pointer to Bochs be mapped in “physical” memory, so that the instrumentation can write data there.

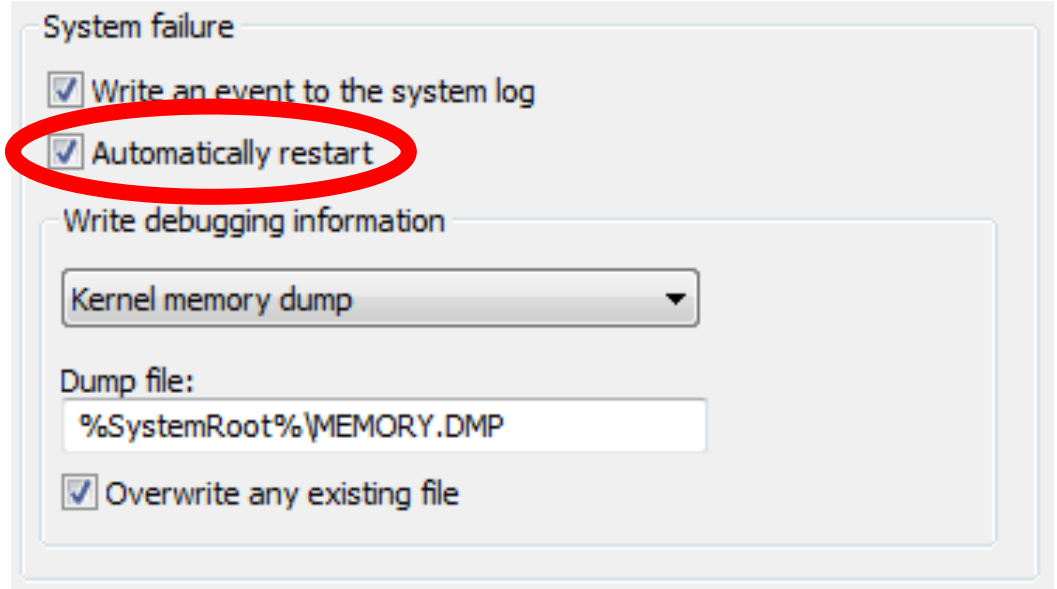
Displaying mutated fonts for best coverage

- Main goal: make sure that all data in the font is processed by the kernel.
  - Request all available `LOGFONT`s via the undocumented `GetFontResourceInfoW` call.
    - A first, unmodified iteration is required.
  - List all glyphs supported by the font via the `GetFontUnicodeRanges` call.
  - Display all of them in several variations (various width, height and properties).

Optimizing the guest operating system

- A number of actions to reduce the volume and background execution in Windows (every cycle is important):
  1. Changed the theme to Classic.
  2. Disabled all services which were not absolutely necessary for the system to work.
  3. Set the boot mode to Minimal with VGA, so that only core drivers were loaded.
  4. Uninstalled all default Windows components (games, web browser, etc.).
  5. Set the “Adjust for best performance” option in System Properties.
  6. Changed the default shell in registry from explorer.exe to the fuzzing harness.
  7. Removed all items from autostart.
  8. Disabled disk indexing.
  9. Disabled paging.
  10. Removed most unnecessary files and libraries from C:\Windows.

Detecting system bugchecks



+

```
void bx_instr_reset(unsigned cpu, unsigned type);
```

