

Intel® Stress Bitstreams and Encoder 2016 – VP9 User Guide

Version 2.1
Updated November 20, 2015

Contents

Introduction.....	3
Feature sets and Profiles	3
General overview.....	3
VP9 specifics	4
Parfile editing basics: how to setup test coverage.....	5
Parfile editing: short summary	7
Description of command-line application options.....	7
Case study: Focus on ADST implementation	9
How to choose input resolution for certain cases	10
How to control frame-scaling randomization	11
Extensive testing and parallel execution.....	12
VP9 Random Encoder as a debugging tool.....	12
Smoke Test Streams <i>New!</i>	13
VP9 FC2 specifics	13
Residual Randomization	14
What VP9 Random Encoder doesn't support.....	15
Memory-bandwidth decoder testing	15
Worst-case performance decoder testing	15
Visually clean content.....	16
Overriding parameters <i>New!</i>	16
Iterating Parameters <i>New!</i>	17

Error Resilience Encoder <i>New!</i>	17
Description.....	18
Important restrictions	18
Broken stream generation.....	18
Common	18
Bit randomization	18
Packet Randomization	19
Syntax randomization	19
Parfiles description	20
Parfile example and parameter description.....	21

Introduction

Testing a decoder is a complex multi-criteria task. Code coverage of 100% lines of a decoder code does not guarantee the decoder is 100% compliant. On another hand, creation of millions of streams to test all possible feature combinations is time and storage consuming. Random Encoder partially solves these two issues. It is lightweight encoder with no mode decision so it is as fast as decoder is. Effective motion estimation and mode decision are not parts of video codec standard and not required to be tested in decoder side. So this most time consuming part is mostly omitted in Random Encoder in favor of speed and flexibility.

Codec developer will not want to keep all streams generated by random encoder, it is enough to keep only basic vectors. If a decoder fails to correctly decode a randomly generated stream then it makes sense to extend the test pool with the stream for future regression validation. Random Encoder is a great extension of codec validation in addition to Stress Bitstreams' basic vectors.

Feature sets and Profiles

VP9 specification has two feature sets: Feature Set 1 (FC1) containing only 8-bit 4:2:0 colorspace (chroma components reduced in both directions) and Feature Set 2 (FC2) allowing 10 and 12-bit color depth and more flexible options of chroma subsampling. Profile is a syntax element in frame header defining color bit depth and chroma subsampling setting scope. Profile 0 allows only 8-bit 4:2:0 color representation. Profile 1 adds option to disable chroma subsampling in vertical, horizontal or both directions, Profile 2 means 10 or 12 bit color depth but 4:2:0 color representation and Profile 3 enables both high-bit color depth and chroma subsampling options.

Random Encoder supports all bit-depth options and chroma-subsampling configurations. All the information in this document is applicable for any Profile and Feature Set. For high-bit-depth and chroma-subsampling specific information, see *VP9* section.

General overview

Random encoder is highly configurable and flexible syntax (VP9) encoder tool. In opposite to regular encoders, it is not intended to achieve compression but only designed to create a valid specification-compliant stream. Compliant streams contain only allowed combinations of syntax elements and their values to test decoder for unusual cases or boundary stress cases where developers usually relax requirements to code development for higher decoding speed. Decoder must be compatible with any stream so its sloppy optimizations have to be carefully tracked for boundary cases where residuals overflow may break visual representation of the picture.

You can find the recommended Decode validation process with Random Encoder below. It is up to user to change the flow and to react on pass and fail events and even decide on the criteria of test passing. We recommend to compare the latest development branch of reference libvpx decoder for VP9.

Random Encoder accepts as an input *an optional YUV file* and a *parfile* describing testing settings: features to utilize, fixed values, random values. As an output, Random Encoder produces *encoded bitstream* and optionally writes *YUV file* with internal reconstruction data. This file is used to validate that Encoder generated proper compressed file and that resulted bitstream is valid.

If there is a mismatch between encoder reconstruct and reference-decoder result, you are welcome to report to Software Publisher (your Intel contact) with the case configuration to request for the fix if it fits to your license agreement with Intel. We will always appreciate your feedback.

Random Encoder has *seed* parameter (-s) defining initial random-engine state. Changing it allows to use the same parfile to produce totally different streams with the same scope of randomization defined by parfile. The main purpose of this seed feature is extensive testing with all possible syntax-element combinations. In addition, this feature can be used to create small bug-reproducers (setting frame number parameter "-f" to some small value) for the parfiles which are known to generate the streams causing failure of examined decoder.

To summarize, typical workflow of compliance testing for VP9 consists of the following steps:

1. Produce test stream by feeding Random Encoder with a parfile and an optional input YUV file,
2. Decode the stream with reference decoder,
3. Verify that Encoder's reconstruct matches with reference-decoder result,
4. Decode the stream with your decoder and verify that its result matches with reference-decoder result,
5. Increment *seed* parameter and go to the step 1.

VP9 specifics

To start validation cycle with VP9 Random Encoder make a decision regarding testing agenda and setup certain features and value range in random-encode parfile. Parfile is a JSON formatted file that contains setting for all syntax elements and some additional entries related to frame scaling and residual randomization. Wide range of values in parfile will result in very detailed testing with baby steps. Random Encoder accepts an YUV file and parfile and iterate with changing "--seed" command-line option producing a new output stream for every iteration.

Here is a simple example when it is reasonable to narrow the scope of search. If you want to check how decoder handles whole range of motion vectors then put "mv_len" parameter in "superblock" section of parfile to maximum value. Magnitude of each MV component will be uniformly distributed in range [-mv_len..mv_len]. But if you need to test interpolation filters and convolution implementation, you need to ensure that 1/8-pixel MV accuracy is utilized, and it is considered only in case if neighbor-motion-vector components have magnitude not greater than 8 pixels. For proper testing you need to set "mv_len" to value 64 (see detailed description of this parameter explaining why all values are multiplied by 8), this will provide uniform distribution between all convolution vectors of each interpolation filter. Otherwise, the chances

to produce 1/8-pel accurate MV are very small, so proper testing of convolution implementation will take a long time.

