



# **Vulnerability Extrapolation**

## **‘Give me more bugs like that’**

## **Blackhat Briefings 2011**

Fabian ‘fabs’ Yamaguchi  
Reecurity Labs GmbH, Germany

# **Agenda**



- Patterns you find when auditing code
- Exploiting these patterns:  
**Vulnerability Extrapolation**
- Using machine learning to get there
- A method to assist in manual code audits based on this idea
- The method in practice
- A detailed showcase

# Exploring a new code base



- Like an area of mathematics you don't yet know.
- It's not completely different from the mathematics you already know.
- But there are secrets specific to this area:
  - Vocabulary
  - Reoccurring patterns in argumentation
  - Weird tricks used in proofs
- Understanding the specifics of the area makes it a lot easier to reason about it.

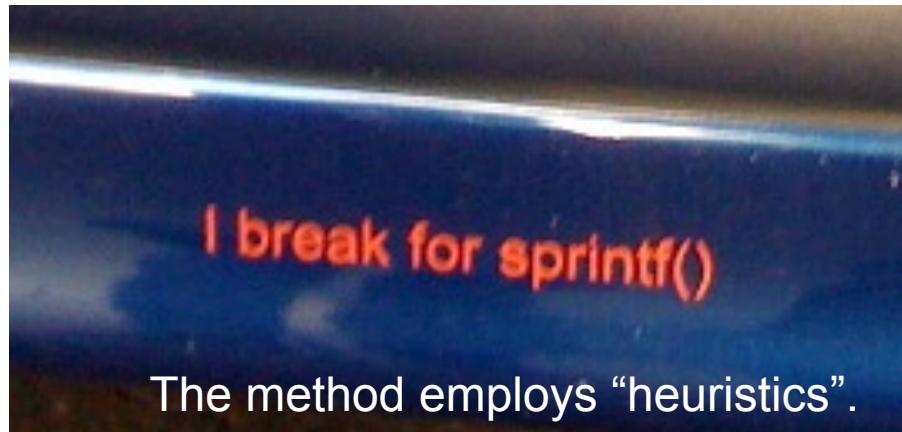
# It's also a lot like DOOM

- Dropped into some code-base, no idea where you are
- Only a handgun to begin with
- Secrets of this particular .WAD matter!
  - Where do I get weapons around here?
  - Effective ways of killing these monsters?



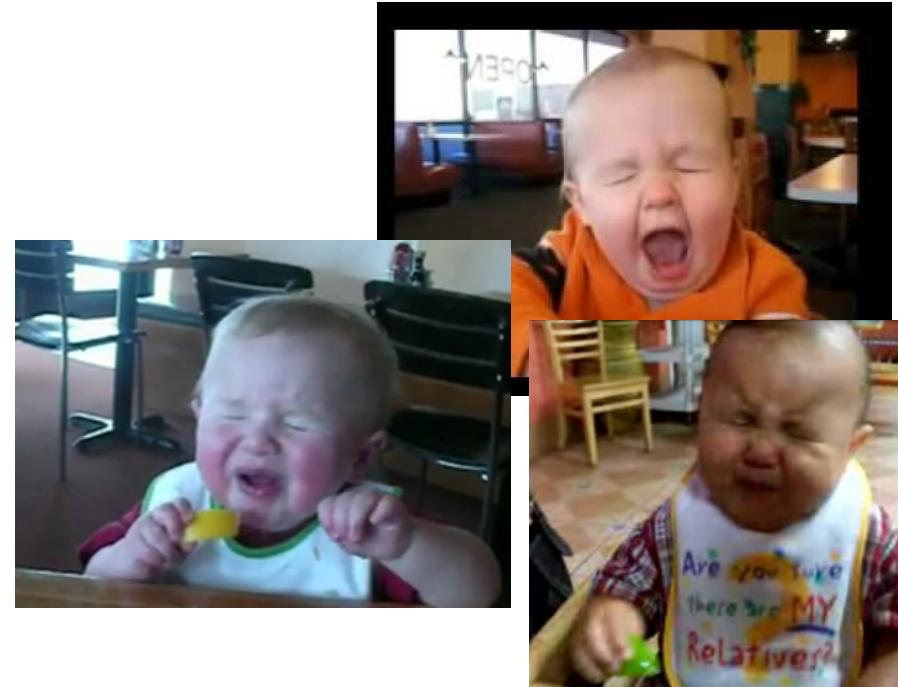
# “It’s all the same”-attitude

- “Let’s not be too romantic about what we do:
- **Things tend to break in the same way over and over again.”**
- Think of how you automatically stop scrolling whenever you see `sprintf`.
- Or those lists of “dangerous functions” used by old-school tools like RATS, ITS4 and flawfinder.



# Or think of the success of fuzzers

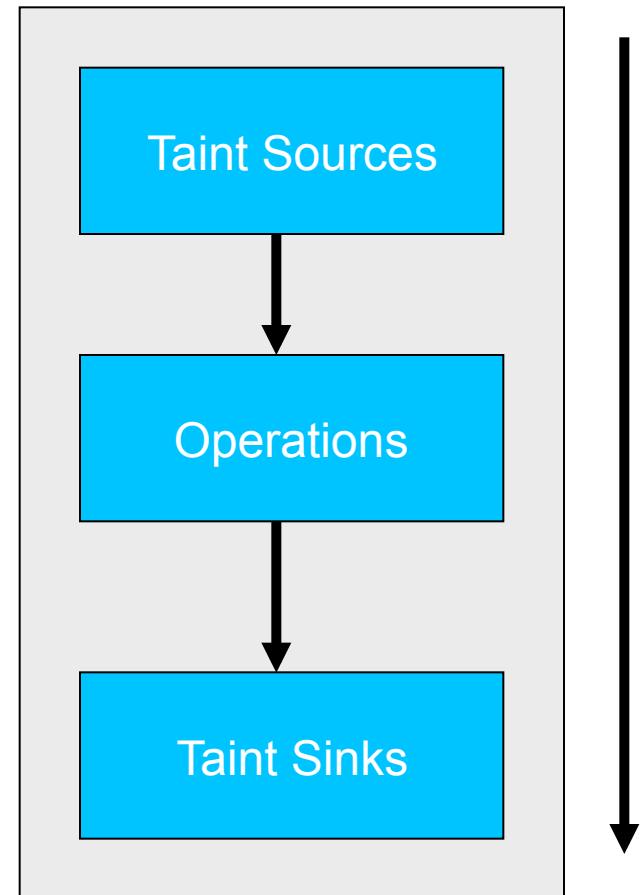
- Fuzzers use patterns in input that will get many targets very upset. Often very effective.
- **Why? Because things tend to break in the same way over and over again.**



“Try giving it a lemon.”

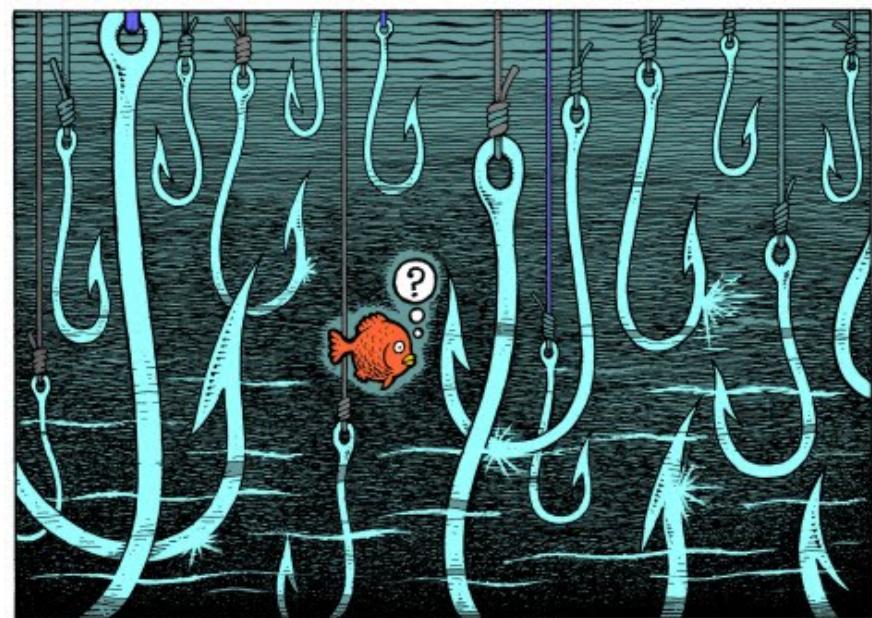
# ... or Taint Propagation

- Example: Monitor flow of integer from read to malloc, detect unsafe operations and monitor if integer is ever checked.
- API usage patterns, we've seen blow up again and again.



