Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning

Fabian Yamaguchi[1], Felix 'FX' Lindner[1], and Konrad Rieck[2]
[1]*Recurity Labs GmbH, Germany*
[2]*Technische Universität Berlin, Germany*

Abstract

Rigorous identification of vulnerabilities in program code is a key to implementing and operating secure systems. Unfortunately, only some types of vulnerabilities can be detected automatically. While techniques from software testing can accelerate the search for security flaws, in the general case discovery of vulnerabilities is a tedious process that requires significant expertise and time. In this paper, we propose a method for assisted discovery of vulnerabilities in source code. Our method proceeds by embedding code in a vector space and automatically determining API usage patterns using machine learning. Starting from a known vulnerability, these patterns can be exploited to guide the auditing of code and to identify potentially vulnerable code with similar characteristics—a process we refer to as *vulnerability extrapolation*. We empirically demonstrate the capabilities of our method in different experiments. In a case study with the library FFmpeg, we are able to narrow the search for interesting code from 6,778 to 20 functions and discover two security flaws, one being a known flaw and the other constituting a zero-day vulnerability.

1 Introduction

The security of computer systems critically depends on the quality and security of its underlying program code. Unfortunately, there is a persistent deficit of security awareness in software development [37] and often the pressure of business competition rules out the design and implementation of secure software. As a result, there exist numerous examples of programming flaws that have led to severe security incidents and the proliferation of malicious software [e.g., 11, 19, 24]. Often these flaws emerge as zero-day vulnerabilities, rendering defense using reactive security tools almost impossible.

From its early days, computer security has been concerned with developing methods for discovery and elimination of vulnerabilities in program code. Due to the fundamental inability of a program to completely analyse another program's code however, determining vulnerabilities automatically has proved to be an involved and often daunting task. Current tools for automatic code analysis, such as Fortify 360 and Microsoft PREfast, are thus limited to detecting vulnerabilities following well-known programming patterns. While techniques derived from software testing, such as fuzz testing [32], taint analysis [20] and symbolic execution [3, 29], may accelerate analysis of program code, the general discovery of vulnerabilities still rests on tedious manual auditing that requires considerable expertise and resources.

As a remedy, we propose a method for assisted discovery of vulnerabilities in source code. Instead of struggling with the limitations of automatic analysis, our method aims at rendering manual auditing more effective by assisting and guiding the inspection of source code. To this end, the method embeds code in a vector space, such that typical patterns of API usage can be determined automatically using machine learning techniques. These patterns implicitly capture semantics of the code and allow to "extrapolate" known vulnerabilities by identifying potentially vulnerable code with similar characteristics. This process of *vulnerability extrapolation* can suggest candidates for investigation to the analyst as well as ease the browsing of source code during auditing.

We empirically demonstrate the capabilities of this method to identify usage patterns and to accelerate code auditing in different experiments. In a case study with the popular library FFmpeg and a known vulnerability (CVE-2010-3429), our method narrows the search for interesting code from 6,778 to 20 functions. Out of these 20 functions, we can identify two security flaws, one being another known weakness and the other constituting a zero-day vulnerability. We prove the relevance of this finding by providing a working exploit.

The rest of this paper is structured as follows: we introduce our method for vulnerability extrapolation in Section 2. An evaluation and a case study with FFmpeg

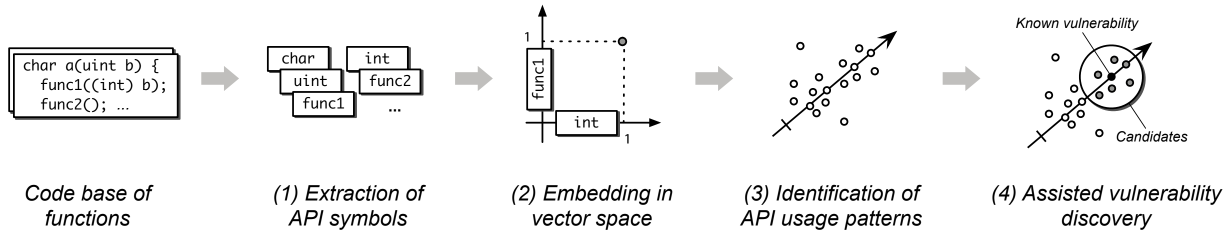

char a(uint b) {
func1((int) b);
func2(); ...
char
int
uint
func2
func1
...
1
func1
int
1
Known vulnerability
Candidates
Code base of functions
(1) Extraction of API symbols
(2) Embedding in vector space
(3) Identification of API usage patterns
(4) Assisted vulnerability discovery

Figure 1: A schematic overview of our method for vulnerability extrapolation.

are presented in Section 3. We discuss related work in Section 4 and conclude in Section 5.

