2010

Detection of recurring software vulnerabilities

Nam Hoai Pham
Iowa State University

Detection of recurring software vulnerabilities

by

Nam H. Pham

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Tien N. Nguyen, Major Professor
Akhilesh Tyagi
Samik Basu

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF TABLES iv

LIST OF FIGURES v

ACKNOWLEDGEMENTS vi

ABSTRACT vii

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. BACKGROUND 5

2.1 Terminology and Concepts 5

2.2 Bug Detection and Localization 6

2.3 Vulnerability Databases 7

CHAPTER 3. EMPIRICAL STUDY 9

3.1 Hypotheses and Process 9

3.2 Representative Examples 10

3.3 Results and Implications 15

3.4 Threats to Validity 16

CHAPTER 4. APPROACH OVERVIEW 17

4.1 Problem Formulation 18

4.2 Algorithmic Solution and Techniques 19

CHAPTER 5. SOFTWARE VULNERABILITY DETECTION 21

5.1 Type 1 Vulnerability Detection 21

5.1.1 Representation 21

5.1.2 Feature Extraction and Similarity Measure 21

5.1.3 Candidate Searching 23
5.1.4 Origin Analysis 24
5.2 Type 2 Vulnerability Detection 25
5.2.1 Representation 25
5.2.2 Feature Extraction and Similarity Measure 27
5.2.3 Candidate Searching 29
CHAPTER 6. EMPIRICAL EVALUATION 31
6.1 Evaluation of Type 1 Vulnerability Detection 31
6.2 Evaluation of Type 2 Vulnerability Detection 33
6.3 Patching Recommendation 37
CHAPTER 7. CONCLUSIONS AND FUTURE WORK 39
APPENDIX A. ADDITIONAL TECHNIQUES USED IN SECURESYNC 42
BIBLIOGRAPHY 46

LIST OF TABLES

Table 3.1 Recurring Software Vulnerabilities 15

Table 6.1 Recurring Vulnerability Type 1 Detection Evaluation 32

Table 6.2 Recurring Vulnerability Type 2 Detection Evaluation 34

Table 6.3 Recurring Vulnerability Type 2 Recommendation 37

Table A.1 Extracted Patterns and Features 44

Table A.2 Feature Indexing and Occurrence Count 44

LIST OF FIGURES

Figure 3.1 Vulnerable Code in Firefox 3.0.3 10
Figure 3.2 Vulnerable Code in SeaMonkey 1.1.12 10
Figure 3.3 Patched Code in Firefox 3.0.4 11
Figure 3.4 Patched Code in SeaMonkey 1.1.13 12
Figure 3.5 Recurring Vulnerability in NTP 4.2.5 13
Figure 3.6 Recurring Vulnerability in Gale 0.99 13

Figure 4.1 SecureSync's Working Process 17
Figure 4.2 Detection of Recurring Vulnerabilities 19

Figure 5.1 xAST from Code in Figure 3.1 and Figure 3.2 22
Figure 5.2 xAST from Patched Code in Figure 3.3 and Figure 3.4 22
Figure 5.3 xGRUMs from Vulnerable and Patched Code in Figure 3.5 26
Figure 5.4 Graph Alignment Algorithm 28

Figure 6.1 Vulnerable Code in Thunderbird 2.0.17 33
Figure 6.2 Vulnerable Code in Arronwork 1.2 35
Figure 6.3 Vulnerable and Patched Code in GLib 2.12.3 35
Figure 6.4 Vulnerable Code in SeaHorse 1.0.1 36

Figure 7.1 The SecureSync Framework 40

Figure A.1 The Simulink Model and Graph Representation 43

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis.

First and foremost, Dr. Tien N. Nguyen for his guidance, patience and support throughout this research and the writing of this thesis. Her insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Akhilesh Tyagi and Dr. Samik Basu.

I would additionally like to thank Tung Thanh Nguyen and Hoan Anh Nguyen for their comments and support throughout the all stages of this thesis.

ABSTRACT

Software security vulnerabilities are discovered on an almost daily basis and have caused substantial damage. It is vital to be able to detect and resolve them as early as possible. One of early detection approaches is to consult with the prior known vulnerabilities and corresponding patches. With the hypothesis that recurring software vulnerabilities are due to *software reuse*, we conducted an empirical study on several databases for security vulnerabilities and found several recurring and similar software security vulnerabilities occurring in *different* software systems. Most of recurring vulnerabilities occur in the systems that reuse source code, share libraries/APIs or reuse at a higher level of abstraction (e.g. algorithms, protocols, or specifications).

The finding suggests that one could effectively detect and resolve some unreported vulnerabilities in one software system by consulting the prior known and reported vulnerabilities in the other systems that reuse/share source code, libraries/APIs, or specifications. To help developers with this task, we developed SecureSync, a supporting tool to automatically detect recurring software vulnerabilities in different systems that share source code or libraries, which are the most frequent types of recurring vulnerabilities. SecureSync is designed to work with a semi-automatically built knowledge base of the prior known/reported vulnerabilities, including the corresponding systems, libraries, and vulnerable and patched code. To help developers check and fix the vulnerable code, SecureSync also provides some suggestions such as adding missed function calls, adding checking of an input/output of a function call, replacing the operators in an expression, etc.

We conducted an evaluation on 60 vulnerabilities of with the totals of 176 releases in 119 open-source software systems. The result shows that SecureSync is able to detect recurring vulnerabilities with high accuracy and to identify several vulnerable code locations that are not yet reported or fixed even in mature systems.

CHAPTER 1. INTRODUCTION

New software security vulnerabilities are discovered on an almost daily basis [4]. Attacks against computer software, which is one of the key infrastructures of our modern society and economies, can cause substantial damage. For example, according to the CSI Computer Crime and Security Survey 2008 [30], 522 US companies were reported to have lost in total $3.5 billion per year due to the attacks on critical business software applications. Many systems are developed, deployed, and used over years that contain significant security weaknesses. Over 90% of security incidents reported to the Computer Emergency Response Team (CERT) Coordination Center result from software defects [17]. Because late corrections of errors could cost up to 200 times as much as early correction [23], it is vital to be able to detect and resolve them as early as possible. One of early detection approaches is to consult with the prior known vulnerabilities and corresponding patches. In current practice, known software security vulnerabilities and/or patches are often reported in public databases (e.g. National Vulnerability Database (NVD) [17], Common Vulnerabilities and Exposures database (CVE) [4]), or on public websites of specific software applications.