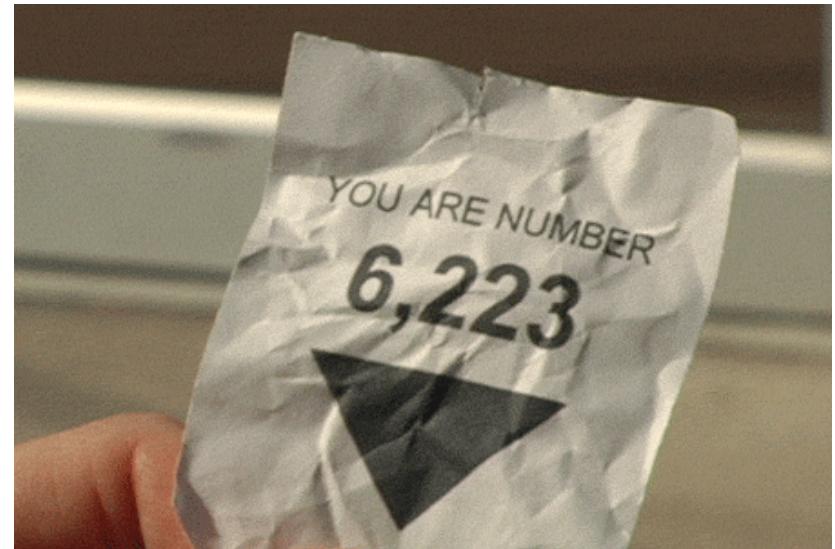
# Overfished

- These methods are too generic.
- They find what people screw up in *most* applications.
- And that's also what *most* people have looked at in the application.



# Specifics matter!

- For software that people actually care about, you can be sure all the low-hanging fruit is gone.
- “grep’ing for memcpy”: not much of a strategy anymore.
- Why? Because that’s what the last 100 people did before you came along.



It is no longer optional to learn about the specifics of the code base!

# Which is why manual audits are so successful

- Because auditors find the weak programming patterns used in *this* application.
- Find the interfaces that are causing trouble in *this* application!
- Find the secrets in this .WAD!



(Seriously, there was a ghostbusters.WAD)

# An example Poppler (CVE-2009-3607)

```
static cairo_surface_t *
create_surface_from_thumbnail_data (guchar *data,
                                    gint   width,
                                    gint   height,
                                    gint   rowstride)
{
    guchar *cairo_pixels;
    cairo_surface_t *surface;
    static cairo_user_data_key_t key;
    int j;

    cairo_pixels = (guchar *)g_malloc (4 * width * height);
    surface = cairo_image_surface_create_for_data ((unsigned char *)
cairo_pixels,
                                                CAIRO_FORMAT_RGB24,
                                                width, height, 4 * width);
    cairo_surface_set_user_data (surface, &key, cairo_pixels,
(cairo_destroy_func_t)g_free);

    [...]
    return surface;
}
```



# Do you see the bugs?



- The integer overflow is obvious.
- The missing check, not so much. You need to know the API to see this!

```
/* This function always returns a valid pointer, but it will return a
 * pointer to a "nil" surface in the case of an error such as out of
 * memory or an invalid stride value. In case of invalid stride value
 * the error status of the returned surface will be
 * %CAIRO_STATUS_INVALID_STRIDE. You can use
 * cairo_surface_status() to check for this.
 */
```

# Evince Bug, silently fixed.

```
static cairo_surface_t *
djvu_document_render (EvDocument      *document,
                      EvRenderContext *rc)
{
    [...]
#ifndef HAVE_CAIRO_FORMAT_STRIDE_FOR_WIDTH
    rowstride = cairo_format_stride_for_width (CAIRO_FORMAT_RGB24,
page_width);
#else
    rowstride = page_width * 4;
#endif
    pixels = (gchar *) g_malloc (page_height * rowstride);
    surface = cairo_image_surface_create_for_data ((guchar *)pixels,
                                                CAIRO_FORMAT_RGB24,
                                                page_width,
                                                page_height,
                                                rowstride);
    cairo_surface_set_user_data (surface, &key,
                                pixels, (cairo_destroy_func_t)g_free);
    [...]
    return surface;
}
```



# A simple case



- This case is simple, because the API-symbols, which lead to these two bugs were exactly the same.
- But does that have to be the case?

# Another Example: libTIFF

## CVE-2006-3459 | CVE-2010-2067

```
static int
TIFFFetchShortPair(TIFF* tif, TIFFDirEntry* dir)
{

    switch (dir->tdir_type) {
        case TIFF_BYTE:
        case TIFF_SBYTE:
            {
                uint8 v[4];
                return TIFFFetchByteArray(tif, dir, v)
                    && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
            }
        case TIFF_SHORT:
        case TIFF_SSHORT:
            {
                uint16 v[2];
                return TIFFFetchShortArray(tif, dir, v)
                    && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
            }
        default:
            return 0;
    }
}
```

# Another Example: libTIFF

## CVE-2006-3459 | CVE-2010-2067

```
static int
TIFFFetchSubjectDistance(TIFF* tif, TIFFDirEntry* dir)

static int
TIFFFetchShortPair(TIFF* tif, TIFFDirEntry* dir)
{
    uint32 v[2];
    float v;
    int ok = 0;

    switch (dir->tdir_type) {
        case TIFF_BYTE:
        case TIFF_SBYTE:
            if (TIFFFetchData(tif, dir, (char *)v) && cvtRational(tif, dir, v[0], v[1], &v)) {
                return TIFFFetchByteArray(tif, dir, v)
                    && TIFFSetField(tif, dir->ttag, 0xFFFFFFFF); means that we have infinite
            }
            * distance. Indicate that with a negative floating point
        case TIFF_SHORT:
            * SubjectDistance value.
        case TIFF_SSHORT:
            /*
            ok = TIFFSetField(tif, dir->ttag,
                (v[0] != 0xFFFFFFFF) ? v : -v);
            && TIFFSetField(tif, dir->ttag, v[0], v[1]);
            */
        default:
            return ok;
    }
}
```

# LibTIFF: Bug Analysis

- `TIFFFetchShortArray` is actually a wrapper around `TIFFFetchData`.
- The two are pretty much synonyms.
- These functions are part of an API local to libTIFF.
- Badly designed API: the amount of data to be copied into the buffer is passed in one of the fields of the dir-structure and not explicitly!
- Developers missed this in both cases and it's hard to blame them.



The times of “grep ‘memcpy’ ./\*.c” may be over. But that does not mean *patterns of API use that lead to vulnerabilities* no longer exist!

# Vulnerability Extrapolation

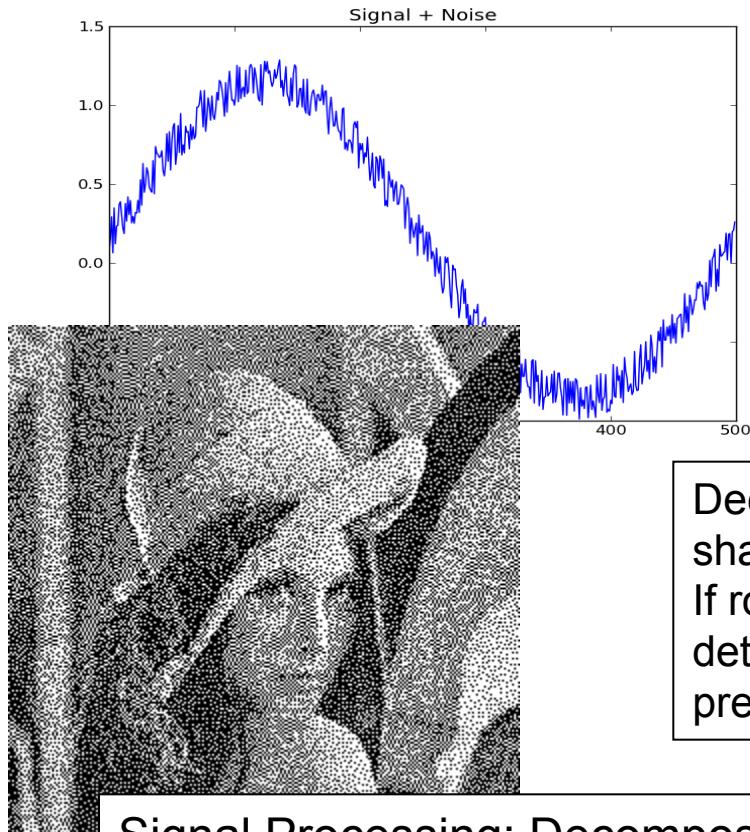
- Given a function known to be vulnerable, determine functions **similar** to this one in terms of application-specific API usage patterns.
  - Why? Because these are most likely to contain another incarnation of this bug.
  - Why? Because developers **tend to make the same mistakes over and over and over again.**
  - Especially if motivated by a bad API.
- Vulnerability Extrapolation exploits the information leak you get every time a vulnerability is disclosed!

