VrankenFuzz

a multi-sensor, multi-generator mutational fuzz testing engine

By Guido Vranken <guido@guidovranken.com>, May and July 2018
Website: https://guidovranken.com/

Introduction

Intended audience

This document is directed at computer security specialists, specifically those with an interest in leveraging fuzz testing to find bugs in software applications.

This research and its implementation is especially suited for people who perform hands-on (active, not passive) vulnerability research.

Most of the examples set out from a situation where the analyst has access to the source code.

Core concepts

Years of hands-on experience with fuzzers have lead me to identify what in my estimation, and in relation to my needs, are several shortcomings in the existing fuzzing engines. To overcome these shortcomings, I have reasoned about the essence of fuzz testing and have broken it down into several general concepts. Each of these concepts have been implemented separately in VrankenFuzz. In addition to this, whereas a traditional fuzzer will only manipulate the target, VrankenFuzz allows the target to dynamically manipulate the fuzzer. The atomization of core fuzz testing primitives and bidirectional influencing is what summarizes VrankenFuzz and what makes it unique (to the best of my knowledge).

One of these primitives is *sensoring*. Earlier, I released libfuzzer-gv[1], which implemented stack depth guided fuzzing, code intensity guided fuzzing, allocation size guided fuzzing, and custom guided fuzzing. VrankenFuzz proceeds to build on the realization that *not just code coverage, but any quantifier* can be used to guide fuzzing.

Another primitive is the *Generator*. Generators allow a target application to have not just one, but multiple inputs. An obvious use case is targets that simply require multiple input variables. But there are also less obvious reasons that merit the use of multiple Generators, even if a target (normally) takes just one input. Why and how I will explain below.

The third primitive is the *Processor*. The Processor defines how to interpret Sensor data. Different interpretations are useful in different circumstances.

1 https://github.com/guidovranken/libfuzzer-gv

Use cases

Fuzzing anything. Fuzzing programs written in other languages that do not support traditional instrumentation. If you have some way of knowing which code paths are taken, you can probably fuzz it with VrankenFuzz.

Differential fuzzing. Each of multiple implementations of the same specification can have its own code coverage tracked separately by leveraging VrankenFuzz' multi-sensor facilities.

Black-box fuzzing, even remotely. A logical categorization of a black-box application that is in line with its internal state allows it to be fuzzed. Create a sensor for a remote web application's execution time (eg. round-trip time minus average round-trip time) and the fuzzer will tend to generate inputs that lead to an ever-greater execution time[2] (= denial-of-service). Create a sensor for the total number of unique error codes that a HTTP server has returned for a body of inputs, and you will end up with a corpus that likely constitutes a set of diverse internal states within the remote server.

Fuzzing "unfuzzable" code. Reaching code that is, *by definition*, not reachable with a traditional fuzzer, for example because it essentially requires cryptography to be defeated. An example can be found later in this document.

Fuzzing binaries. In conjunction with a processor emulator interface framework like Unicorn[3], it is possible to provide precise code coverage guided fuzzing of binaries (by representing the processor's program counter with a Sensor).

A supplement to manual audits. VrankenFuzz is very versatile and once you internalize its logic, several approaches can be used to exert fine-grained control. For example, you can force it to focus on a certain function within the target, instead of hoping the fuzzer will hit that function repeatedly by happenstance. You can force certain inputs, for example inputs that take a long time to execute, or inputs that consume a lot of memory, to be discarded (not added to the corpus), thereby creating a corpus of fast-executing or low-memory inputs. Etcetera.

Interactive fuzzing

What and why

If your objective is to find bugs in an application, there are several tactics at your disposal, one of which is fuzz testing. But writing a fuzzer for a target and letting it run in the background may

2 This assumes a certain inherent relationship between input data and execution time that can be explored gradually.
3 https://www.unicorn-engine.org/

not always be as efficient as you'd expect.

Fuzzers are an excellent tool for very quickly covering code of *a certain type of complexity*.

A fuzzer's reach can rapidly extend across a maze of functions and branches that appear byzantine to the human mind, for whom the overarching cohesion is lost in syntactic or semantic complexity.

The human mind is better at reasoning about other types of code, that conversely are "blockers" for the fuzzer. The mind can recognize certain constructs in the target code as being unresolvable through sheer trial-and-error testing (as the fuzzer essentially does), such as verifying a MAC that was computed over the (dynamic) input data.

In the quest for program bugs, the fuzzer and the human mind can complement one another in an endeavor I call *interactive fuzzing*.

