Generalized Vulnerability Extrapolation using Abstract Syntax Trees

Fabian Yamaguchi
University of Göttingen
Göttingen, Germany

Markus Lottmann
Technische Universität Berlin
Berlin, Germany

Konrad Rieck
University of Göttingen
Göttingen, Germany

ABSTRACT

The discovery of vulnerabilities in source code is a key for securing computer systems. While specific types of security flaws can be identified automatically, in the general case the process of finding vulnerabilities cannot be automated and vulnerabilities are mainly discovered by manual analysis. In this paper, we propose a method for assisting a security analyst during auditing of source code. Our method proceeds by extracting abstract syntax trees from the code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of these patterns. This representation enables us to decompose a known vulnerability and extrapolate it to a code base, such that functions potentially suffering from the same flaw can be suggested to the analyst. We evaluate our method on the source code of four popular open-source projects: LibTIFF, FFmpeg, Pidgin and Asterisk. For three of these projects, we are able to identify zero-day vulnerabilities by inspecting only a small fraction of the code bases.

1. INTRODUCTION

The security of computer systems critically depends on the quality of its underlying code. Even minor flaws in a code base can severely undermine the security of a computer system and make it an easy victim for attackers. There exist several examples of vulnerabilities that have led to security incidents and the proliferation of malicious code in the past [e.g. 21, 26]. A drastic case is the malware Stuxnet [7] that featured code for exploiting four unknown vulnerabilities in the Windows operating system, rendering conventional defense techniques ineffective in practice.

The discovery of vulnerabilities in source code is a central issue of computer security. Unfortunately, the process of finding vulnerabilities cannot be automated in the general case. According to Rice's theorem a computer program is unable to generally decide whether another program contains vulnerable code [10]. Consequently, security research has focused on devising methods for identifying specific types of vulnerabilities.

Several approaches have been proposed that statically identify patterns of specific vulnerabilities [e.g., 4, 18, 28, 32], such as the use of certain insecure functions. Moreover, concepts from the area of software verification have been successfully adapted for tracking vulnerabilities, for example, in form of fuzz testing [27], taint analysis [22] and symbolic execution [1, 8]. Many of these approaches, however, are limited to specific conditions and types of vulnerabilities. The discovery of vulnerabilities in practice still mainly rests on tedious manual auditing that requires considerable time and expertise.

In this paper, we propose a method for assisting a security analyst during auditing of source code. Instead of striving for an automated solution, we aim at rendering manual auditing more effective by guiding the search for vulnerabilities. Based on the idea of *vulnerability extrapolation* [33], our method proceeds by extracting abstract syntax trees from the source code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of the extracted patterns. The patterns contain subtrees with nodes corresponding to types, functions and syntactical constructs of the code base. This representation enables our method to decompose a known vulnerability and to suggest code with similar properties—potentially suffering from the same flaw—to the analyst for auditing.

We evaluate the efficacy of our method using the source code of four popular open-source projects: LibTIFF, FFmpeg, Pidgin and Asterisk. We first demonstrate in an quantitative evaluation how functions are decomposed into structural patterns and how similar code can be identified automatically. In a controlled experiment we are able to narrow the search for a given vulnerability to 8.7% of the code base and consistently outperform non-structured approaches for vulnerability extrapolation. We also study the discovery of real vulnerabilities in a qualitative evaluation, where we are able to discover 10 zero-day vulnerabilities in the source code of the four open-source projects.

In summary, we make the following contributions:

- *Generalized vulnerability extrapolation:* We present a general approach to the extrapolation of vulnerabilities, allowing both the content and structure of code to be considered for finding similar flaws in a code base.
- *Structural comparison of code:* We present a method for robust extraction and analysis of abstract syntax trees that allows for automatic comparison of code with respect to structural patterns.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

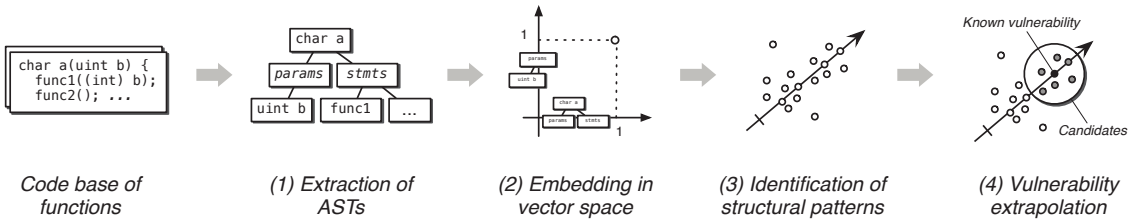


Figure 1: Schematic overview of our method for vulnerability extrapolation.

- *Evaluation and cases studies:* We study the capabilities of our method in different empirical experiments, where we identify real zero-day vulnerabilities in popular open-source projects.

The rest of this paper is structured as follows: our method for vulnerability discovery is introduced in Section 2 and its evaluation is presented in Section 3. Limitations and related work are discussed in Section 4 and 5, respectively. Section 6 concludes this paper.

2. VULNERABILITY EXTRAPOLATION

The concept of vulnerability extrapolation builds on the observation that source code often contains several vulnerabilities linked to the same flawed programming patterns, such as missing checks before or after function calls. Given a known vulnerability, it is thus often possible to discover previously unknown vulnerabilities by finding functions sharing similar code structure.

In practice, such extrapolation of vulnerabilities is attractive for two reasons: First, it is a general approach that is not limited to any specific vulnerability type. Second, the extrapolation does not hinge on any involved analysis machinery: a robust parser and an initial vulnerability are sufficient for starting an analysis. However, assessing the similarity of code is a challenging task, as it requires analyzing and comparing structured objects, such as subtrees of syntax trees. Previous work has thus only considered flat representations, such as function and type names, for extrapolating vulnerabilities [see 33].