2 Vulnerability Extrapolation

The concept of *vulnerability extrapolation* builds on the idea of identifying unknown vulnerabilities using programming patterns observed in known security flaws. The rational underlying this concept is that vulnerabilities are often directly linked to patterns of specific API usage. For example, the unfortunate interactions of several basic routines for string and memory processing have been used for decades to identify "low-hanging fruit" vulnerabilities. This intuitive concept of extrapolation, however, poses two challenges: First, how can we capture patterns of API usage automatically, and second, how can we transfer these patterns from known vulnerabilities to other code fragments?

To tackle these problems, we combine techniques from static code analysis and machine learning. In particular, our method proceeds by mapping the source code under investigation to an expressive vector space, such that patterns of API usage can be geometrically inferred and used to guide the search for vulnerabilities. This extrapolation process can be described in four steps.

1. **Extraction of API symbols.** In the first step the source code is tokenized and parsed into individual functions. For each function, we extract the names of referenced types and functions. We refer to these extracted symbols as *API symbols*.

2. **Embedding in a vector space.** Using the extracted symbols, each function is embedded in a vector space, such that each dimension is associated with one API symbol. This representation allows us to model and identify API usage geometrically.

3. **Identification of API usage patterns.** In the third step, we apply the technique of Principal Component Analysis that enables us to infer discriptive directions in the vector space, which correspond to *dominant API usage patterns*.

4. **Assisted vulnerability discovery.** Finally, we express each function as a mixture of dominant API usage patterns. Starting from the vectorial location of a known vulnerability, this representation enables us to identify functions sharing similar API usage and possibly containing similar vulnerabilities.

In the following sections, we discuss these steps in more detail and provide the required theoretical and technical background.

2.1 API Symbols and Usage Patterns

The term *Application Programming Interface* or *API* usually refers to interfaces of software libraries. For discovery of vulnerabilities, we make use of this term in a broader sense and denote all source-level interfaces referencing semantically related code as APIs. Such interfaces arise naturally from modular program design and may correspond to classes, libraries, frameworks as well as collections of utility functions.

We refer to any identifier used to access functionality of an API as an *API symbol*. In particular, we consider names of types, names of functions and type casts as API symbols. As an example, Figure 2 shows the API symbols associated with a simple C function. The accessed API corresponds to the functions `func2` and `func3` and involves different types, such as `int` and `struct bar`.

```
static char func1(unsigned int a, struct foo *b)
{
    int c = 0;
    struct bar *d;

    if (a == 0) {
        d = func2((int) a);
    } else {
        c = func3((struct bar *) b);
    }

    return c;
}
```

Figure 2: Source code of an exemplary C function. API symbols are indicated by bold typeface.

Based on the API symbols, we can define the notion of an *API usage pattern*, which simply corresponds to a set of symbols used in several functions. These patterns can cover different functionality of the source code. For example, an API usage pattern may correspond to lock functions, such as `mutex_lock` and `mutex_unlock`, whereas another pattern might reflect typical string functions, such as `strcpy`, `strcat` and `strlen`. To distinguish random combinations of symbols from relevant code, we consider only *dominant* usage patterns which occur frequently in the code base. We will see in Section 2.3 how dominant API usage patterns can be identified automatically.

As the first step of our analysis, we thus tokenize and parse the source code under investigation into individual functions (or alternatively functional blocks). For each function, we then extract its API symbols and store these as sets for subsequent processing.

2.2 From API Symbols to Vectors

API symbols and usage patterns are intuitive to a human analyst, yet both concepts are not directly suitable for application of machine learning. Learning techniques usually operate on numerical vectors and often express patterns as combinations of vectors. Inspired by the vector space model from information retrieval [28], we thus embed the functions of our code base in a vector space using the API symbols. This allows us to conduct the search for vulnerable code in a geometric manner.

To present this embedding, we first need to introduce some basic notation. We denote by $\mathcal{X} = \{x_1, \ldots, x_n\}$ the set of functions in our code base and refer to $\mathcal{S}$ as the set of all API symbols contained in $\mathcal{X}$. We can then define a mapping ϕ from $\mathcal{X}$ to an $|\mathcal{S}|$-dimensional vector space, whose dimensions are associated with the API symbols $\mathcal{S}$. Formally, this map ϕ is defined as

$$\phi : \mathcal{X} \longmapsto \mathbb{R}^{|\mathcal{S}|}, \quad \phi(x) \longrightarrow (\phi_s(x))_{s \in \mathcal{S}} \,.$$

For a given function $x \in \mathcal{X}$ the value at the dimension associated with the symbol $s \in \mathcal{S}$ is computed by

$$\phi_s(x) := I(s,x) \cdot \text{TFIDF}(s)$$

where I is simply an indicator function

$$I(s,x) = \begin{cases} 1 & \text{if the symbol } s \text{ is contained in } x \\ 0 & \text{otherwise} \end{cases}$$

and TFIDF(s) corresponds to a standard weighting term used in information retrieval. This weighting ensures that the contribution of very frequent API symbols is lowered, similar to stop words in natural language text. A detailed introduction to this mapping technique is provided in the book of Salton and McGill [27].