With the hypothesis that recurring software vulnerabilities are due to *software reuse*, we conducted an empirical study on several databases for security vulnerabilities including NVD [17], CVE [4], and others. We found several recurring and similar software security vulnerabilities occurring in *different* software systems. Most of recurring vulnerabilities occur in the systems that reuse source code (e.g. having the same code base, deriving from the same source, or being developed on top of a common framework). That is, a system has some vulnerable code fragments. Then, such code fragments are reused in other systems (e.g. by copy-and-paste practice, by branching/duplicating the code base and then developing new versions or new systems). Patches in one of such systems were late propagated into other systems. Due to the reuse of source code, the recurring vulnerable code fragments are

identical or highly similar in code structure and names of function calls, variables, constants, literals, or operators. Let us call them Type 1.

Another type of recurring vulnerabilities occurs across different systems that share APIs/libraries (Type 2). For example, such systems use the same function from a library and have the same errors in API usages, e.g. missing or wrongly checking the input/output of the function; missing or incorrectly ordering function calls, etc. The corresponding vulnerable code fragments on such systems tend to misuse the same APIs in a similar manner, e.g., using the incorrect orders, missing step(s) in function calls, missing the same checking statements, incorrectly using the same comparison expression, etc.

There are also some systems having recurring or similar vulnerabilities due to the reuse at a higher level of abstraction. For example, such systems share the same algorithms, protocols, specifications, standards, and then have the same bugs or programming faults. We call such recurring vulnerabilities Type 3. The examples and detailed results of all three types will be discussed in Chapter 3.

This finding suggests that one could effectively detect and resolve some unreported vulnerabilities in one software system by consulting the prior known and reported vulnerabilities in the other systems that reuse/share source code, libraries, or specifications. To help developers with this task, we developed SecureSync, a supporting tool that is able to automatically detect recurring software vulnerabilities in different systems that share source code or libraries, which are the most frequent types of recurring vulnerabilities. Detecting recurring vulnerabilities in systems reusing at higher levels of abstraction will be investigated in future work.

SecureSync is designed to work with a semi-automatically built knowledge base of the prior known/reported vulnerabilities, including the corresponding systems, libraries, and vulnerable and patched code. It could support detecting and resolving vulnerabilities in the two following scenarios:

1. Given a vulnerability report in a system A with corresponding vulnerable and patched code, SecureSync analyzes the patches and stores the information in its knowledge base. Then, via Google Code Search [6], it searches for all other systems B that share source code and libraries with A, checks if B has the similarly vulnerable code, and reports such locations (if any).

2. Given a system X for analysis, SecureSync will check whether X reuses some code fragments or libraries with another system Y in its knowledge base. Then if the shared code in X is sufficiently

similar to the vulnerable code in Y, SecureSync will report it to be likely vulnerable and point out the vulnerable location(s).

In those scenarios, to help developers check and fix the vulnerable code, SecureSync also provides some suggestions such as adding missed function calls, adding checking of input/output before or after a call, replacing the operators in an expression, etc.

The key technical goals of SecureSync are how to represent vulnerable and patched code and how to detect code fragments that are similar to vulnerable ones. We have developed two core techniques for those problems to address two kinds of recurring vulnerabilities. For recurring vulnerabilities of Type 1 (reusing source code), SecureSync represents vulnerable code fragments as Abstract Syntax Tree (AST)-like structures, with the labels of nodes representing both node types and node attributes. For example, if a node represents a function call, its label will include the node type `FUNCTION CALL`, the function name, and the parameter list. The similarity of code fragments is measured by the similarity of structures of such labeled trees. Our prior technique, Exas [44, 49], is used to approximate structure information of labeled trees and graphs by vectors and to measure the similarity of such trees via vector distance.

For recurring vulnerabilities of Type 2 (systems sharing libraries), the traditional code clone detection techniques do not work in these cases because the similarity measurement must involve program semantics such as API usages and relevant semantic information. SecureSync represents vulnerable code fragments as graphs, with the nodes representing function calls, condition checking blocks (as control nodes) in statements such as `if`, `while`, or `for`, and operations such as `==`, `!`, or `<`. Labels of nodes include their types and names. The edges represent the relations between nodes, e.g. control/data dependencies, and orders of function calls. The similarity of such graphs is measured based on their largest common subgraphs.

To improve the performance, SecureSync also uses several filtering techniques. For example, it uses text-based filtering to keep only source files containing identifiers/tokens related to function names appearing in vulnerable code in its knowledge base. It also uses locality-sensitive hashing (LSH) [20] to perform fast searching for similar trees in its knowledge base: only trees having the same hash code are compared to each other. SecureSync uses set-based filtering to find the candidates of Type 2:

only the graphs containing the nodes having the same/similar labels with the nodes of the graphs in its knowledge base are kept as candidates for comparison.

We conducted an evaluation on 48 and 12 vulnerabilities of Type 1 and Type 2 with the totals of 51 and 125 releases, respectively, in 119 open-source software systems. The result shows that SecureSync is highly accurate. It is able to correctly locate most of locations of vulnerable code in the systems that share source code/libraries with the systems in its knowledge base. Interestingly, it detects 90 releases having potentially vulnerable code locations (fragments having vulnerabilities) that, to the best of our knowledge, have not been detected, reported, or patched yet even in mature systems. Based on the recommendations from our tool, we produced the patches for such vulnerabilities and reported to the developers of those systems. Some of such vulnerabilities and patches were actually confirmed. We still wait for replies on others.

The contribution of this thesis includes:

1. An empirical study that confirms the existence of recurring/similar software vulnerabilities and our aforementioned software reuse hypothesis, and provides us insights on their characteristics;

2. Two representations and algorithms to detect recurring vulnerabilities on systems sharing source code and/or APIs/libraries;

3. SecureSync: An automatic prototype tool that detects recurring vulnerabilities and recommends the resolution for them;

4. An empirical evaluation of our tool on real-world datasets of vulnerabilities and systems shows its accuracy and usefulness.

The outline of the thesis is as follow: Chapter 2 discusses about the literature review. Chapter 3 reports our empirical study on recurring vulnerabilities. Chapter 4 and Chapter 5 present the overview of our approach and detailed techniques. The empirical evaluation is in Chapter 6. Finally, conclusions appear last in Chapter 7.

