Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery

Fabian Yamaguchi
University of Göttingen
Göttingen, Germany

Christian Wressnegger
idalab GmbH
Berlin, Germany

Hugo Gascon
University of Göttingen
Göttingen, Germany

Konrad Rieck
University of Göttingen
Göttingen, Germany

Abstract

Uncovering security vulnerabilities in software is a key for operating secure systems. Unfortunately, only some security flaws can be detected automatically and the vast majority of vulnerabilities is still identified by tedious auditing of source code. In this paper, we strive to improve this situation by accelerating the process of manual auditing. We introduce CHUCKY, a method to expose missing checks in source code. Many vulnerabilities result from insufficient input validation and thus omitted or false checks provide valuable clues for finding security flaws. Our method proceeds by statically tainting source code and identifying anomalous or missing conditions linked to security-critical objects. In an empirical evaluation with five popular open-source projects, CHUCKY is able to accurately identify artificial and real missing checks, which ultimately enables us to uncover 12 previously unknown vulnerabilities in two of the projects (Pidgin and LibTIFF).

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Vulnerabilities; Static Analysis; Anomaly Detection

1. INTRODUCTION

Detecting and eliminating vulnerabilities in software is a key for operating secure computer systems. The slightest flaw in the design or implementation of software can severely undermine its security and make it an easy victim for attackers. Several security incidents of the last years are actually the result of critical vulnerabilities in software, for example, the attacks conducted by Stuxnet [8], the drive-by download attacks exploiting Java flaws [24] and the recently discovered UPNP flaws in millions of home routers [21].

Finding vulnerabilities in software is a classic problem of computer security. Unfortunately, an automatic approach for finding arbitrary vulnerabilities cannot exist. According to Rice's theorem, checking whether a program contains vulnerable code using another program is undecidable in the general case [12].

As a result of this limitation, security research has focused on discovering specific types of vulnerabilities. For example, the usage of potentially dangerous functions, such as `strcpy` and `strcat`, can be easily detected by scanning for these functions [e.g., 3, 36, 39]. Similarly, analysis techniques such as fuzzing [e.g., 22, 33] and taint analysis [e.g., 23, 26] can help in spotting insecure flows of data in software. Sophisticated methods using symbolic execution can even discover flaws in unusual code branches [e.g., 1, 9, 37]. Many of these approaches, however, are hard to operate effectively in practice and opaque to a security analyst [see 11, 14]. Consequently, the vast majority of security flaws is still discovered by tedious and time-consuming manual analysis.

In this paper, we strive to improve this situation by accelerating the process of manual auditing. To this end, we introduce CHUCKY, a method for automatically identifying missing checks in source code. Many types of vulnerabilities result from insufficient validation of input and thus omitted checks provide valuable clues for finding security flaws. For example, if the length of data copied to a buffer is checked in 9 out of 10 functions in a program, it is evident that the function missing the check is a prime candidate for security auditing. To identify such functions, CHUCKY statically taints the source code and detects anomalous or missing conditions linked to security-critical objects, such as memory buffers. By comparing regular checks with missing ones, CHUCKY is able to suggest correct conditions and potential fixes to the analyst.

Our method operates independent of external information and additional annotations, as for example used by the static security checker *Splint* [7]. Instead we build on the assumption that missing or faulty checks are rare events and the majority of conditions imposed on security-critical objects in a software project are correct. While this assumption is satisfied for mature software projects, it does not hold true in all cases. We discuss limitations of our approach in Section 5 specifically.

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516665.

We demonstrate the efficacy of our approach in a qualitative and quantitative evaluation, where we analyze missing checks in the code of the following popular open-source projects: Firefox, Linux, LibPNG, LibTIFF and Pidgin. For all projects, CHUCKY is able to identify artificial and real missing checks accurately with few false positives. This accuracy ultimately enables us to identify 12 different previously unknown vulnerabilities in two of the projects, namely Pidgin and LibTIFF.

In summary, we make the following contributions:

- **Identification of missing checks.** We introduce a novel method for static analysis of source code that is able to automatically identify missing checks for vulnerability discovery.
- **Anomaly detection on conditions.** Our method embeds functions in a vector space, such that missing and unusual expressions in their conditions can be identified automatically.
- **Top-down and bottom-up analysis.** Taint analysis enables us to spot missing checks on untrusted input sources (top-down) as well as when accessing security-critical sinks (bottom-up).
- **Suggestion of corrections.** During auditing our method is able to suggest potential fixes to an analyst by highlighting differences between a missing check and regular ones.

The rest of this paper is structured as follows: we review missing-check vulnerabilities in Section 2 and introduce CHUCKY in Section 3 along with technical details. We evaluate its ability to expose missing checks on real source code in Section 4. Limitations and related work are discussed in Section 5 and 6, respectively. Section 7 concludes the paper.

2. MISSING-CHECK VULNERABILITIES

Many critical classes of vulnerabilities in software are a direct consequence of missing checks. This includes flaws in access control, such as missing checks of user permissions, as well as purely technical defects, such as buffer and integer overflows resulting from missing range checks. The consequences of these classes of vulnerabilities can be dramatic. For example, in January 2013 a vulnerability in the Java 7 runtime caused by a missing check in an access control component allowed attackers to install malware on millions of hosts (CVE-2013-0422). Thus, finding missing checks is crucial for securing software and computer systems.

Throughout this paper, we adopt the terminology established in the field of taint analysis for discussing missing checks [see 29]. In taint analysis, data entering a program via a *source* is monitored as it propagates to a *sink*, possibly undergoing validation in the process. Using this terminology we can discriminate two types of security checks in source code, both of which CHUCKY is designed to analyze.

- **Checks implementing security logic.** Programs implementing access control, such as Web applications or operating system kernels, perform security checks to restrict access to resources. Methods to detect specific types of these flaws have been proposed for Web applications [31] as well as Linux kernel code [34]. In this setting, parameters or global variables act as input sources, which need to be validated. As an example, consider the first check in Figure 1, where the parameter `user` is validated before allowing the remaining code of the function to be executed.

```
1  int foo(char *user, char *str, size_t n)
2  {
3    char buf[BUF_SIZE], *ar;
4    size_t len = strlen(str);
5
6    if(!is_privileged(user))
7      return ERROR;
8
9    if(len >= BUF_SIZE) return ERROR;
10   memcpy(buf, str, len);
11
12   ar = malloc(n);
13   if(!ar) return ERROR;
14
15   return process(ar, buf, len);
16 }
```

Figure 1: Exemplary security checks in a C function: a check implementing security logic (line 6) and two checks ensuring secure API usage (line 9 and 13).