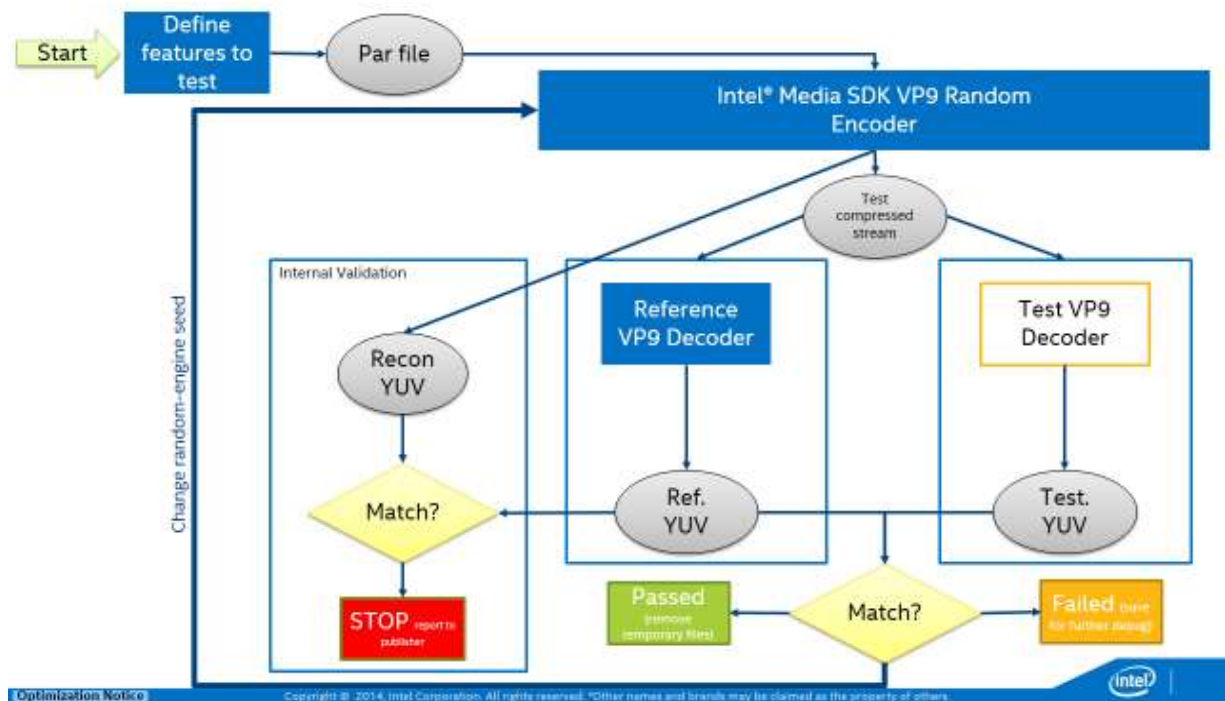


Figure 1. Testing workflow for Random Encoder for VP9. The scope of testing is defined in parfile. Encoder produces compressed bitstream with randomized mode decisions and YUV file with internal reconstruct (optional). To validate your decoder compare its results of randomized bitstream decompression against the results of reference decoder. The purpose of YUV file with reconstruct is validation of the Random Encoder—it also must match to reference decoder result.

Parfile editing basics: how to setup test coverage

This chapter explains in details the format that was chosen for test coverage.

Let's start with a basic question: why we need a parfile? Random Encoder makes each decision randomly, there is no any constraints or behavior patterns. You can expect from the Encoder any possible syntax-element combination, until this combination is VP9-format compliant. Two possible problems arise here. The first one is what if we don't want *any* combination: we are in early development stage and don't have all the features implemented, so it doesn't make sense to test those features. The second problem is test-coverage control.

For example, Encoder have to decide for current frame, which frame type it will be: Key or Inter. We could give equal chances for both variants, or put key frame about every second or every 10 seconds. Equal chances may be not good for inter-prediction feature coverage, since inter-prediction features may form more complicated feature combinations and require more detailed testing; 10 seconds may be too rare for accurate key frame testing, intra-prediction testing and key/inter frame combinations; one second may seem fair but it still doesn't look like

enough flexible solution. In addition, we need possibility to disable key frames for extensive testing of intra prediction.

The solution we came with is definition of chances for certain values of enumerated syntax element by probability weights. Frame type has two options: Key and Inter, and to have equal chances for both we add the following line in parfile: "frame_type" : [1, 1]. If we want to have key frame about every second, we increase the chances of Inter frame: "frame_type" : [1, 29]. This define chances of getting key frame as 1/30, and, what is important, doesn't define any strict pattern allowing to produce any possible combination of key and inter frames. Sometimes it is even possible to have two consecutive key frames, but the average ratio of key-frame number to the total length will be about 1/30 for the long enough streams. And if we want to disable inter frames we just give them zero chance: "frame_type" : [1, 0]. The solution appeared rather flexible and being working for all enumerated syntax elements. Most of the elements are named the same or similar way as variables in libvpx source code (reference VP9 decoder implementation).

Some syntax elements would be unreasonable to enumerate, for example, *base_qindex* has range of 256 acceptable values, and there is no sense to sit and reason "I want to avoid qp=13, it is unlucky, and would prefer better chances for 3, 7, 23 and maybe some other prime numbers". Typically, you would prefer one of the following options:

- fixed value;
- range of small or large qp values;
- full range;
- full range except zero, which may turn lossless mode on.

Such range parameters are defined in parfile with minimum and maximum values, and random values will be distributed uniformly in this range. For example, full range of *base_qindex* is defined as following: "base_qindex_range" : [0, 255]. If we want to exclude zero value, we write "base_qindex_range" : [1, 255]. To make QP fixed, for example, equal to 40, we set min=max: "base_qindex_range" : [40, 40].

JSON syntax was chosen since it convenient both for reading and for manual editing, but initially such format also had some drawbacks. There is no any syntax cues to recognize weight-based parameters from range-based ones, and for weight-based parameters the order of enumeration may be unobvious. These drawbacks are compensated by the comments for each parameter prompting what is the type of parameter, what are acceptable values for range-based parameters, and what are actual enumeration values of weight-based parameters. Parameters not corresponding to VP9 syntax-elements and related to Encoder's behavior control have comments with additional explanation.

Such JSON definition extended with comments turned out to be a good tradeoff between flexibility of test-coverage setup, human readability and compatibility with automatic parsers.

Parfile editing: short summary

Parfile is file in JSON format with comments. It includes four general sections related to different groups of parameters. "frame" and "superblock" sections allow to control syntax-element randomization scope in frame header and at superblock level. Most elements in these two section are named according to variables in libvpx source code. "scaling" section defines the way how frame resolution will be varied in the encoded stream. "residual" section contains settings related to residual randomization at superblock level. "override" section allows to redefine settings for specified frames.

There are two types of parameters: weights and range. In most cases logical or enumerated parameters are defined by relative weights. For example, setting "frame_type" : [1, 10] means one chance of key frame against 10 chances of inter frame. Numeric parameters with wide range are defined by specifying minimum and maximum values. For example, loop filter level acceptable values are from 0 to 63, and in parfile it is defined as "lf_filter_level" : [0, 63]. Actual values will be spread uniformly in this range.

To disable randomization for weights-defined parameters, weights for all except one values must be set to zero. To fix range-defined parameter you should set min and max values to the same value.

Description of command-line application options

Random Encoder is a command-line application that accepts the following parameters:

```
vp9_random_encoder.exe -i <input.yuv> -o <output.vp9> -w <width> -h <height>
                        [-p <config.json> -s <seed> -r <reconstruct.yuv>]
                        [-f <# of frames to process>]
```

More detailed parameter description:

Parameter	Short description	Details
-i STRING, --input=STRING	Input file (raw YUV)	File path, currently only I420 YUV is accepted
-o STRING, --output=STRING	Output VP9 bistream	File is output in IVF container
-w INT -h INT, --width=INT --height=INT	Width and height of input YUV	
f INT, --frames=INT	Number of frames to process	Number of frames which random encoder will consume and encode. Decoding result may have different number of frames because some of encoded frames may be hidden and also bitstream may contain one-byte-repeated frames. If hidden frames and one-byte frame repeats are disabled, output will contain the same number of frames
-r STRING, --recon=STRING	Output YUV file with encoder's reconstruct (optional)	Used to validate random encoder. If not set, encoder's reconstruct won't be written.
-p STRING, --parfile=STRING	JSON file with distribution parameters (optional)	Parfile setting limitations on syntax element randomization. If not set, the default values, which actually are not very practical, will be used.
-s INT, --seed=INT	Seed for random engine (optional)	Acceptable range: 0..INT_MAX