CHAPTER 2. BACKGROUND

First, we discuss about the terminology and concepts used within the thesis. Second, we present related approaches in bug detection and localization. Finally, an overview of the current state of open software security databases appears last in this chapter.

2.1 Terminology and Concepts

A system is a software product, program, module, or library under investigation (e.g. Firefox, OpenSSL, etc). **A release** refers to a specific release/version of a software system (e.g. Firefox 3.0.5, OpenSSL 0.9.8).

A software bug is the common term used to describe an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways [13].

A patch is a piece of software designed to fix problems with, or update a computer program or its supporting data. This includes fixing security vulnerabilities and other bugs, and improving the usability or performance [11].

A vulnerability is an exploitable software fault occurring on specific release(s) of a system. For example, CVE-2008-5023 reported a vulnerability in security checks on Firefox 3.0.0 to 3.0.4. A software vulnerability can be caused by a software bug which may allow an attacker to misuse an application.

A recurring vulnerability is a vulnerability that occurs and should be fixed/patched on at least two different releases (of the same or different systems). This term also refers to a group of vulnerabilities having the same causes on different systems/releases. Examples of recurring vulnerabilities are in Chapter 3.

2.2 Bug Detection and Localization

Our research is closely related to *static* approaches to detect *similar* and *recurring* bugs. Static approaches could be categorized into two types: *rule/pattern-based* and *peer-based*. The strategy in pattern-based approaches is first to detect the correct code patterns via mining frequent code/usages, via mining rules from existing code, or predefined rules. The anomaly usages deviated from such patterns are considered as potential bugs. In contrast, a peer-based approach attempts to match the code fragment under investigation against its databases of cases and provides a fix recommendation if a match occurs. Peer-based approach was chosen for SecureSync because a vulnerability must be detected early even though it does not necessarily repeat frequently yet. It is desirable that a vulnerability is detected even if it matches with one case in the past. SecureSync efficiently stores only features for each case.

Several bug finding approaches are based on mining of *code patterns* [32,38,55–57]. Sun *et al.* [54] present a *template-* and *rule-based* approach to automatically propagate bug fixes. Their approach supports some *pre-defined* templates/rules (e.g. orders of pairs of method calls, condition checking around the calls) and requires a fix to be extracted/expressed as rules in order to propagate it. Other tools also detect *pre-defined*, *common* bug patterns using syntactic pattern matching [26,32]. In contrast to those pattern-based approaches, SecureSync is based on our general graph-based representation for API-related code and its feature extraction that could support any vulnerable and corresponding patched code.

JADET [56] and GrouMiner [46] perform mining object usage patterns and detect the violations as potential bugs. Several approaches mine usage patterns in term of the orders of pairs of method calls [18, 38, 57], or association rules [53, 55]. Error-handling bugs are detected via mining sequence association rules [55]. Chang *et al.* [24] find patterns on condition nodes on program dependence graphs and detect bugs involving negligent conditions. Hipikat [27] extracts *lexical* information while building the project's memories and recommends relevant artifacts. Song *et al.* [53] detect association rules between six types of bugs from the project's history for bug prediction. However, those approaches focus only on specific sets of patterns and bugs (object usages [56], error-handling [55], condition checking [24]). BugMem [35] uses a *textual* difference approach to identify changed texts and detects similar bugs. SecureSync captures better the contexts of API usages with its graph representation,

thus, it could detect any type of API-related recurring vulnerabilities more precisely and generally than specific sets of bug patterns. It is designed toward API better than GrouMiner [46] with data and operator nodes, and their relations with function calls.

FixWizard [45] is a peer-based approach for detecting recurring bugs. It relies on code peers, i.e. methods/classes with similar interactions in the *same* system, thus, cannot work across systems. Patch Miner [12] finds all code snapshots with similar snippets (i.e. cloned code) to the fragment that was fixed. CP-Miner [37] mines frequent subsequences of tokens to detect bugs caused by inconsistent editing to cloned code. Jiang *et al.* [34] detect clone-related bugs via formulating context-based inconsistencies. Existing supports for *consistent editing* of cloned code are limited to interactive synchronization in editors such as CloneTracker [28]. Compared to clone-related bug detection approaches [34,37], our detection for code-reused vulnerabilities could also handle *non-continuous* clones (e.g. source code in Figure 3.1 and Figure 3.2). More importantly, our graph-based representation and feature extraction are more specialized toward finding fragments with *similar API usages*, and they are more flexible to support the detection of API-related bugs *across systems*.

Several approaches have been proposed to help users localize buggy code areas [22,31,36,40,42]. Some leverage the project's historical information: the amount of changed LOC over the total in a time period [42], frequently/recently modified/fixed modules [31], code co-changes and bug locality [36], change and complexity metrics [41,48,52], social network among developers [21,22,51,58], etc.

In software security, vulnerability prediction approaches that relied on projects' components and history include [29,43]. Other researchers study the characteristics of vulnerabilities via discovery rates [25], or time and efforts of patching [19]. Longstaff [39] defined a vulnerability classification called Ease of Exploit Classification, based on the difficulty of exploitation. In general, none of existing software security approaches has studied recurring software security vulnerabilities and their characteristics.

2.3 Vulnerability Databases

There are many publicly available vulnerability databases in the Internet including, but not limited to, Common Vulnerabilities and Exposure (CVE) [4], Internet Security Systems (ISS) [7], National

Vulnerability Database (NVD) [17], The Open Source Vulnerability Database (ovsbd) [15]. There is also a suite of selected open standards (The Security Content Automation Protocol - SCAP) that enumerate software flaws, security related configuration issues, and product names; measure systems to determine the presence of vulnerabilities; and provide mechanisms to rank the results of these measurements in order to evaluate the impacts of the discovered security issues. SCAP standards [16] are comprised of Common Vulnerabilities and Exposures (CVE) [4], Common Configuration Enumeration (CCE) [2], Common Platform Enumeration (CPE) [3], Common Vulnerability Scoring System (CVSS) [5], Extensible Configuration Checklist Description Format (XCCDF) [14], and Open Vulnerability and Assessment Language (OVAL) [10].