- **Checks ensuring secure API usage.** Regardless of security logic, checks to ensure secure operation of internal and external APIs are required. As an example, consider the last two checks shown in Figure 1. To protect from buffer overflows, the first check validates the variable `len` derived from the source `strlen` before it is propagated to the sink `memcpy`. Moreover, the second check validates the return value of the source `malloc` to ensure that subsequent code does not cause a denial of service condition by dereferencing a NULL-pointer.

CHUCKY is based on the key observation that sources and sinks are usually employed many times within a code base, each time requiring similar security checks to be implemented. As a consequence, there often exist typical patterns for checking data retrieved from sources or propagated to sinks. If such patterns can be automatically inferred from the code, it is possible to detect deviations from these patterns and thereby spot potential vulnerabilities.

3. IDENTIFYING MISSING CHECKS

Exposing missing checks in source code poses two main challenges: First, typical checks must be determined automatically by leveraging information scattered throughout the code base. Second, any deviations from these checks need to be detected and presented to the analyst while browsing the code. Moreover, the analyst must be able to easily comprehend how the method arrives at its results.

To address these problems, our method implements a five-step procedure, which can be executed for each source and sink referenced by a selected function. This procedure combines techniques from static analysis and machine learning to determine missing checks and provide supporting evidence. The five steps are illustrated in Figure 2 and outlined in the following:

1. **Robust Parsing.** Conditions, assignments and API symbols are first extracted from a function's source code using a robust parser [20]. In particular, the

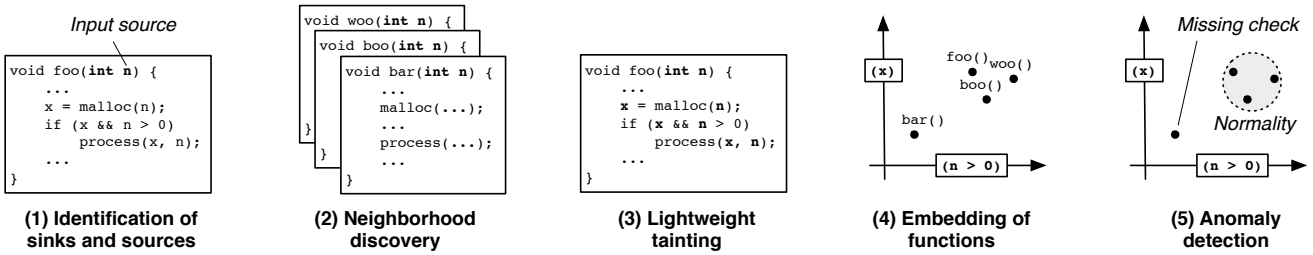


Figure 2: Overview of Chucky: (1) sources and sinks are identified, (2) functions with similar context are grouped, (3) variables depending on the sources/sinks are tainted, (4) functions are embedded in a vector space using tainted conditions, and (5) functions with anomalous or missing conditions are detected.

sources and sinks used in each of the functions are determined. The remaining four steps are then executed for each source or sink independently.

2. **Neighborhood discovery.** The necessity of a check highly depends on the context code operates in. Our method thus identifies functions in the code base operating in a similar context to that of the selected function using techniques inspired by natural language processing (Section 3.2).

3. **Lightweight tainting.** To determine only those checks associated with a given source or sink, lightweight tainting is performed for the function under examination and all its neighbors in top-down and bottom-up direction (Section 3.3).

4. **Embedding of functions.** The selected function and its neighbors are then embedded in a vector space using the tainted conditions such that they can be analyzed using machine learning techniques (Section 3.4).

5. **Anomaly Detection.** The embedding of functions enables us to geometrically search for missing checks. In particular, we compute a model of normality over the functions, such that anomalous checks can be identified by large distances from this model (Section 3.5).

In the following sections, we describe these steps in more detail and provide the necessary technical as well as theoretical background.

3.1 Robust Parsing

Reasoning about missing checks requires a deep understanding of program syntax. Chucky therefore begins by parsing code using a robust parser for C/C++ developed during our research. The parser is based on an island grammar [see 20] for the parser generator ANTLR [25] and provides abstract syntax trees (ASTs) for all functions of a code base even when declarations can only be resolved partially. This allows Chucky to be directly employed without requiring code to be compiled or a build environment to be configured. To encourage more research in the area, we have made our parser available as open-source software[1]. Chucky employs this parser to extract the following information from each function.

[1]https://github.com/fabsx00/joern

- **Sources and sinks.** All function parameters, function calls as well as global and local variables are potential sources or sinks of information, each of which may be tied to a unique set of conditions required for secure operation. Chucky therefore begins by extracting all sources and sinks from each function. The granularity of the analysis is further increased by taking fields of structures into account. As an example, consider Figure 3 where all sources and sinks of the function `foo` are marked.

- **API symbols**. Additionally, we extract API symbols as a prerequisite for neighborhood discovery (see Section 3.2). All types used in parameter and local variable declarations as well as the names of all functions called are considered as API symbols.

- **Assignments.** Assignments describe the flow of information between variables, which we exploit to determine conditions related to a sink by performing lightweight tainting (see Section 3.3). For each assignment, we store the subtree of the AST referring to the left- and right-value of the assignment.

- **Conditions.** Ultimately, functions are compared in terms of the conditions they impose when using a source or sink (see Sections 3.4 and 3.5). As conditions, we consider all expressions of control statements such as those introduced by the keywords `if`, `for` or `while` as well as those found in conditional expressions. To provide access to partial expressions, we store conditions as references to corresponding subtrees of the AST rather than as flat strings.

Upon completion of this step, patterns can be determined for each of the extracted sources and sinks and analyzed for missing checks, as explained in the following sections.