To tackle the challenge of structured data, our method combines concepts from static analysis, robust parsing and machine learning. It proceeds in four steps that are illustrated in Figure 1 and described in the following:

1. *Extraction of abstract syntax trees.* In the first step, abstract syntax trees (AST) are extracted for all functions of the code base using a robust parser. This parser is based on the concept of island grammars [20] and capable of extracting syntax trees from C/C++ source code even without a working build environment (Section 2.1).

2. *Embedding in a vector space.* The abstract syntax trees of the functions are then embedded in a vector space, such that techniques from machine learning can be applied to analyze the code. The embedding is accomplished by disregarding irrelevant nodes in the trees and representing each function as a vector of contained subtrees (Section 2.2).

3. *Identification of structural patterns.* Based on the vectorial representation, structural patterns are identified in the code using the technique of latent semantic analysis [5]. This analysis technique determines dominant directions (structural patterns) in the vector space corresponding to combinations of AST subtrees frequently occurring in the code base (Section 2.3).

4. *Vulnerability Extrapolation.* Finally, the functions of the code base are described as mixtures of the identified structural patterns. This representation enables identifying code similar to a known vulnerability by finding functions with a similar mixture of structural patterns (Section 2.4).

In the following, these four steps are described in detail and the required theoretical and technical background is presented where necessary.

2.1 Robust AST Extraction

Determining arbitrary structural patterns in code demands a fine grained representation of code, similar in precision to the abstract syntax trees (AST) generated by compilers. Obtaining these trees directly from a compiler is only possible if a working build environment is available. Unfortunately, constructing such an environment is often non-trivial in practice, as all dependencies of the code including correct versions of build tools and header files need to be available. Finally, when analyzing legacy code, parts of the code may simply not be available anymore.

As a remedy, we employ a robust parser for C/C++ based on the concept of *island grammars* [20]. This parser allows for extracting ASTs from individual source files. In contrast to parsers integrated with a compiler, this parser does not aim to validate the syntax of the code it processes. Instead, the objective is to extract as much information from the code as possible, assuming that it is syntactically valid in the first place.

Our parser is based on a single grammar definition for the ANTLR parser generator [23] and publicly available[1]. The parser outputs ASTs in a serialized text format as shown in Figure 2. This serialized format is well suited for subsequent processing and provides generic access to the structure of the parsed code. A graphical version of the corresponding tree is presented in Figure 3.

For our analysis we distinguish between different types of nodes in the syntax tree. We refer to all nodes associated with parameter types, declaration types and function calls as *API nodes* (dashed in Figure 3), as they define how the code interfaces with other functions and libraries. Moreover, we denote all nodes describing syntactical elements as *syntax nodes* (dotted in Figure 3).

[1]http://codeexploration.blogspot.de/

```
1 int foo(int y)
2 {
3     int n = bar(y);
4
5     if (n == 0)
6         return 1;
7
8     return (n + y);
9 }
```

(a) Exemplary C function

```
# type  depth  value1   value2
func    0       int     foo
params  1
param   2       int     y
stmts   1
decl    2       int     n
op      2       =
call    3       bar
arg     4       y
if      2       (n == 0)
cond    3       n == 0
op      4       ==
stmts   3
return  4       1
return  2       (n + y)
op      3       +
```

(b) Serialized AST

Figure 2: Example of a C function and a serialized AST

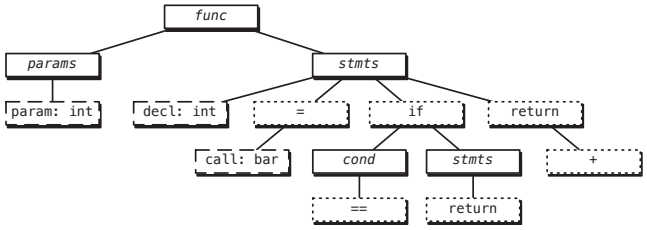


Figure 3: Abstract syntax tree with API nodes (dashed) and syntax nodes (dotted).

2.2 Embedding of ASTs in a Vector Space

Abstract syntax trees offer a rich source of information for extraction of code patterns. However, machine learning techniques cannot be applied directly to this type of data, as they usually operate on numerical vectors.

To address this problem, a suitable map is required, allowing ASTs to be transformed into vectors. This map needs to capture the structure and content of the trees, and thus is crucial for the success of vulnerability extrapolation. To construct this map, we describe the AST of each functions in our code base using a set of subtrees $\mathcal{S}$. In particular, we experiment with the following three definitions of the set:

1. *API nodes.* We consider only flat function and type names [see 33]. The set $\mathcal{S}$ simply consists of all individual API nodes found in the ASTs of the code base. All other nodes are ignored.

2. *API subtrees.* The set $\mathcal{S}$ is defined as all subtrees of depth D in the code base that contain at least one API node. The subtrees are generalized by replacing all non-API nodes with placeholders (empty nodes).

3. *API/S subtrees.* The set $\mathcal{S}$ consists of all subtrees of depth D containing at least one API or syntax node. Again, all non-API and non-syntax nodes are replaced by placeholders (empty nodes).

Depending on this definition of $\mathcal{S}$, we obtain different views on the functions of the code base. If we consider API nodes only, our characterization is shallow and we only capture the interfacing of the functions. If we choose API subtrees as the set $\mathcal{S}$, we describe the functions in terms of API nodes and the structural context these nodes occur in. Finally, if we consider API/S subtrees, we obtain a view on our code base that reflects API usage as well as the occurrences of syntactical elements in the functions. In the following we fix the depth of subtrees to $D = 3$ in all experiments, as this setting provides a good balance between a shallow representation and overly complex subtrees.