CHAPTER 3. EMPIRICAL STUDY

3.1 Hypotheses and Process

Hypotheses. In this study, our research is based on the philosophy that *similar code tends to have similar properties*. In the context of this thesis, the similar properties are software bugs, programming flaws, or vulnerabilities. Due to software reuse, in reality, there exist many systems that have similar code and/or share libraries/components. For example, they could be developed from a common framework, share some code fragments due to the copy-and-paste programming practice, use the same API libraries, or implement the same algorithms or specifications, etc. Therefore, we make hypotheses that (H1) there exist software vulnerabilities recurring in different systems, and (H2) one of the causes of such existence is software reuse. In this study, we use the following important terms.

Analysis Process. To confirm those two hypotheses, we considered around 3,000 vulnerability reports in several security databases: National Vulnerability Database (NVD [17]), Open Source Computer Emergency Response Team Advisories (oCERT) [9], Mozilla Foundation Security Advisories (MFSA [8]), and Apache Security Team (ASF) [1]. Generally, each report describes a vulnerability, thus, a recurring vulnerability would be described in multiple reports. Sometimes, one report (such as in oCERT) describes more than one recurring vulnerabilities. Since it is impossible to manually analyze all those reports, we used a textual analysis technique to cluster them into different groups having similar textual contents and manually read such groups. Interestingly, we found some groups which really report recurring vulnerabilities. To verify such recurring vulnerabilities and gain more knowledge about them (e.g. causes and patches), we also collected and analyzed all available source code, bug reports, discussions relevant to them. Let us discuss representative examples and then present detailed results.

```
PRBool nsXBLBinding::AllowScripts() {
...
  JSContext* cx = (JSContext*) context->GetNativeContext();
  nsCOMPtr<nsIDocument> ourDocument;
  mPrototypeBinding->XBLDocumentInfo()->GetDocument(getter_AddRefs(ourDocument));

  /* Vulnerable code: allows remote attackers to bypass the protection mechanism for codebase
       principals and execute arbitrary script via the -moz-binding CSS property in a signed
      JAR file */

  PRBool canExecute;
  nsresult rv = mgr->CanExecuteScripts(cx, ourDocument->NodePrincipal(), &canExecute);
  return NS_SUCCEEDED(rv) && canExecute;
}
```

Figure 3.1 Vulnerable Code in Firefox 3.0.3

```
PRBool nsXBLBinding::AllowScripts() {
...
  JSContext* cx = (JSContext*) context->GetNativeContext();
  nsCOMPtr<nsIDocument> ourDocument;
  mPrototypeBinding->XBLDocumentInfo()->GetDocument(getter_AddRefs(ourDocument));

  nsIPrincipal* principal = ourDocument->GetPrincipal();
  if (!principal){
   return PR_FALSE;
  }

  // Similarly vulnerable code
  PRBool canExecute;
  nsresult rv = mgr->CanExecuteScripts(cx, principal, &canExecute);
  return NS_SUCCEEDED(rv) && canExecute;
}
```

Figure 3.2 Vulnerable Code in SeaMonkey 1.1.12

3.2 Representative Examples

Example 1. In CVE-2008-5023, it is reported that there is a vulnerability which *"allows remote attackers to bypass the protection mechanism for codebase principals and execute arbitrary script via the -moz-binding CSS property in a signed JAR file"*. Importantly, this vulnerability *recurs* in different software systems: all versions of Firefox 3.x before 3.0.4, Firefox 2.x before 2.0.0.18, and SeaMonkey 1.x before 1.1.13 have this vulnerability.

Figure 3.1 and Figure 3.2 show the vulnerable code fragments extracted from Firefox 3.0.3 and SeaMonkey 1.1.12. Figure 3.3 and Figure 3.4 show the corresponding patched code in Firefox 3.0.4 and SeaMonkey 1.1.13. As we could see, the vulnerable code is patched by adding more checking mechanisms via two functions `GetHasCertificate` and `Subsumes`. Further analyzing, we figure

```
PRBool nsXBLBinding::AllowScripts() {
....
  JSContext* cx = (JSContext*) context->GetNativeContext();
  nsCOMPtr<nsIDocument> ourDocument;
  mPrototypeBinding->XBLDocumentInfo()->GetDocument(getter_AddRefs(ourDocument));

  // PATCHED CODE ---------------------
  PRBool canExecute;
  nsresult rv = mgr->CanExecuteScripts(cx, ourDocument->NodePrincipal(), &canExecute);
  if (NS_FAILED(rv) || !canExecute) {
    return PR_FALSE;
  }

  PRBool haveCert;
  doc->NodePrincipal()->GetHasCertificate(&haveCert);
  if (!haveCert){
   return PR_TRUE;
  }

  PRBool subsumes;
  rv = ourDocument->NodePrincipal()->Subsumes(doc->NodePrincipal(), &subsumes);
  return NS_SUCCEEDED(rv) && subsumes;
}
```

Figure 3.3 Patched Code in Firefox 3.0.4

out that the vulnerability is recurring due to the *reuse of source code*. In Figure 3.1 and Figure 3.2, the vulnerable code fragments are highly similar. Because Firefox and SeaMonkey are developed from the common framework Mozilla, they largely share source code, including the vulnerable code in those fragments. Therefore, it causes a recurring vulnerability in those two systems.

This example is a representative example of the recurring vulnerabilities that we classify as Type 1 (denoted by **RV1**). The vulnerabilities of this type are recurring due to the reuse of source code i.e. a code fragment in one system has some undetected flaws and is reused in another system. Therefore, when the flaws are exploited as a vulnerability, the vulnerability is recurring in both systems. In general, the reuse could be made via copy-and-paste practice, or via branching the whole code base to create a new version of a system. In other cases, systems could be derived from the same codebase/framework, but are later independently developed as a new product. Because of reuse, the vulnerable code tends to be highly *similar in texts* (e.g. names of called functions, variables, names/values of literals, etc) and *in structure* (e.g. the structure of statements, branches, expressions, etc). Due to this nature, those similar features could be used to identify them.