When performing interactive fuzzing, the task of the analyst is twofold:

- See to the efficacy of the fuzzer: is it still reaching new code? Why is it not reaching new code? Is code coverage of the highest importance in this target, or can other variables play a meaningful role as well?
- Push the fuzzer in the right direction: comment out code (if available) in the target where it makes sense. Construct an input manually in a hex editor. Drive it towards other types bugs than memory violations, like excessive memory consumption, or deep recursions, or very slow executions (denial-of-service). Add assertions (eg. `if ( ... ) abort()`) where that makes sense. When using VrankenFuzz: can I implement Sensors and Generators at strategic places?

Circumventing blockers

Cryptographic blockers

A noteworthy example of blockers can be found in applications that employ cryptography. If we imagine a network protocol that authenticates incoming data by computing a Message Authentication Code (MAC)[4], execution will never move past the data authentication phase, because a fuzzer simply does not possess the ability to crack the MAC, and the crucial comparison between expected and computed value never resolves to true.

In the case of MAC or hash verification, dictionaries are certainly not a viable strategy: no manageable repository of static dictionary values can ever match the MAC or hash that the application computes over the dynamically generated input messages.

So what I would do in this case is comment out the original comparison between computed

4 https://en.wikipedia.org/wiki/Message_authentication_code

MAC and expected MAC, and replace it with a choice based on a pseudo-random boolean that decides whether to proceed or exist. Essentially the following:

```
/* Original */
bool process_message(const uint8_t* msg_data, size_t msg_size)
{
     const auto expected = expectedMAC();
     const auto computed = computeMAC(msg_data, msg_size);

     if ( expected != computed ) {
         return false;
     } else {
         return true;
     }
}

/* Version used in audit */
bool process_message(const uint8_t* msg_data, size_t msg_size)
{
     const auto expected = expectedMAC();
     const auto computed = computeMAC(msg_data, msg_size);

     if ( (rand() % 2) == 0 ) {
         return false;
     } else {
         return true;
     }
}
```

But editing the target source code to suit your needs, isn't that defying the purpose of the audit, because your objective is to find inputs that crash the application *in its original form*?
No, in this case it is not, because with this modification we mimic what could also happen in an unaltered production environment: each input message can either contain a valid MAC or an invalid MAC. So by altering the source code, we have not *deviated* from the similarity to a production environment, but we have made the fuzzing environment *more like it*.

But there is one problem with the modified `process_message()`. It succeeds or fails based on `rand()`. But `rand()` is fickle – unless repeatedly seeded with the same value, its PRNG state is retained across iterations without regard for the input at hand, and this breaks the determinism of the fuzzer. Concretely, if we have input *I*, `rand() % 2` may produce 0 if we run *I* one time, and produce 1 if we run *I* again. This leaves us in a situation where it will be more difficult to reproduce crashes, and a sub-optimal functioning of the fuzzing engine's algorithms (because it cannot determine whether it was a mutation that lead to new code coverage, or if it was the output of `rand()`, and it will decide that it was the former, leading to unwarranted

addition of inputs to the corpus).

It would be helpful if we could just pull some additional data from the fuzzer engine in the middle of the target. This is what Generators are for in VrankenFuzz.

On trickle-down logic

Consider an application that implements a network protocol. It expects a well-formed message to be encapsulated within several layers of coding schemes.

One layer re-assembles several separate network packets, into a single, meaningful message.
The next layer takes this message, which it expects to be base64-encoded, and decodes it from base64 into binary.
The third layer uses an ASN1 decoder to decode the binary message into several distinct data types.
The fourth layer validates these variables with some custom logic.

```
enum {
    PROTOCOL_HANDSHAKE = 0x9999;
};

std::vector<uint8_t*> reassemble_packets(std::vector<packet_t>&
packets)
{
    /* validate sequence numbers */
    size_t prevSeq = 0;
    for ( const auto& p : packets ) {
        if ( p.sequenceNumber < prevSeq ) {
            return false;
        }
        prevSeq = p.sequenceNumber;
    }

    /* re-assemble the packets into an InputMessage */
    /* ... */

    /* upon success: */
    return inputMessage;

    /* upon failure: */
    return nullptr;
}

std::vector<uint8_t*> base64Decode(std::vector<uint8_t*> data)
{
```

```
    /* Attempt to base64-decode data */
    /* ... */

    /* upon success: */
    return base64Decoded;

    /* upon failure: */
    return nullptr;
}

bool decode(std::vector<packet_t>& packets)
{
    std::vector<uint8_t>* inputMessage;
    if ( (inputMessage = reassemble_packets(packets)) == nullptr ) {
        return false;
    }

    std::vector<uint8_t>* base64Decoded;
    if ( (base64Decoded = base64Decode(inputMessage)) == nullptr ) {
        return false;
    }

    ASN1Data* asn1Decoded;
    if ( (asn1Decoded = asn1Decode(base64Decoded)) == nullptr ) {
        return false;
    }

    if ( asn1Decoded->version != 0x1234 ) {
        return false;
    }

    if ( asn1Decoded->operation != PROTOCOL_HANDSHAKE ) {
        return false;
    }

    /* and so on */
    /* ... */
}
```