Based on the set $\mathcal{S}$ we can define a map ϕ that embeds an AST x in a vector space, where each dimension is associated with one element of $\mathcal{S}$. Formally, this map is given by

$$\phi : \mathcal{X} \longmapsto \mathbb{R}^{|\mathcal{S}|}, \quad \phi(x) \longrightarrow \big(\#(s,x) \cdot w_s\big)_{s \in \mathcal{S}}$$

where $\mathcal{X}$ refers to all ASTs of functions in our code base and $\#(s,x)$ returns the number of occurrences of the subtree $s \in \mathcal{S}$ in x. For convenience and later processing, we store the vectors of all ASTs in our code base in a matrix M, where one cell of the matrix is defined as $M_{s,x} = \#(s,x) \cdot w_s$.

The term w_s in the map ϕ corresponds to a TF-IDF weighting. This weighting ensures that subtrees occurring very frequently in the code base have little effect when assessing function similarity. Furthermore, it removes the bias towards longer functions, which can contain similar subtrees to a lot of different functions but are not particularly similar to any of them. A detailed description of this weighing scheme is given by Salton and McGill [25].

Let us, as an example, consider the AST given in Figure 3 and the set $\mathcal{S}$ of API nodes. The tree x contains only three API nodes, namely `param: int`, `decl: int` and `call: bar`. As a result, the corresponding three dimensions in the vector $\phi(x)$ are non-zero, whereas all other dimensions are zero. The vector space constructed by the map ϕ may contain hundred thousands of dimensions, yet the vectors are extremely sparse. This sparsity can be exploited for efficiently storing and comparing the vectors in practice.

2.3 Identification of Structural Patterns

By calculating distances between vectors, the representation obtained in the previous step already allows functions to be compared in terms of the subtrees they share. However, we cannot yet compare functions with respect to more involved patterns. For example, the code base of a server application may contain functions related to network communication, message parsing and thread scheduling. In this setting, it would be better to compare the functions with respect to these functionalities rather than looking at the plain subtrees of the ASTs.

Fortunately, we can adapt the technique of *latent semantic analysis* [5] to solve this problem. Latent semantic analysis is a classic technique of natural language processing that is used for identifying topics in text documents. Each topic is represented by a vector of related words. In our setting these topics correspond to types of functionality in the code base and the respective vectors are associated with subtrees related to these functionalities.

Latent semantic analysis identifies topics by determining dominant directions in the vector space, that is, subtrees frequently occurring together in ASTs of our code base. We refer to these directions of related subtrees as *structural patterns*. By projecting the original vectors on the identified directions, one obtains a low-dimensional representation of the data. Each AST of a function is described as a mixture of the structural patterns. For example, a function related to communication and parsing is represented as a mixture of patterns corresponding to these types of functionality.

Formally, latent semantic analysis seeks d orthogonal directions in the vector space that capture as much of the variance inside the data as possible. One technical way to obtain these d directions is by performing a singular value decomposition (SVD) of the matrix M. That is, M is decomposed into three matrices U, Σ and V as follows

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

The decomposition provides a wealth of information and contains the projected vectors as well as the structural patterns identified in the matrix M.

1. The d columns of the unitary matrix U correspond to the dominant directions in the vector space and define the d structural patterns of subtrees identified in the code base.

2. The diagonal matrix Σ contains the singular values of M. The values indicate the variances of the directions and allow us to assess the importance of the d structural patterns.

3. The rows of V contain the projected representations of the embedded ASTs, where each AST is described by a mixture of the d structural patterns contained in the matrix U.

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

2.4 Extrapolation of Vulnerabilities

Once the decomposition has been calculated, which takes a fraction of the time required for code parsing, an analyst is able to access the information encoded in the three matrices. In particular, the following three activities can be performed instantaneously to assist code auditing.

- *Vulnerability extrapolation.* The rows of the matrix V describe all functions as mixtures of structural patterns. Finding structurally similar functions is thus as simple as comparing the rows of V using a suitable measure, such as the cosine distance [25]. This operation forms the basis for the extrapolation of vulnerabilities. Clearly, there is no guarantee that functions with similar structure are plagued by the same vulnerabilities, however, examples presented in Section 3 provide some evidence for this correspondence.

- *Code base decomposition.* At the beginning of an audit, little is known about the overall structure of the code base. In this setting, the matrix U storing the most prevalent structural patterns in its columns gives important insight into the structure of the code base. This information can be used to uncover major clusters of similar functions, such as sets of functions employing similar programming patterns. This allows an analyst to select interesting parts of the code early in the audit and concentrate on promising functions first.

- *Detection of unusual functions.* Finally, the representation of functions in terms of structural patterns is fully transparent, allowing an analyst to discover the most prominent patterns used in any particular function by comparing rows of V with columns of U. This enables determining deviations from common programming patterns by analyzing differences in the representations of functions. For example, it might be interesting to audit a function related to message parsing that deviates from other such functions by also sharing structural patterns with network communication.

3. EVALUATION

We proceed to evaluate our method with real source code. In particular, we are interested in studying the ability of our method to assess the similarity of code and to identify potentially vulnerable functions in practice. We first conduct a *quantitative evaluation*, where we apply our method in a controlled experiment on different code bases. We then present a *qualitative evaluation* and examine the extrapolation of real vulnerabilities in two case studies.