3.2 Neighborhood Discovery

Security checks are often highly context dependent. For example, omitting checks on string operations may be perfectly acceptable when parsing configuration files, while posing a serious threat when processing network data. Chucky accounts for this difference by only comparing the function under examination to functions sharing a similar context. This is achieved by identifying the *neighborhood* of the function, that is, related functions using similar API symbols. The rationale behind this choice is that the combination of interfaces used by a function is characteristic for the subsystem it operates in as well as the functionality it implements.

```
1  int foo(char *user, char *str, size_t n)
2  {
3    char buf[BUF_SIZE], *ar;
4    size_t len = strlen(str);
5
6    if(!is_privileged(user))
7      return ERROR;
8
9    if(len >= BUF_SIZE) return ERROR;
10   memcpy(buf, str, len);
11
12   ar = malloc(n);
13   if(!ar) return ERROR;
14
15   return process(ar, buf, len);
16 }
```

Figure 3: C function with sources and sinks: All parameters (line 1), global and local variables (lines 3,4 and 9) and function calls (line 4, 6, 10, 12 and 15) are marked for analysis.

For discovering these neighboring functions, we adapt the classic *bag-of-words model* from natural language processing that is commonly used to compare text documents [28]. Similar to the words in these documents, we represent each function in the code base by the API symbols it contains. We then map the functions to a vector space, whose dimensions are associated with the frequencies of these symbols [see 41]. Functions using a similar API lie close to each other in this vector space, whereas functions with different context are separated by large distances.

Formally, we define a mapping ϕ from the set of all functions $X = \{x_1, \ldots, x_m\}$ to $\mathbb{R}^{|A|}$ where A is the set of all API symbols contained in X. This mapping is given by

$$\phi : X \to \mathbb{R}^{|A|}, \quad \phi(x) \mapsto \big(I(x,a) \cdot \mathrm{TFIDF}(x,a,X)\big)_{a \in A}$$

where I is an indicator function defined as

$$I(x,a) = \begin{cases} 1 & \text{if } x \text{ contains API symbol } a \\ 0 & \text{otherwise} \end{cases}$$

and $\mathrm{TFIDF}(x,a,X)$ a standard weighting term from information retrieval [28]. The rationale behind the use of this term is to lower the impact of very frequently used API symbols on the similarity of functions.

This geometric representation enables us to identify the k-nearest neighbors $\mathcal{N} \subset X$ of the selected function x with respect to the source or sink s. We determine this set $\mathcal{N}$ by first extracting all functions of the code base that reference s. We then calculate the cosine distance of their corresponding vectors to $\phi(x)$ and finally select those k functions with the smallest distance.

Note, that Chucky is not very sensitive to the choice of the parameter k, as we demonstrate in Section 4, where values between 10 and 30 provide good performance. As a result of the neighborhood discovery, only the function under examination and its neighbors need to be processed in the following steps of the analysis.

3.3 Lightweight Tainting

Among the many checks present in a regular function, only a subset is relevant for a chosen source or sink. Analyzing the function in terms of its use of a specific source or sink therefore requires unrelated checks to be discarded automatically. However, this selection of relevant checks is not trivial, as it requires the flow of data between variables to be taken into account.

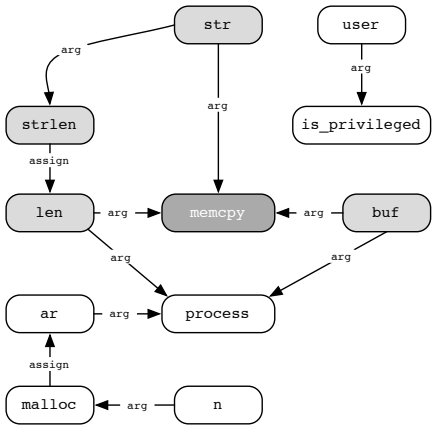


Figure 4: Dependency graph for function `foo`. Nodes reachable from `memcpy` are shaded, where the taint propagation stops at function boundaries. Isolated nodes are omitted.

To address this problem, Chucky performs lightweight tainting of the code of the target function and all its neighbors for each source or sink in two stages:

1. **Dependency modelling.** A directed graph is created to model the dependencies between variables. The nodes of the graph correspond to the identifiers used in the function (i.e., sources and sinks), while the edges reflect assignments between identifiers. Edges are also added if identifiers are parameters of functions.

2. **Taint propagation.** Starting from the identifier corresponding to a selected source or sink, the graph is traversed in top-down as well as bottom-up direction to discover all related identifiers. The propagation stops at function boundaries, that is, edges from parameters to functions are not followed.

An example of a dependency graph for the function `foo` from Figure 3 is shown in Figure 4. The identifier `memcpy` has been picked as source/sink for the analysis. Three identifiers are tainted (gray shading) including directly connected nodes, such as `len` and `buf`, as well as indirectly linked nodes, such as `strlen`.

Once the tainted identifiers for a function are known, we examine its conditions (see Section 3.1) and remove a condition if it does not reference at least one of the tainted identifiers. We thus restrict the conditions analyzed in subsequent steps only to those conditions related to the source or sink under examination.

3.4 Embedding of Functions

In the last two steps, we leverage the information distributed across the function neighborhood to determine typical checks and deviations thereof using anomaly detection. Similar to the technique for neighborhood discovery (Section 3.2), this method for anomaly detection operates on numerical vectors and thus the functions need to be again embedded in a suitable vector space. We implement this embedding as follows.

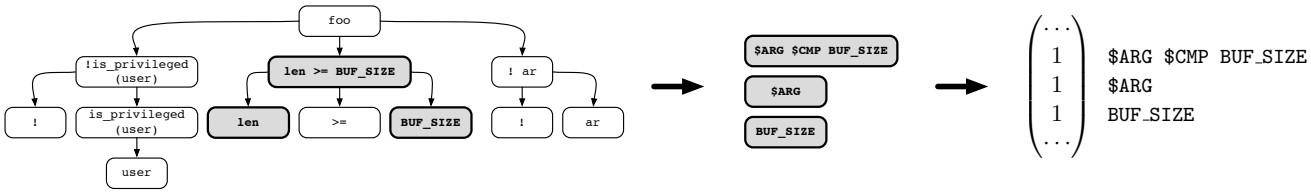


Figure 5: Schematic depiction of embedding. Conditions related to the specified source/sink are extracted from the abstract syntax tree and mapped to a vector space using their expressions.

Upon completion of the previous step, conditions related to the source or sink s are known for each function. We proceed to extract all expressions (including sub-expressions) contained in each of these conditions. The expressions then undergo a simple normalization to account for small syntactical differences in the formulation of checks. In detail, the normalization consists of two transformations.

1. **Removal of negations.** Since CHUCKY does not account for the actions taken upon evaluation of an expression, negations are removed as the first step of normalization. For the same reason, relational operators are replaced by the symbol `$CMP`. Furthermore, numbers are replaced by `$NUM` since for example, the expression `x < 10` is equivalent to `x < 1 + 9`.