You could call this code pattern tricke-down logic:

- *If and only if* the packet re-assembly succeeds will Base64 decoding be attempted.
- *If and only if* the Base64 decoding succeeds will ASN1 decoding be attempted
- *If and only if* ASN1 decoding succeeds will value validation be attempted

Let’s assume some success ratio’s: (made-up, for simplicity)

For a random input, the odds that packet re-assembly succeeds is 30%
For a random input, the odds that Base64 decoding succeeds is 10%
For a random input, the odds that ASN1 decoding succeeds is 20%

So for a random input, the odds that all three layers can successfully parse the message that the previous layer produced, is 20% of 10% of 30% = 0.6%.
So out of 1000 random inputs, only 6 (on average) pass through all three layers.

By reasoning about the code, the following can be stated;

- reassemble_packets() takes a set, of variable length, of values in the range 0 - 255
- reassemble_packets() *produces* a set, of variable length, of values in the range 0 - 255
- base64Decode() takes a set, of variable length, of values in the range 0 – 255
- base64Decode() *produces* a set, of variable length, of values in the range 0 – 255
- asn1Decode() takes a set, of variable length, of values in the range 0 - 255
- asn1Decode() *produces* a set, of variable length, of values in the range 0 – 255

During an actual audit, the analyst must always reason about if and how the target code is eligible for customization. But let’s say that in this case, for the sake of demonstration, the previous statements are true, then it follows that if the following, altered version produces a crash, it must also be possible to construct an input that crashes the original version:

(pseudocode-ish for simplicity)

```
bool decode(std::vector<packet_t>& packets)
{
    std::vector<uint8_t>* inputMessage = reassemble_packets(packets);

    if ( vf_generate_bool() == true ) {
        return false;
    }

    {
        std::vector<uint8_t> generated = vf_generate(...);
        std::vector<uint8_t>* base64Decoded = base64Decode(generated);
    }

    if ( vf_generate_bool() == true ) {
        return false;
    }

    ASN1Data* asn1Decoded
```

```
    {
        std::vector<uint8_t> generated = vf_generate(...);
        asn1Decoded = asn1Decode(generated);
    }

    if ( vf_generate_bool() == true ) {
        return false;
    }

    if ( asn1Decoded->version != 0x1234 ) {
        return false;
    }

    if ( asn1Decoded->operation != PROTOCOL_HANDSHAKE ) {
        return false;
    }
}
```

This version is, at least for our purposes, sufficiently like the original, but there is now a 50% of 50% of 50% = 12.5% chance of reaching the version and protocol checks. The possibility to return with an error (`return false;`) is retained only to emulate the original, as it may cause relevant state changes in the code that precedes `decode()`. If you remove these, you are testing all three parsing layers 100% of the time. That's *a lot* more efficient than only 6 out of a 1000 times.

So you do this and you find a memory violation, in `base64Decode()`. Does this mean, then, that the original, unaltered application also contains a path towards this violation?
Yes, because we previously established that `reassemble_packets()` can produce any sequence of bytes. And we now know that a certain sequence of input bytes to `base64Decode()` results in a memory violation. Hence, to create an input that affects the original application, take the offending input and encode it such that it is a valid input to `reassemble_packets()`.

When I am doing audits, I use these tricks all the time. With native multi-generator support this has become a lot easier.

Generators

Generators create data for the target to work with. Other fuzzing engines typically only have a single generator. For instance, the libFuzzer entry point function, which has the following signature:

```
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)
```

gives the target a single block of data (defined by the tuple <data, size>) per iteration.