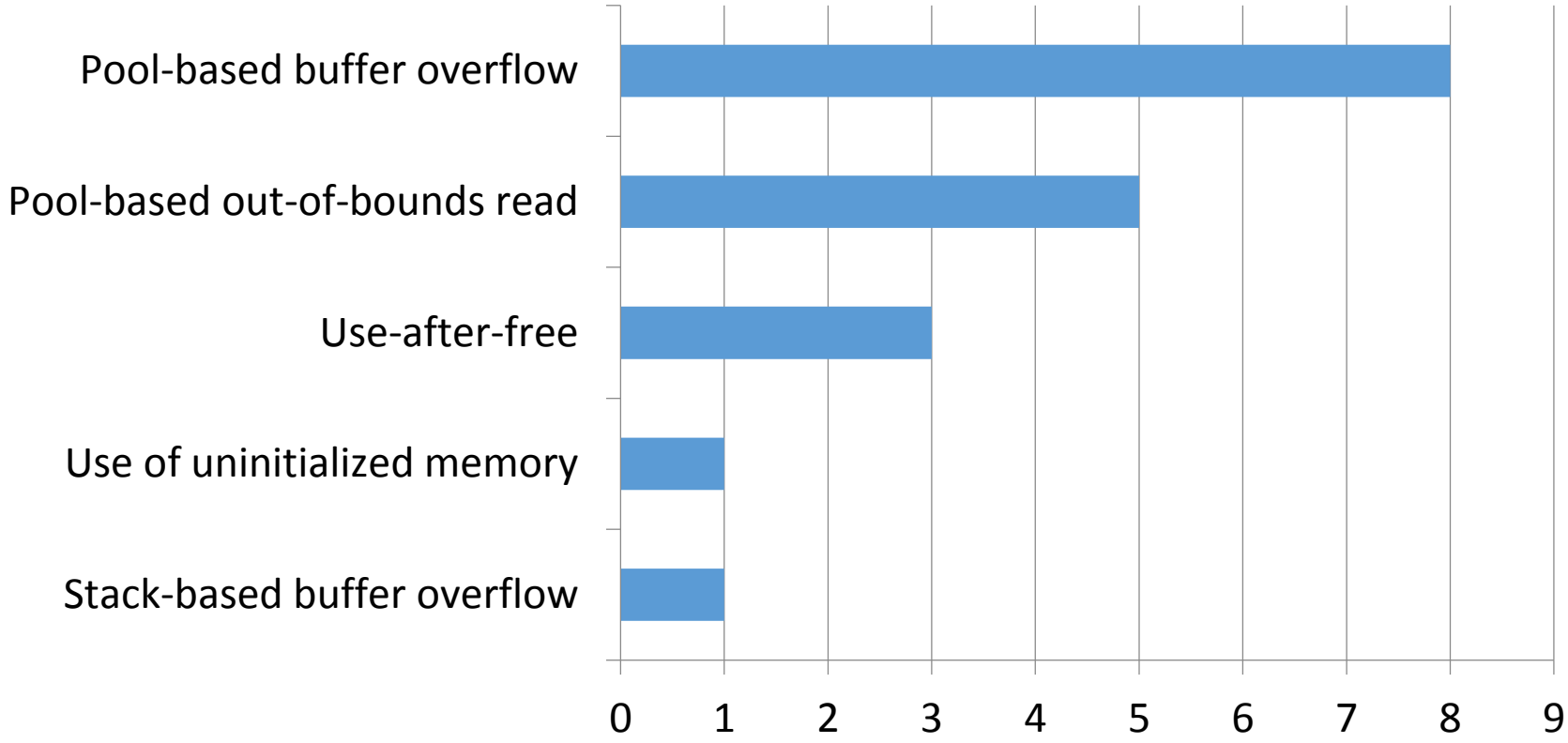
Reproducing crashes

- Reproduction performed on a VirtualBox VM with the same installation as Bochs, with some very simple logic:
  - Check if there is a crash dump in `C:\Windows\Minidump`.
  - If so, generate a textual `!analyze -v` report with WinDbg, and copy together with the dump and last processed file to external directory.
  - Test the next sample with the test harness a few times (8 or more).
  - If the system doesn't crash, restart it manually to avoid propagating any pool corruptions across different samples.

Minimizing offending samples

- Two stages:
  - Table-granular minimization to determine which malformed tables are causing the crash.
  - Byte-granular minimization to check what the exact mutations are.

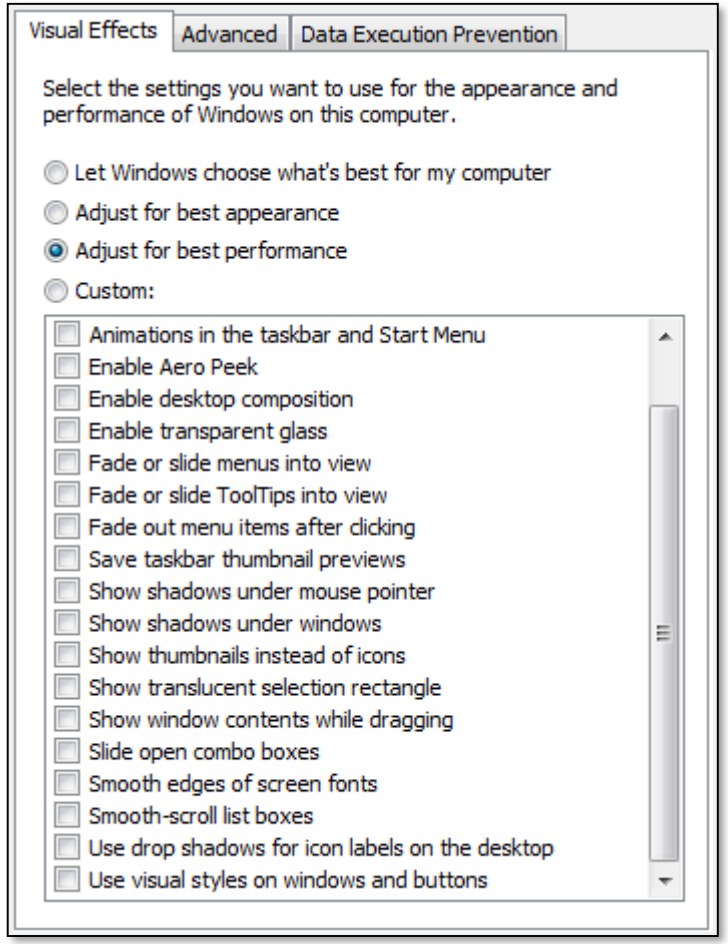
Results – summary



Results – timeline

- **21 May 2015:** iteration #1, **11** vulnerabilities reported (4 TTF, 7 OTF).
- **18 Aug 2015:** iteration #2, **2** vulnerabilities reported (2 TTF).
- **22 Dec 2015:** iteration #3, **3** vulnerabilities reported (1 TTF, 2 OTF).
- **29 June 2016:** iteration #4, **2** vulnerabilities reported (2 TTF).

CVE-2016-0145 fix delayed by a month, because…



Closing thoughts

- Hopefully after this effort, no more bugs are lurking there, right?
- This will become less important, as Microsoft has moved font handling out of the privileged kernel context in Windows 10.
- Remember it can still be used as an RCE vector, keeping it a sensitive code region.

Thanks!

@j00ru

http://j00ru.vexillium.org/

j00ru.vx@gmail.com