For the evaluation we consider four popular open-source projects, namely LibTIFF, FFmpeg, Pidgin and Asterisk. For each of these projects we pick one known vulnerability as a starting point for the vulnerability extrapolation and proceed to manually label *candidate functions* which should be reviewed for the same type of vulnerability.

In the following, we describe the code bases of these projects and the choice of candidate functions in detail:

1. LibTIFF (`http://www.libtiff.org`) is a library for processing images in the TIFF format. Its source code covers 1,292 functions and 52,650 lines of code. Version 3.8.1 of the library contains a stack-based buffer overflow in the parsing of TLV elements that allows an attacker to execute arbitrary code using specifically crafted images (CVE-2006-3459). Candidate functions are all parsers for TLV elements.

2. Pidgin (`http://www.pidgin.im`) is a client for instant messaging implementing several communication protocols. The implementation contains 11,505 functions and 272,866 lines of code. Version 2.10.0 of the client contains a vulnerability in the implementation of the AIM protocol (CVE-2011-4601). An attacker is able to remotely crash the client using crafted messages. Candidate functions are all AIM protocol handlers converting incoming binary messages to strings.

3. FFmpeg (`http://www.ffmpeg.org`) is a library for conversion of audio and video streams. Its code base spans 6,941 functions with a total of 298,723 lines of code. A

	API nodes			API subtrees			API/S subtrees		
	75%	90%	100%	75%	90%	100%	75%	90%	100%
Pidgin	0.1	0.36	2.00	0.35	0.22	0.98	0.22	0.67	25.98
LibTIFF	6.35	6.97	7.58	5.65	6.66	7.27	6.49	9.36	17.32
FFmpeg	6.17	8.10	19.61	5.00	8.66	11.09	7.71	15.21	28.35
Asterisk	0.06	10.64	15.29	0.24	10.23	15.54	1.19	16.50	28.45
Average	3.17	6.52	11.12	2.81	6.44	8.72	3.90	10.44	25.03

Table 1: Performance of vulnerability extrapolation in a controlled experiment. The performance is given as amount of code (%) to be audited to find 75%, 90% and 100% of the potentially vulnerable functions.

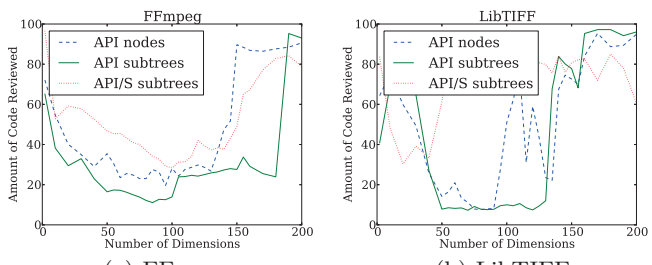
vulnerability has been identified in version 0.6 (CVE-2010-3429). During the decoding of video frames, indices are incorrectly computed, enabling the execution of arbitrary code. Candidate functions are all video decoding routines, which write decoded video frames to a pixel buffer.

4. Asterisk (`http://www.asterisk.org`) is a framework for Voice-over-IP communication. The code base covers 8,155 functions and 283,883 lines of code. Version 1.6.1.0 of the framework contains a vulnerability (CVE-2011-2529), which allows a remote attacker to corrupt memory of the server and to cause a denial of service via a crafted packet. Candidate functions are all functions reading incoming packets from UDP/TCP sockets.

3.1 Quantitative Evaluation

In our first experiment, we study the ability of our method to identify functions sharing similarities with a known vulnerability on the four code bases. To conduct a controlled experiment we thoroughly inspect each code base and manually label all candidate functions, that is, all functions that potentially contain the same vulnerability. Note that this manual analysis process required several weeks of work and can hardly be seen as an alternative to the concept of vulnerability extrapolation.

For each of the four code bases, we apply our method and rank the functions according to the selected target vulnerabilities. We vary the embedding of syntax trees by considering flat API nodes, API subtrees and API/S subtrees (see Section 2.2). Moreover, we compute the ranking for different numbers of structural patterns identified by latent semantic analysis (see Section 2.3). As performance measure we assess the efficacy of the vulnerability extrapolation by measuring the amount of code that needs to be inspected for finding all candidate functions.



(a) FFmpeg (b) LibTIFF

Figure 4: Performance of vulnerability extrapolation in a controlled experiment.

Figure 4 shows the results of this experiment for FFmpeg and LibTIFF, where results for the other two code bases are similar. The API subtrees clearly outperform the other representations of the code and enable narrowing the set of functions to be inspected to 8.7% on average. By contrast, the flat representation of API nodes requires 11.1% of the functions to be reviewed, while for the API/S subtrees even every 4th function (25%) needs to be inspected. Furthermore, Figure 4 also shows that the number of extracted structural patterns is not a critical parameter for vulnerability extrapolation. Our method performs well on all code bases when this number is between 50 to 100 dimensions, despite the fact that FFmpeg contains 6,941 and LibTIFF only 1,292 functions. In the following case studies, we fix this parameter to 70.

Table 1 presents a fine-grained analysis of the performance for each code base, where the amount of code that needs to be audited for revealing 75%, 90% and 100% of the candidate functions is shown. All numbers are expressed in percent of the code base to account for their different sizes. The API subtrees perform best, where 75% of the candidate functions are discovered by reading under 3% of the code bases. In the case of Pidgin and Asterisk, this number further reduces to less than 1% of the entire code base providing a significant advantage over manual auditing.

Nevertheless, the results also show room for improvement, particularly when all candidate functions need to be discovered. In this case, the amount of code to be read reaches 8.7% on average and up to 16% in the worst case. However, even in the worst case the amount of code that needs to be inspected is reduced by 84% and vulnerability extrapolation clearly accelerates manual code auditing in practice.