# **What needs to be done**

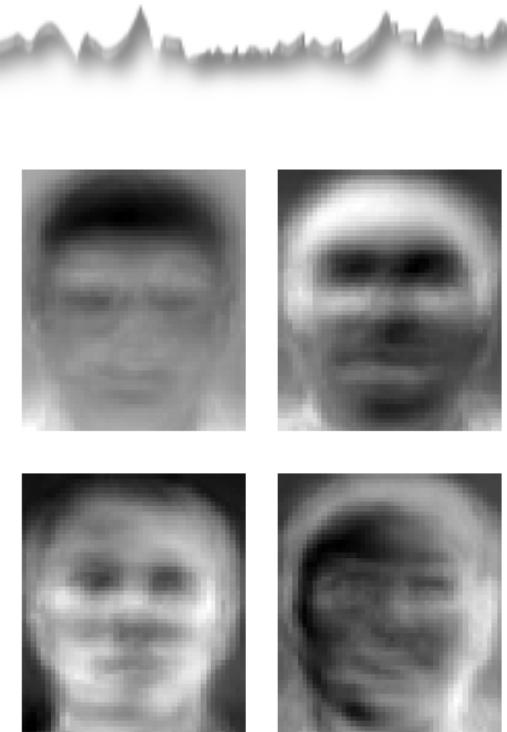


- We need to be able to determine how “similar” functions are in terms of these programming patterns.
- We need to find a way to extract these programming patterns from a code-base in the first place.
- How do we do that?

# Similarity – A decomposition



Decomposition into shape and rotation:  
If rotation is just a detail, these are pretty similar.



In Face-Recognition, faces are decomposed into weighted sums of **commonly found patterns** + a noise-term.

# Signal and Noise

- Checking if two things are similar always requires a decomposition into “The big picture” and “the details” or “signal” and “noise”.
- In general, if the big picture is the same and only the details differ, things are pretty similar.
- What’s signal and what’s noise depends on the problem you’re dealing with.

# Decomposing Code

```
static int
TIFFFetchSubjectDistance(TIFF* tif, TIFFDirEntry* dir)
{
    uint32 I[2];
    float v;
    int ok = 0;

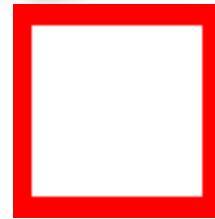
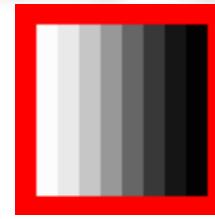
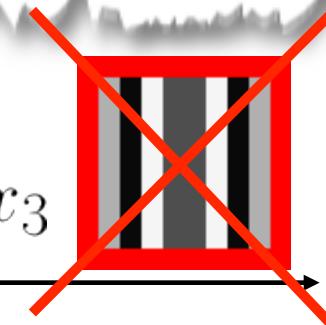
    if (TIFFFetchData(tif, dir, (char *)I)
        && cvtRational(tif, dir, I[0], I[1], &v)) {
        /*
         * XXX: Numerator 0xFFFFFFFF means that we have infinite
         * distance. Indicate that with a negative floating point
         * SubjectDistance value.
         */
        ok = TIFFSetField(tif, dir->tdir_tag,
                          (I[0] != 0xFFFFFFFF) ? v : -v);
    }
    return ok;
}
```

Big picture

Function = Dominant Pattern + Noise

- Once you know a code-base, you start to decompose automatically:
  - Dominant patterns of API use: Some FetchData-Function followed by TIFFSetField
  - Symbols occurring in this function but not necessarily in any of the other functions employing the pattern

# Think of it as 'zooming out'

 $\approx x_1$  $+x_2$  $+x_3$ 

Increasing level of detail/frequency

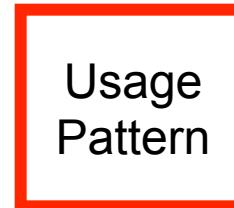
 $\approx x_1$  $+x_2$  $+x_3$ 

```
static int
TIFFFetchSubjectDistance(TIFF* tif, TIFFDirEntry* dir)
{
    uint32 [2];
    float v;
    int ok = 0;

    if (TIFFFetchData(tif, dir, (char *)) &&
        cvtRational(tif, dir, ([0], [1]), &v)) {
        /*
         * XXX: Numerator 0xFFFFFFFF means that we have infinite
         * distance. Indicate that with a negative floating point
         * SubjectDistance value.
         */
        ok = TIFFSetField(tif, dir->tdir_tag,
                          ([0] != 0xFFFFFFFF) ? v : -v);
    }

    return ok;
}
```

Decreasing dominance of pattern

 $\approx x_1$  $+x_2$  $+x_3$ 

Linear approximation of each function by the most dominant API usage patterns of the code-base it is contained in!

# Extracting dominant patterns

How do we identify the most dominant API usage patterns of a code-base?



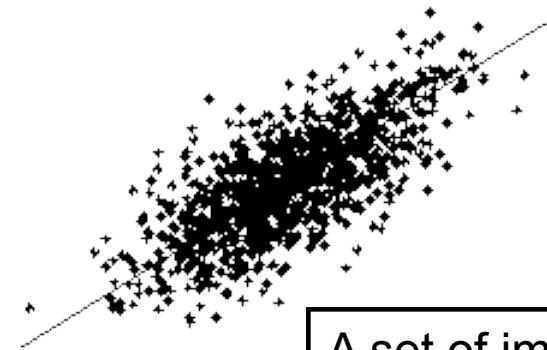
How do other fields identify dominant patterns in their data?

# Principal Component Analysis in Face Recognition

- Images have a natural vectorial representation.
- Each image can be interpreted as a `$numberOfPixels`-dimensional vector.
- Directions in this space correspond to dependencies among pixels.

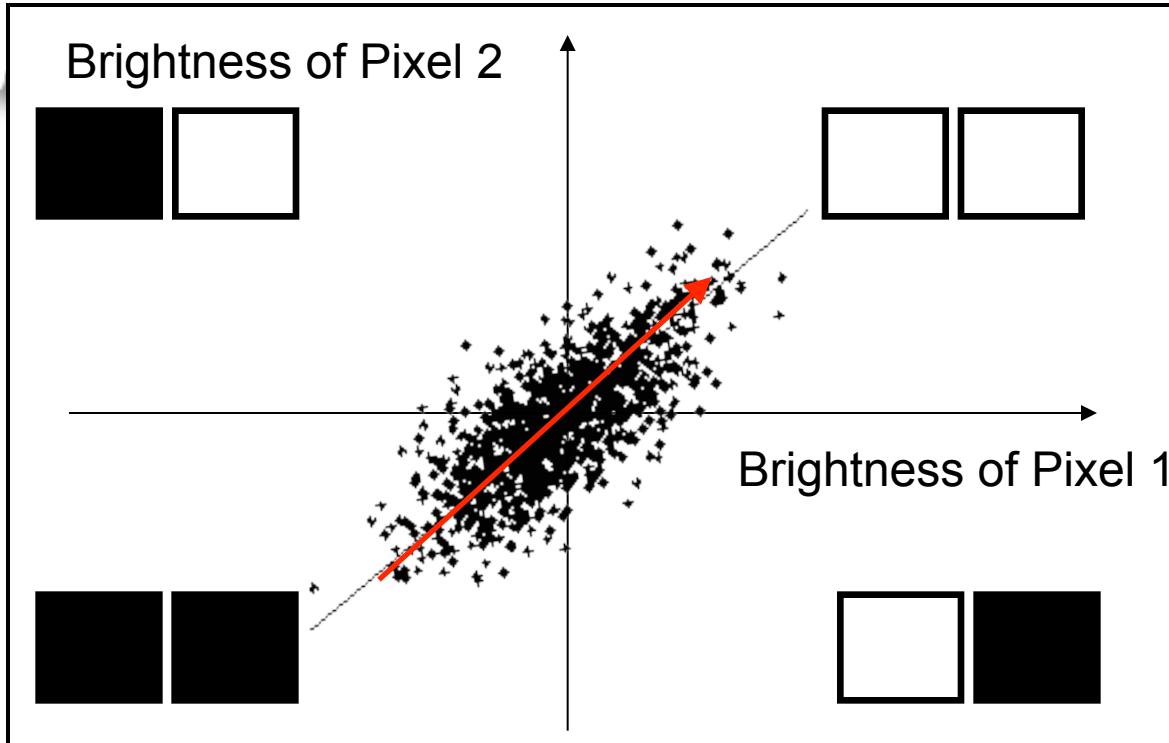
$$\begin{pmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.5 \end{pmatrix}$$

Brightness of first pixel  
... of second pixel  
... of last pixel



A set of images can be represented by a set of vectors.