<code>--dump_hidden=BOOL</code>	Write invisible frames to reconstruct file (optional)	0 or 1, default is 0. If is set to 1, encoder will output hidden frames to reconstruct. This option was added to provide ability to use reference libvpx decoder without modification, because there is not any option ask him to dump hidden frames. On the other hand, for thorough validation, you need to modify ref decoder and use output containing hidden frames.
<code>--output_state=STRING</code>	Path where to write states of random engine at keyframes (optional)	This is another way to get short-reproducer stream, but it is not well designed, requires to be accurate and we encourage you to iterate with <i>seed</i> instead. But if you want the details. If you use this option providing file path, the encoder will dump the whole state of internal Mersenne-twister random-engine at each key frame. The state is a single line with a lot of numbers. And if you want to start from the state of certain frame, you can copy certain line (without frame number) to separate file and provide path to this file by <code>--input_state</code> option. Since the random encoder mostly doesn't rely in input frames for mode decision, there are very good chances to get the same combination of modes in the beginning of the bitstream, not somewhere in the middle. But again, there are much better ways to get short reproducers. In addition to iterating with <i>seed</i> , you can just cut the certain part with Intel Video Pro Analyzer or any other tool parsing and editing IVF format.
<code>--input_state=STRING</code>	Path to input file with random engine state (optional)	See <code>--output_state</code> option description.
<code>--bit_depth=INT</code>	Bitstream bit depth (8, 10, 12), overwrites parfile setting	Only for the version with FC2 support, see <i>VP9</i> section for description.
<code>--recon_bit_depth=INT</code>	Encoder-reconstruct bit depth (8, 10, 12, 16), for easier comparison with results of reference decoder	Only for the version with FC2 support, see <i>VP9</i> section for description.
<code>--colorspace=STRING</code>	Bitstream colorspace: yuv420 yuv422 yuv440 yuv444, overwrites parfile setting	Only for the version with FC2 support, see <i>VP9</i> section for description.
<code>-t STRING,</code> <code>--target=STRING</code>	Target mode: random visual mem_bw <ul style="list-style-type: none"> • random (default) - no restriction, full randomization • visual - try to avoid some configurations yielding visual artifacts • mem_bw - optimize for memory bandwidth testing 	At the current stage visual mode does two things: <ul style="list-style-type: none"> • repeats each invisible frame with <i>show_existing_frame</i> flag. This allows make playback smooth without disabling invisible frames in parfile; • disables residual randomization. All other settings potentially yielding visual artifacts are to be controlled through parfile. "mem_bw" (memory_bandwidth) mode enforces reference frame sign bias flags have different values, so compound prediction is available at each inter frame. All other settings recommended for memory bandwidth testing are provided in 401th parfile
<code>--allow_idct_overflow=BOOL</code>	Disable check that randomized residual doesn't yield inverse-transform overflow	Residual randomization in combination with high QP may produce illegal combination of coefficients yielding inverse transform overflow. During residual randomization, Encoder performs additional checks dropping illegal combinations. To disable this checks, set this option to 1.
<code>--extreme_residual=BOOL</code>	Use only extreme values for random residual	Effective if residual randomization is enabled in parfile, helps to hit inverse-transform overflow.

<code>--idct_overflow_hw=BOOL</code>	Enable Emulate-Hardware-Highbitdepth behavior for iDCT calculation	Affects inverse transform, copies behavior of corresponding configuration define in libvpx code.
<code>--verbose</code>	Print reference frames' resolution for each frame (optional)	Also option for debug purposes. When working with certain stream and parfile and frame scaling is enabled helps to understand dependencies between reference frames.
<code>--help</code>	Output help message	

Case study: Focus on ADST implementation

For example, you have optimized inverse ADST computation and want to validate new implementation against the reference one. ADST is applied only to intra blocks, but for default config the portion of key frames is 10% and portion of intra blocks in inter frames is 25%. Taking into account that not all intra modes utilize ADST, about 75% of bitstream content will be irrelevant to the validation task.

It will be reasonable not to use inter frames for ADST testing. So *008_intra_stress.json* parfile is a good starting point, and it will be modified to setup Random Encoder for ADST testing. First of all, let's perform smoke test using this parfile without modification on a small number of frames (about 20 at SD resolution). If it passes, you can move ahead. If not, you can check that each size of transform works correctly by creating separate parfiles for each transform modifying the following line in "frame" section:

```
"tx_mode" : [1, 1, 1, 1, 2], // 4x4, 8x8, 16x16, 32x32, select
```

Let's disable all transform except 4x4:

```
"tx_mode" : [1, 0, 0, 0, 0], // 4x4, 8x8, 16x16, 32x32, select
```

Then do the same for 8x8:

```
"tx_mode" : [0, 1, 0, 0, 0], // 4x4, 8x8, 16x16, 32x32, select
```

And the same way for 16x16. Actually, only-8x8 "tx_mode" setting doesn't fully disable 4x4 transforms, they are still used for superblocks less than 8x8 and for chroma in blocks less than 16x16. If it is necessary to disable small transforms totally, "partition" setting in "superblock" section needs to be modified in order to disable superblock splitting at certain level.

If there were any basic problems with new ADST implementation, they would be revealed by such basic tests. Now let's focus on validation of extreme cases. Since it is important to check extreme values, increase the portion of blocks with randomized residual up to 50% changing the following line in "residual" section:

```
"sb_randomized" : [9, 1], // no | yes
```

to

```
"sb_randomized" : [1, 1], // no | yes
```

Original version defines chances that superblock has random residual as one against nine, what yields only 10% of blocks with random residual. In modified version, the changes are one-to-one, encoder will produce 50% of such blocks.

"tx_mode" parameter described above should be set to allow 4x4, 8x8 and 16x16 transforms with equal chances (32x32 blocks are processed only with DCT):

```
"tx_mode" : [1, 1, 1, 0, 0], // 4x4, 8x8, 16x16, 32x32, select
```

In addition, you may disable intra-prediction mode, which don't utilize ADST (DC and 45°) changing "intra_mode" setting in "superblock" section the following way:

```
// //DC, V, H, 45,135,117,153,207, 63, TM  
"intra_mode" : [0, 1, 1, 0, 1, 1, 1, 1, 1, 1],
```

Summary: to validate ADST implementation we selected the parfile without inter frames as starting pointed; portion of blocks with random residual was increased in order to test inverse transform for handling extreme values; 32x32 transform size and transform-size definition at superblock level were disabled; also intra modes not utilizing ADST were disabled. This configuration is dedicated to validation of ADST transform, and if there is a possible mismatch with reference implementation, the chances that it will reveal it are several times higher than those for default all-enabled configuration are.

How to choose input resolution for certain cases

Recommended resolution for input video is 432x240 or close to it. This is trade-off between performance and test coverage. Globally we have two separate scopes of value randomization: frame header and superblock elements. Larger resolution (for example, Full HD) will lead to lower Encoder's speed in terms of FPS, what means that we will write lesser amount of frames and will try less amount of frame-header element combinations in the same period of time. For superblock-element-combination coverage, resolution has almost no matter. But using resolution than 432x240 will yield degenerating to 1-or-2-superblock-high video, what reduces coverage of dependencies between neighbor superblocks.

Some features need input video of large resolution for proper testing. For example, maximum magnitude of MV component is limited by resolution and is not reached at 432x240, so for testing MV extreme values Full HD input is recommended. Another example is number of vertical tiles. Minimum tile width is 256 pixels, so at bitstreams with frame width less than 512 pixels vertical tiles can't appear at all. It's recommended to test tiles on 4K+ content and also try frame width not divisible by 256.

Another specific feature is a frame scaling. First of all, it is recommended to separate validation of frame scaling and other features. The main reason is that scaling complicates the debugging process. And for frame-scaling test you need input of rather large resolution to test the edge cases of different reference-frame resolution: for proper testing of all combination it's recommended to have input file of dimensions which produce reasonable values after division by 32 (for example, 4096x2048 is ok).

Summary: basic recommended resolution is 432×240, it is a good trade-off between performance and frame-header feature coverage; some cases require Full HD or larger input resolution; frame scaling also requires large input resolution and we recommend to test other features thoroughly prior to frame-scaling validation.

How to control frame-scaling randomization

"scaling" section is the most unobvious one comparing to other parfile sections, but it is rather important because on-fly frame scaling is new feature in VP9 and requires special attention.

The idea of this parameter section is providing enough flexibility to generate basic scaling cases, corner cases and extensive-testing streams with arbitrary resolution. VP9 has limitation on reference scaling factor (2x downscale and 16x upscale), so the parameters don't control absolute values in pixels, only relative factors.

Random Encoder doesn't upscale the input frames, only downscales them. So the maximum resolution is defined by resolution of input video. Minimum frame size can be limited with "min_frame_size" parameter. Each frame is scaled independently on other frames, but 0.5x–16x ref scaling limit is taken into account.

"resolution_change" is the basic parameter with enables frame scaling. It is weight-based parameter, which defines the chances for the current frame to be scaled.

Three similar parameters are defined by pairs of floats: "intra_max_downscale", "max_upscale_factor" and "max_downscale_factor". Their first number defines max scale factor for frame width, the second one does for frame height. But they are applied in different cases. "intra_max_downscale" applies to key frames and intra-only frames; these types of frames are free from restrictions on reference frame size, so intra frames are allowed to be any size.