A special case of Type 1 is related to different releases of a system (e.g. Firefox 2.x and 3.x in this example). Generally, such versions/releases are developed from the same codebase, e.g. later versions

```
PRBool nsXBLBinding::AllowScripts() {
...
  JSContext* cx = (JSContext*) context->GetNativeContext();
  nsCOMPtr<nsIDocument> ourDocument;
  mPrototypeBinding->XBLDocumentInfo()->GetDocument(getter_AddRefs(ourDocument));

  nsIPrincipal* principal = ourDocument->GetPrincipal();
  if (!principal){
   return PR_FALSE;
  }

  // PATCHED CODE ---------------------
   PRBool canExecute;
   nsresult rv = mgr->CanExecuteScripts(cx, ourDocument->NodePrincipal(), &canExecute);
   if (NS_FAILED(rv) || !canExecute) {
    return PR_FALSE;
   }

  PRBool haveCert;
  doc->NodePrincipal()->GetHasCertificate(&haveCert);
  if (!haveCert){
   return PR_TRUE;
  }

  PRBool subsumes;
  rv = ourDocument->NodePrincipal()->Subsumes(doc->NodePrincipal(), &subsumes);
  return NS_SUCCEEDED(rv) && subsumes;
}
```

Figure 3.4 Patched Code in SeaMonkey 1.1.13

are copied from earlier versions, thus, most likely have the same vulnerable code. When vulnerable code is fixed in the later versions, it should also be fixed in the earlier versions. This kind of patching is referred as *back-porting*.

Example 2. OpenSSL, an open source implementation of the SSL and TLS protocols, provides an API library named EVP as a high-level interface to its cryptographic functions. As described in EVP documentation, EVP has a protocol for signature verification, which could be used in the following procedure. First, `EVP_VerifyInit` is called to initialize a *verification context* object. Then, `EVP_VerifyUpdate` is used to hash the data for verification into that verification context. Finally, that data is verified against corresponding public key(s) via `EVP_VerifyFinal`. `EVP_VerifyFinal` would return one of **three** values: 1 if the data is verified to be correct; 0 if it is incorrect; and -1 *if there is any failure in the verification process*. However, the return value of -1 is overlooked by several developers. In other words, they might understand that, `EVP_VerifyFinal` would return only **two** values: 1 and 0 for correct and incorrect verification. Therefore, in several systems, the flaw state-

```
static int crypto_verify() {
...
   EVP_VerifyInit(&ctx, peer->digest);
   EVP_VerifyUpdate(&ctx, (u_char *)&ep->tstamp, vallen + 12);

   /* Vulnerable code: EVP\_VerifyFinal returns 1: correct, 0: incorrect, and -1: failure.
       This expression is false for both 1 and -1, thus, verification failure is mishandled
       as correct verification */
   if (!EVP_VerifyFinal(&ctx, (u_char *)&ep->pkt[i], siglen, pkey))
     return (XEVNT_SIG);
...
}
```

Figure 3.5 Recurring Vulnerability in NTP 4.2.5

```
int gale_crypto_verify_raw(...) {
...
   EVP_VerifyInit(&context,EVP_md5());
   EVP_VerifyUpdate(&context,data.p,data.l);
   for (i = 0; is_valid && i < key_count; ++i) {
   ... //Similarly vulnerable code
     if (!EVP_VerifyFinal(&context,sigs[i].p,sigs[i].l,key)) {
       crypto_i_error();
       is_valid = 0;
       goto cleanup;
     }
   }
   cleanup: EVP_PKEY_free(key);
}
```

Figure 3.6 Recurring Vulnerability in Gale 0.99

ment `if (!EVP_VerifyFinal(...))` is used to check for some error(s) in verification. Thus, when `EVP_VerifyFinal` returns -1, i.e. a failure occurs in the verification process, the control expression `(!EVP_VerifyFinal(...))` is false as in the case when 1 is returned. As a result, the program would behave as if the verification is correct, i.e. it is vulnerable to this exploitation.

From CVE-2009-0021 and CVE-2009-0047, this programming flaw appeared in two systems NTP and Gale using EVP library, and really caused a recurring vulnerability that *"allows remote attackers to bypass validation of the certificate chain via a malformed SSL/TLS signature for DSA and ECDSA keys"*. Figure 3.5 and Figure 3.6 show the corresponding vulnerable code that we found from NTP 4.2.5 and Gale 0.99. Despite detailed differences, both of them use the signature verification protocol provided by EVP and incorrectly process the return value of `EVP_VerifyFinal` by the aforementioned `if` statement.

We classify this vulnerability into Type 2, i.e. API-shared/reused recurring vulnerability (denoted

by **RV2**). The vulnerabilities of this type occur on the systems that share APIs/libraries. Generally, APIs should be used following a usage protocol specified by API designers. For example, the API functions must be called in the correct orders; the input/output provided to/returned from API function calls must be properly checked. However, developers could wrongly use such APIs, i.e. do not follow the intended protocols or specifications. They could call the functions in an incorrect order, miss an essential call, pass an unchecked/wrong-typed input parameter, or incorrectly handle the return values. Since they reuse the same library in different systems, they could make such similar erroneous usages, thus, create similar faulty code, and make their programs vulnerable to the same or similar vulnerabilities. Generally, each RV2 is related to a misused API function or protocol.

Example 3. Besides those two types, the other identified recurring vulnerabilities are classified as Type 3, denoted by **RV3**. Here is an example of type 3. According to CVE-2006-4339 and CVE-2006-7140, two systems OpenSSL 0.9.7 and Sun Solaris 9 *"when using an RSA key with exponent 3, removes PKCS-1 padding before generating a hash, which allows remote attackers to forge a PKCS #1 v1.5 signature that is signed by that RSA key and prevents libike from correctly verifying X.509 and other certificates that use PKCS #1"*. Those two systems realize the same RSA encryption algorithm, even though with different implementations. Unfortunately, the developers of both systems make the same mistake in their corresponding implementations of that algorithm, i.e. *"removes PKCS-1 padding before generating a hash"*, thus make both systems vulnerable to the same exploitation.

Generally, recurring vulnerabilities of Type 3 occur in the systems with the reuse of artifacts at a higher level of abstraction. For example, they could implement the same algorithms, specifications, or same designs to satisfy the same requirements. Then, if their developers made the same implementation mistakes, or the shared algorithms/specifications had some flaws, the corresponding systems would have the same or similar vulnerabilities. However, unlike Type 1 and Type 2, vulnerable code of Type 3 is harder to recognize/localize/match in those systems due to the wide varieties of implementation choices and differences in design, architecture, programming language, etc among systems.