# Images with two pixels



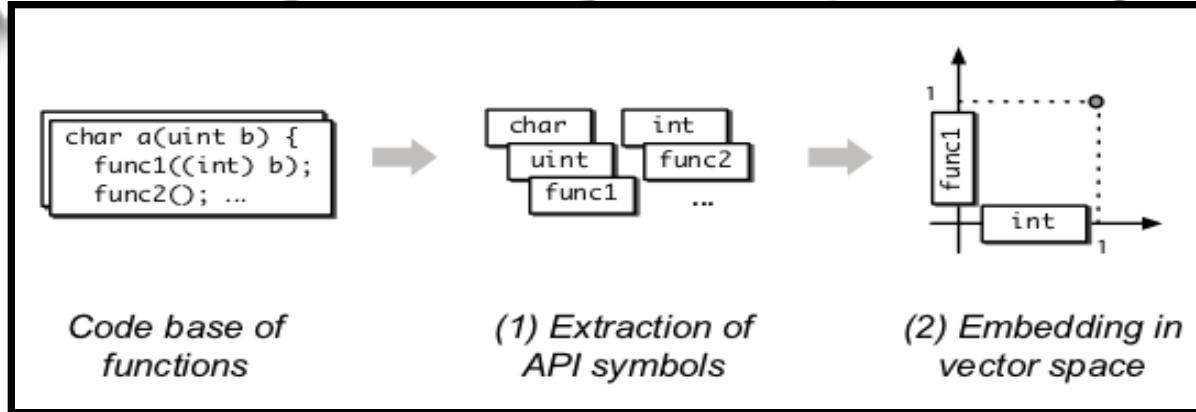
- The most dominant pattern: Either both pixels light up, or both don't.
- As opposed to, for example, either pixel 1 lights up and pixel 2 doesn't and vice versa.
- Geometrically, this corresponds to the direction where the data varies most.
- In other words, if you were to project onto the red vector, you'd best describe the data with a single dimension.

# We can make direct use of this!



- Directions of highest variance correspond to dominant patterns in the data.
- These correspond to the eigenvectors of the data covariance matrix.
- A singular value decomposition can be used to obtain these.
- Let's make use of this to determine dominant API usage patterns!

# Mapping code to the vector space



- Describe functions by the API-symbols they contain.
- API-symbols are extracted using a fuzzy parser.
- Each API-symbol is associated with a dimension.

```
func1(){  
    int *ptr = malloc(64);  
    fetchArray(pb, ptr);  
}
```

$$\xrightarrow{\hspace{1cm}} \begin{matrix} \textit{malloc} \\ \textit{printf} \\ \textit{int} \\ \vdots \\ \textit{fetchArray} \end{matrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

# Approximation of functions by most dominant API usage patterns!

PCA implemented by truncated Singular Value Decomposition.

$$\begin{matrix} & \textit{func}_1 & \textit{func}_2 & \dots & \textit{func}_n \\ \textit{strcpy} & 0 & 1 & \dots & 1 \\ \textit{foo} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_m & ,1 & 0 & \dots & 0 \end{matrix} \approx U \Sigma V^T$$

Directions of highest variance

Strength of pattern on diagonal

Representation of functions in terms of these dominant patterns

# A closer look at the decomposition

Data Matrix (Contains all function-vectors)

Each column of U  
is a dominant  
pattern.

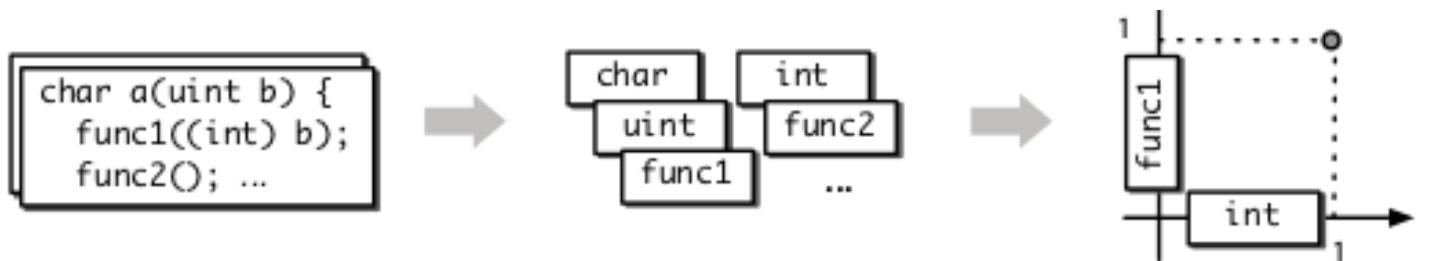
Strength of pattern

$$M \approx U \Sigma V^T =$$
$$\left( \begin{array}{c} \leftarrow u_1 \rightarrow \\ \leftarrow u_2 \rightarrow \\ \vdots \\ \leftarrow u_{|\mathcal{S}|} \rightarrow \end{array} \right) \left( \begin{array}{cccc} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_d \end{array} \right) \left( \begin{array}{c} \leftarrow v_1 \rightarrow \\ \leftarrow v_2 \rightarrow \\ \vdots \\ \leftarrow v_{|\mathcal{X}|} \rightarrow \end{array} \right)^T .$$

Each row is a representation  
of an API-symbol in terms of  
the most dominant patterns

Representation of functions in terms  
of the most dominant patterns

# In summary



*Code base of  
functions*

*(1) Extraction of  
API symbols*

*(2) Embedding in  
vector space*



*(3) Identification of  
API usage patterns*

*(4) Assisted vulnerability  
discovery*

# A toy problem to gain an intuition

## Group 1

```
void guiFunc1(GtkWidget *widget)
{
    int j;
    gui_make_window(widget);
    GtkWidget *button;
    button = gui_new_button();
    gui_show_window();
}
```

```
void guiFunc2(GtkWidget *widget)
{
    gui_make_window(widget);
    GtkWidget *myButton;
    button1 = gui_new_button();
    button2 = gui_new_button();
    button3 = gui_new_button();

    for(int i = 10; i != i; i++)
        do_gui_stuff();
}
```

# Group2

```
void netFunc1()
{
    int fd;
    int i = 0;
    struct sockaddr_in in;
    fd = socket(arguments);
    recv(fd, moreArguments);

    if(condition){
        i++;
        send(fd, i, arg);
    }
    send(fd, i, arg);
    close(fd);
}
```

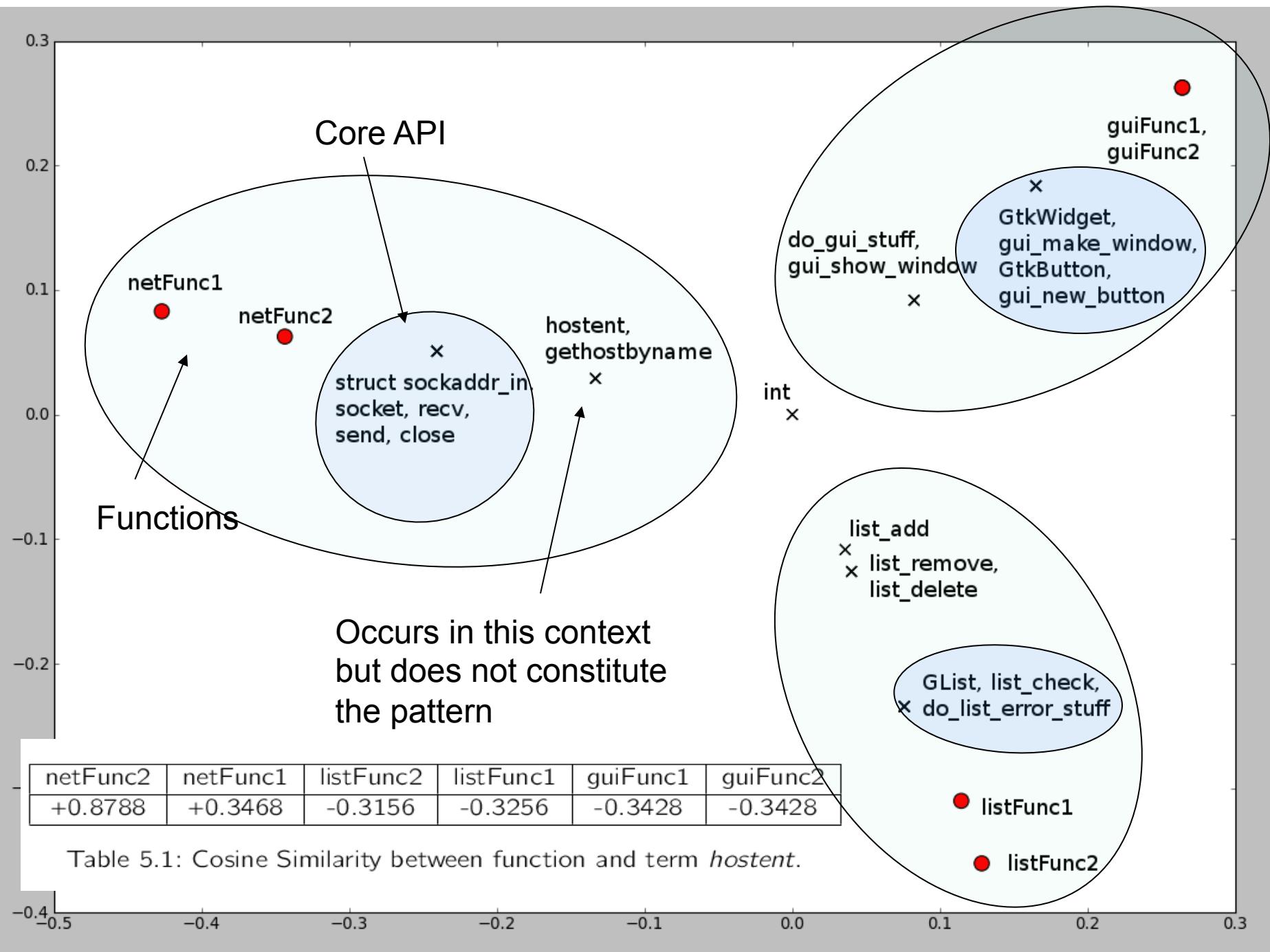
```
void netFunc2()
{
    int fd;
    struct sockaddr_in in;
    hostent host;
    fd = socket(arguments);
    recv(fd, moreArguments);
    gethostbyname(host)

    if(condition){
        int i = 0;
        i++;
        send(fd, i, arg);
    }
    close(fd);
}
```