Randomizing resolution of inter frames, Encoder picks one reference frame with ref frame index from range defined by "base_ref_frame" parameter. Then for each dimension, Encoder decides will it be increased or decreased by chances from "downscale_or_upscale". After it Encoder picks random value in range between original size and size defined by max upscale/downscale factor. Minimum allowed value is 4, maximum is limited by input video resolution.

There is an extra parameter "scale_first_frame", which defines do we downscale the first frame or not. Why do we need this? It may be more simple case for decoder to start with the largest frame size to know how much memory to allocate. Don't expect such behavior in real world, but feel free to use it for debugging purposes. This parameter is defined by string, possible values are "no", "yes" or "maybe". "maybe" option leaves to decoder the decision to scale the first frame for the current stream or not.

In order to specify exact dimensions, parameter "frame_size_px" is used.

Extensive testing and parallel execution

Random Encoder as fast as ~50 000 cases a week for a single process. For satisfactory validation, it will need a month execution with NO fails in decoder on a corner case. It is possible to execute several processes in parallel on a single computer to increase coverage for the shorter time period as many as hardware memory and CPU cores allows.

You may split parallel tasks by parfiles, input resolutions and seed interval. It is reasonable to spend most of CPU time testing all-enabled stress parfile at small resolution, but also have some additional launches focusing on three following targets:

1. Large resolutions, tiles and long motion vectors;
2. Interpolation filters, motion vectors ≤ 8 pixels, randomized residual to test inverse transform and convolution for extreme cases;
3. Frame scaling with very large range of possible resolutions.

VP9 Random Encoder as a debugging tool

In addition to stress-testing after basic validation VP9 Random Encoder is useful validation tool, when it is necessary to understand what exactly feature has been broken after changes in decoder. Starting with 1.2.0 release, we include into Encoder's delivery the parfiles used for Stress Bitstreams generation.

Parfiles contains the same information as the spreadsheet defining the syntax-element utilization, but are more convenient for debugging purposes because they don't need any preparation for feeding to Encoder.

The Stress Bitstreams are organized in such way that complexity and number of utilized features increases with the stream index. So when you get a portion of streams failed validation we recommend to start with one having the lowest stream index value.

We propose the following workflow for debugging with Stress Bitstreams and Random Encoder:

1. Pick the failed stream with the lowest stream-index value and the stream with preceding stream-index value (which passes the validation).
2. Pick the corresponding parfiles and generate new streams with Random Encoder. Ensure that behavior of tested decoder on generated streams is the same as on original ones.
3. Eliminate from the "bad" parfile the syntax elements that don't cause fail. Most of parfile lines are mapped directly to VP9 syntax elements. And changes between consecutive parfiles include several syntax elements, usually related to one feature or similar features. Open "good" and "bad" parfile in side-by-side text comparison tool and look what if the feature difference between parfiles. Try to disable each feature and determine does it affect decoder behavior. For some case this information combined with knowledge about recent decoder modifications will be enough to figure out the cause of fail.
4. Narrow the scope of possible values for problem syntax element. When the problem syntax element is defined, sometimes it is possible to narrow the

scope of utilized values. We had investigated a case when a decoder produced wrong output on streams generated by Random Encoder, but behaved properly if we disable sharp interpolation filter. This was a rather good hint that problem is with the handling of sharp interpolation filter.

5. Continue simplification of the “bad” parfile by disabling other features.
This step pursues two goals: check cross-feature dependencies and simplify further debugging.
 - a. To illustrate the first goal, *intra_only* flag in header causes `setup_past_independence()` function call, which resets some of decoder contexts and buffers. Disabling *intra_only* element in parfile checks does this case affect the considered feature of “bad” parfile. Other noticeable features to be tested first for cross-dependence: frame resolution change, *show_frame* flag (frame visibility), segmentation, tiling, frame context operation flags.
 - b. Even if all this stuff didn’t help, it would be easier to go into deeper debugging if we have the stream with only the largest partitions (simpler call stack and debugging path) and the smallest transforms (easier to observe the values of the whole block). Of course, if the bug still occurs on such configuration. Also you can try to set max motion vector length to 0, leave only one interpolation filter, disable intra prediction on intra frames, disable compound prediction and so on.
6. Iterate with “--seed” option to find shorter reproducer.
Sometime the bug is occurring in rather rare cases and far from the beginning of the stream. To get shorter reproducer, limit the length of bitstream produced by random encoder by some small value and iterate with *seed* parameter value, which controls the initial state of random engine. For intra frame bugs, just one frame usually will be enough. For inter-prediction bugs, we recommend to do the reduction of stream length in several steps, because in case of specific dependency between several frames 2-frames reproducer may be not enough.

Smoke Test Streams

Since version 2.1 Random Encoder package in addition to default streams includes small set of streams for fast smoke testing. This set has the same reference code coverage as the default set, and allows to perform much faster test and validation cycle. Stream set has much smaller memory footprint in comparison with default set. As a drawback, it has minor degradation in syntax element values coverage and in cross-coverage of multiple syntax elements.

VP9 FC2 specifics

The Random Encoder supports all Profiles of VP9 format: color depth of 8, 10 and 12 bits with different settings of chroma subsampling. To enable high-bit-depth encoding use “--bit_depth” and “--colorspace” command-line options or “color” section in parfile. When command-line and parfile settings are used together, the command-line one has a priority. Generation of bitstreams with mixed bit depth or colorspace is possible only with parfile color setting. Supported values for “--bit_depth” option are 8, 10 and 12; for “--colorspace” option: “yuv420”, “yuv422”, “yuv440”, “yuv444” (without quotes).

The Random Encoder supports only 8-bit YUV 4:2:0 input and performs necessary color conversions internally. Bit-depth conversion is performed as shifting the 8-bit value by 2 or 4 bits. The lower bits are randomized. For chroma upscale the Encoder uses nearest-neighbor method.

To simplify validation and debugging we provide "--recon_bit_depth" option defining bit depth of YUV file with decoders reconstruct. Supported values are 8, 10, 12 and 16. libvpx decoder has the similar option "--output-bit-depth". It may be useful to decode all high-bit-depth enabled streams to 16 bits for easier comparison against ref decoder. Especially important case is stream with mixed bit depth: reference decoder defines output parameters as the first-frame parameters, and information loss is possible. The reference decoder has no any option to unify output chroma subsampling format, so neither has the Random Encoder.

Profile, bit depth and subsampling randomization settings can be defined in "color" section of JSON parfile. All the parameters in "color" section are weight-based. If you use --bit_depth or --colospace command-line options, "color" section in parfile is ignored.

VP9 format allows profile, bit depth and subsampling setting to change only at key and intra-only frames.

Residual Randomization

To stress-test residual decoding, Random Encoder provides two modes of residual randomization: randomized residual and randomized transform coefficients. Mode is defined with "type" parameter in "residual" section, available options are "off" (no residual randomization), "diff" (plain residual) and "transform" (transform coefficients). For each superblock, Random Encoder makes a decision to fill it with random residual or not. You can control the portion of such superblocks with "sb_randomized" parameter.

In randomized-residual mode, randomization is performed independently for each pixel of superblock. Pixel-residual value is picked from range defined by "residual_range" parameter (-255..255 by default). Please take into account that for 10- and 12-bit color depth this range adapted by the encoder internally, no adjustments are required. So if you need max range for 10-bit color, parameter value still needs to be [-255, 255] and Random Encoder will interpret as [-255..255] for 8-bit frames and as [-1023..1023] for 10-bit frames.

Transform-coefficient randomization is a little more complicated. Not all the coefficient configurations will have valid back-projection, so the encoder should be accurate in a way he randomizes. First, it picks end-of-block (EOB) token position. EOB token means that all coefficients starting from this position are equal to zero. Then the encoder picks random position in a range from DC coefficient to EOB and puts in this position a random value according coefficient position, transform size and type. If the result is valid transform (inverse transform yields residual in acceptable range), the encoder tries to put one more coefficient. When encoder finally fails to get a valid transform, it steps one iteration back and stops. Coefficient values are generated in a way to yield uniform distribution of the most significant bit, what allows producing a reasonable amount of extreme-value cases.