The fuzzer programmer can write code to extract these variables in an ad-hoc fashion from the input data:

```
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)
{
    size_t val;
    char* string = NULL;
    unsigned char* blob;

    if ( size < sizeof(val) ) {
        return 0;
    }

    memcpy(&val, data, sizeof(val));
    data += sizeof(val);
    size -= sizeof(val);

    {
        uint8_t string_size;
        if ( size < sizeof(string_size) ) {
            return 0;
        }
        memcpy(&string_size, data, sizeof(string_size));
        data += sizeof(string_size);
        size -= sizeof(string_size);

        if ( string_size > size ) {
            return 0;
        }

        string = malloc(string_size + 1);
        memcpy(string, data, string_size);
        string[string_size] = 0;
    }

    {
        uint8_t blob_size;
        if ( size < sizeof(blob_size) ) {
```

```
            free(string);
            return 0;
        }
        memcpy(&blob_size, data, sizeof(blob_size));
        data += sizeof(blob_size);
        size -= sizeof(blob_size);

        if ( blob_size > size ) {
            free(string);
            return 0;
        }

        blob = malloc(blob_size);
        memcpy(blob, data, blob_size);
    }

    {
        /* Perform operations on 'val', 'string' and 'blob' */
    }

    free(string);
    free(blob);

    return 0;
}
```

This is rather convoluted. While it can be simplified by utilizing a general-purpose extraction library, this general approach will tend to retain the following sub-optimal properties:

Extraction failures. The input data isn't suited to be decoded into various data types. See the various `return 0;` statements in my example, where extraction is aborted due to the input block being too small. Deserialization is started and stopped without serving any purpose and costly cycles are wasted. VrankenFuzz Generators cannot fail – they will always serve the requested data.

Execution speed. Execution speed is of utmost importance. Valuable cycles are lost to breaking down the input into several variables.

Input exhaustion. Sometimes you need some specific data block extracted from the input to be larger. Yes, you can simply make the fuzzer generate larger inputs. Depending on your deserialization logic, this might also make other extracted data buffers larger, which may not be what you want. What's more, the bigger an input is, the less efficient the fuzzer becomes, because the probability that a mutation changes *just* those bytes that lead to new code coverage, decrease with the increase of the input size. With Generators you have fine-grained control over the minimum and maximum size of the data it generates.

Inflexible serialization. The example above will write inputs to disk that are serializations of three variables. What if you add code to extract a fourth variable? The inputs may not deserialize into the same values anymore.

Cross-variable pollution. The variable contents will “bleed” into each other at the level of the mutator; `string` and `blob` may have very different semantics within the domain of the target, while they are extracted from a single long block, that is treated as such by the fuzzer. So, at one time an input may be encapsulating some meaningful values for `string` and `blob`, but a subsequent mutation will mingle the values, whereas ideally, you want a mutated `string` to be similar to the previous `string`, and a previous `blob` be similar to the previous `blob`. In short, there is no strict separation between the two, and this makes the mutation/coverage correlation sub-optimal.

Useless extractions. You might be extracting variables that you do not need during an iteration (depending on the code path, which you do not know in advance). With Generators you can generate data exactly there where you need it.

Generators do not have to be invoked at the logical beginning of the target. Data can be requested from VrankenFuzz anywhere in the target. This property gives rise to several interesting possibilities, as I will show in the “Examples” chapter.

You don’t need to free generated data. VrankenFuzz allocates the data and frees it after the iteration.

Generators also have a horizontal dimension. If a Generator is called for the first time during an iteration, it stores the generated data in “chunk 1” internally. If the same Generator ID is called a second time during iteration, the data is internally stored in “chunk 2”.

Seeding

Each Generator can be seeded (populated with a set of inputs) separately. The “On trickle-down logic” paragraph showed that it is sometimes useful to break down a single data stream (like a network socket) into several streams (Generators). By performing this seperation based on semantics, you can seed a Generator corpus exclusively with suited inputs.

A real example is SSL/TLS software, whose parsing of input mostly involves unpacking the protocol-specific encoding, but also requires interpreting DER-encoded X509 certificates. The call to the code that parses the DER-coded data (eg. `d2i_X509` in OpenSSL) can be replaced with a Generator, and this Generator can be seeded with one of several publicly available X509 fuzzing corpora[56]. This can be a shortcut to reaching a lot of code paths. Further, a Generator being an independent container, X509 data will not “bleed” into TLS protocol data structures,

5 https://github.com/openssl/openssl/tree/master/fuzz/corpora/x509
6 https://github.com/mozilla/nss-fuzzing-corpus/tree/master/certs

keying material, extension payloads etc and vice versa when the inputs are being mutated, as is the case when only a single data stream is used.