# Group 3

```
void listFunc1(int elem)
{
    GList myList;
    if(!list_check(myList)){
        do_list_error_stuff();
        return;
    }
    list_add(myList, elem);
}
```

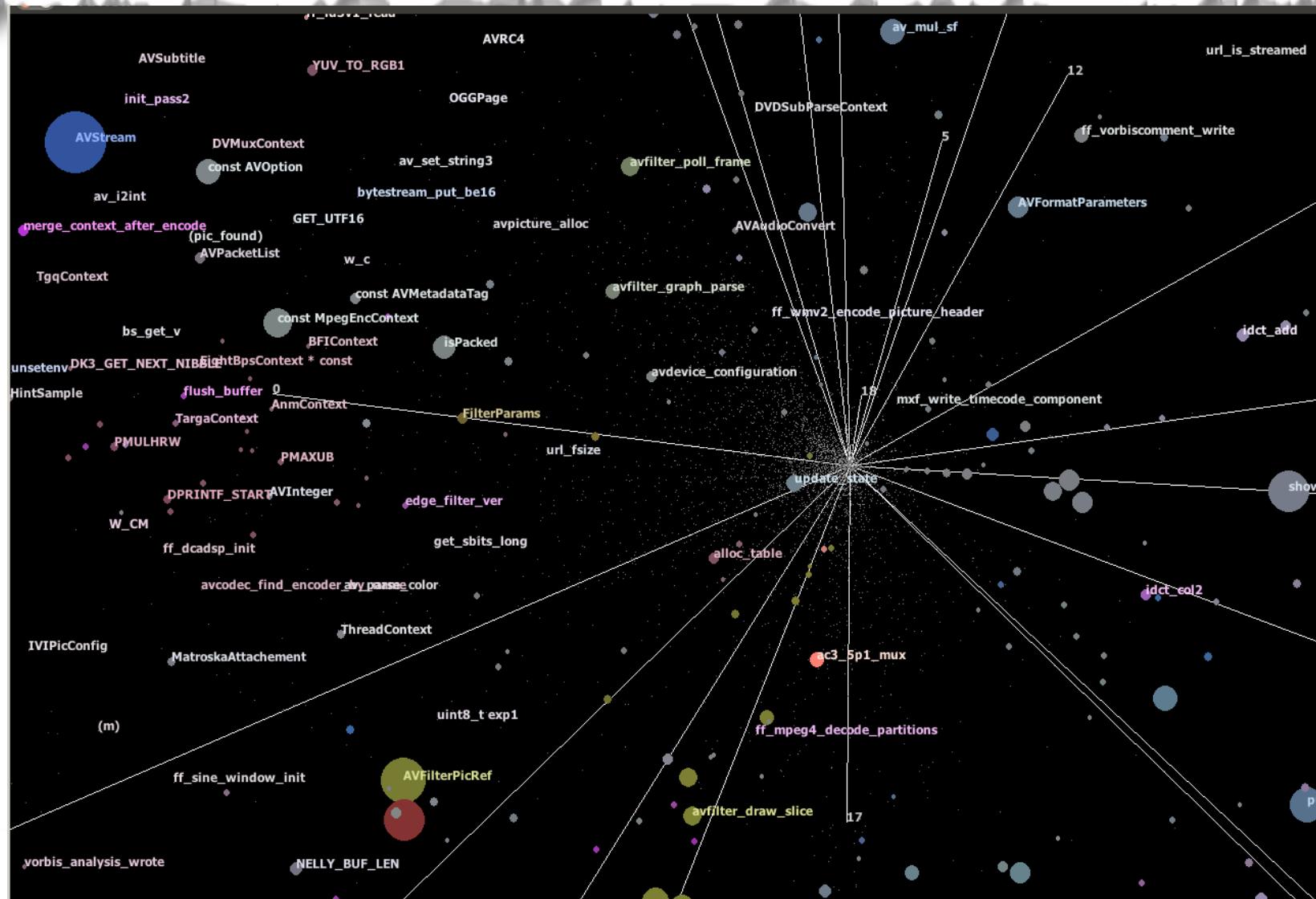
```
void listFunc2(int elem)
{
    GList myList;
    if(!list_check(myList)){
        do_list_error_stuff();
        return;
    }
    list_remove(myList, elem);
    list_delete(myList);
}
```



# We get a lot more than just a method to extrapolate!

- We get a projection of all functions and API symbols into a space where...
  - ... API symbols constituting a pattern are close to one another.
  - ... functions using the same pattern are close to one another.
  - ... functions are close to the dominant API symbols of their most dominant patterns.
- You can browse code in this space.

# **But visualization isn't the way to go for real code-bases ;)**



# **It turns out...**



- ... tables don't look as fancy but are a lot more useful. At least if the visualization you're doing is as bare-bones as the one just shown.
- Let's browse FFmpeg.

# DecodingContexts: Synonyms

- Decoders in FFmpeg form a group of functions related by API use.
- Each decoder has its own decoding-context.
- They are thus ‘noise’-terms to be found at a similar angle.

Score	Symbol
1.00	FlicDecodeContext
0.88,	EightBpsContext * const
0.88,	AnmContext
0.87,	TiffContext * const
0.87,	IdcinContext
0.86,	free_bundles
0.86,	C93DecoderContext * const
0.86,	SgiState * const
0.86,	VCR1Context
0.86,	ff_lzw_decode_close
0.84,	release_buffer
0.81,	BinkContext * const
0.80,	VmdVideoContext
0.80,	LibOpenJPEGContext
0.80,	MsrleContext
0.77,	TMVContext
0.77,	DVVideoContext
0.76,	Msvideo1Context
0.76,	ff_ivi_free_buffers
0.75,	VqaContext
0.74,	KmvcContext *const

# String-API in FFmpeg

Symbol	Score
char	69.12
const char	46.98
strcmp	33.82
snprintf	19.17
exit	16.78
strtol	15.93
fprintf	12.23
strlen	10.47
get_byte	8.28
BytelOContext	7.31
av_strlcpy	6.61
HTTPContext	6.55
enum AVMediaType	6.11
FF_ARRAY_ELEMS	5.55
av_log	5.44
AVFilterGraph	5.13
strncmp	5.05
av_strlcatf	4.94
ADPCMChannelStatus	4.76

Score	Function Name
0.92	matroskadec.c matroska_convert_tag
0.92	sdp.c sdp_get_address
0.92	avfiltergraph.c avfilter_graph_get_filter
0.90	rtsp.c rtsp_parse_transport
0.90	utils.c find_info_tag
0.89	rtsp.c make_setup_request
0.87	movenc.c mov_write_string_metadata
0.84	nutdec.c set_disposition_bits
0.84	tests/rotozoom.c init_demo
0.82	libavutil/error.c av_strerror
0.82	opt.c hexchar2int
0.80	httpauth.c choose_qop
0.80	pnm.c ff_pnm_decode_header
0.80	ffserver.c socket_open_listen
0.80	rdt.c ff_rdt_subscribe_rule
0.80	ffserver.c http_send_too_busy_reply
0.79	audioconvert.c avcodec_sample_fmt_string
0.79	sdp.c sdp_write_header
0.79	ffmpeg.c opt_vstats
0.79	httpauth.c handle_digest_update
0.79	httpauth.c handle_digest_params
0.79	pnm.c pnm_get
0.78	httpauth.c handle_basic_params
0.78	sdp.c sdp_write_address
...	...