Another option related to transform randomization is "tx_zero_coeff_tail". Its purpose is to deliver coverage of the following corner case. In a transform block, the last coefficient can't be zero, EOB token must be always preceded with nonzero coefficient value. But in case when the block has all N×N coefficients, EOB token is omitted. A block filled with tail of zero coefficients instead of single EOB token will be valid one for VP9 decoder. The "tx_zero_coeff_tail" parameter is responsible for chances to get such zero-coefficient-tail ended block instead of EOB-token ended block. The parameter is applicable only for blocks with randomized, not reconstructed residual.

Residual and transform-coefficient randomization modes can't be used simultaneously in a single stream. Also, take into account that enabling any mode of residual randomization increases memory footprint proportionally to size of input frame in pixels.

What VP9 Random Encoder doesn't support

- Superframes. This feature is related to IVF container, not to VP9 format, you can find streams testing superframes in publicly available libvpx test stream pool;
- display_width and display_height different from source-frame size. Frame header may have flag, telling that display size is different, set equal to 1 (controlled through parfile), but actual display size values will be equal to source frame size. In case of different display size the part performing rescale is not related to decoder, so it will be almost impossible to achieve binary match for this, except you use the same interpolation algorithm.

Memory-bandwidth decoder testing

VP9 Random Encoder allows to stress-test decoder memory throughput at inter-prediction. *401_memory_bandwidth.json* parfile contains the encoder settings, which are targeted to maximize the amount of reads of reference frame surfaces. The key distinction of this parfile is usage only of 4×4 partitions. In combination with heavy interpolation utilization and compound prediction, 4×4 block requires reference-frame area of 13×13 pixels to be read twice. In addition, we disable all features using intra-prediction (like intra-only frames or segmentation reference-frame setting) and ZEROMV mode. To avoid the possibility of single reference prediction mode we use "--target mem_bw" command-line option, which enforces reference frame sign bias flags to have different values and adds in average 12% reads in inter-prediction.

The stream package contains the example stream of 1920×1080 resolution. To test other resolutions, you will need to generate test streams using the Random Encoder and the memory-bandwidth parfile.

Worst-case performance decoder testing

In addition to memory-bandwidth test we deliver parfiles for decoder-performance stress testing. We provide parfiles *402_worst_performance_19mbps.json*, *403_worst_performance_80mbps.json*, *404_worst_performance_200mbps.json*, which are targeted to stress the most CPU consuming parts of VP9 decoding pipeline: bool decoder, inter-prediction and loopfilter.

The main idea is to stress intra-prediction and loopfilter by small-prediction and transform block configuration. Key difference between 19mbps and 80mbps parfiles is switch from 8x8 prediction blocks to 4x4 ones.

The main tools to control bool-decoder load are QP and mode of forward probability update estimation. 200mbps parfile is derived from 80mbps one, the only different option is probability update mode. For 80mbps case encoder estimates optimal probability using on the basis of mode-decision stats, for 200mbps case it writes fully random probabilities.

Bitrates in the parfile names are approximate and were adjusted for the type of content we use for other streams at Full HD resolution. The Random Encoder doesn't have direct control for bitrate. The main way to adjust result bitrate are QP-related parameters and portion of blocks with randomized residual.

Visually clean content

Visual parfiles provide ability to check decoding results visually when it is impossible to validate binary match. They are based on the original parfiles, but have some modifications to eliminate visual artifacts: reference-frame scaling, skip-coefficients flag and residual randomization are disabled, range of possible QP is reduced. Random Encoder has a special mode to repeat all invisible frames with `show_existing_frame` flag; to enable it, add `--target visual` to encoder's options.

Visual parfiles are organized in the same way as standard ones, and have letter "V" in front of the parfile index.

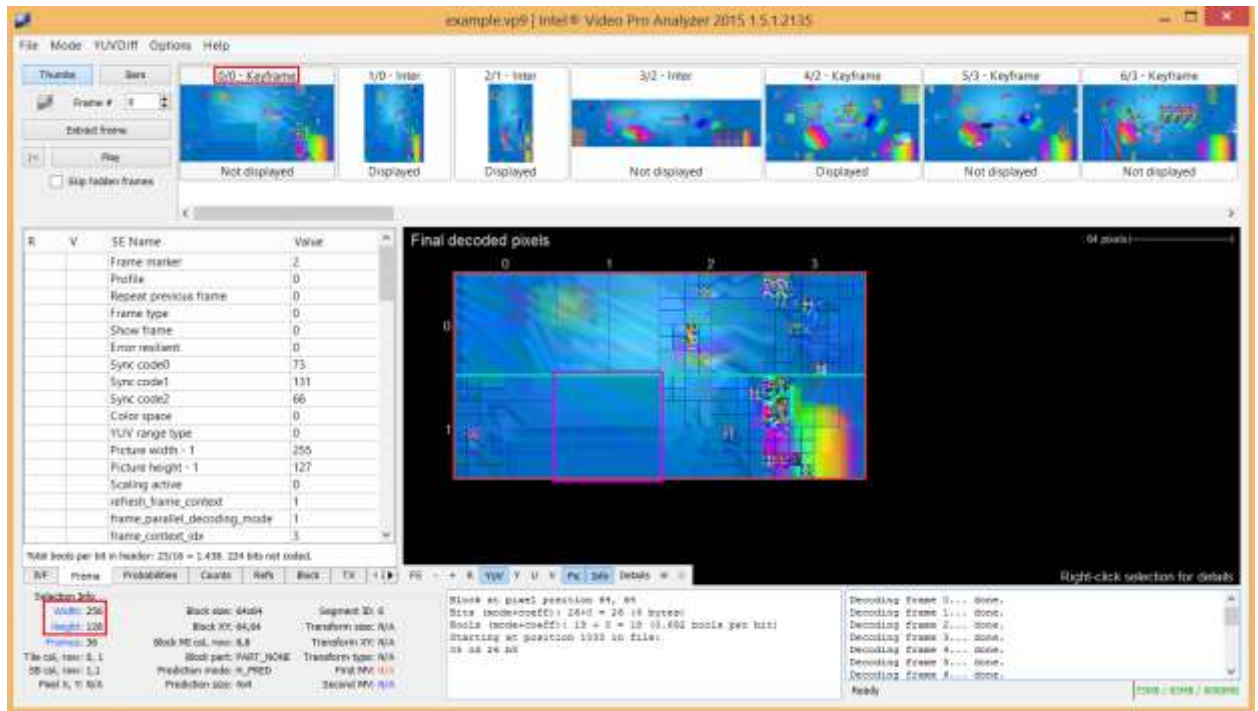
Overriding parameters

To redefine parameters for specific frames you should put them in "override" section of the JSON file. This section has the following structure:

```
"override" : {
  "0,4..6" : { // frames 0, 4, 5 and 6 will use settings specified below instead of default ones
    "frame" : {
      "frame_type" : [1, 0] // only key frames
    },
    "scaling" : {
      "frame_size_px" : [256, 128] //exact frame size [width, height]
    }
  },
  "1..3" : {
    "frame" : {
      "frame_type" : [0, 1] // only inter frames
    }
  }
}
```

In this example "0,4..6" is a string, specifying for which frames new settings will apply. You could set individual frames and frame ranges a..b, separated by commas. Attribute "frame_type" is redefined in both parameter sets shown in example, making frames 0, 4, 5, 6 key-frames, and 1, 2, 3 inter-frames. Parameters which are not set in overriding parameter set are taken from main parfile.

It's easy to visualize effect of this settings using [Intel® Video Pro Analyzer](#):



If we'd open a stream, generated by example parfile below, in Video Pro Analyzer we can actually see that frame types are set according to our settings, and frames 0, 4-6 have the size of 256x128 pixels.