For convenience and later processing, we store the vectors of all functions in our code base in a matrix M, where one element of the matrix is defined as

$$M_{s,x} := \phi_s(x).$$

As a result, the matrix M consists of $|\mathcal{X}|$ column vectors each containing $|\mathcal{S}|$ elements.

Apparently, the embedding of source code introduces a dilemma: On the one hand, it is desirable to analyse as many API symbols as possible, while on the other hand storing billions of elements in a matrix M may get intractable. However, the map ϕ is *sparse*, that is, a function x contains only few of all possible API symbols and thus the majority of elements in M is zero. This sparsity can be exploited to extract and compare vectors $\phi(x)$ with linear-time complexity using data structures, such as hash maps and sorted arrays [see 25].

2.3 Principal Component Analysis

The mapping outlined in the previous section allows for comparison of functions in terms of API symbols, simply by computing distances between the respective vectors. However, this vectorial representation alone is not sufficient for effective discovery of vulnerabilities, as these are not characterized by individual API symbols but patterns of symbols. For example, functions may use the same API but utilize different subsets, such that the underlying usage pattern is only reflected in the combination of all subsets.

This problem of composing usage patterns can be addressed by *Principal Component Analysis*—a standard technique of machine learning for automatically determining descriptive directions in a vector space [9]. In our setting, these directions are associated with combinations of API symbols and can be interpreted as dominant API usage patterns. Moreover, the directions define a low-dimensional subspace that the original vectors can be projected to. Functions that do not share any symbols but make us of the same API lie close to each other in this subspace, as they fall onto the same direction identified by PCA. This technique of projecting data to a low-dimensional subspace using PCA is also referred to as Latent Semantic Analysis [8], a name that indicates the ability to extract latent semantic relations from data.

Formally, PCA seeks d orthogonal directions in the vector space that capture as much of the variance inside the data as possible. One way to obtain these d directions is by performing a *truncated Singular Value Decomposition* (SVD) of the matrix M. This decomposition can be implemented efficiently using the Lanczos algorithm, an iterative procedure suited for high-dimensional and sparse data. For computing this decomposition we make use of the popular library SVDLIBC [26].

The truncated SVD decomposes the matrix M into three matrices U, Σ and V which offer a wealth of information about the code base and API usage. This decomposition has the following form

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

In particular, we obtain three relevant sources of information that describe the dominant patterns of API usage, their relevance and the relation of function and symbols to these patterns.

1. The d columns of the unitary matrix U correspond to the principal components of PCA and thus reflect the d most dominant API usage patterns—prevalent combinations of API symbols in the code base.
2. The diagonal matrix Σ contains the singular values of M. The values indicate the variances of the principal components and allow us to assess the individual importance of the d API usage patterns.
3. The rows of U and V contain the projected representations of API symbols and functions, respectively. While the matrix V can be used to measure the similarity of functions, U comes handy if API symbols need to be traced back to usage patterns.

As we will see in the following, these three matrices provide the basis for assisted discovery of vulnerabilities and conclude the rather theoretical presentation of our practical method.

2.4 Assisted Vulnerability Discovery

Once the decomposition has been calculated, which takes minutes on average consumer hardware, the analyst can query the obtained information and matrices in real time. Hence, our method can be directly applied to assist an analyst while browsing and auditing source code. In particular, the following three activities can be conducted during an auditing session.

Vulnerability extrapolation. By comparing the row vectors of V using a similarity measure, such as the cosine similarity [27], the relations of all functions in the code base can be assessed. This allows for quickly discovering functions that share similar API usage patterns and builds the basis for extrapolating vulnerabilities. Given a function containing a known vulnerability, the analyst can scan the code base for occurrences of similar API usage and focus on functions related to the vulnerability. This guided search for vulnerabilities can significantly reduce the number of functions to be audited. We demonstrate this practice on a real-life example in Section 3.2.

Extracting dominant usage patterns. The proposed method can also be used as a pre-processing step for in-depth analysis. The column vectors of the matrix U correspond to the d most dominant API usage patterns and their respective combinations of symbols. Using these patterns, an analyst can easily group the code base into different subsets and concentrate on particular usage patterns, for example, by restricting the audit only to functions making use of security-critical APIs, such as network and authentication routines.