# **Functions similar to function - Vulnerability Extrapolation**

- Take a function that used to be vulnerable as an input.
- Measure distances to other functions to determine those functions, which are most similar.
- Let's try that for FFmpeg.

# Original bug: CVE-2010-3429

```
static int flic_decode_frame_8BPP(AVCodecContext *avctx,
                                 void *data, int *data_size,
                                 const uint8_t *buf, int buf_size)
{
    [...]
    pixels = s->frame.data[0]; [...]
    case FLI_DELTA:
        y_ptr = 0;
        compressed_lines = AV_RL16(&buf[stream_ptr]);
        stream_ptr += 2;
        while (compressed_lines > 0) {
            line_packets = AV_RL16(&buf[stream_ptr]);
            stream_ptr += 2;
            if ((line_packets & 0xC000) == 0xC000) {
                // line skip opcode
                line_packets = -line_packets;
                y_ptr += line_packets * s->frame.linesize[0];
            } else if ((line_packets & 0xC000) == 0x4000) {
                [...]
            } else if ((line_packets & 0xC000) == 0x8000) {
                // "last byte" opcode
                pixels[y_ptr + s->frame.linesize[0]-1] = line_packets & 0xff;
            } else {
                [...]
                y_ptr += s->frame.linesize[0];
            }
        }
        break;
    [...]
}
```

## Decoder-Pattern:

Usually a variable of type AvCodecContext  
AV\_RL\*-Functions used as sources.

Lot's of primitive types with specified width used.

Use of memcpy, memset, etc.

unchecked index,  
Write to arbitrary location in memory.

# Extrapolation

- The closest match contained the same vulnerability but it was fixed when the initial function was fixed.
- [You cannot expect this decoder to be the optimal prototype for the decoder class, so yes, it will find non-decoders.]

Score 1	Function Name
1.000000	flic_decode_frame_8BPP (libavcodec/flicvideo.c)
0.964096	flic_decode_frame_15_16BPP (libavcodec/flicvideo.c)
0.826979	lz_unpack (libavcodec/vmdav.c)
0.803331	decode_frame (libavcodec/lcldec.c)
0.796700	raw_encode (libavcodec/rawenc.c)
0.756951	vmdvideo_decode_init (libavcodec/vmdav.c)
0.723750	ymd_decode (libavcodec/vmdav.c)
0.702356	aasc_decode_frame (libavcodec/aasc.c)
0.684610	flic_decode_init (libavcodec/flicvideo.c)
0.665167	decode_format80 (libavcodec/vqavideo.c)
0.664279	targa_decode_rle (libavcodec/targa.c)
0.660454	adpcm_decode_init (libavcodec/adpcm.c)
0.659811	decode_frame (libavcodec/zmbv.c)
0.655338	decode_frame (libavcodec/8bps.c)
0.651587	msrle_decode_8_16_24_32 (libavcodec/msrledec.c)
0.648321	wmavoice_decode_init (libavcodec/wmavoice.c)
0.646872	get_quant (libavcodec/nuv.c)
0.641871	MP3lame_encode_frame (libavcodec/libmp3lame.c)
0.641642	mpegtts_write_section (libavformat/mpegttsenc.c)
0.634922	tgv_decode_frame (libavcodec/eatgv.c)

0-Bug

# 0-Bug

```
static void
[...]
int
int
int
frame_x = AV_RL16(&s->buf[6]);
frame_y = AV_RL16(&s->buf[8]);
frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
frame_height = AV_RL16(&s->buf[12]) - frame_y + 1;
[...]
if          0
/* originally UnpackFrame in VAG's code */

[...]
dp = &s->frame.data[0][frame_y * s->frame.linesize[0] + frame_x];
          0
          0
          0
switch
[...]
case 2
for      0
    memcpy(dp, pb, frame_width);
          0
          0
break
[...]
```

## Decoder-Pattern:

Usually a variable of type AvCodecContext

AV\_RL\*-Functions used as sources.

Lot's of primitive types with specified width used.

Use of memcpy, memset, etc.

Again an unchecked index into the pixel-buffer!

# From 0-Bug to 0-day

Demonstrate that this...



... can be turned into this.



# Writing a binary exploit in 2011

- Writing binary exploits in 2011 is an adventure you do not want to miss.
  - ASLR and DEP have arrived to stay.
  - The heap is hardened.
  - Every bug is kind of different right now and new generic techniques are just emerging.

But when you look deep into the code and use it creatively, you can get it done ☺



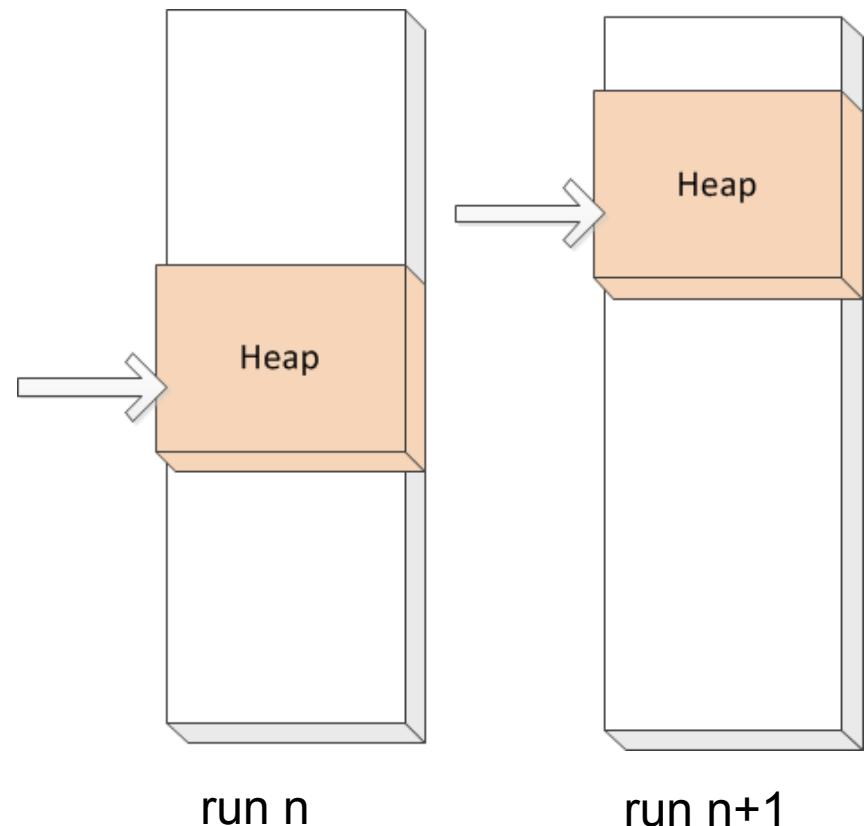
```
static void
[...]
int
int
int
frame_x = AV_RL16(&s->buf[6]);
frame_y = AV_RL16(&s->buf[8]);
frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
frame_height = AV_RL16(&s->buf[12]) - frame_y + 1;
[...]
if          0
/* originally UnpackFrame in VAG's code */

if ((frame_width == 0) && (frame_x || frame_y))
[...]
s->x_off = frame_x;
s->y_off = frame_y;
frame_x -= s->x_off;
frame_y -= s->y_off;

if (frame_x || frame_y)
(frame_height != 0)
memcpy(s->frame.data[0] + frame_y * s->frame.linesize[0] + frame_x,
       0
       0
       0
switch
[...]
case 2      0
for          0
    memcpy(dp, pb, frame_width);
       0
       0
switch (meth)
[...]
case 2:
for (i = 0; i < frame_width; i++)
    memcpy(dp, pb + frame_x + i * frame_width, frame_height);
    pb += frame_width;
    dp += frame_height;
    pp += frame_height;
}
break;
[...]
}
}
```

# Our Bug v.s. ASLR

- Due to ASLR, the start-address of the heap will change from run to run.
- The relative address of the pixel-buffer within the heap will not be affected by ASLR.
- As long as we choose an offset that remains within the boundaries of the heap, ASLR does not hurt us yet.