Iterating Parameters

Since version 2.1 most of the parameter ranges in JSON file could be replaced with iterations:

(Start, Step, End)

Each call to certain parameter will return previous value of this parameter incremented by *Step*, but not greater than *End*. First used value will be equal to *Start*. When value exceeds *End*, iteration starts again from *Start*. This behaviour could be useful if you want to cover certain range in a deterministic way.

For example, "frame_type" : [1, 1] will give key (parameter value 0) and inter (parameter value 1) frames with equal probabilities, while "frame_type" : (0, 1, 1) will produce sequence of alternating frame types (key, inter, key, inter, ...).

As another example, "show_existing_idx" : (0, 1, 7) will allow you to cover each possible existing frame buffer index one by one.

Currently, some elements, which represent size in "scaling" section, such as "frame_size_px" or "min_frame_size", are not supported.

Also note that with iterations specified in braces, parfile is no longer a valid JSON file.

Error Resilience Encoder

Description

Error Resilience Encoder for VP9 is a tool that allows you to generate broken video streams to test behavior of VP9 decoder on various types of errors. What's more important – it allows to control type and positioning of errors, which makes stream invalid. This tool can be used to pinpoint flaws in error handling and to more precisely define expected behavior on the wide range of possible errors.

Error Resilience Encoder is based on Intel® Random Encoder: at first, valid compliant stream is generated with Random Encoder, utilizing all of its flexibility, then destructive changes are applied to it, based on user-set parameters.

Error Resilience Encoder accepts as an input a *YUV file* and a *parfile* – a JSON-formatted file, describing testing settings: features to utilize, fixed values, random values. As an output, Encoder produces encoded broken bitstream.

Important restrictions

Although Error Resilience Encoder is an extension of Random Encoder, it's not intended to generate valid streams. Currently, only first 15 frames of the stream are fully customizable by user. For the rest of the frames, if user-set parameters don't meet the required level of degradation, additional errors are introduced into stream. Error Resilience Encoder comes as a separate product from Random Encoder.

Broken stream generation

Rules of error generation are described by "broken" section in parfile. If in doubt, [see example of the parfile below](#). This chapter explains its parameters in detail. There are 3 invalidation options: bitwise randomization, packet-level failures and corrupted syntax elements. Corresponding subsections in JSON are: "bit", "packet" and "syntax".

Common

Each of the subsections has string parameter "frames". If specified, parameters of this subsection will apply only to selected frames. Format of this string is similar to overrides: "0,1..3" means "generate this type of errors only for frames 0, 1, 2 and 3".

Parameter "keep_file_header" in "broken", if set to non-zero guards ivf file header from changes. Otherwise, "ivf_header_prob" probabilities are used to corrupt file header in "bit" and "packet" modes.

Bit randomization

The idea of bit randomization is to simply invert some bits in valid VP9 stream. Main parameter here is the probability with which specific bit will be inverted – parameter "prob" in subsection "bit". Each frame of the generated VP9 stream consists of separately coded parts: container (ivf) header, uncompressed header, compressed header (probabilities), and compressed frame data. Error Resilience Encoders allows to specify separate probabilities for each of this parts.

Corresponding parameters are: "ivf_header_prob", "unc_header_prob", "comp_header_prob", "frame_data_prob". If some of this parameters are not specified explicitly, default "prob" is used instead. Bit randomization allows to simulate low-level network and storage errors.

Packet Randomization

Main idea is to operate on packet level, i.e. with set of bytes. "prob" in "packet" section stands for probability for each byte whether corrupted packet starts with this byte. Parameter "size" : [min, max] determines ranged distribution of the length of this packet. This packet could be then cut from the stream, turned to zeroes, or duplicated, depending on parameter "mode", which is weighted distribution. As special case, packet-level randomization allows you to cut sections or entire frames from stream (see parfiles *004_drop_frames.json* and *005_header_mismatch.json* in Error Resilience package for example).

Syntax randomization

This section allows you to corrupt syntax elements, where bitstream specification made it possible. Mostly it consists of the elements of the uncompressed header. Each parameter is a probability with which this feature will be enabled. The following table lists all of the features.

"frame_marker"	Change in constant value, which marks each VP9 frame
"sync_code"	Change in sync code, used in keyframes
"reserved_bit"	Set bit, which is reserved to zero
"header_size"	Corrupt header size, written after uncompressed header
"profile"	Set invalid profile value
"subsampling"	Set 4:2:0 subsampling for 1 and 3 profiles
"reference"	Use invalid reference, if possible. Invalid references might be either uninitialized (stream starts with intra-only), or of wrong size (VP9 has restriction on relation between frame sizes), or of wrong profile
"context"	Use uninitialized entropy context
"transform_coeff"	Use coefficients which results in overflow after inverse transform
"tile_count"	Corrupt tile count encoding
"dimensions"	Write wrong dimensions in header

Following values are valid, but not fully covered by randomization in Random Encoder.

"force_write_dimensions"	Explicitly write dimensions on inter-frames
"color_range "	YUV color range
"colorspace"	Random colorspace

Parfiles description

000_all_features.json	All features written in parfile
001_corrupted_headers.json	Invert bits in headers (ivf, uncompressed, compressed)
002_packet_failures.json	Packet randomization
003_frame_parallel.json	Generated streams contains two independent streams, of which only one is corrupted. Demonstrates use of override feature to create independent streams
004_drop_frames.json	Demonstrates ability to cut entire frames
005_header_mismatch.json	Demonstrates ability to cut specific parts
006_uninitialized_structures.json	Starts with intra-only, generates randomly either uninitialized entropy context, or uninitialized reference
007_invalid_scale_reference.json	Cut valid reference to produce error
008_missing_existing_frame.json	Cut first frame, trying to show existing