Built-in Generator functions

The following built-in generators are currently offered:

```
void vf_generate(generator_id gid, uint8_t** data, size_t* size);
void vf_generate_max(generator_id gid, uint8_t** data, size_t* size,
size_t max);
void vf_generate_min(generator_id gid, uint8_t** data, size_t* size,
size_t min);
void vf_generate_min_max(generator_id gid, uint8_t** data, size_t*
size, size_t min, size_t max);

size_t vf_generate_direct_max(generator_id gid, uint8_t* data, size_t
max);
size_t vf_generate_direct_min_max(generator_id gid, uint8_t* data,
size_t min, size_t max);

bool vf_generate_bool(generator_id gid);

uint8_t vf_generate_8(generator_id gid);
uint16_t vf_generate_16(generator_id gid);
uint32_t vf_generate_32(generator_id gid);
uint64_t vf_generate_64(generator_id gid);

char* vf_generate_string(generator_id gid);
char* vf_generate_string_max(generator_id gid, size_t max);
char* vf_generate_string_min(generator_id gid, size_t max);
char* vf_generate_string_min_max(generator_id gid, size_t min, size_t
max);

size_t vf_generate_val(generator_id gid, size_t min, size_t max);
```

In summary

- Generators generate data for the target to work with.
- There can be any amount of active Generators within a target.
- Each Generator can provide an infinite stream of data.
- Each consecutive call to a specific Generator during an iteration is internally treated as a ‘chunk’ within the Generator, and serialized as such in the corpus on disk.
- Several predefined generator types are available; those returning `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `bool`, `uint64_t` with min. and max. limits, and buffers of

arbitrary size.

Sensors

Built-in sensors

Core sensors

VrankenFuzz has the following Core Sensors:

- **PCsensor** - receives program location data. PC stands for program counter.
- **StackSensor** - receives stack pointer data.
- **AllocationSensor** - receives data on allocations and de-allocations ([*malloc()*, *calloc()*, *realloc()*] and *free()*).
- **ValueProfileSensor** – receives data on value comparisons and from other miscellaneous instrumented code.

Built-in Sensor/Processor combinations

VrankenFuzz offers the following pre-defined shorthands for combinations of Core Sensors and Processors:

Code coverage. A PCSensor records all locations accessed in the code (reported by `__sanitizer_cov_trace_pc_guard`) and propagates them to a Unique Processor. The Unique Processor reports the total amount of unique locations seen.

Code intensity. A PCSensor records the *amount* of locations accessed in the code (reported by `__sanitizer_cov_trace_pc_guard`) and propagates them to a Highest Processor. The Highest Processor reports the total of locations seen.
Code intensity sensoring may have *some* use in affirming the constant-timeness of functions (hypothesized).

Stack depth maximum. A StackSensor records all stack pointer values at every instrumented block (computed within each `__sanitizer_cov_trace_pc_guard` invocation) and propagates them to a Highest Processor. The Highest Processor reports the highest stack pointer value seen.

Stack unique. A StackSensor records all stack pointer values at every instrumented block (computed within each `__sanitizer_cov_trace_pc_guard` invocation) and propagates them to a Unique Processor. The Unique Processor reports the total amount of unique stack pointer values seen.

Allocation maximum. An AllocationSensor records all allocations and frees. It keeps track of

the current number of bytes allocated by subtracting the frees from the allocations and propagates this value to a Highest Processor. The Highest Processor reports the highest collective memory consumption seen.

Value Profile. Similar to how `-use_value_profile` is implemented in libFuzzer.

Sensor scoping

Sensor scoping is enabling a sensor only during the execution of a certain portion of code.

Sometimes you are interested in fuzzing only a specific part of the target, let's call it *X()*, but getting to *X()* requires executing code that logically precedes it. Getting the fuzzer to focus specifically on `X()` can be achieved through Sensor scoping: enable a Sensor before entering *X()*, and disable it right after.

This construct leads to some interesting things:

- Only inputs that lead to *X()* will be added to the corpus
- The fuzzing gets faster because all instrumentation code outside *X()* is essentially a no-op; VrankenFuzz ignores instrumentation if no Sensor for that instrumentation is not enabled.
- Because there are only inputs that reach *X()* in the corpus, each new mutation has a high likelihood of reaching *X()* too.
- If *A()* calls B(), then *B()* won't trigger the Sensor, but if *X()* calls *B()*, *B()* will trigger the Sensor, simply because in *A()* (which calls *B()*) the Sensor is disabled, but in *X()* (which calls *B()*) it is enabled.

Sensors can be enabled and disabled with:

void vf_sensor_enable(ID);
void vf_sensor_disable(ID);

Custom sensors

Everything that can be quantified, eg. be represented as a value, can be turned into a sensor.

void vf_sensor_callback(ID, value, processorType);

If a sensor with this ID does not exist yet , it is internally constructed and connected to a Processor of *processorType*.

Binding

Two sensors can be bound together in order to create the Cartesian product[7] of the two output sets.

7 https://en.wikipedia.org/wiki/Cartesian_product

If during an iteration sensor 1 outputs the set `[1, 4, 9]` and sensor 2 outputs the set `[100, 150, 200]`, then the Cartesian product of these sets, which the following set of tuples:

```
<1, 100>
<1, 150>
<1, 200>
<4, 100>
<4, 150>
<4, 200>
<9, 100>
<9, 150>
<9, 200>
```

Is forwarded to a Processor.