2. **Normalization of arguments and return values** If the source or sink of interest is a function, its return value is renamed to `$RET` while any variables directly influencing its arguments are renamed to `$ARG`. This allows return values and arguments of functions to be compared regardless of the names of variables chosen.

Finally, the functions can be embedded in a vector space by applying a mapping φ, which transforms functions into numerical vectors. To represent each function by all normalized expressions contained in its conditions, we define E to be the set of all normalized expressions and φ as

$$\varphi : X \to \mathbb{R}^{|E|}, \quad \varphi(x) \mapsto \big(I(x,e)\big)_{e \in E}$$

where I is an indicator function defined as

$$I(x,e) = \begin{cases} 1 & \text{if } x \text{ contains } e \text{ in a condition} \\ 0 & \text{otherwise.} \end{cases}$$

Note that while the total number of expressions contained in the code base may be very large, the vast majority of functions contains only few of these expressions. In practice, this allows for memory efficient implementations using hash maps or sorted arrays [27].

Figure 5 illustrates this process for the function `foo` from Figure 1. The single condition related to the sink `memcpy` is first determined and all sub-expressions are extracted and normalized. Finally, the vectorial representation is obtained by applying the mapping φ.

3.5 Anomaly Detection

Based on the embedding of functions, we are finally able to determine missing checks geometrically. To this end, a model of normality quantifying the importance of each expression contained in a check is derived from the embedded neighbors. Measuring the distance to this model allows us to determine checks that are present in most neighbors but are missing in the examined function. This enables CHUCKY to pinpoint the exact missing check and report its absence to the analyst. Moreover, an anomaly score can be calculated, allowing particularly interesting code to be returned to the analyst for immediate inspection. Mathematically, this process is implemented as follows.

For each source or sink s used in a function x under consideration, a model of normality is calculated based on its neighbors $\mathcal{N}$. Recalling that in the previous step, each neighbor is mapped to a vector in the space spanned by the expressions contained in its conditions, a natural choice for this model is the center of mass of all embedded neighbor vectors. The model $\mu \in \mathbb{R}^{|E|}$ is thus computed over all k embedded neighbour functions as

$$\mu = \frac{1}{|\mathcal{N}|} \sum_{n \in \mathcal{N}} \varphi(n)$$

Each coordinate of μ represents the fraction of the neighbors, that contain a particular Boolean expression in its conditions as a number between 0 and 1. For example, a score of 0.9 in the coordinate associated with the expression `$RET $CMP $NUM` indicates that 90% the function's neighbors check the return value against a literal number while only 10% do not.

Identifying missing checks is now easy as it merely requires assessing the difference between the embedded function vector $\varphi(x)$ and the model of normality μ. To this end, we calculate the distance vector $d \in \mathbb{R}^{|E|}$ given by

$$d = \mu - \varphi(x).$$

Each coordinate of the vector d is a value between -1 and +1. Positive numbers denote Boolean expressions that are checked by a fraction of neighbors but missing in the function under consideration x, i.e., *missing checks*. In contrast, negative numbers denote expressions checked in x but not present in any of its neighbors.

Finally, from the distance vector we compute an anomaly score suitable to rank functions according to the likelihood that it is omitting a check. We define this anomaly score for a function x as

$$f(x) = ||\mu - \varphi(x)||_\infty = \max_{e \in E} \big(\mu_e - I(x,e)\big)$$

that is, the anomaly score is given by the largest coefficient of the vector d. Recalling that positive numbers denote missing expressions, the rationale behind this choice is to rank functions high if many neighbors contain an expression, which is not present in the function of interest. Furthermore, the maximum norm is chosen, because functions deviating from its neighbors in a single concrete expression—while containing all other expressions—are usually more interesting than functions differing from their neighbors entirely.

4. EMPIRICAL EVALUATION

We proceed to evaluate our method on the source code of five popular software projects: Firefox, Linux, LibPNG, LibTIFF and Pidgin. In particular, we are interested in CHUCKY's detection performance when tasked to identify missing checks as well as its practical value in real source code audits. We begin by conducting a controlled experiment (Section 4.1) involving artificial as well as real missing checks leading to vulnerabilities discovered in the past. Finally, we study our method's ability to assist in the discovery of previously unknown vulnerabilities by providing case studies (Section 4.2).

4.1 Missing Check Detection

To evaluate the detection performance of CHUCKY, the security history of each of the five projects is reviewed. In each code base, we uncover cases where a security check is present in many functions but omitted in others, thus causing vulnerabilities.

In all but one case, these vulnerabilities are critical, allowing an attacker to fully compromise the software. Additionally, we take care to choose samples involving different cases of vulnerabilities, e.g., missing checks for security logic, function arguments, function return values. In the following we provide a detailed description of our dataset. Table 1 summarizes this information.

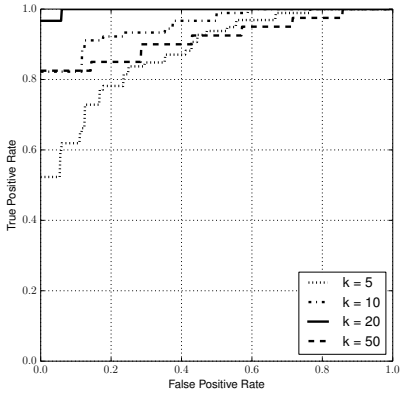
- **Firefox.** The JavaScript engine of the popular Web browser Firefox (version 4.0) contains 5,649 functions and 372,450 lines of code. A failure to check the number of arguments passed to native code implementations of JavaScript functions (i.e., the parameter `argc`) leads to a use-after-free vulnerability (CVE-2010-3183). Ten utility functions implementing array operations perform the same security check to avoid this.

- **Linux.** The filesystem code of the Linux operating system kernel (version 2.6.34.13) contains 19,178 functions and 955,943 lines of code. A missing check before setting an ACL allows to bypass file system permissions (CVE-2010-2071). The check involves the parameter `dentry` and its structure field `dentry->d_inode`. Eight functions of different filesystems implement a corresponding security check correctly.