# **What this bug gives us**



- The ability to write up to 65535 bytes of data specified by us to a location relative to the pixel-buffer.
- **Constraint: Offsets in the interval [-65535; -1] cannot be specified.**

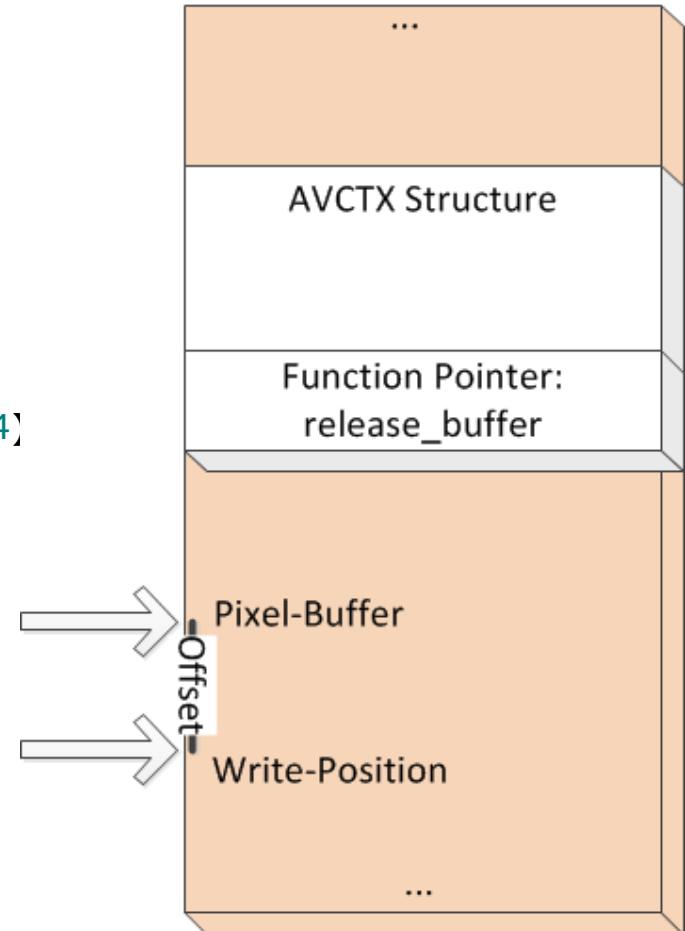
# What do we overwrite?

```
static int vmdvideo_decode_frame(AVCodecContext *avctx,
                                void *data, int *data_size,
                                AVPacket *avpkt)
{
    [...]
    vmd_decode(s);

    /* make the palette available on the way out */
    memcpy(s->frame.data[1], s->palette, PALETTE_COUNT * 4)

    /* shuffle frames */
    FFSWAP(AVFrame, s->frame, s->prev_frame);
    if (s->frame.data[0])
        avctx->release_buffer(avctx, &s->frame);

    [...]
    return buf_size;
}
```



Overwrite the `release_buffer` pointer!

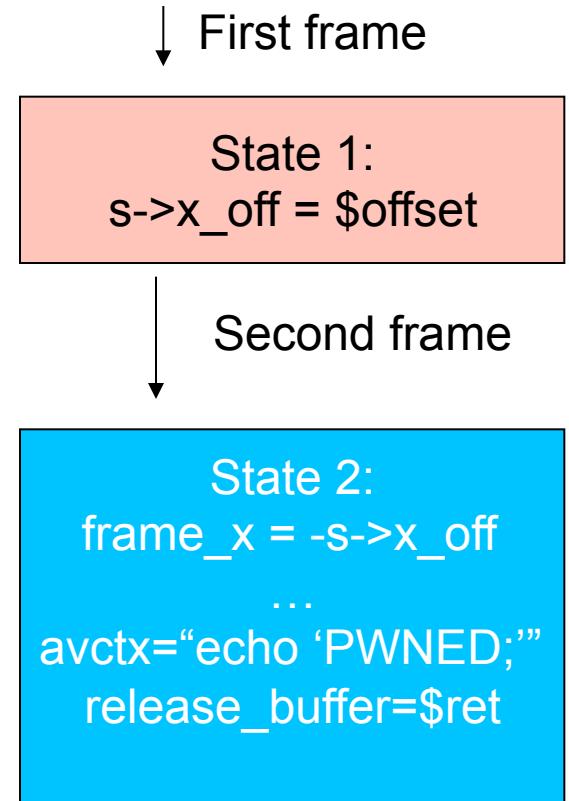
Of course, that's in the interval of offsets we can't use 😞

# The state changing aspect

```
static void vmd_decode(VmdVideoContext *s)
{
    static void vmd_decode(VmdVideoContext *s)
    {
        int frame_x, frame_y;
        int frame_width, frame_height;
        int dp_size;
        frame_x = AV_RL16(&s->buf[6]);
        frame_y = AV_RL16(&s->buf[8]);
        frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
        frame_height = AV_RL16(&s->buf[12]) - frame_y + 1;
        if ((frame_width == s->avctx->width && frame_height == s->avctx->height) &&
            (frame_x || frame_y)) {
            s->x_off = frame_x;
            s->y_off = frame_y;
            frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
            frame_x -= s->x_off;
            frame_y -= s->y_off;
            if ((frame_width == s->avctx->width && frame_height == s->avctx->height) &&
                (frame_x || frame_y)) {
                mem_cpy(s->frame.data[0], s->prev_frame.data[0],
                        s->avctx->height * s->frame.linesize[0]);
                if (s->sizes[0] != 0) {
                    /* originally UnpackFrame in VAG's code */
                    pb = p;
                    meth = *pb++;
                    [...] s->x_off = frame_x;
                    s->y_off = frame_y;
                    switch (meth) {
                    [...] case 2:
                    frame_x -= s->x_off;
                    frame_y -= s->y_off;
                    [...] p += s->prev_frame.linesize[0];
                }
            }
        }
    }
}
```

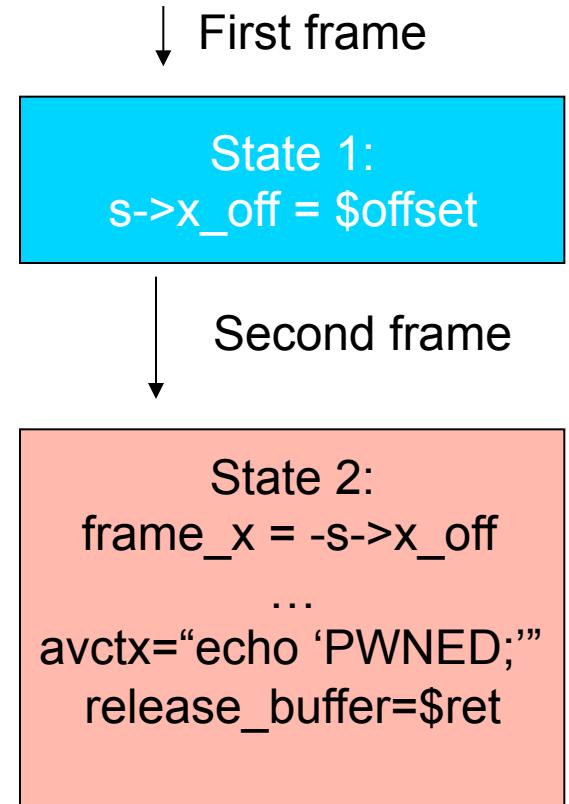
# First frame

- We exploit the bug using two video frames.
- First frame: Put the decoder in a state where  $s->x\_off$  contains our desired sign-inverted offset.



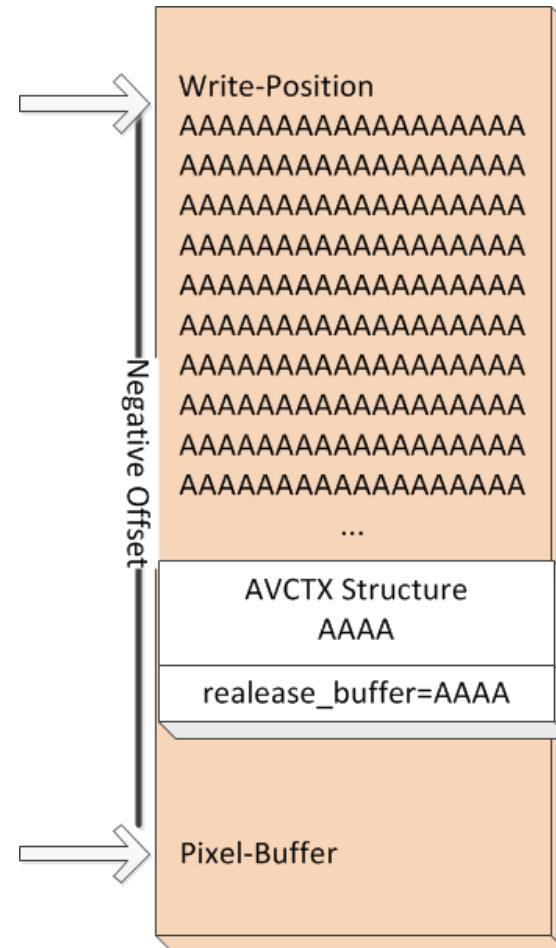
# Second frame

- Second frame: Makes the decoder set `frame_x` to `-s->x_off` and thus we add a negative offset to the pixel-buffer.
- At this negative offset, we overwrite `avctx`:
  - First few 100 bytes: Shell-Commands.
  - Last 4 byte: Address of `<system>-call`.



# Now we can overwrite the pointer!

- Specify the sign-inverted desired offset in the first frame's `frame_x` and do not trigger memory corruption. The offset will be saved in `s->x_off`.
- In the second frame, set `frame_x` to 0 so that `frame_x` will be set to `-s->x_off`.
- Trigger memory corruption with this second frame.