Parfile example and parameter description

Here we provide the example parfile, which has randomization of all the features enabled. Each syntax element is provided with comment about it values. If it is unobvious what if the type of the parameter (weight or range), check the table below.

```
{
"frame" : {
  "frame_type"      : [1, 10],          // key | inter
  "start_w_intra_only" : [4, 1],        // no | yes, start bitstream with intre-only instead
                                          // of key frame
  "tx_mode"         : [1, 1, 1, 1, 2], // 4x4, 8x8, 16x16, 32x32, select
  "interp_filter"   : [1, 1, 1, 1, 4], // reg, smooth, sharp, bilin, switchable
  "horz_tile_range" : [0, 2],          // log2 val, [0..2]
  "vert_tile_range" : [0, 6],          // log2 val, [0..6], 2 is max reasonable for FullHD
  "frame_context_idx" : [1, 1, 1, 1], // context id
  "allow_hp"        : [1, 1],          // 1/8-pe1 MV accuracy, off/on
  "error_resilient" : [5, 1],          // no | yes
  "frame_parallel"  : [1, 1],          // no | yes
  "reset_context"   : [1, 1, 1, 1],    // do nothing | invalid | reset current | reset all
  "refresh_context" : [1, 1],          // no | yes
  // Lossless forces base_qindex and q deltas to zero.
  // Without this control it would be difficult to generate proper mix of lossless
  // and lossy frames in one stream.
  "lossless"        : [4, 1],          // no | yes
  "write_display_size" : [1, 1],        // no | yes -- write display size flag
                                          // the frame size is kept from source
  "base_qindex_range" : [0, 255],      // 0 to 255, zero means lossless
  "y_dc_delta_q_range" : [-15, 15],    // [-15..15]
  "uv_ac_delta_q_range" : [-15, 15],   // [-15..15]
  "uv_dc_delta_q_range" : [-15, 15],   // [-15..15]
  "lf_filter_level"   : [0, 63],        // [0..63], 6 bit
  "lf_sharpness_level" : [0, 7],        // 3 bit
  "lf_delta_enable"   : [1, 1],         // no | yes
  "lf_delta_update"   : [1, 1],         // no | yes
  "lf_mode_delta_range" : [-63, 63],    // 6 bit + sign
  "lf_ref_delta_range" : [-63, 63],    // 6 bit + sign
  "seg_enabled"       : [1, 1],         // no | yes
  "seg_update_map"    : [1, 1],         // no | yes
  "seg_update_data"   : [1, 1],         // no | yes
  "seg_abs_delta"     : [1, 1],         // delta | abs value
  "seg_temporal_update" : [1, 1],       // no | yes
  "seg_alt_q_enable"  : [1, 1],         // no | yes
  "seg_alt_q_range"   : [0, 127],      // [0..127], if seg_abs_delta == 1
  "seg_alt_q_delta"   : [-127, 127],   // [-127..127], if seg_abs_delta == 0
  "seg_alt_lf_enable" : [1, 1],         // no | yes
  "seg_alt_lf_range"  : [0, 63],        // if seg_abs_delta == 1
  "seg_alt_lf_delta"  : [-63, 63],     // if seg_abs_delta == 0
  "seg_ref_enable"    : [4, 1],         // no | yes
  "seg_ref"           : [1, 1, 1, 1],   // intra | last | golden | altref
  "seg_skip_zeromv"   : [10, 1],       // no | yes
  // Ref-frame-index settings are ignored in case when not all reference buffers are suitable
  // for prediction: configurations with enabled frame scaling, bit-depth or
  // chroma-subsampling randomization and when first frame is intra-only and all buffers were
  // initialized.
  "lst_fb_idx_range" : [0, 7],          // [0..7] last frame ref id range
  "gld_fb_idx_range" : [0, 7],          // [0..7] golden frame ref id range
  "alt_fb_idx_range"  : [0, 7],          // [0..7] alt frame ref id range
  "refresh_mask_bit"  : [2, 1],         // probs that bit in refresh mask = 0 | 1
  "refresh_mask"      : "01000100",    // explicitly set which frames to refresh;
                                          // overrides refresh_mask_bit
  "ref_frame_sign_bias" : [2, 2],       // probs for ref frame sign bias = 0 | 1
  "lst_frame_sign_bias" : [1, 0],       // sign bias when last frame is selected as reference;
                                          // when not specified defaults to ref_frame_sign_bias
  "gld_frame_sign_bias" : [0, 1],       // sign bias when gold frame is reference
  "alt_frame_sign_bias" : [0, 1],       // sign bias when alt frame is reference
  "show_frame"        : [1, 2],         // no | yes
  "intra_only"        : [2, 1],         // no | yes
  "show_existing_frame" : [5, 1],       // no | yes - repeat existing frame from ref buffer
  "show_existing_idx" : [1, 1, 1, 1, 1, 1, 1, 1], // prob weights for existing frame ref idx
  "comp_pred_mode"    : [1, 1, 1],     // single only | compound only | hybrid
  // frame context update

```

```

// * no update -- keep the probabilities from the previous frame
// * optimal probs -- estimate optimal probabilities for the frame and write them
// in forward update
// * random probs -- randomize frame context probs for the frame
"fc_update_type" : [1, 1, 1], // no update | optimal probs | random probs
"fc_rnd_prob_range" : [1, 255] // [1..255] (only for random probs)
},
"superblock" : {
  "ref_frame" : [1, 1, 1, 1], // intra | last | golden | altref
  // //DC, V, H, 45,135,117,153,207, 63, TM
  "intra_mode" : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
  "inter_mode" : [1, 1, 1, 1], // nearestmv | nearmv | zeromv | newmv
  "mv_len" : 16376, // max len * 8 (with fractional part bits)
  "skip" : [9, 1], // no | yes
  "interp_filter" : [1, 1, 1], // reg, smooth, sharp
  "comp_pred" : [1, 1], // no | yes, if comp_pred_mode == hybrid
  "comp_pred_var_ref" : [1, 1], // ref_idx, 0 or 1
  "seg_id" : [1, 1, 1, 1, 1, 1, 1, 1], // 8 segments
  "seg_id_predicted" : [3, 1], // no | yes

  "partitions" : [
    // N, H, V, S
    [1, 1, 1, 1], // 8x8
    [1, 1, 1, 3], // 16x16
    [1, 1, 1, 6], // 32x32
    [1, 1, 1, 9] // 64x64
  ],

  "tx_size" : [
    // 4, 8,16,32
    [1, 1], // blocks less than 16x16
    [1, 1, 1], // blocks less than 32x32
    [1, 1, 1, 1], // when any transform size suits
  ]
},
"color" : {
  // Bitstream profiles indicated by 2-3 bits in the uncompressed header.
  // Profile 0. 8-bit 4:2:0 only.
  // Profile 1. 8-bit 4:4:4, 4:2:2, and 4:4:0.
  // Profile 2. 10-bit and 12-bit color only, with 4:2:0 sampling.
  // Profile 3. 10-bit and 12-bit color only, with 4:2:2/4:4:4/4:4:0 sampling.
  "profile" : [1, 0, 0, 0], // bitstream profile, 0..3
  "bitdepth" : [1, 0], // 10 or 12 bit if profile >= 2
  "subsampling_x" : [0, 1], // chroma subsampling, no | yes, if profile == 1 or 3
  "subsampling_y" : [0, 1] // chroma subsampling, no | yes, if profile == 1 or 3
},
"scaling" : {
  // Input file defines maximum resolution.
  // intra_max_downscale pair defines maximum possible downscale on key frames and intra-only
  // frames for width and height correspondingly.
  //
  // For example, if we have source 1920x1080 and "intra_max_downscale" : [2, 4],
  // then frame width for intra frames will vary in range of 960..1920 (up to 2x downscale)
  // and the range for frame height will be 270..1080 (up to 4x downscale).
  //
  // Randomizing resolution of inter frames, first we pick one reference frame with
  // ref frame index from range defined by "base_ref_frame" parameter. Then for each dimension
  // we decide will it be increased or decreased by chances from "downscale_or_upscale".
  // After it we pick random value in range between original size and size defined by max
  // upscale/downscale factor.

  "resolution_change" : [0, 1], // probs of resolution change <no | yes>
  "scale_first_frame" : "yes", // "no" | "maybe" | "yes"
  "intra_max_downscale" : [16, 16], // width, height (floating point allowed)
  "base_ref_frame" : [0, 7], // ref frame to borrow resolution if size_from_ref == 1
  "downscale_or_upscale" : [1, 1], // chances to decrease and increase frame size
  "max_upscale_factor" : [16, 16], // width, height (floating point allowed), max = 16
  "max_downscale_factor" : [2, 2], // width, height (floating point allowed), max = 2
  "min_frame_size" : [4, 4] // width and height in pixels,
  // min frame size restrictions
},
"residual" : {
  // Only one mode of residual randomization per stream can be used.
  "type" : "diff", // off | diff | transform

```

```

"sb_randomized" : [9, 1], // no | yes
"residual_range" : [-255, 255], // min and max residual value (max range is [-255, 255])

// For tx mode if inverse transform of random-generated block doesn't produce valid
// residual, encoder tries to regenerate it, and stops after several unsuccessful tries.
// Increasing the range will reduce the chances to generate valid transform and the portion
// of random blocks in bitstream.
"tx_eob_range" : [
  [1, 16], // tx 4x4
  [1, 64], // tx 8x8
  [1, 256], // tx 16x16
  [1, 1024] // tx 32x32
]
},
"broken" : {
  "keep_file_header" : 1,
  "bit" : {
    "frames" : "0..5,10..15",
    //"prob" : 0.005,

    "ivf_header_prob" : 0.001,
    "unc_header_prob" : 0.001,
    "comp_header_prob" : 0.0001,
    "frame_data_prob" : 0.00001
  },
  "packet" : {
    "frames" : "6,8,10,12",
    "size" : [2, 8],
    //"prob" : 0.001,

    "ivf_header_prob" : 0,
    "unc_header_prob" : 0.01,
    "comp_header_prob" : 0.001,
    "frame_data_prob" : 0.0001,
    "mode" : [1, 1, 1] //loss, duplicate, zero
  },
  "syntax" : {
    "frames" : "7..9,11,13",
    //"prob" : 0.1,

    "frame_marker" : 0.2,
    "sync_code" : 0.2,
    "reserved_bit" : 0.2,
    "header_size" : 0.2,
    "profile" : 0.2,
    "subsampling" : 0,
    "reference" : 0.5,
    "context" : 0.4,
    "color_range" : 0,
    "colorspace" : 0,
    "dimensions" : 0,
    "force_write_dimensions" : 1,
    "tile_count" : 0,
    "transform_coeff" : 1,
  }
},
"override" : {
  "0,4..6" : { // frames 0, 4, 5 and 6 will use settings specified below instead of default ones
    "frame" : {
      "frame_type" : [1, 0] // only key frames
    },
    "scaling" : {
      "frame_size_px" : [256, 128] //exact frame size [width, height]
    }
  },
  "1..3" : {
    "frame" : {
      "frame_type" : [0, 1] // only inter frames
    }
  }
}
}
}