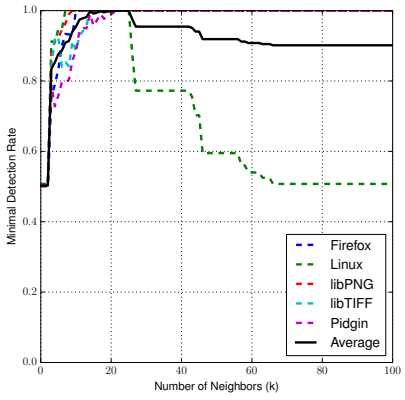
- **LibPNG.** The image processing library LibPNG (version 1.2.44) contains 437 functions and 40,255 lines of code. A missing check of the PNG chunk's size (i.e., the parameter `length`) results in a memory corruption (CVE-2011-2692). Nineteen functions processing PNG chunks perform the same critical check to avoid this.

- **LibTIFF.** The image processing library LibTIFF (version 3.9.4) contains 609 functions and 33,335 lines of code. Missing checks of the length field of TIFF directory entries (i.e., the parameter `dir` and its field `dir->tdir_count`) lead to two independent stack-based buffer-overflows (CVE-2006-3459 and CVE-2010-2067). Nine functions processing TIFF directory entries perform a security check to avoid this problem.

- **Pidgin.** The instant messaging library of the popular instant messenger Pidgin (version 2.7.3) contains 7,390 functions and 332,762 lines of code. A missing check of the return value of the internal base64-decoding routine `purple_base64_decode` leads to a denial-of-service vulnerability (CVE-2010-3711). Eighteen functions parsing network data in Pidgin perform a corresponding security check correctly to avoid this.



(a) Averaged ROC curve



(b) AUC (Minimal Detection Rate)

Figure 6: Detection performance for the five projects with neighborhoods of different size.

For each code base we begin our evaluation by patching the known vulnerabilities. We then proceed to create several experiments in a round-robin fashion for each security check, where we remove the check from one function while leaving all other functions untouched. Vulnerabilities are thus deliberately introduced in order to allow us to measure our methods ability to identify them.

CHUCKY is then employed to rank functions by analyzing their use of the source or sink requiring validation. Note that while only those functions known to be vulnerable or non-vulnerable are included in the ranking, the entire code base is considered for neighborhood selection. The experiment thus reflects the situation encountered in practice and performs no simplifications.

These experiments allow us to assess our method's ability to rediscover those vulnerabilities reported to the projects in the past but go one step further by exploring CHUCKY's capability to discover artificially introduced vulnerabilities. In total, the dataset allows us to perform 64 experiments independently (see last column of Table 1).

Project	Component	Vulnerability	LOC	# functions	# with check
Firefox 4.0	JavaScript engine	CVE-2010-3183	372450	5649	10
Linux 2.6.34.13	Filesystem code	CVE-2010-2071	955943	19178	8
LibPNG 1.2.44	Entire library	CVE-2011-2692	40255	473	19
LibTIFF 3.9.4	Entire library	CVE-2010-2067	33335	609	9
Pidgin 2.7.3	Messaging	CVE-2010-3711	332762	7390	18

Table 1: Overview of our dataset. For each project the missing-check vulnerability, the lines of code (LOC), the number of functions and the number of functions involving the check is listed.

Figure 6(a) shows the ROC curves of our method averaged over all projects for neigborhoods of different size k, where the detection rate is plotted against the false-positive rate for different thresholds. For low values of k, such as $k = 5$, already 50% of the missing checks can be detected with few false positives. With increasing k, CHUCKY is able to identify more missing conditions, where finally almost all missing checks in the five code bases are detected with $k = 20$ at a detection rate of 96%.

We further investigate the effect of the number of neighbors on the detection performance by generating individual ROC curves for values of k between 1 and 100 using the *Area Under Curve* (AUC) as a performance measure [2]. Figure 6(b) shows the results of this experiment. For $k = 25$ we attain *perfect results* across all code bases allowing all vulnerabilities to be detected with no false positives. While we do not assume that an optimal choice of k exists for arbitrary code bases, for the projects under consideration, the average number of functions operating in a similar context thus seems to be around 25. For all further evaluations we thus fix k to 25.

Moreover, we can observe that for those code bases where APIs have been insecurely used (Firefox, LibPNG, LibTIFF and Pidgin), the maximum performance is achieved when k is chosen above a certain threshold. This confirms that in these cases, the majority of functions employing the source or sink perform the security check, thus making neighborhood discovery rather dispensable. In the case of Linux, where missing checks in security logic need to be identified, the performance drops when k becomes too large. We examine this case in detail and find that the source `dentry` is used many times across the code base while a security check is only required in few cases. Neighborhood discovery thus becomes essential in order to create a model of normality only from functions that operate in a similar context.

This quantitative evaluation demonstrates the potential of CHUCKY. All of the considered vulnerabilities are successfully identified by our method and would have been spotted if CHUCKY had been applied to the respective software projects in the past.

4.2 Discovery of Vulnerabilities

First and foremost, CHUCKY is designed to assist an analyst in day-to-day auditing. In the following, we therefore study our method's ability to assist in the discovery of previously unknown vulnerabilities in practice. In particular, we describe how 7 previously unknown vulnerabilities have been discovered by applying CHUCKY on two real-world projects, namely, LibTIFF and Pidgin. We have conducted further studies uncovering 5 more unknown vulnerabilities. For the sake of brevity however, we omit these and details of the vulnerabilities here.

To ensure that all vulnerabilities found by CHUCKY are previously unknown, we update the code of both software projects to the most recent version. At the time of writing, these are version 4.0.3 for LibTIFF and 2.10.6 for Pidgin.

4.2.1 *LibTIFF Case Study*

In the first case study, we employ CHUCKY to uncover vulnerabilities in LibTIFF, an image processing library and suite of tools for the Tagged Image File Format (TIFF). Image processing libraries are a popular target for attackers as parsing images securely is a challenging task. In particular, integer overflow vulnerabilities when dealing with image dimensions (i.e., image width and height) are a common problem. We therefore use CHUCKY to rank all functions of the code base according to anomalous use of any parameters or local variables named `width`, `height`, `w` or `h`.

```
1  static int
2  tiffcvt(TIFF* in, TIFF* out)
3  {
4    uint32 width, height; /* image width & height */
5    uint16 shortv;
6    float floatv;
7    char *stringv;
8    uint32 longv;
9    uint16 v[1];
10
11   TIFFGetField(in, TIFFTAG_IMAGEWIDTH, &width);
12   TIFFGetField(in, TIFFTAG_IMAGELENGTH, &height);
13
14   CopyField(TIFFTAG_SUBFILETYPE, longv);
15   [...]
16   if( process_by_block && TIFFIsTiled( in ) )
17     return( cvt_by_tile( in, out ) );
18   else if( process_by_block )
19     return( cvt_by_strip( in, out ) );
20   else
21     return( cvt_whole_image( in, out ) );
22 }
```

Figure 7: Missing checks of the variables width and height in the function `tiffcvt`.