A bound sensor is itself a sensor that can be bound to another sensor (internally it is derived from the *BaseSensor* class, from which all sensors are derived), so constructing a Sensor chain of arbitrary length is possible.

Internally, the Cartesian product of two sensor outputs is computed by taking the set of unique values outputs of each of the two Sensors, and representing each combination of the two sets into a single value by computing `(((uint64_t)val1 << 32) | (uint64_t)(val2 & 0xFFFFFFFF))`. From this it is apparent that for values exceeding upper bound of an `uint32_t`, an imprecise Cartesian product may be produced. In most practical cases, this won't happen, and even if it happens, it isn't necessarily a critical flaw.
I've experimented with variable-sized Sensor and Processor inputs and outputs, but the speed penalty incurred is too severe to prefer it to using a native data type in almost all use cases. If you should require a precise Cartesian product, it can be achieved computing the Cartesian product yourself and feeding the resulting value to a Custom Sensor.

An interesting use of bound sensors is the following.

```
Msg* parseMsg(const std::vector<uint8_t> data)
{
    Msg* msg = new Msg();

    /* Perform operations to parse 'data' */

    /* upon success: */
    return msg;

    /* upon failure: */
    throw std::runtime_error("error");
```

```
}

void process(
        const std::vector<uint8_t> data1,
        const std::vector<uint8_t> data2,
        const std::vector<uint8_t> data3)
{
    Msg* msg1 = parseMsg(data1);
    Msg* msg2 = parseMsg(data2);
    Msg* msg3 = parseMsg(data2);
}
```

Before reaching the end of `process`(), the three input buffers must be parsed *successfully* (they will otherwise throw an error).
Once `process()` has run a number of times, each instrumented block inside *parseMsg()* has reported activity to the fuzzer, eg. the `parseMsg()` code coverage is now 100%. But this does not necessarily mean that all three calls from `process()` to `parseMsg()` will succeed. This is a problematic situation for a traditional fuzzer because its sole source of guidance, the code coverage, is already exhausted!

With VrankenFuzz you can do this:

```
Msg* parseMsg(const std::vector<uint8_t> data)
{
    Msg* msg = new Msg();

    /* Perform operations to parse 'data' */

    /* upon success: */
    return msg;

    /* upon failure: */
    throw std::runtime_error("error");
}

#define PC_SENSOR_ID 1
#define STEP_SENSOR_ID 2
#define PC_STEP_SENSOR_ID 3

void process(
        const std::vector<uint8_t> data1,
        const std::vector<uint8_t> data2,
        const std::vector<uint8_t> data3)
{
    /* Create a Sensor for registering code coverage */
```

```
    sensor_from_core_builtin(PC_SENSOR_ID,
VRANKENFUZZ_SENSOR_BUILTIN_CORE_CODE_COVERAGE,
VRANKENFUZZ_PROCESSOR_TYPE_NOOP);

    /* Bind the code coverage sensor to step sensor (see below) */
    sensor_combine(PC_STEP_SENSOR_ID, PC_SENSOR_ID, STEP_SENSOR_ID,
VRANKENFUZZ_PROCESSOR_TYPE_UNIQUE);

    /* First step in the three-step process of parsing all mesasges,
     * so set STEP_SENSOR_ID to 1 */
    sensor_callback(STEP_SENSOR_ID, 1,
VRANKENFUZZ_PROCESSOR_TYPE_NOOP);
    Msg* msg1 = parseMsg(data1);

    /* Second step in the three-step process of parsing all mesasges,
     * so set STEP_SENSOR_ID to 2 */
    sensor_callback(STEP_SENSOR_ID, 2,
VRANKENFUZZ_PROCESSOR_TYPE_NOOP);
    Msg* msg2 = parseMsg(data2);

    /* Third step in the three-step process of parsing all mesasges,
     * so set STEP_SENSOR_ID to 3 */
    sensor_callback(STEP_SENSOR_ID, 3,
VRANKENFUZZ_PROCESSOR_TYPE_NOOP);
    Msg* msg3 = parseMsg(data2);
}
```

Here we create a bound sensor that is the amalgam of 1) code coverage and 2) the step (1, 2, 3) in the process. Each new <code location, step> is an indicator to the fuzzer where along the way it is, and we are thus *guiding* it towards the end of `process()`.

You can bind sensors together with:

```
void vf_sensor_combine(newID, ID1, ID2, processorType);
```

Sensor *ID1* and Sensor *ID2* do not already have to exist. Sensor *newID* will be automatically created once both Sensor *ID1* and Sensor *ID2* have reported at least one value.