```

Table 1. Randomization-control parameter types and acceptable values

Parameter	Type	Enumeration (for weights) and acceptable range (for other types)	Default value
Frame section			
frame_type	weights	KEY, INTER	[1, 24]
start_w_intra_only	weights	False, True	[1, 0]
tx_mode	weights	ONLY_4X4, ALLOW_8X8, ALLOW_16X16, ALLOW_32X32, TX_MODE_SELECT	[1, 1, 1, 1, 1]
interp_filter	weights	EIGHTTAP, EIGHTTAP_SMOOTH, EIGHTTAP_SHARP, BILINEAR, SWITCHABLE	[1, 1, 1, 1, 1]
horz_tile_range	range	[0..2]	[0, 0]
vert_tile_range	range	[0..6]	[0, 0]
frame_context_idx	weights	0, 1, 2, 3	[1, 1, 1, 1]
allow_hp	weights	False, True	[1, 1]
error_resilient	weights	False, True	[1, 0]
frame_parallel	weights	False, True	[1, 0]
reset_context	weights	0, 1, 2, 3	[1, 0]
refresh_context	weights	False, True	[1, 1]
lossless	weights	False, True	[1, 0]
write_display_size	weights	False, True	[1, 1]
base_qindex_range	range	[0..255]	[40, 40]
y_dc_delta_q_range	range	[-15..15]	[0, 0]
uv_ac_delta_q_range	range	[-15..15]	[0, 0]
uv_dc_delta_q_range	range	[-15..15]	[0, 0]
lf_filter_level	range	[0..63]	[0, 63]
lf_sharpness_level	range	[0..7]	[0, 7]
lf_delta_enable	weights	False, True	[1, 1]
lf_delta_update	weights	False, True	[1, 1]
lf_mode_delta_range	range	[-63, 63]	[-63, 63]
lf_ref_delta_range	range	[-63, 63]	[-63, 63]
seg_enabled	weights	False, True	[1, 1]
seg_update_map	weights	False, True	[1, 1]
seg_update_data	weights	False, True	[1, 1]
seg_abs_delta	weights	False, True	[1, 1]
seg_temporal_update	weights	False, True	[1, 1]
seg_alt_q_enable	weights	False, True	[1, 1]
seg_alt_q_range	range	[0, 127]	[20, 40]
seg_alt_q_delta	range	[-127, 127]	[-20, 20]
seg_alt_lf_enable	weights	False, True	[1, 1]
seg_alt_lf_range	range	[0, 63]	[0, 63]
seg_alt_lf_delta	range	[-63, 63]	[-63, 63]
seg_ref_enable	weights	False, True	[1, 1]
seg_ref	weights	INTRA_FRAME, LAST_FRAME, GOLDEN_FRAME, ALTREF_FRAME	[1, 1, 1, 1]
seg_skip_zeromv	weights	False, True	[1, 1]
lst_fb_idx_range	range	[0..7]	[0, 0]
gld_fb_idx_range	range	[0..7]	[1, 1]
alt_fb_idx_range	range	[0..7]	[2, 2]
refresh_mask_bit	weights	0, 1	[1, 2]
refresh_mask	bitmask	[00000000..11111111]	refresh_mask_bit
ref_frame_sign_bias	weights	0, 1	[2, 1]
lst_frame_sign_bias	weights	0, 1	ref_frame_sign_bias
alt_frame_sign_bias	weights	0, 1	ref_frame_sign_bias
gld_frame_sign_bias	weights	0, 1	ref_frame_sign_bias
show_frame	weights	False, True	[0, 1]
intra_only	weights	False, True	[2, 1]
show_existing_frame	weights	False, True	[1, 0]
show_existing_idx	weights	0, 1, 2, 3, 4, 5, 6, 7	[1, 1, 1, 1, 1, 1, 1, 1]
comp_pred_mode	weights	SINGLE_REFERENCE, COMPOUND_REFERENCE,	[2, 1, 2]

REFERENCE_MODE_SELECT			
fc_update_type	weights	No update, optimal probabilities, random probabilities	[1, 1, 1]
fc_rnd_prob_range	range	[1..255]	[1, 254]
Superblock section			
ref_frame	weights	INTRA_FRAME, LAST_FRAME, GOLDEN_FRAME, ALTREF_FRAME	[1, 1, 1, 1]
intra_mode	weights	DC_PRED, V_PRED, H_PRED, D45_PRED, D135_PRED, D117_PRED, D153_PRED, D207_PRED, D63_PRED, TM_PRED	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
inter_mode	weights	NEARESTMV, NEARMV, ZEROMV, NEWMV	[1, 1, 1, 1]
mv_len	integer	0..16384	1280
mv_joint_zero	weights	False, True	[1, 0]
skip	weights	False, True	[10, 1]
interp_filter	weights	EIGHTTAP, EIGHTTAP_SMOOTH, EIGHTTAP_SHARP	[1, 1, 1]
comp_pred	weights	SINGLE_REFERENCE, COMPOUND_REFERENCE	[1, 1]
comp_pred_var_ref	weights	0, 1	[1, 1]
seg_id	weights	0, 1, 2, 3, 4, 5, 6, 7	[1, 1, 1, 1, 1, 1, 1, 1]
seg_id_predicted	weights	False, True	[3, 1]
partitions	weights	PARTITION_NONE, PARTITION_HORZ, PARTITION_VERT, PARTITION_SPLIT	// N, H, V, S [1, 1, 1, 1], // 8x8 [1, 1, 1, 3], // 16x16 [1, 1, 1, 6], // 32x32 [1, 1, 1, 9] // 64x64
tx_size	weights	TX_4X4, TX_8X8, TX_16X16, TX_32X32	[1, 1], [1, 1, 1], [1, 1, 1, 1]
Color section			
profile	weights	Profile 0, profile 1, profile 2, profile 3	[1, 0, 0, 0]
bit_depth	weights	10 bit, 12 bit	[1, 0]
subsampling_x	weights	False, True	[0, 1]
subsampling_y	weights	False, True	[0, 1]
Scaling section			
resolution_change	weights	False, True	[1, 0]
frame_size_px	int	[width, height], no more than input_frame_size	[width, height]
intra_max_downscale	float	[1.0 .. 16.0]	1.0
base_ref_frame [0, 7],	range	[0..7]	[0, 7]
downscale_or_upscale	weights	Downscale, upscale	[1, 1]
max_upscale_factor	float	[1.0 .. 16.0]	1.0
max_downscale_factor	float	[1.0 .. 2.0]	1.0
scale_first_frame	string	"no", "maybe", "yes"	"no"
min_frame_size	Int	[1..input_frame_size]	[4, 4]
Residual section			
type	string	"off", "diff", "transform"	"off"
sb_randomized	weights	False, True	[1, 0]
residual_range	range	[-255..255]	[-255, 255]
tx_eob_range	range	[0, 16], // tx 4x4 [0, 64], // tx 8x8 [0, 256], // tx 16x16 [0, 1024] // tx 32x32	[0, 16], // tx 4x4 [0, 64], // tx 8x8 [0, 256], // tx 16x16 [0, 1024] // tx 32x32