Database	Report	Group	RV	RV1	RV2	RV3
NVD	2,598	151	143	74	36	33
oCERT	30	30	34	18	14	2
MFSA	103	77	77	77	0	0
ASF	234	59	59	59	0	0
TOTAL	2,965	299	313	228	50	35

Table 3.1 Recurring Software Vulnerabilities

3.3 Results and Implications

Table 3.1 summarizes the result of our study. Column `Report` shows the total number of vulnerability reports we collected and column `Group` shows the total number of groups of reports about recurring vulnerabilities we manually analyzed in each database. Column `RV` is the number of identified recurring vulnerabilities. The last three columns display the numbers for each type. The result confirms our hypotheses H1 and H2: there exist many recurring vulnerabilities in different systems (see column RV), and those vulnerabilities recur due to the reuse of source code (RV1), APIs/libraries (RV2), and other artifacts at higher levels of abstraction, e.g. algorithms, specifications (RV3). Note that, each group in oCERT contains only one report, however, each report in oCERT generally describes several vulnerabilities, and many of them are recurring. Thus, the number in column RV is larger than that in column Group.

The result also shows that the numbers of RV1s (source code-reused recurring vulnerabilities) and RV2s (API-reused) are considerable. All vulnerabilities reported on Mozilla and Apache are RV1 because Mozilla and Apache are two frameworks on which the systems in analysis are developed. Therefore, such systems share a large amount of code including vulnerable code fragments. Recurring vulnerabilities of Type 3 (RV3s) are less than RV1s and RV2s partly because the chance that developers make the same mistakes when implementing an algorithm might be less than the chance that they create a flaw in source code or misuse libraries in similar ways. Moreover, the systems sharing designs or algorithms might be not as many as the ones reusing source code and libraries.

Implications. The study confirms our hypotheses on recurring software vulnerabilities. Those vulnerabilities are classified in three types based on the artifacts that their systems reuse. This finding suggests that we could use the knowledge of prior known vulnerabilities in reported systems to detect

and resolve not-yet-reported vulnerabilities recurring in other systems/releases that reuse the related source code/libraries/algorithms, etc.

The study also provides some insights about the characteristics of vulnerable code of Types 1 and 2. While Type 1 vulnerable code is generally similar in texts and structure, Type 2 vulnerable code tends to have similar method calls, and similar input checking and output handling before and after such calls. Those insights are used in our detection and resolution of recurring vulnerabilities.

3.4 Threats to Validity

Public vulnerability databases used in our research could be incomplete because some of vulnerabilities are not disclosed or do not have patches available to the public yet. Since our empirical study is based on the reported vulnerabilities with available patches, it results might be affected. Furthermore, recurring vulnerabilities in our study are examined and identified by human beings. Therefore, the result could be biased due to human subjective views and mistakes.

CHAPTER 4. APPROACH OVERVIEW

We have developed SecureSync, an automatic tool to support the detection and resolution recommendation for RV1s and RV2s. The tool builds a knowledge base of the prior known/reported vulnerabilities and locates the vulnerable code fragments in a given system that are similar to the ones in its knowledge base. The working process of SecureSync is illustrated in Figure 4.1.

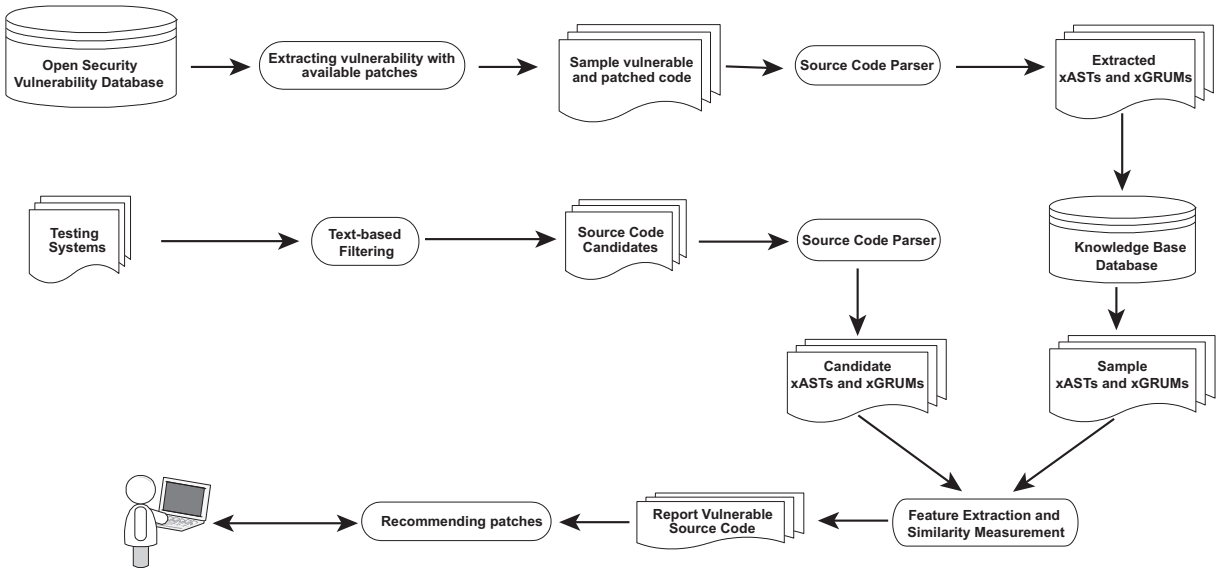

Figure 4.1 SecureSync's Working Process

In order to build the knowledge base, SecureSync searches for vulnerabilities with available patches in *Open Security Vulnerability Databases*. The vulnerable and patched samples are then extracted and stored in the *Knowledge Base Database* as xASTs and xGRUMs. When a testing system arrives, SecureSync performs text-based filtering to keep only the source files related to sample vulnerabilities (i.e similar identifiers/tokens). Then, these source files are parsed to build a set of xASTs and xGRUMs. Given candidates X and samples Y as a set of xASTs and xGRUMs extracted from the testing system

and the knowledge base respectively, SecureSync calculates the similarity between pairs of xASTs (Type 1) and pairs of xGRUMs (Type 2) from X and Y and reports the candidates which are sufficiently similar to the vulnerable sample and less similar to the corresponding patched one. SecureSync also helps developers by pointing out the specific locations of vulnerable code and providing corresponding patches derived from sample patches.