3.2 Qualitative Evaluation

In a case study with FFmpeg and Pidgin, we now demonstrate the practical merit of vulnerability extrapolation and show how our method plays the key role in identifying eight zero-day vulnerabilities. We have conducted two further studies with Pidgin and Asterisk uncovering two more zero-day vulnerabilities. For the sake of brevity however, we omit these case studies and details of the vulnerabilities here.

3.2.1 *Case study: FFmpeg.*

Flaws in the indexing of arrays are a frequently occurring problem in media libraries. In many cases these vulnerabilities allow attackers to write data to arbitrary locations in memory, an exploit primitive that can often be leveraged for arbitrary code execution. In this case study, we show how a publicly known vulnerability in the video decoder for FLIC media files of FFmpeg (CVE-2010-3429) is used to uncover three further vulnerabilities of this type, two of which were previously unknown. Note that this is the same vulnerabil-

```
1  static int flic_decode_frame_8BPP(AVCodecContext *avctx,
2                                    void *data, int *data_size,
3                                    const uint8_t *buf,
4                                    int buf_size)
5  { [...]
6    signed short line_packets; int y_ptr;
7    [...]
8    pixels = s->frame.data[0];
9    pixel_limit = s->avctx->height * s->frame.linesize[0];
10   frame_size = AV_RL32(&buf[stream_ptr]); [...]
11   frame_size -= 16;
12   /* iterate through the chunks */
13   while ((frame_size > 0) && (num_chunks > 0)) { [...]
14     chunk_type = AV_RL16(&buf[stream_ptr]);
15     stream_ptr += 2;
16     switch (chunk_type) { [...]
17     case FLI_DELTA:
18         y_ptr = 0;
19         compressed_lines = AV_RL16(&buf[stream_ptr]);
20         stream_ptr += 2;
21         while (compressed_lines > 0) {
22             line_packets = AV_RL16(&buf[stream_ptr]);
23             stream_ptr += 2;
24             if ((line_packets & 0xC000) == 0xC000) {
25                 // line skip opcode
26                 line_packets = -line_packets;
27                 y_ptr += line_packets * s->frame.linesize[0];
28             } else if ((line_packets & 0xC000) == 0x4000) {
29                 [...]
30             } else if ((line_packets & 0xC000) == 0x8000) {
31                 // "last byte" opcode
32                 pixels[y_ptr + s->frame.linesize[0] - 1] =
33                     line_packets & 0xff;
34             } else { [...]
35                 y_ptr += s->frame.linesize[0];
36             }
37         }
38         break; [...]
39     }
40   [...]
41   } [...]
42   return buf_size;
43 }
```

```
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) {
      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: Original vulnerability in `flic_decode_frame_8BPP` (left) and zero-day vulnerability found in `vmd_decode` (right)

ity extrapolated in our previous work [33], thus allowing the advantages of our improved method to be demonstrated in direct comparison to a non-structured approach.

Original vulnerability. Video decoding in general involves processing image data in form of video frames. These frames contain the encoded pixels as well as meta information of the image, such as width and offset values. Decoders usually proceed by allocating an array and then using the offsets provided in the video frame to populate the array with the image data. In this context, it must be carefully verified that offsets specified by the frame refer to locations within the array. In the case of the decoding routine `flic_decode_frame_8BPP` shown in Figure 5, no verification of this kind is performed allowing an attacker to reference locations outside the pixel array.

The critical write is performed on line 32, where the least significant byte of the user-supplied integer `line_packets` is written to a location relative to the 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. Due to the loop starting at line 21, 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 buffer.

Extrapolation. We proceed by using our method to generate the ranking shown in Table 2. This ranking contains the 30 most similar functions to the vulnerable function `flic_decode_frame_8BPP` selected from a total of 6,941 functions in the FFmpeg code base. Candidate functions for the vulnerabilities are depicted with light shading, discovered vulnerabilities are indicated by dark shading.

First, we note that 20 out of 30 functions are candidate functions, i.e., they are decoders performing a write operation to a pixel buffer. Furthermore, no irrelevant functions are part of the first 13 results and we can spot four vulnerable functions in the ranking corresponding to the following three vulnerabilities:

1. The function `flic_decode_frame_15_16BPP` is located in the same file as the original vulnerability and likewise processes FLIC video frames. The function has been found by FFmpeg developers to contain a similar vulnerability, which was patched along with the original flaw. Our method returns a similarity of 98%.

2. The function `vmd_decode` depicted in Figure 5 contains the vulnerability discovered in [33]. The function proceeds by employing the same API functions used in the original vulnerability to read frame meta data on line 7 to 10 and then uses these values to calculate an incorrect index into the pixel buffer on line 28. Our method returns a similarity of 89%, leading us almost instantly to this vulnerability.