From the 74 functions dealing with these variables, CHUCKY reports only a single function with an anomaly score of 100%. We examine the reported function `tiffcvt` (Figure 7) to find that the `width` and `height` fields are obtained directly from the image file at lines 11 and 12 and are not checked. CHUCKY reports that all neighbors of the function perform a check on the variable `height` while 79% additionally perform a check on the variable `width`.

Indeed, this missing check leads to an integer overflow when calling the function `cvt_by_strip` shown in Figure 8, for which 50% of its neighbors suggest an additional check for the `width` field. Triggering this overflow, a buffer smaller than expected can be allocated at line 11, resulting in a heap-based buffer overflow when calling `TIFFReadRGBAStrip`

Score	Source File	Function Name
0.92	tools/thumbnail.c	initScale
0.88,	tools/rgb2ycbcr.c	cvtRaster
0.88	tools/rgb2ycbcr.c	setupLuma
0.88	tools/ycbcr.c	setupLuma
0.84	tools/pal2rgb.c	main
0.84	tools/tiff2bw.c	main
0.80	libtiff/tif_print.c	TIFFPrintDirectory
0.80	tools/raw2tiff.c	guessSize
0.76	tools/sgisv.c	svRGBContig
0.76	tools/sgisv.c	svRGBSeparate

Table 2: Top ten functions returned for the sink *_TIFFmalloc*. All 10 functions fail to check the return value of the sink. Vulnerabilities are indicated by dark shading.

```
1  static int
2  cvt_by_strip( TIFF *in, TIFF *out )
3
4  {
5    uint32* raster; /* retrieve RGBA image */
6    uint32  width, height; /* image width & height */
7    [...]
8    TIFFGetField(in, TIFFTAG_IMAGEWIDTH, &width);
9    TIFFGetField(in, TIFFTAG_IMAGELENGTH, &height);
10   /* Allocate strip buffer */
11   raster = (uint32*)
12    _TIFFmalloc(width*rowsperstrip*sizeof (uint32));
13   if (raster == 0) {
14     TIFFError(TIFFFileName(in),
15               "No space for raster buffer");
16     return (0);
17   } [...]
18   for(row=0;ok&&row<height;row+=rowsperstrip )
19     { [...]
20       /* Read the strip into an RGBA array */
21       if (!TIFFReadRGBAStrip(in,row,raster)) {
22         [...]
23       } [...]
24     }
25   _TIFFfree( raster ); [...]
26   return ok;
27 }
```

Figure 8: Integer overflow in the function `cvt_by_strip` caused by the missing check in the caller `tiffcvt`. In effect, a buffer overflow results when calling `TIFFReadRGBAStrip`.

on line 21.Chucky thus leads us almost directly to a possibly exploitable memory corruption vulnerability.

In a second example, our method is used to uncover NULL pointer dereferenciations. To this end, all functions of the code base are analyzed for missing or unusual checks for `_TIFFMalloc`, a simple wrapper around `malloc`.

In total 237 functions call `_TIFFMalloc`. Table 2 shows the top ten of these functions ranked by our method according to anomalous use of `_TIFFMalloc`. In each of the ten cases, Chucky reports that a check for the return value (expression `$RET`) is performed by the vast majority of neighbors, while it is missing in the identified functions. Note, that at no point, the checks required for the use of `_TIFFMalloc` have been specified explicitly; instead Chucky leverages the information distributed across the code base to determine these security checks automatically. We confirm the missing checks in all ten cases. In four of these, the omitted check allows attackers to cause a denial-of-service condition by providing specifically crafted input, whereas in other cases, only a software defect is identified.

```
1  cvtRaster(TIFF* tif, uint32* raster,
2            uint32 width, uint32 height)
3  {
4    uint32 y;
5    tstrip_t strip = 0;
6    tsize_t cc, acc;
7    unsigned char* buf;
8    uint32 rwidth = roundup(width, horizSubSampling);
9    uint32 rheight = roundup(height, vertSubSampling);
10   uint32 nrows = (rowsperstrip > rheight ?
11                   rheight : rowsperstrip);
12   uint32 rnrows = roundup(nrows,vertSubSampling);
13
14   cc = rnrows*rwidth + 2*((rnrows*rwidth) /
15        (horizSubSampling*vertSubSampling));
16   buf = (unsigned char*)_TIFFmalloc(cc);
17   // FIXME unchecked malloc
18   for (y = height; (int32) y > 0; y -= nrows){
19     uint32 nr = (y > nrows ? nrows : y);
20     cvtStrip(buf, raster + (y-1)*width, nr, width);
21     nr = roundup(nr, vertSubSampling);
22     acc = nr*rwidth + 2*((nr*rwidth)/
23      (horizSubSampling*vertSubSampling));
24     if(!TIFFWriteEncodedStrip(tif,strip++,
25                               buf,acc)){
26       _TIFFfree(buf); return (0);
27     }
28   }
29   _TIFFfree(buf); return (1);
30 }
```

Figure 9: A missing check detected in the function `cvtRaster` of the library LibTIFF.

As an example, let us consider the function `cvtRaster` shown in Figure 9. This function provides an illustrative example because the programmer confirms that the return value of `_TIFFMalloc` requires validation in the comment on line 17. In this particular case, the method reports that 85% of the function's neighbors validate the return value and 40% compare it to the constant *NULL*. No other Boolean expression is found to consistently occur across all neighbors in more than 30% of the cases. Furthermore, from all symbols used in the function, the deviation in its use of `_TIFFMalloc` is most pronounced. The function is thus among the top 15% in the global ranking and thus Chucky points the analyst to vulnerability even if an interest in `_TIFFMalloc` is not expressed explicitly.

4.2.2 *Pidgin Case Study*

In the second case study, we employ Chucky to uncover two denial-of-service vulnerabilities in Pidgin's implementation of the Microsoft Instant Messaging Protocol. In particular, we find that a vulnerability exists allowing users to remotely crash the instant messengers without requiring cooperation from the victims side. As starting point for our analysis, we review the C standard library for commonly used functions, which crash upon reception of a NULL pointer as an argument. As an example, the sinks `atoi` and `strchr` are chosen and Chucky is employed to rank all functions according to missing or faulty checks for these sinks.