API browsing. As the majority of software is developed in a modular manner, any code base of reasonable size necessarily contains internal APIs [6]. Often these internal APIs are scarcely documented and scattered across different files in the code base. Nonetheless, an understanding of these APIs can be crucial for identifying more subtle vulnerabilities. Our method assists an analyst in understanding public as well as internal APIs. By comparing rows in V (functions) with rows in U (symbols), an analyst can determine important API symbols associated with the APIs used in a function. In the same manner, it is possible to determine functions, which best match a constructed set of API symbols. This allows a very directed search for occurrences of particular patterns known to commonly cause problems.

3 Evaluation and Case Study

Thus far we have seen how source code can be modeled and analysed for discovery of vulnerabilities using machine learning techniques. In practice however, it is not the sophisticated design of a method that matters, but its ability to really assist in day-to-day auditing. To study the efficacy of the proposed method in practice, we thus conduct two experiments with real source code. In the first experiment, we quantitatively evaluate the ability of our method to automatically identify API usage patterns and to structure source code (Section 3.1). In the second experiment—a case study—we apply our method for vulnerability extrapolation to the library FFmpeg and construct a working exploit for a discovered zero-day vulnerability (Section 3.2).

3.1 Quantitative Evaluation

The effectivity of vulnerability extrapolation rests on the accurate identification of API usage patterns. To validate this capability, we construct an evaluation code base that comprises functions from different classes of API usage. We ensure that these classes contain distinct usage patterns by selecting functions from different contexts and applications. In particular, we consider a total of 420 functions from the Linux Kernel (2.6.32) and the media-decoding library FFmpeg (0.6.0) which are assigned to the following five classes:

1. Functions for sending network data (Linux kernel)
2. Functions for probing keyboards (Linux kernel)
3. Functions for probing sound drivers (Linux Kernel)
4. Functions for media demuxing (FFmpeg)
5. Functions for media decoding (FFmpeg)

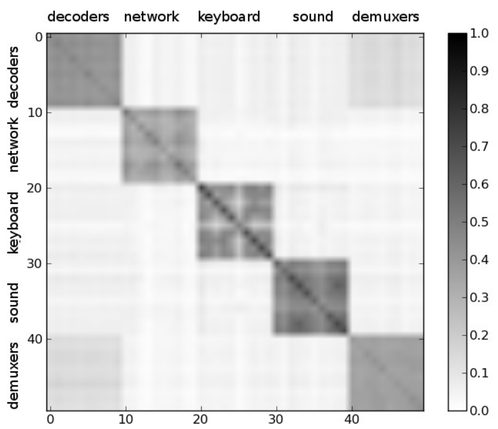
The functions in each class are randomly partitioned into subsets, such that each subset has approximately the same size and each class is split into ten subsets. We then apply our method to the resulting code base and study how inner-class and intra-class relationships are captured by embedding functions in a vector space and by projecting the functions to directions determined by PCA.

Figure 3(a) presents the pairwise similarities between the subsets of the five classes directly measured in the vector space, that is, prior to application of PCA. The similarities are depicted as a matrix, where each cell shows the average cosine similarity between the functions in one subset and another. While the matrix shows some structured contour, most of its surface appears blurred. A notable variance between similarity scores within a class is observable and several subsets of different classes can hardly be discriminated. It is evident that embedding of functions alone is not sufficient for determining usage patterns in source code.

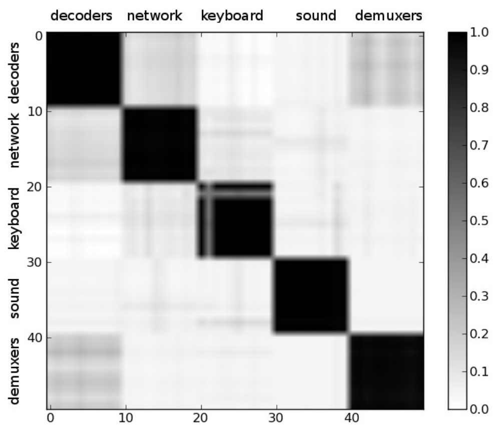
Figure 3(b) shows the pairwise similarities between the subsets measured after the embedded functions have been projected to the top five directions identified by PCA. That is, the functions are represented as mixtures of API usage patterns instead of individual symbols. In this projected representation, inter-class similarities are significantly higher than in the original vector space and high distances between functions of different classes can be observed. The application of PCA removes “noise” from the code base and thereby allows to infer relevant patterns for discriminating the five classes—a prerequisite for effective extrapolation of vulnerabilities.

3.2 Case Study: FFmpeg

We finally demonstrate by example how the proposed method can be integrated into a real auditing task, where it plays a key role in the identification of a zero-day vulnerability. For this case study, we consider the



(a) Similarity matrix for embedded functions.



(b) Similarity matrix for projected functions using PCA.

Figure 3: Similarity matrix for (a) embedded functions and (b) embedded functions projected to the top 5 directions of PCA. Dark shading indicates high similarity.

widely used open-source media decoding library FFmpeg (0.6.0) and extrapolate a recently discovered vulnerability found in the processing of FLIC videos.