```
1  static void vqa_decode_chunk(VqaContext *s)
2  {
3    [...]
4    int lobytes = 0;
5    int hibytes = s->decode_buffer_size / 2; [...]
6    for (y = 0; y < s->frame.linesize[0] * s->height;
7      y += s->frame.linesize[0] * s->vector_height){
8      for (x = y; x < y + s->width; x += 4, lobytes++, hibytes++)
9        {
10       pixel_ptr = x;
11       /* get the vector index, the method for
12          which varies according to
13        * VQA file version */
14       switch (s->vqa_version) {
15       case 1: [...]
16       case 2:
17       lobyte = s->decode_buffer[lobytes];
18       hibyte = s->decode_buffer[hibytes];
19       vector_index = (hibyte << 8) | lobyte;
20       vector_index <<= index_shift;
21       lines = s->vector_height;
22       break;
23       case 3: [...]
24        }
25       while (lines--) {
26         s->frame.data[0][pixel_ptr + 0] =
27           s->codebook[vector_index++];
28         s->frame.data[0][pixel_ptr + 1] =
29           s->codebook[vector_index++];
30         s->frame.data[0][pixel_ptr + 2] =
31           s->codebook[vector_index++];
32         s->frame.data[0][pixel_ptr + 3] =
33           s->codebook[vector_index++];
34         pixel_ptr += s->frame.linesize[0];
35         }
36     }
37   }
38 }
39
40 static av_cold int vqa_decode_init(AVCodecContext *avctx)
41 {
42     VqaContext *s = avctx->priv_data;
43     unsigned char *vqa_header;
44     int i, j, codebook_index;
45     s->avctx = avctx;
46     avctx->pix_fmt = PIX_FMT_PAL8; [...]
47       /* load up the VQA parameters from the header */
48     vqa_header = (unsigned char *)s->avctx->extradata;
49     s->vqa_version = vqa_header[0];
50      s->width = AV_RL16(&vqa_header[6]);
51     s->height = AV_RL16(&vqa_header[8]); [...]
52     /* allocate decode buffer */
53     s->decode_buffer_size = (s->width / s->vector_width) *
54         (s->height / s->vector_height) * 2;
55     s->decode_buffer = av_malloc(s->decode_buffer_size);
56     s->frame.data[0] = NULL;
57     return 0;
58 }
```

Figure 6: The second zero-day vulnerability found by extrapolation of CVE-2010-3429 in `vqa_decode_chunk`.

3. The function `vqa_decode_chunk` contains another previously unknown vulnerability shown in Figure 6. In this case, however, reading of frame meta data using the characteristic API is performed by a different function, `vqa_decode_init`, on line 21 and 22. Despite the missing API information, our method is able to uncover the decoder based on its characteristic code structure. In particular, the two nested loops iterating over the pixel buffer on line 6 and 8 are common to most decoders.

In summary, this example shows that our method is useful to identify zero-day vulnerabilities in real-world code. Furthermore, by combining information about symbol usage and code structure, our method gains increased robustness over approaches resting solely on symbol information.

Sim.	Function name	Sim.	Function name
0.98	flic_decode_frame_15_16BPP	0.87	wmavoice_decode_init
0.92	decode_frame	0.85	decode_frame
0.92	decode_frame	0.84	smc_decode_stream
0.91	flac_decode_frame	0.84	rl2_decode_init
0.90	decode_format80	0.84	xvid_encode_init
0.89	decode_frame	0.84	vmdvideo_decode_init
0.89	tgv_decode_frame	0.83	mjpega_dump_header
0.89	vmd_decode	0.82	ff_flac_is_._valid
0.89	wavpack_decode_frame	0.82	decode_init
0.88	adpcm_decode_frame	0.82	ws_snd_decode_frame
0.88	decode_frame	0.81	bmp_decode_frame
0.88	aasc_decode_frame	0.81	sbr_make_f_master
0.88	vqa_decode_chunk	0.80	ff_h264_decode_ref_pic_.
0.87	cmv_process_header	0.80	decode_frame
0.87	msrle_decode_8_16_24_32	0.79	vqa_decode_init

Table 2: Top 30 most similar functions to a known vulnerability in FFmpeg.

3.2.2 *Case study: Pidgin*

To demonstrate the generality of our method, we perform a second extrapolation on a different code base and for a different vulnerability type. In this case study, the set of relevant functions is very small in comparison to the size of the code base, that is, only 67 out of 11,505 functions are candidate functions. Despite this increased difficulty, we are able to identify nine vulnerabilities similar to a known vulnerability among the first 30 hits, six of which were previously unknown.

Original Vulnerability. The function `receiveauthgrant` shown in Figure 7 does not perform sufficient validation of UTF-8 strings received from the network allowing attackers to cause a denial of service condition and possibly execute arbitrary code. The function reads the username and message from an incoming binary data packet on line 15 and 20 respectively. It then passes these values to a suitable packet handler on line 27. In general, packet handlers assume that strings passed in parameters are valid UTF8-strings, however, the function does not ensure that this is the case.

Extrapolation. We again apply our method to obtain the 30 most similar functions to the original vulnerability as shown in Table 3. We first note that 28 of the first 30 hits are candidates selected from a total of 11,505 functions. After a short inspection of the suggested functions, we are able to spot the same type of vulnerability as in the original function in nine of the candidates. As an example, consider function `parseicon` shown in Figure 7. The username is read from the binary data packet on line 16 and is passed to a handler function unchecked on line 25–similar to the vulnerability in `receiveauthgrant`. It has been verified that this again allows to cause a denial of service condition.

The second case study demonstrates that our method works well even when the number of relevant functions is small compared to the total size of the code base. We are able to narrow down the search for potentially vulnerable

```
1  static int
2  receiveauthgrant(OscarData *od,
3                   FlapConnection *conn,
4                   aim_module_t *mod,
5                   FlapFrame *frame,
6                   aim_modsnac_t *snac,
7                   ByteStream *bs)
8  {
9    int ret = 0;
10   aim_rxcallback_t userfunc;
11   guint16 tmp;
12   char *bn, *msg;
13   /* Read buddy name */
14   if ((tmp = byte_stream_get8(bs)))
15     bn = byte_stream_getstr(bs, tmp);
16   else
17     bn = NULL;
18   /* Read message (null terminated) */
19   if ((tmp = byte_stream_get16(bs)))
20     msg = byte_stream_getstr(bs, tmp);
21   else
22     msg = NULL;
23   /* Unknown */
24   tmp = byte_stream_get16(bs);
25   if ((userfunc =
26        aim_callhandler(od, snac->family, snac->subtype)))
27     ret = userfunc(od, conn, frame, bn, msg);
28   g_free(bn);
29   g_free(msg);
30   return ret;
31 }
```