Table 3 shows all functions with an anomaly score of over 50%, that is, cases where more than half of the neighbors suggest a check to be introduced. Furthermore, in all cases, Chucky indicates that the arguments passed to the sinks need to be checked. With this information, we are able to

Score	Source File	Function Name
0.84	msn.c	msn_normalize
0.76	oim.c	msn_oim_report_to_user
0.72	oim.c	msn_parse_oim_xml
0.72	msnutils.c	msn_import_html
0.64	switchboard.c	msn_switchboard_add_user
0.64	slpcall.c	msn_slp_sip_recv
0.60	msnutils.c	msn_parse_socket
0.60	contact.c	msn_parse_addr..._contacts
0.60	contact.c	msn_parse_each_member
0.60	command.c	msn_command_from_string
0.56	msg.c	msn_message_parse_payload

Table 3: Top ten functions returned for the sinks `atoi` and `strchr` in Pidgin's implementation of the Microsoft Instant Messenger Protocol. Vulnerabilities are indicated by dark shading.

discover two cases among the top ten missing checks allowing attackers to remotely crash Pidgin.

First Example.

For the function `msn_parse_oim_xml` shown in Figure 10, CHUCKY reports that 72% of its neighbors validate arguments passed to `atoi` while this function does not. Indeed, this is the case on line 19 where the variable `unread` is passed to `atoi` unchecked. Moreover, CHUCKY reports that 75% of the function neighbors check the return value of `xmlnode_get_data` while this function does not. Combined, these two missing checks allow Pidgin to be crashed by sending an XML-message with an empty "E/UI" node. This causes `xmlnode_get_data` to return a NULL pointer on line 13, which is then propagated to `atoi` resulting in a crash.

```
1  static void
2  msn_parse_oim_xml(MsnOim *oim, xmlnode *node)
3  {
4    xmlnode *mNode;
5    xmlnode *iu_node;
6    MsnSession *session = oim->session;
7    [...]
8    iu_node = xmlnode_get_child(node, "E/IU");
9
10   if(iu_node != NULL &&
11    purple_account_get_check_mail(session->account))
12    {
13      char *unread = xmlnode_get_data(iu_node);
14      const char *passports[2] =
15        { msn_user_get_passport(session->user) };
16      const char *urls[2] =
17        { session->passport_info.mail_url };
18
19      int count = atoi(unread);
20
21      /* XXX/khc: pretty sure this is wrong */
22      if (count > 0)
23        purple_notify_emails(session->account->gc,
24                             count, FALSE, NULL,
25                             NULL, passports,
26                             urls, NULL, NULL);
27      g_free(unread);
28    }
29  [...]
30 }
```

Figure 10: Missing check in the function `msn_parse_oim_xml` of the instant messenger Pidgin.

Second Example.

For the function `msn_message_parse_payload` shown in Figure 11, CHUCKY reports a failure to check the argument passed to `strchr` with an anomaly score of 56%. The vulnerable call can be seen on line 15 and can be triggered by sending a message containing the string "Content-Type" immediately followed by two successive carriage return line feed sequences. This causes the variable `value` to be set to NULL on line 10. This value propagates to `strchr` on line 15 causing the crash of Pidgin. This vulnerability is particularly interesting as it can be triggered by other users of the MSN service.

```
1  void
2  msn_message_parse_payload(MsnMessage *msg, [...])
3  {
4    [...]
5    for (cur = elems; *cur != NULL; cur++)
6      {
7        const char *key, *value; [...]
8        tokens = g_strsplit(*cur, ": ", 2);
9        key = tokens[0];
10       value = tokens[1];
11       [...]
12       if (!strcmp(key, "Content-Type"))
13         {
14           char *charset, *c;
15           if ((c = strchr(value, ';')) != NULL)
16             {
17               [...]
18             }
19           msn_message_set_content_type(msg, value);
20         }
21       else
22         {
23           msn_message_set_attr(msg, key, value);
24         }
25       g_strfreev(tokens);
26     }
27   g_strfreev(elems);
28   /* Proceed to the end of the "\r\n\r\n" */
29   tmp = end + strlen(body_dem);
30   [...]
31   g_free(tmp_base);
32 }
```

Figure 11: Missing check in the function `msn_message_parse_payload` of the instant messenger Pidgin.

5. LIMITATIONS

Similar to other methods for the discovery of security flaws, CHUCKY cannot overcome the inherent limitations of vulnerability identification. While our method is able to expose missing security checks effectively, it cannot verify whether these truly lead to vulnerabilities in practice. This limitation, however, can be alleviated by the analyst, as he can guide the search for vulnerabilities by only inspecting security-critical sources and sinks, such as common memory, network and parsing functions. As our experiments demonstrate, this often enables CHUCKY to pinpoint missing checks related to vulnerabilities with little manual effort.

In contrast to other methods, CHUCKY does not require external information or code annotations to identify missing checks. The method is capable of operating on the code base of a software project alone. This advantage comes at price: our approach is based on the assumption that the majority of checks in a code base are correct and missing checks are rare. While this assumption holds true for mature software projects, it is not necessary satisfied by software at

an early stage of development. Consequently, CHUCKY is better suited for finding vulnerabilities in stable code.

It is also important to note that CHUCKY makes no attempts to evaluate expressions. Semantically equivalent checks thus cannot be detected, which may lead to false positives in practice. Moreover, our method is only able to detect checks if they are missing entirely and cannot detect checks performed too late. Combining our method with existing techniques from data flow analysis and symbolic execution therefore seems to be an interesting direction for future work.

Finally, there exist many types of vulnerabilities that have no relation to missing checks and thus cannot be exposed by CHUCKY. Nonetheless, it is a common practice of developers to add checks before potential vulnerabilities and security-critical code, for example for protecting buffers, limiting access to resources or changing the program flow. All these checks can be exposed by CHUCKY and hence our method addresses a wide range of possible security flaws.

6. RELATED WORK

The theory and practice of finding vulnerabilities in software is a classic field of computer security. Due to their generic nature, missing-check vulnerabilities border on a wide range of previous research. In the following, we point out relations to this work and related approaches.

Static Code Analysis.