# Now, where do we jump?

- We're in luck: the base image of mplayer is not compiled as position-independent code.
- A small portion of code will be at constant addresses despite ASLR!

What about this? ☺

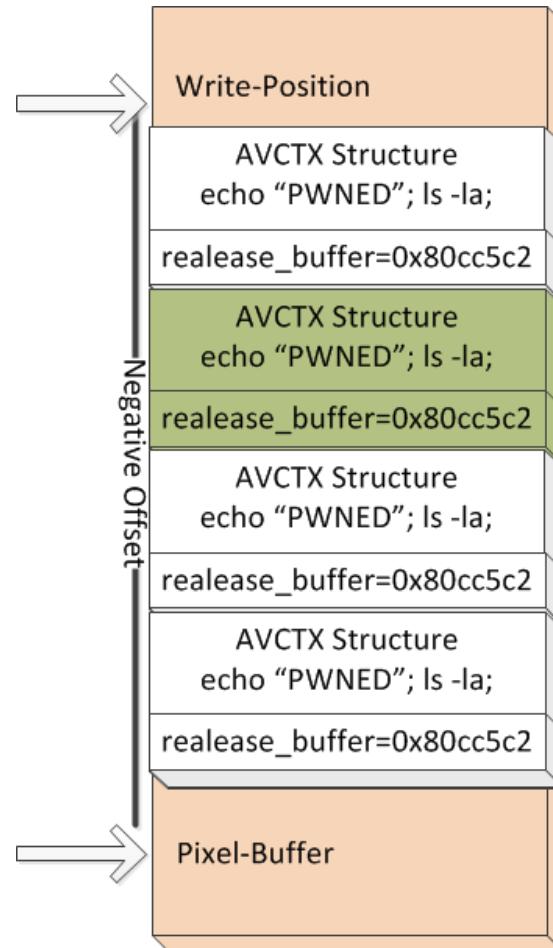
**0x080cc5c2: mov %eax, (%esp)**

**0x080cc5c5: call <system@plt>**



# When shellcode contains shell-commands

- When the call is made, %eax has just been used as an auxiliary register to hold a pointer to avctx.
- avctx will be interpreted as a string of commands for /bin/sh!
- Lightly spray the heap with avctx-structures accordingly.
- Of course, this is not 100% stable but works remarkably well ☺





**DEMO**

---

# **Evaluation: How good does it work in general?**

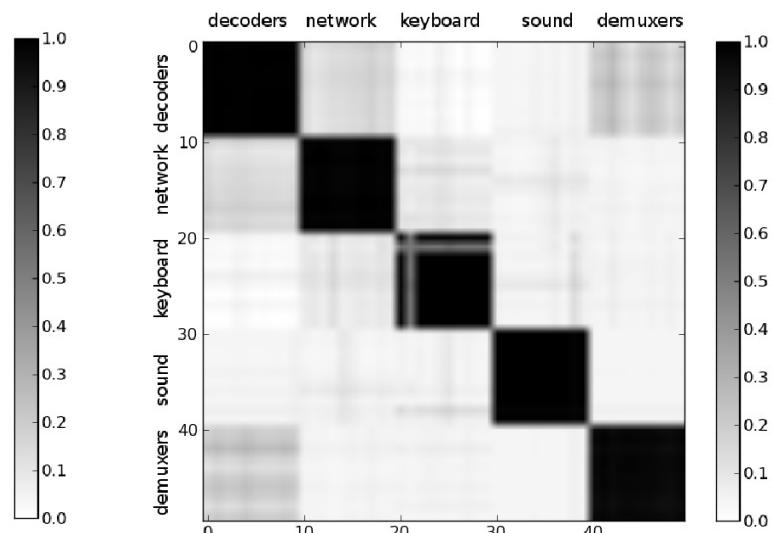
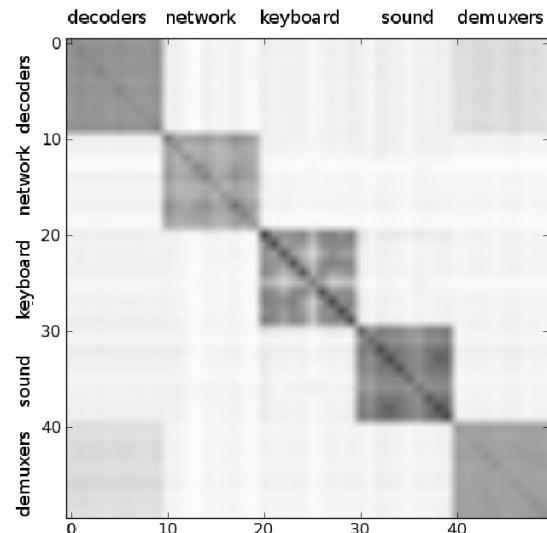
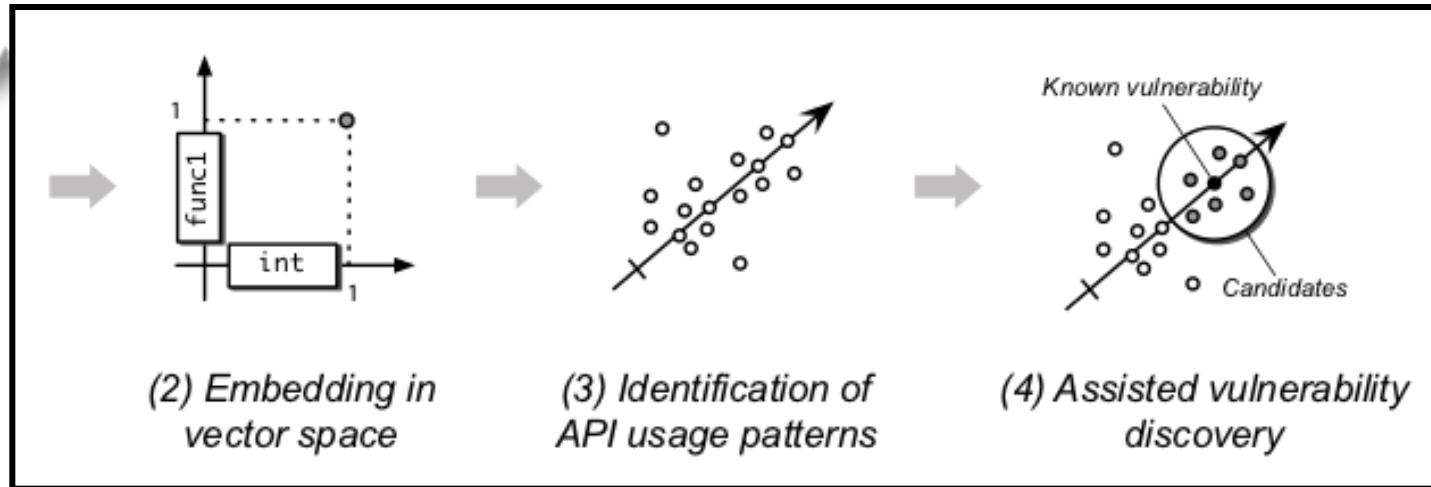
- Of course, if no further similar bug exists in the code-base, this will not work.
- Second: We did not look for the same bug but for the same usage-pattern.
- That does not mean the pattern has to be used wrong in the cases we find as well.
- We just know the things to look for with this pattern.

# **What we *can* show**



- If we do API-discovery by hand, we can check whether the method would have found these groups as well.
- Manually extracted code from the Linux kernel and FFmpeg was used to evaluate the method.

# Evaluation



# Summary

- There are lots of patterns in your bugs.
- There's an info-leak when you disclose a bug: You're providing a *sample* of what went wrong in this specific code-base.
- Fixing a bug without performing proper extrapolation may be contra-productive
- You may be disclosing related 0-day!

# **Where to go from here**



- ... for this method in particular.
- This will probably work well on binaries.
- While PCA was the vanilla algorithm to try out, there are better representations: NMF looks a lot more promising.
- We're currently investigating whether structural features can improve the method.

# Some ideas

- ... and for security in general.
- “Learn” context-free grammars from input to generate fuzzers.
- Cluster fuzz-traces to group input that hits the same bug.
- More robust OS- and rate-limiter detection in port-scanning.
- Heap-chunk usage patterns to identify how APIs interact at runtime?

# Final words



*Whenever you encounter patterns in security research or need to make a fuzzy decision, you may want to give machine learning a shot.*

*It can be beneficial and refreshing to find ways of applying research from other fields to your problems.*

# Questions?



Fabian Yamaguchi  
Vulnerability Researcher

[fabs@recurity-labs.com](mailto:fabs@recurity-labs.com)

Recurity Labs GmbH, Berlin, Germany  
<http://www.recurity-labs.com>