The original vulnerability. In September 2010, the open source CERT reported a security vulnerability (CVE-2010-3429) in FFmpeg attributed to Cesar Bernardini, which allows an attacker to write data to arbitrary locations in memory relative to a pointer on the heap via crafted FLIC media frames [1]. The vulnerability is contained in the function `flic_decode_frame_8BPP` displayed in Figure 4, which is called for each frame of an 8 bit-per-pixel video.

The critical write operation is performed on line 29, where the least significant byte of the user-supplied integer `line_packets` is written to a location rela-

```
 1  static int flic_decode_frame_8BPP(AVCodecContext *avctx,
 2                                    void *data, int *data_size,
 3                                    const uint8_t *buf, int buf_size)
 4  {   [...] signed short line_packets; int y_ptr; [...]
 5      pixels = s->frame.data[0];
 6      pixel_limit = s->avctx->height * s->frame.linesize[0];
 7      frame_size = AV_RL32(&buf[stream_ptr]); [...]
 8      frame_size -= 16;
 9      /* iterate through the chunks */
10      while ((frame_size > 0) && (num_chunks > 0)) { [...]
11          chunk_type = AV_RL16(&buf[stream_ptr]);
12          stream_ptr += 2;
13          switch (chunk_type) { [...]
14          case FLI_DELTA:
15              y_ptr = 0;
16              compressed_lines = AV_RL16(&buf[stream_ptr]);
17              stream_ptr += 2;
18              while (compressed_lines > 0) {
19                  line_packets = AV_RL16(&buf[stream_ptr]);
20                  stream_ptr += 2;
21                  if ((line_packets & 0xC000) == 0xC000) {
22                      // line skip opcode
23                      line_packets = -line_packets;
24                      y_ptr += line_packets * s->frame.linesize[0];
25                  } else if ((line_packets & 0xC000) == 0x4000) {
26                      [...]
27                  } else if ((line_packets & 0xC000) == 0x8000) {
28                      // "last byte" opcode
29                      pixels[y_ptr + s->frame.linesize[0] - 1] =
30                          line_packets & 0xff;
31                  } else { [...]
32                      y_ptr += s->frame.linesize[0];
33                  }
34              }
35              break; [...]
36          } [...]
37      } [...]
38      return buf_size;
39  }
```

Figure 4: Original vulnerability (CVE-2010-3429).

tive to the heap-based buffer pixels. It has been overlooked that the offset is dependent on y_ptr and s->frame.linesize[0], both of which can be controlled by an attacker. In fact, due to the loop starting at line 18, it is possible to assign an arbitrary value to y_ptr independent of the last value stored in line_packets and no check is performed to verify whether the offset remains within the confined regions of the buffer. It is thus possible for an attacker to write arbitrary bytes to arbitrary locations in memory.

Extrapolation. For discovery of similar vulnerabilities, we apply our method to the code base of FFmpeg consisting of 6,778 functions. For PCA, we choose $d = 200$ and thereby project the embedded functions to a subspace capturing up to 200 unique API usage patterns. Table 1 lists the 20 most similar functions to flic_decode_frame_8BPP in this subspace. Note that we have found 20 to be a reasonable number of functions to consider in one batch and, as we will see shortly, sufficiently large for identification of vulnerabilities.

Inspecting the functions listed in Table 1, we first spot a similar flaw in flic_decode_frame_15_16BPP, where our method reports a similarity of 96%. This vulnerability has been discovered previously and is patched in the current versions of FFmpeg. Surprisingly however, another similar bug in function vmd_decode located in a different source file has not discovered by the developers. Our method reports a similarity of 72% for vmd_decode leading us almost instantly to this unknown vulnerability. The vulnerability is shown in Figure 5 and 6.

Similarity	Function name
1.00	flic_decode_frame_8BPP
0.96	flic_decode_frame_15_16BPP
0.83	lz_unpack
0.80	decode_frame (lcldec.c)
0.80	raw_encode
0.76	vmdvideo_decode_init
0.72	vmd_decode
0.70	aasc_decode_frame
0.68	flic_decode_init
0.67	decode_format80
0.66	targa_decode_rle
0.66	adpcm_decode_init
0.66	decode_frame (zmbv.c)
0.66	decode_frame (8bps.c)
0.65	msrle_decode_8_16_24_32
0.65	wmavoice_decode_init
0.65	get_quant
0.64	MP3lame_encode_frame
0.64	mpegts_write_section
0.64	tgv_decode_frame

Table 1: Top 20 of 6,778 functions ranked by cosine similarity to flic_decode_frame_8BPP. Discovered vulnerabilities are indicated by a shaded background.