4.1 Problem Formulation

To build SecureSync, there are two main challenges: how to represent and measure the similarity of RV1s and RV2s, and how to localize the recurring ones in different systems. In SecureSync, we represent vulnerabilities via the *features* extracted from their vulnerable and patched code, and calculate the *similarity* of those vulnerabilities via such features. Feature extraction and similarity measure functions are defined differently for the detection of RV1s and RV2s, due to the differences in their characteristics. The problem of detecting recurring vulnerabilities is formulated as follows.

Definition 1 (Feature and Similarity) *Two functions $F()$ and $Sim()$ are called the feature extraction and similarity measure functions for the code fragments. $F(A)$ is called the feature set of a fragment A. $Sim(A, B)$ is the similarity measurement of two fragments A and B.*

Definition 2 (Recurring Vulnerable Code) *Given a vulnerable code fragment A and its corresponding patched code A'. If a code fragment B is sufficiently similar to A and less similar to A', i.e. $Sim(B, A) \geq \sigma$ and $Sim(B, A') < Sim(B, A)$, then B is considered as a recurring vulnerable code fragment of A. σ is a chosen threshold.*

A and A' could be similar because A' is modified from A. Thus, B could be similar to both A and A'. The second condition requires B to be more similar to vulnerable code than to patched code.

Definition 3 (Detecting Recurring Vulnerability) *Given a knowledge base as a set of vulnerable and patched code fragments $K = \{(A_1, A'_1), (A_2, A'_2), ..., (A_n, A'_n)\}$ and a program as a set of code fragments $P = \{B_1, B_2, ..., B_m\}$. Find fragment B_i(s) that is recurring vulnerable code of some fragment A_j(s).*

4.2 Algorithmic Solution and Techniques

The general process for detection of RV1s and RV2s is illustrated in Figure 4.2. First, SecureSync produces candidates fragments from the program P under investigation (line 2). Then, each candidate is compared against vulnerable and patched code of the vulnerabilities in the knowledge base K to find the recurring ones (lines 3-4). Detected vulnerabilities are reported to the users with the recommendation.

```
1  function Detect(P, K, σ) //detect recurring vulnerability
2    C = Candidates(P, K) //produce candidates
3    for each fragment B ∈ C: //check against knowledge base for recurring
4      if ∃(A, A') ∈ K : Sim(B, A) ≥ σ ∧ Sim(B, A') < Sim(B, A)
5        ReportAndRecommend(B)
```

Figure 4.2 Detection of Recurring Vulnerabilities

This algorithm requires the following techniques:

Feature Extraction and Similarity Measure. For RV1s, SecureSync uses a tree-based representation, called extended AST (xAST), that incorporates textual and structural features of code fragments. The similarity of fragments is computed based on the similarity of such trees via Exas, an approach for structural approximation and similarity measure of trees and graphs [44]. For RV2s, SecureSync uses a novel graph-based representation, called xGRUM. Each code fragment is represented as a graph, in which nodes represent function calls, variables, operators and branching points of control statements (e.g. `if`, `while`); and edges represent control/data dependencies between nodes. With this, SecureSync could represent the API usage information relevant to the orders of function calls, the checking of input or handling of output of function calls. Then, the similarity of code fragments is measured by the similarity of those xGRUMs based on their aligned nodes (Section 5).

Building Knowledge Base of Reported Vulnerabilities. We build the knowledge base for SecureSync using a semi-automated method. First, we access to vulnerability databases and manually analyze each report to choose vulnerabilities. Then, using code search, we find the corresponding vulnerable and patched code for the chosen vulnerabilities. We use SecureSync to automatically produce corresponding xASTs and xGRUMs from those collected code fragments as their features. Note that,

this knowledge building process could be fully automated if the vulnerability databases provide the information on the vulnerable and corresponding patched code.

Producing Candidate Code Fragments. After having functions for feature extraction, similarity measure, and the knowledge base, SecureSync produces code fragments from the program under investigation to find recurring vulnerable code using Definition 2. To improve the detection performance, SecureSync uses a text-based filtering technique to keep for further processing only the files having some tokens (i.e. words) identical or similar to the names of the functions in vulnerable code of the knowledge base.

Recommending Patches. For Type 1 with the nature of source code reuse, the patch in the knowledge base might be applicable to the detected vulnerable code with little modification. Thus, SecureSync does recommendation by pointing out the vulnerable statements and the sample patch taken from its knowledge base. For Type 2 with the nature of API usage, via its novel graph alignment algorithm, SecureSync suggests the addition of missed function calls, or the checking of input/output before/after calls, etc.

CHAPTER 5. SOFTWARE VULNERABILITY DETECTION

5.1 Type 1 Vulnerability Detection

5.1.1 Representation

To detect Type 1 recurring vulnerabilities, SecureSync represents code fragments, including vulnerable and patched code in its knowledge base, via an AST-like structure, which we call *extended AST* (xAST). An xAST is an augmented AST in which a node representing a function call, a variable, a literal, or an operator has its label containing the node's type, the signature, the data type, the value, or the token of the corresponding program entity. This labeling provides more semantic information for the tree (e.g. two nodes of the same type of function call but different labels would represent different calls). Figure 5.1 illustrates two xASTs of similar vulnerable statements in Figure 3.1 and Figure 5.2 represents the corresponding patch code. Two trees have mostly similar structures and nodes' labels, e.g. the nodes representing the function calls `CanExecuteScripts`, the variable of data type `nsresult`, etc. Therefore, they similarly have the same patch. (For simplicity, the node types or parameter lists are not drawn).

5.1.2 Feature Extraction and Similarity Measure

SecureSync considers feature set $F(A)$ of a code fragment A is a set of xASTs, each represents a statement of A. For example, vulnerable code fragments in Figure 3.1 have feature sets of sex and eight xASTs, respectively. Then, the similarity of two fragments is measured via the similarity of corresponding feature sets of xASTs.