```
static int
parseicon(OscarData *od,
          FlapConnection *conn,
          aim_module_t *mod,
          FlapFrame *frame,
          aim_modsnac_t *snac,
          ByteStream *bs)
{

  int ret = 0;
  aim_rxcallback_t userfunc;
  char *bn;
  guint16 flags, iconlen;
  guint8 iconcsumtype, iconcsumlen, *iconcsum, *icon;

  bn = byte_stream_getstr(bs, byte_stream_get8(bs));
  flags = byte_stream_get16(bs);
  iconcsumtype = byte_stream_get8(bs);
  iconcsumlen = byte_stream_get8(bs);
  iconcsum = byte_stream_getraw(bs, iconcsumlen);
  iconlen = byte_stream_get16(bs);
  icon = byte_stream_getraw(bs, iconlen);
  if ((userfunc =
       aim_callhandler(od, snac->family, snac->subtype)))
    ret = userfunc(od, conn, frame, bn, iconcsumtype,
                   iconcsum, iconcsumlen, icon, iconlen);
  g_free(bn);
  g_free(iconcsum);
  g_free(icon);
  return ret;
}
```

Figure 7: Original vulnerability (CVE-2011-4601) in `receiveauthgrant` (left), zero-day vulnerability in `parseicon` (right).

Sim.	Function name	Sim.	Function name
1.00	receiveauthgrant	0.98	incomingim_ch4
1.00	receiveauthreply	0.98	parse_flap_ch4
1.00	parsepopup	0.98	infoupdate
1.00	parseicon	0.98	parserights
1.00	generror	0.98	incomingim
0.99	incoming_..._buddylist	0.98	parseadd
0.99	motd	0.97	userinfo
0.99	receiveadded	0.97	parsemod
0.99	mtn_receive	0.97	parsedata
0.99	msgack	0.97	rights
0.99	keyparse	0.97	rights
0.99	hostversions	0.97	uploadack
0.98	userlistchange	0.96	incomingim_ch2_sendfile
0.98	migrate	0.96	rights
0.98	error	0.96	parseinfo_create

Table 3: Top 30 most similar functions to a known vulnerability in Pidgin.

code to a handful of functions. Again, the extraploation enables us to identify previously unknown vulnerabilities by inspecting only a small fraction of the code base.

4. LIMITATIONS

The discovery of vulnerable code in software is a hard problem. Due to the fundamental inability of one program to completely analyze another program's code, a generic technique for finding arbitrary vulnerabilities does not exist [10]. As a consequence, all practical approaches either limit the search to specific types of vulnerabilities or, as in the case of vulnerability extrapolation, only identify *potentially* vulnerable code. In the following we discuss the limitations of our approach in more detail.

Our method builds on techniques from machine learning, such as the embedding in a vector space and latent semantic analysis. These techniques are effective in identifying potentially vulnerable code, yet they do not provide any guarantees whether the identified code truly contains a vulnerability. This limitation is inherent to the application of machine learning, which considers the statistics of the source code rather than the true semantics. Due to Rice's theorem, however, a generic discovery of vulnerabilities is impossible anyway and thus even the discovery of potential vulnerabilities is beneficial in practice.

A prerequisite for extrapolation is the existence of a starting vulnerability. In cases where no known vulnerability is available, our method cannot be applied. In practice such cases are rare. For large software projects, it is not the discovery of a single vulnerability that is challenging but making sure that similar flaws are not spread across the entire code base. Extrapolation addresses exactly this setting. Moreover, related techniques such as fuzz testing, taint analysis and symbolic execution can be easily coupled with vulnerability extrapolation and provide starting vulnerabilities automatically.

Finally, the discovery of our method is limited to vulnerabilities present in few functions of source code. Complex flaws that span several functions across a code base can be difficult to detect for our method. However, our case study with FFmpeg shows that vulnerabilities distributed over two functions can still be effectively identified, as long as both functions share some structural patterns with the original vulnerability.

5. RELATED WORK

The identification of vulnerabilities has been a vivid area of security research. Various contrasting concepts have been devised for finding and eliminating security flaws in source

code. Our method is related to several of these approaches, as we point out in this section.

5.1 Code Clone Detection

In the simplest case, functions containing similar vulnerabilities exist because code has been copied. The detection of such *copy-&-paste code clones* has been an ongoing research topic. In particular, Kontogiannis et al. [14] explore the use of numerical features, such as the number of called functions, to detect code clones, while Baxter et al. [2] suggest a more fine-grained method, which compares ASTs. Li et al. present *CP-Miner*, a tool for code clone detection based on frequent itemset mining [16]. They demonstrate the superiority of their approach to the well-known tool *CCFinder* developed by Kamiya et al. [13], a token-based detection method. Maletic et al. [19] propose a method for code clone detection, which compares functions in terms of comments and identifiers. Similarly, Jang et al. have introduced a method for finding unpatched copies of code using n-gram analysis [11]. A thorough evaluation of existing methods is provided by Bellon et al. [3].

Code clone detection shares some similarities with vulnerability extrapolation. However, corresponding methods address a fundamentally different problem and are specifically tailored to finding copied code. As result, they can only uncover vulnerabilities that have been introduced by duplication of code.

5.2 Static Code Analysis

The idea of vulnerability extrapolation hinges on the observation that patterns of API usage are often indicative for vulnerable code. This correspondence has been recognized for a long time and is reflected in several static analysis tools, such as *Flawfinder* [30], *RATS* [24] or *ITS4* [28]. These tools offer databases of API symbols commonly found in conjunction with vulnerabilities and allow a code base to be automatically scanned for their occurrences. The effectiveness of these tools critically depends on the quality and coverage of the databases. Vulnerabilities related to internal APIs or unknown patterns of API usage cannot be uncovered.

Engler et al. [6] are among the first to explore the link between vulnerabilities and programming patterns. Their method is capable of detecting vulnerabilities given a set of manually defined programming patterns. As an extension, Li and Zhou [15] present an approach for mining similar patterns automatically and detecting their violation in code. An inherent problem of this approach is that frequent programming mistakes will lead to the inference of valid patterns and thus common flaws cannot be detected. Williams et al. [31] and Livshits et al. [17] address this problem by incorporating software revision histories into the analysis. Our method is related to these approaches. However, we focus on the analysis of code structure for finding vulnerabilities, rather than modelling common programming templates.