Just like the original function, vmd_decode reads the frame dimensions and offsets specified by the individual frame on line 8 to 11 and then calculates an offset into the pixel buffer based on these values on line 34. The function fails to validate whether the given offset references a location within the buffer. Therefore, as user-supplied data is copied to the specified location on line 43, an attacker can corrupt memory by choosing an offset outside of the buffer

In this case study, our method is mainly used to identify functions sharing similar API usage patterns, yet this search for semantic similarities is pivotal for discovery of vulnerabilities. Note that the bodies of the original function and the discovered vulnerability differ significantly and a simple comparison would have been insufficient to spot their relation. By contrast, this study demonstrates that API usage patterns are commonly linked to sets of semantically related functions, simply because similar tasks are usually solved by similar means within a code base. Consequently, functions similar by these terms are often plagued by related vulnerabilities.

Exploit. To demonstrate the relevance of this finding, we craft an exploit for the discovered vulnerability targeting the popular media player MPlayer that is linked to the FFmpeg library on Ubuntu Linux (10.04 LTS). On this platform, MPlayer is not compiled as a position-

```c
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_x -= s->x_off;
    frame_y -= s->y_off;
    [...]
    if (frame_x || frame_y || (frame_width != s->avctx->width) ||
        (frame_height != s->avctx->height)) {
        memcpy(s->frame.data[0], s->prev_frame.data[0],
            s->avctx->height * s->frame.linesize[0]);
    }
    [...]
    if (s->size >= 0) {
        /* originally UnpackFrame in VAG's code */
        pb = p;
        meth = *pb++;
        [...]

        dp = &s->frame.data[0][frame_y * s->frame.linesize[0]
             + frame_x];
        dp_size = s->frame.linesize[0] * s->avctx->height;
        pp = &s->prev_frame.data[0][frame_y *
             s->prev_frame.linesize[0] + frame_x];

        switch (meth) {
        [...]
        case 2:
            for (i = 0; i < frame_height; i++) {
                memcpy(dp, pb, frame_width);
                pb += frame_width;
                dp += s->frame.linesize[0];
                pp += s->prev_frame.linesize[0];
            }
            break;
                [...]
        }
    }
}
```

Figure 5: Discovered zero-day vulnerability

```c
static int vmdvideo_decode_frame(AVCodecContext *avctx,
                                 void *data, int *data_size,
                                 AVPacket *avpkt)
{
    const uint8_t *buf = avpkt->data;
    int buf_size = avpkt->size;
    VmdVideoContext *s = avctx->priv_data;

    s->buf = buf;
    s->size = buf_size;

    [...]

    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);

    [...]
}
```

Figure 6: Caller of vulnerable function `vmd_decode`.

independent executable and thus the image of the executable is located at a predictable offset. As a result, we can successfully exploit the vulnerability, despite contemporary anti-exploitation techniques, such as Address Space Layout Randomization. A detailed description of the exploit is presented in Appendix A and further background on this case study is presented in [38].

4 Related Work

Code analysis and methods for detection of vulnerabilities have been a vivid area of research in computer security. Over the last years, many different concepts and techniques have been devised to tackle this problem. Our contribution is related to several of these approaches, as we discuss in the following.

Our concept of representing code based on patterns of API usage is motivated by the fact that classes of vulnerabilities can often be directly linked to distinct API symbols, a correspondence that is well-known to practitioners and reflected in several static analysis tools, such as *Flawfinder* [35], *RATS* [2] or *ITS4* [33]. These tools offer fixed databases of API symbols commonly found in conjunction with vulnerabilities and allow a target code base to be scanned for their occurrences.

In academic security research, the connection between API symbols and vulnerability classes has also been recognized and provides a basis for taint analysis [20, 29]. In taint analysis, an analyst can describe a class of vulnerabilities by a *source-sink* system defined over API symbols. If data tainted by an attacker and stemming from one of the sources propagates to a sink without undergoing validation, a vulnerability is detected. The success of this approach has been demonstrated for different types of vulnerabilities and attacks, such as SQL injection [18], Cross Site Scripting [14] and integer-based vulnerabilities [34]. In most realizations, taint analysis is a dynamic process and thus limited to discovery of vulnerabilities observable during execution of a program.

A second strain of research has considered symbolic execution as an extension to taint analysis for detecting vulnerabilities in source code [3, 29]. Most notably is the work of Avgerinos et al. [3] that introduces a framework for finding and even exploiting vulnerabilities using symbolic execution. Despite some amazing results, symbolic execution suffers from the infeasibility of exploring all possible execution paths and its application in practice critically depends on heuristics for pruning off execution branches. In the general case, the search for vulnerabilities thus remains a manual process, which however can be accelerated by assisted analysis, as shown in this work

For static code analysis, Engler et al. [10] are among the first to introduce a method suitable for detecting flaws attributed to programming patterns. However, their method requires a manual definition of these patterns. As an extension, Li and Zhou [16] present an approach for mining programming rules and automatically detecting their violation. An inherent problem of this approach is that a frequent programming mistakes will lead to the inference of a valid pattern and thus common flaws cannot be detected. Williams et al. [36] as well as Livshits et al. [17] address this problem and incorporate software revision histories into the analysis. The detection of programming rules not related to corrections of code, therefore becomes less likely. On the downside, only programming rules violated in the past can be detected, making the discovery of previously unknown flaw patterns impossible.