Naming

A Sensor can be given a descriptive name using

void sensor_set_name(ID, const char name);*

If a Sensor name is set, its name instead of its ID will be displayed by the logger.

Sensor *ID* does not already have to exist. As soon as a sensor with id *ID* reports a value, its name will be registered automatically.

This function is optimized in that it performs the name registration only once no matter how many times it is called, so you don't have to worry about unnecessary overhead even if the target calls it multiple times.

In summary

- Sensors consume values.
- A Sensor is always connected to a Processor.
- Hence, a Sensor has an *input* and an *output*.
- There can be any amount of active Sensors within a target.
- There are built-in Sensors that consume variables that are related to the lowest level of the target's execution, such as code location, stack pointer value and memory allocations.
- Sensors can be named.

Processors

A Processor is some tiny logic that consumes a sensor value and transforms it into a different value. VrankenFuzz offers four types of Processors.

The **Highest Processor** consumes a sensor value and outputs it only if the value is higher than any value previously seen (and always outputs the input value if there is no previous value).

Eg.:

```
if ( input > highest ) { highest = input; propagate(highest); }
```

The **Lowest Processor** consumes a sensor value and outputs it only if the value is lower than any value previously seen (and always outputs the input value if there is no previous value).

Eg.:

```
if ( input < lowest ) { lowest = input; propagate(lowest); }
```

The **Unique Processor** consumes a sensor value and outputs the number of previously seen values if the value wasn't previously seen.

Eg.:

```
if ( previous_values.count(input) == 0 )
{ previous_values.insert(input); propagate(previous_values.size()); }
```

Where `previous_values` is an `std::unordered_set`.

The **No-Operation Processor** consumes a sensor value and outputs the same value. This is a construct that is required in order to construct bound Sensors, eg. two Sensors connected to a Processor.

Eg.:

```
propagate(input);
```

Note: all code examples above are pseudo-code and not actual excerpts from the VrankenFuzz source code.

Discarding inputs

Inputs can be discarded (not added to the corpus) by some custom condition C.

If C is true for an input I, I is discarded.

Conversely, the analyst may choose to discard all inputs by default, and allow addition to the corpus if C is true for I.

Why is this useful?

You could be interested in fuzzing a particular function F in a target. For the reasons touched upon in the “On trickle-down logic” paragraph, only a small subset of all inputs that are fired at the target will naturally reach F.
By discarding all inputs by default, and explicitly allowing inclusion (“undiscarding”) in F, newly mutated inputs will be ever more likely to reach F.

The standard library and Generators

Standalone software, like servers, is cumbersome to fuzz, because it tends to contain a lot of code that performs IO, such as communicating with the network and reading files from disk.

Transforming standalone software into an effective fuzzer is time-consuming because it requires making placeholders for its IO interfaces.
This is necessary because you don’t want actual interaction with the network or disk; this breaks the determinism of the fuzzer and pollutes the network and storage.
Ideally, you want the application to source its data from fuzzer Generators instead of from a live network or disk in a fuzzing environment.

For several of my previous efforts to fuzz standalone software, like my OpenVPN and SoftEther VPN audits, I replaced each and every call to an IO function (eg. `recv()`) with a custom implementation that acted according to the POSIX specification, but sourced data from the fuzzer input data (eg. supplied by `LLVMFuzzerTestOneInput()`). This is essentially what you want, but this is a time consuming process.

This is why I am currently implementing a replacement for the POSIX network functions, along these lines:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags)
{
    uint8_t* data;
    size_t size;

    if ( vf_generate_bool(GENERATOR_ID_RECV_BOOL) == false ) {
        /* TODO set errno */
        return -1;
    }

    if ( len == 0 ) {
        return 0;
    }

    vf_generate_max(GENERATOR_ID_RECV_DATA, &data, &size, len);

    memcpy(buf, data, size);