5.3 Taint Analysis and Symbolic Execution

A more generic approach to vulnerability discovery builds on taint analysis, where vulnerabilities are identified by a source-sink system. If tainted data stemming from a source propagates to a sink without undergoing validation, a potential vulnerability is detected. The success of this approach has been demonstrated for several types of vulnerabilities, including SQL injection and cross-site scripting [12, 18] as well as integer-based vulnerabilities [29]. In most realizations, taint analysis is a dynamic process and thus limited to discovery of vulnerabilities observable during execution of a program.

The limitations of taint analysis have been addressed by several authors using symbolic execution [e.g., 1, 4, 8, 22]. Instead of passively monitoring the flow from the a source to a sink, these approaches try to actively explore different execution paths. Most notably is the work of Avgerinos et al. [1] that introduces a framework for finding and even exploiting security vulnerabilities. The power of symbolic execution, however, comes at a prize: the analysis often suffers from a vast space of possible execution paths. In practice, different assumptions and heuristics are necessary to trim down this space to a tractable number of branches. As a consequence, the application of symbolic execution for regular code auditing is still far from being practical [9].

5.4 Vulnerability Extrapolation

The concept of vulnerability extrapolation has been first introduced in [33]. In this work a method for vulnerability extrapolation is proposed that analyzes the usage of function and type names for finding vulnerabilities. However, neither the syntax nor the structure of the code are considered and thus the analysis is limited to flaws reflected in particular API symbols. Our method significantly extends this work by extracting and analyzing structural patterns in ASTs. As a result, we are able to discover vulnerabilities that are related to API usage as well as the structure of code. Moreover, we demonstrate the efficacy of our approach on a significantly larger set of code.

In comparison with related approaches, it is noteworthy that vulnerability extrapolation does not aim at identifying vulnerabilities automatically, but rendering manual auditing of source code more effective. The underlying rationale is that manual auditing—though time-consuming—is still superior to automatic methods and indispensable in practice. The assisted approach of vulnerability extrapolation here better fits the needs of security practitioners.

6. CONCLUSIONS

A key to strengthening the security of computer systems is the rigorous elimination of vulnerabilities in the underlying source code. To this end, we have introduced a method for accelerating the process of manual code auditing by suggesting potentially vulnerable functions to an analyst. Our method extrapolates known vulnerabilities using structural patterns of the code and enables efficiently finding similar flaws in large code bases. Empirically, we have demonstrated this capability by identifying real zero-day vulnerabilities in open-source projects, including Pidgin and FFmpeg.

The concept of vulnerability extrapolation is orthogonal to other approaches for finding vulnerabilities and can be directly applied to complement current instruments for code auditing. For example, if a novel vulnerability is identified using fuzz testing or symbolic execution, it can be extrapolated to the entire code base, such that similar flaws can be immediately patched. This extrapolation raises the bar for attackers, as they are required to continue searching for novel vulnerabilities, once an existing flaw has been sufficiently extrapolated and related holes have been closed in the source code.

Reporting of Vulnerabilities

The discovered vulnerabilities have been reported to the respective developers before submission of this paper. The flaws should be fixed in upcoming versions of the projects.

References

[1] 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.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 1998.

[3] S. Bellon, R. Koschke, I. C. Society, G. Antoniol, J. Krinke, I. C. Society, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33:577–591, 2007.

[4] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 269–278, 2006.

[5] 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.

[6] 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.

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

[8] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[9] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, 2011.

[10] J. Hopcroft and J. Motwani, R. Ullmann. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2 edition, 2001.

[11] J. Jang, A. Agrawal, , and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[12] 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.

[13] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002.

[14] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:108, 1996.

[15] 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.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.

[17] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proc. of 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] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proc. of International Conference on Automated Software Engineering (ASE)*, page 107, 2001.

[20] L. Moonen. Generating robust parsers using island grammars. In *Proc. of Working Conference on Reverse Engineering (WCRE)*, pages 13–22, 2001.

[21] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[22] 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.

[23] T. Parr and R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25: 789–810, 1995.

[24] rats. Rough auditing tool for security. Fortify Software Inc., `https://www.fortify.com/ssa-elements/threat-intelligence/rats.html`, visited April, 2012.

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

[26] C. Shannon and D. Moore. The spread of the Witty worm. *IEEE Security and Privacy*, 2(4):46–50, 2004.

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

[28] 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.

[29] 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.

[30] D. A. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder/`, visited April, 2012.

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

[32] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. of USENIX Security Symposium*, 2006.

[33] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2011.