Finally, techniques from the field of machine learning have been successfully applied in several areas of security, such as for intrusion detection [e.g., 7, 12, 15] and analysis of malicious software [e.g., 4, 5, 23]. A large body of research has been concerned with the design of learning-based security systems, as well as with their shortcomings [13, 30] and evasion possibilities [21, 22, 31]. However, to our knowledge, the application of machine learning to problems of offensive security, such as vulnerability discovery, has gained almost no attention so far.

5 Conclusion

We have introduced a method for assisted discovery of vulnerabilities in source code, deliberately leaving aside the known difficulties of fully automated analysis. Our method accelerates the process of manual code auditing by quickly identifying patterns of API usage in a code base and suggesting code related to known vulnerabilities to a security analyst. We have demonstrated on real production code that once a vulnerability is known, similar vulnerabilities can be easily identified by cycling through similar functions determined by our method.

The proposed method uses API usage patterns for analysing the code base. Many vulnerabilities can be captured well by API usage, yet there also exist cases where the code structure of a function is more relevant for auditing. While the proposed method can not uncover vulnerabilities building only on these patterns, we are currently investigating techniques for integrating structural information from source code into the process of vulnerability extrapolation. Moreover, the ability of our approach to narrow the auditing process to a few interesting functions may also play well with software testing, for example, for selectively fuzzing functions or performing involved symbolic execution.

In conclusion, we can note that fixing a single vulnerability *without performing sufficient extrapolation*, as currently performed by many vendors of software, can be contra-productive, given that it provides attackers with information that may be used to identify similar yet unpatched vulnerabilities. Vulnerability extrapolation can help here to strengthen software security and to support the elimination of related vulnerabilities in practice.

Acknowledgments

The authors gratefully acknowledge the funding from *Bundesministerium für Bildung und Forschung* under the project PROSEC (FKZ 01BY1145).

References

[1] 2010-004 ffmpeg/libavcodec arbitrary offset dereference. Open Source Computer Emergency Response Team, http://www.ocert.org/advisories/ocert-2010-004.html, visited April, 2011.

[2] Rough auditing tool for security. Fortify Software Inc., https://www.fortify.com/ssa-elements/threat-intelligence/rats.html, visited April, 2011.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2011.

[4] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Recent Adances in Intrusion Detection (RAID)*, pages 178–197, 2007.

[5] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2009.

[6] J. Bloch. How to design a good API and why it matters. In *Proc. of ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 506–507, 2006.

[7] G. Cretu, A. Stavrou, M. Locasto, S. Stolfo, and A. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *Proc. of IEEE Symposium on Security and Privacy*, 2008.

[8] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[9] R. Duda, P.E.Hart, and D.G.Stork. *Pattern classification*. John Wiley & Sons, second edition, 2001.

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72, 2001.

[11] N. Falliere, L. O. Murchu, , and E. Chien. W32.stuxnet dossier. Symantec Corporation, 2011.

[12] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proc. of IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, USA, 1996.

[13] C. Gates and C. Taylor. Challenging the anomaly detection paradigm: A provocative discussion. In *Proc. of New Security Paradigms Workshop (NSPW)*, pages 21–29, 2006.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy*, pages 6–263, 2006.

[15] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proc. of Conference on Computer and Communications Security (CCS)*, pages 251–261, 2003.

[16] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of European Software Engineering Conference (ESEC)*, pages 306–315, 2005.

[17] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proc. of the 10th European Software Engineering Conference (ESEC)*, pages 296–305, 2005.

[18] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proc. of USENIX Security Symposium*, 2005.

[19] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an internet worm. In *Proc. of Internet Measurement Workshop (IMW)*, pages 273–284, 2002.

[20] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2005.

[21] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *Recent Adances in Intrusion Detection (RAID)*, pages 81–105, 2006.

[22] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proc. of IEEE Symposium on Security and Privacy*, pages 17–31, 2006.

[23] R. Perdisci, A. Lanzi, and W. Lee. McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 301–310, 2008.

[24] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Proc. of USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.

[25] K. Rieck and P. Laskov. Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9(Jan):23–48, 2008.

[26] D. Rohde. SVDLIBC - Doug Rohde's SVD C Library. http://tedlab.mit.edu/~dr/SVDLIBC/, visited April, 2011.

[27] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1986.