    return (ssize_t)(size);
}
```

The canonical libc functions are overriden with the LD_PRELOAD trick[8].
I have been succesfully fuzzing a HTTP client using this approach.

Logging

VrankenFuzz currently offers two loggers: a console logger, which prints information about new Sensor data to the terminal (somewhat similar to libFuzzer's output), and a JSON logger, which stores all information about new inputs and sensor data into separate JSON files.

Also included in the project is a tiny web application built with Flask[9] and Chart.js[10] that shows a live visualization as a graph of the various Sensors. It sources its data from the directory

8 https://stackoverflow.com/questions/426230/what-is-the-ld-preload-trick
9 http://flask.pocoo.org/
10 https://www.chartjs.org/

containing the JSON files with each AJAX call.

About the project

Availability

I'd like to release the source code publicly as open source for a compensation.

I will certainly not be selling it commercially in a SaaS or license-based scheme.

I'm also open to discussing the use of the project binaries free of charge by interesting academic, non-commerical research efforts.

Please contact me at guido@guidovranken.com with any reasonable offer or proposal.

Current project state

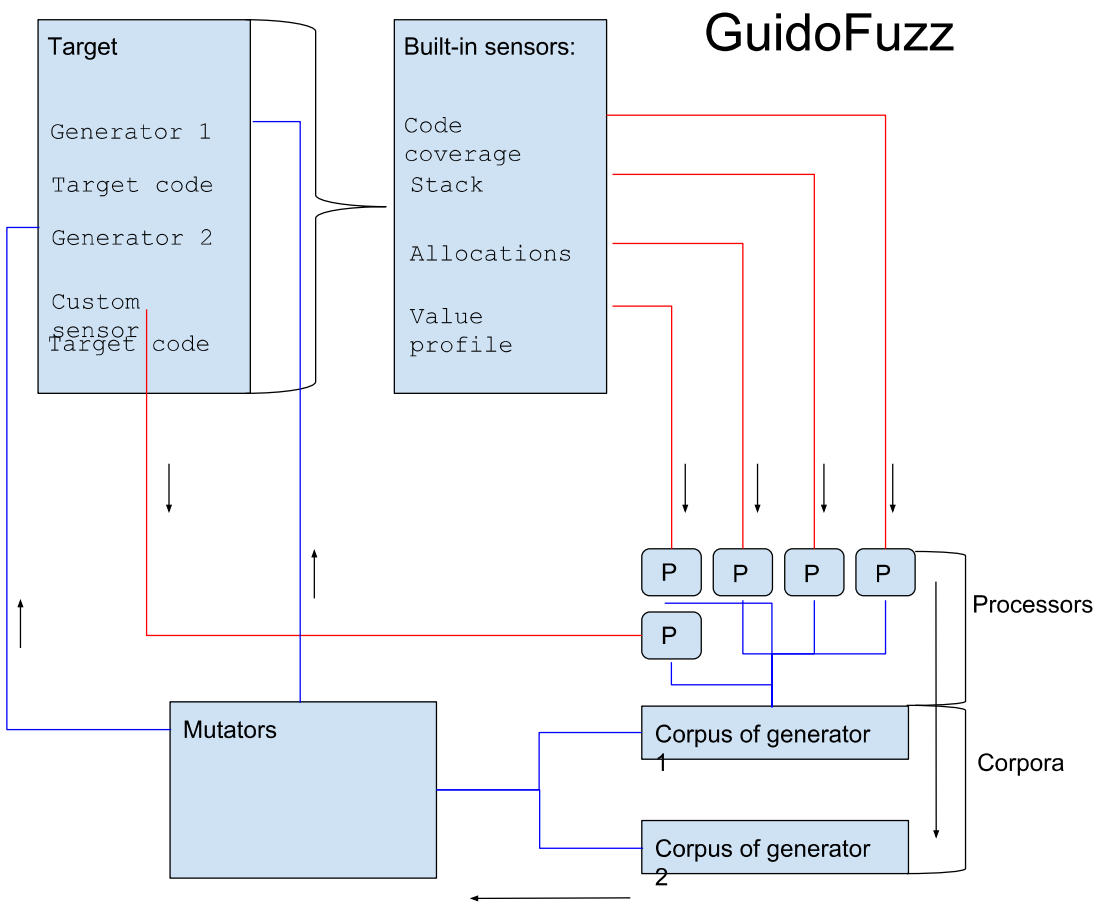
The program is functional and can be used to find bugs in real software, but there is still work left to do on:

- Dictionaries [partially implemented]
- Command-line parameter processing [partially implemented]
- Speed optimizations
- Signal handling [not implemented]
- Multithread support [not implemented]
- Libc implementation [partially implemented]
- Formal documentation [not written]
- Bindings for Python, Go, JavaScript..

Source code

The VrankenFuzz source code consists of several thousands of lines of C++ 11. It is mostly or completely const-correct. Care has been taken to make the code as legible as possible. A consistent variable and method naming scheme is used throughout. It compiles without any warnings (`clang++ -Wall`).

Diagram


GuidoFuzz
Target
Generator 1
Target code
Generator 2
Custom sensor
Target code
Built-in sensors:
Code coverage
Stack
Allocations
Value profile
P
P
P
P
P
Processors
Mutators
Corpus of generator 1
Corpus of generator 2
Corpora