In practice, checking tools such as *Microsoft PREfast* [15] or *PScan* [5] are used to statically find vulnerabilities in source code. These tools possess built-in information about correct API usage and common programming mistakes, which severely limits the kind of vulnerabilities these tools can detect. An alternative route is taken by scanners such as *Splint* [7], which allow code annotations to be supplied by analysts. However, creating these annotations and rules regarding API usage is both time consuming and challenging as it requires an intimate understanding of both internal and external APIs of a target program.

Several approaches seize the idea of inspecting the temporal properties of API usage [e.g., 6, 16, 38] and attempt to provide this information across projects [e.g., 10, 43]. Such properties would, for instance, indicate that a *lock* function call usually is followed by a call to the *unlock* function. Software defects can thus be detected by identifying anomalies, which do not comply with the derived set of rules. Conditions, however, are not analyzed and therefore, these approaches are not suited for finding missing checks.

Tan et al. [34] analyze API call sequences to identify necessary security checks whereas the mapping between a specific check and the event that is required to be checked is specified in advance. Similarly, Livshits et al. [18] focus on inferring information flow specifications by modelling propagation graphs. In contrast, CHUCKY attempts to find missing checks in a more general scope and without additional specification of the sensitive code regions or functions.

Other work goes one step further and makes use of additional auxiliary information like software revision histories [19, 40] or different API implementations [32]. Similar to CHUCKY, Son et al. [31] intend not to make use of annotations or any external specifications. Instead they rely on software engineering patterns commonly used in web applications and SQL queries that alter the database as security-sensitive events. Therefore, this approach is tightly bound to web applications while CHUCKY can be applied in any programming environment if a suitable parser is available.

In many approaches, infrequent but correct patterns might cause false positives due to a biased notion of normality. Thummalapenta and Xie [35] address this by introducing *alternative patterns*. We discussed this limitation with respect to CHUCKY in Section 5. Another more general method coined as vulnerability extrapolation [41, 42] learns from known security flaws and finds locations that might exhibit a similar vulnerability. We build on this method for the neighborhood discovery described in Section 3.2.

Taint Analysis and Symbolic Execution.

Taint analysis or taint tracking is a method for performing information flow analysis. Data of interest is "tainted" and tracked from a source through the system to a specified sink. In principle one differentiates between static and dynamic taint analysis. Dynamic taint tracking has been used for the discovery of vulnerabilities [e.g., 23, 37]. However, since CHUCKY is strictly operating on source code, we do not discuss dynamic approaches at this point. Static taint tracking has been effectively used for detecting vulnerabilities such as format string vulnerabilities in C programs [30] or SQL injections and cross-site scripting [13, 17].

Nevertheless, taint analysis implies some limitations that come down to its passive view on the data flow. Symbolic execution can overcome these by actively exploring the code's state space and execution paths [1, 4, 9, 37]. Due to this state exploration, symbolic execution becomes intractable without heuristics to reduce the number of branches that need to be analyzed. As a result it hardly can be used for code auditing in practice [11].

7. CONCLUSIONS

Vulnerabilities in software are a persistent problem and one of the root causes of many security incidents. Discovering security flaws is a challenging and often daunting task, as automatic approaches are inherently limited in spotting vulnerable code. As a remedy, we introduce CHUCKY in this paper, a method that can automatically detect missing checks in software and thereby help to accelerate the manual auditing of source code. Instead of struggling with the limits of automatic approaches, our method aims at assisting a human analyst by providing information about missing security checks and potential fixes. Our evaluation demonstrates the potential of this approach, since we are able to uncover 12 previously unknown vulnerabilities in popular software projects among the first missing checks.

Finally, CHUCKY can interface with many other techniques for finding vulnerabilities. For example, exposed missing checks might be further analyzed using techniques for fuzzing or symbolic execution. These techniques could allow to narrow down the actual consequences of a missing check and might help to rank detected flaws according to their severity. Moreover, we currently plan to integrate CHUCKY in a visual development environment and analyze its capabilities to expose missing security checks directly during the development of software.

Acknowledgments

The authors acknowledge funding from *BMBF* under the project PROSEC (FKZ 01BY1145).

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] A. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

[3] B. Chess and M. Gerschefske. Rough auditing tool for security. Google Code, `http://code.google.com/p/rough-auditing-tool-for-security/`, visited February, 2013.

[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] A. DeKok. Pscan: A limited problem scanner for c source files. `http://deployingradius.com/pscan/`, visited February, 2013.

[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] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

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

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

[10] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–130, 2010.

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

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

[13] 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.

[14] J. A. Kupsch and B. P. Miller. Manual vs. automated vulnerability assessment: A case study. In *Proc. of Workshop on Managing Insider Security Threats (MIST)*, pages 83–97, 2009.

[15] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[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 M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proc. of USENIX Security Symposium*, 2005.

[18] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *Proc. of ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI)*, 2009.

[19] 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.

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

[21] H. Moore. Security flaws in universal plug and play: Unplug. don't play. Technical report, Rapid 7, 2013.

[22] C. Mulliner, N. Golde, and J.-P. Seifert. Sms of death: From analyzing to attacking mobile phones on a large scale. In *Proc. of USENIX Security Symposium*, 2011.

[23] 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.

[24] J. W. Oh. Recent Java exploitation trends and malware. Presentation at Black Hat Las Vegas, 2012.

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

[26] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4):15–27, Apr. 2006.

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

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

[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] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. of USENIX Security Symposium*, pages 201–218, 2001.

[31] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *Proc. of ACM International Conference on Object Oriented Programming Systems Languages and Applications (SPLASH)*, 2011.

[32] V. Srivastava, M. D. Bond, K. S. Mckinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *Proc. of ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI)*, 2011.

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

[34] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: automatically inferring security specifications and detecting violations. In *Proc. of USENIX Security Symposium*, 2008.

[35] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proc. of the International Conference on Automated Software Engineering (ASE)*, pages 283–294, 2009.

[36] 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.

[37] 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.

[38] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. of European Software Engineering Conference (ESEC)*, pages 35–44, 2007.

[39] D. A. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder/`, visited February, 2013.

[40] 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.

[41] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proc. of 5th USENIX Workshop on Offensive Technologies (WOOT)*, pages 118–127, Aug. 2011.

[42] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proc. of 28th Annual Computer Security Applications Conference (ACSAC)*, pages 359–368, Dec. 2012.

[43] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In *Proc. of the European Conference on Object-Oriented Programming(ECOOP)*, pages 318–343, 2009.