[28] G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[29] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[30] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of IEEE Symposium on Security and Privacy*, pages 305–316, 2010.

[31] Y. Song, M. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode: Re-thinking the role of learning in intrusion detection systems. *Machine Learning*, 2009.

[32] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[33] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 257–267, 2000.

[34] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2009.

[35] D. A. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder/, visited April, 2011.

[36] C. C. Williams, J. K. Hollingsworth, and S. Member. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31:466–480, 2005.

[37] G. Wurster and P. Oorschot. The developer is the enemy. In *Proc. of New Security Paradigms Workshop (NSPW)*, 2008.

[38] F. Yamaguchi. Automated extraction of API usage patterns from source code for vulnerability identification. Diploma thesis, Technische Universität Berlin, 2011.

A Exploit

For the interested reader, we provide a description of a proof-of-concept exploit developed for the identified vulnerability. The exploit demonstrates that attackers can execute arbitrary code with the privileges of the target process if the user can be enticed into opening a crafted media file. The following describes our setup.

1. The target platform is an Ubuntu Linux 10.04 LTS (Lucid Lynx), which uses Address Space Layout Randomization (ASLR), non-executable data regions and a hardened heap implementation to hinder exploitation.

2. FFmpeg is used by a number of frontends. In this study, we assume that the popular media player MPlayer is used as a frontend to FFmpeg.

Recall that the identified vulnerability allows arbitrary data to be written to locations on the heap relative to the pixel buffer `dp`. This allows heap management structures and the contents of other heap chunks to be overwritten. This capability is made use of in the following way.

Identifying a function pointer on the heap. A simple way of redirecting the flow of execution to arbitrary addresses is to overwrite pointers to functions known to be used after the overwrite.

A suitable pointer is stored on the heap in the codec-context structure `avctx` in `avctx->release_buffer`. This pointer is used by `vmdvideo_decode_frame` in a call on line 29 shortly after calling the vulnerable function vmd_decode.

However, given that the context structure avctx is allocated at codec initialization many allocations prior to that of the pixel buffer dp, one must be able to specify negative offsets to overwrite avctx and in particular the release_buffer pointer. This can be achieved as described in the next paragraph.

Crafting of frames to overwrite the pointer. Using a single video frame to exploit the vulnerability exposes a problem: The attacker-supplied values used to calculate the offset are all 16 bit integers and in summary, offsets in the interval $[-65535; -1]$ cannot be specified and thus the avctx cannot be overwritten.

To bypass this limitation, we make use of the fact that vmd_decode allows frames to store an offset in s->x_off, which is then subtracted from consecutive frame offsets. Thus, the vulnerability is exploited using two frames:

First frame. The first frame specifies the sign-inverted desired offset in frame_x. The two other values specified by the frame frame_y and frame_width are set such that the vulnerable write is not triggered. The value of frame_x is then stored in s->x_off on line 16 and subtracted from frame_x on line 19. This implies that frame_x is 0 after execution of line 19, which means that the following block is not executed. The desired offset is now stored in s->x_off.

Second frame. The second frame now specifies frame_x and frame_y to be 0, such that line 16 and 17 are *not* executed. On line 19, the value of s->x_off stored by the previous frame is now subtracted from frame_x, thereby resulting in an underflow. frame_x is now negative as desired. The copy operation on line 43 then copies an amount of attacker-supplied data specified by frame_width to a location before the buffer. We can thus overwrite large portions of the heap before the buffer and in particular the release_buffer pointer.

Redirection of execution. We then make use of the fact that the target frontend MPlayer was not compiled as a position-independent executable on the target platform. Therefore, the image base of the MPlayer executable is located at a predictable address, despite address space layout randomization. This means that the flow of execution can be reliably redirected to any sequence of instructions of the MPlayer code. A suitable sequence of instructions can be identified in the MPlayer executable, which when redirected to, allows the execution of arbitrary attacker-supplied shell-commands. The following sequence is well suited for this task:

```
0x080cc5c2 : mov    %eax,(%esp)
0x080cc5c5 : call   0x809481c <system@plt>
```

As seen on line 29, release_buffer receives avctx as its first argument. To accomplish this, avctx was first moved into the register %eax and then pushed onto the stack, therefore, as the sequence of instructions is invoked, the shell commands saved at avctx are executed. Since we are able to overwrite avctx->release_buffer, it is also possible to overwrite avctx. To exploit the issue, we therefore overwrite the avctx structure such that it begins with the shell commands to be executed and replaces avctx->release_buffer with the address of the instruction sequence presented. Note that avctx and avctx->release_buffer are 260 bytes apart, leaving enough room for shell commands.

It is noteworthy that to further stabilize this exploit, the attacker must gain a more fine grained control over the heap state. However, this simple exploit still proofs that the vulnerability can indeed be exploited on contemporary Linux systems.