SecureSync uses Exas [44] to approximate the xAST structures and measure their similarity. Using Exas, each xAST or a set of xASTs T is represented by a characteristic vector of occurrence-counts of

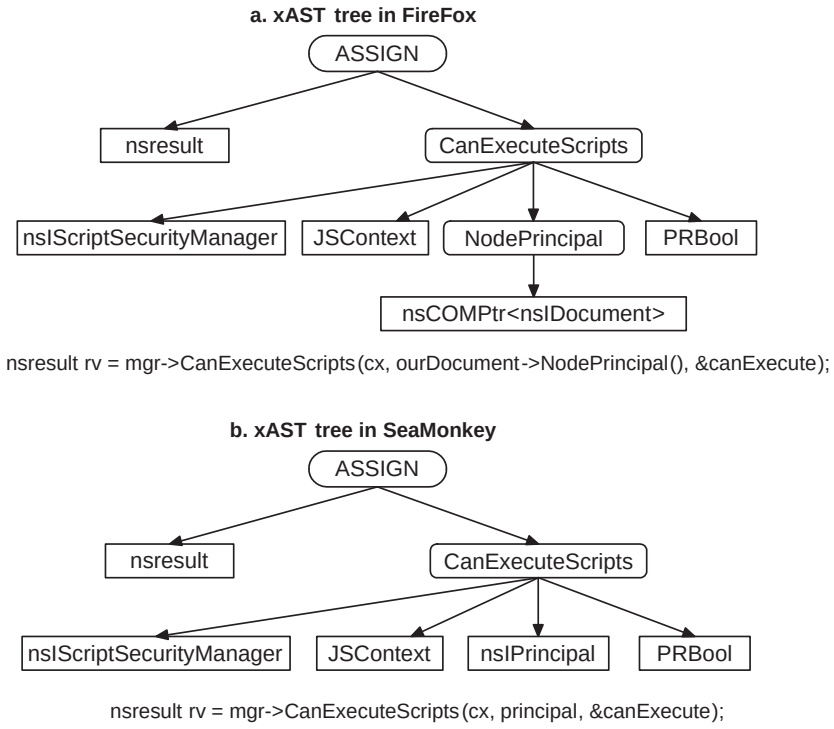

Figure 5.1 xAST from Code in Figure 3.1 and Figure 3.2

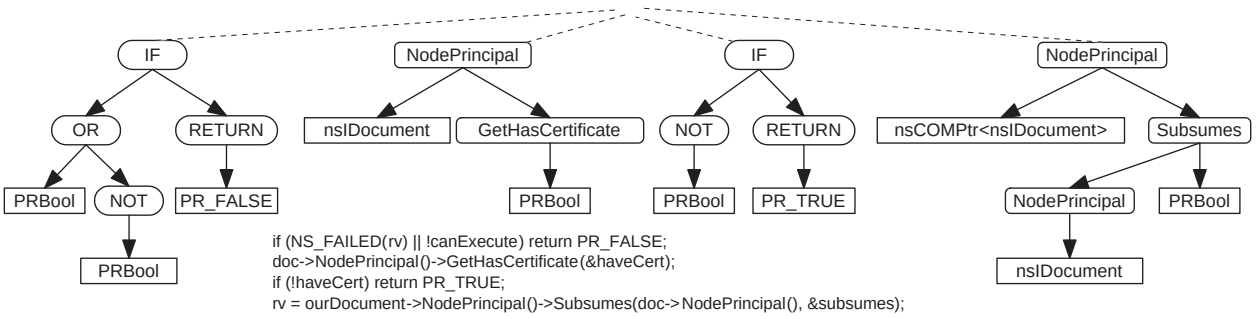

Figure 5.2 xAST from Patched Code in Figure 3.3 and Figure 3.4

its structural features $Ex(T)$. For trees, a structural feature is a sequence of labels of the nodes along a limited length path. For example, both trees in Figure 5.1 has a feature `[ASSIGN]-[nsresult]` and a feature `[ASSIGN]-[CanExecuteScripts]-[PRBool]`.

The similarity of two fragments are measured based on the Manhattan distance of two corresponding Exas vectors of their feature sets:

$$Sim(A, B) = 1 - \frac{|Ex(F(A)) - Ex(F(B))|}{|Ex(F(A))| + |Ex(F(B))|} \tag{5.1}$$

This formula normalizes vector distance with the vectors' sizes. Thus, with the same threshold for similarity, larger trees, which have larger vectors, are allowed to have more different vectors.

5.1.3 Candidate Searching

A vulnerable code fragment of Type 1 generally scatters in several *non-consecutive* statements (see Figure 3.1 and 3.2). Thus, traditional code clone detection techniques could not handle well such similar, non-consecutive fragments. To address that and to find the candidates of such vulnerable code fragments, SecureSync compares every statement of P to vulnerable code statements in its knowledge base and merges such statements into larger fragments. To improve searching, SecureSync uses two levels of filtering.

Text-based Filtering. Text-based filtering aims at filtering out the source files that do not have any code textually similar to vulnerable code in the knowledge base. For each file, SecureSync does lexical analysis, and only keeps the files that contain the tokens/words (e.g. identifiers, literals) identical or similar to the names in the vulnerable code (e.g. function/variable names, literals). This text-based filtering is highly effective. For example, in our evaluation, after filtering Mozilla Firefox with more than 6,000 source files, SecureSync keeps only about 100 files.

Structure-based Filtering. Structure-based filtering aims at keeping only the *statements* that potentially have similar xAST structures to the vulnerable ones in knowledge base. To do this, SecureSync uses locality-sensitive hashing (LSH) [20]. LSH scheme provides the hash codes for the vectors such that the more similar the two vectors are, the higher probability they would have the same hash code [20]. SecureSync first parses each source file kept from the previous step into an xAST.

Then, for each sub-tree representing a statement S in the file, it extracts an Exas feature vector $Ex(S)$. To check whether statement S is similar to a statement T in the knowledge base, SecureSync compares LSH hash codes of $Ex(S)$ with those of $Ex(T)$. If $Ex(S)$ and $Ex(T)$ have some common LSH-hash code, they are likely to be similar vectors, thus, S and T tend to have similar xAST structures. For faster processing, every statement T of the vulnerable code in knowledge base is pre-hashed into a hashing table. Thus, if a statement S does not share any hash code in that hash table, it will be disregarded.

Candidate Producing and Comparing. After previous steps, SecureSync has a set of *candidate statements* that potentially have similar xAST structures with some statement(s) in vulnerable code in its knowledge base. SecureSync now merges consecutive candidate statements into larger code fragments, generally at the method level. Then, candidate code fragments will be compared to vulnerable and patched code fragments in the knowledge base, using Definition 2 Section 4.1 and Formula 5.1. Based on the LSH hash table, SecureSync compares each candidate B with only the code fragment(s) A in the knowledge base that contain(s) the statements Ts correspondingly having some common LSH hash codes with the statements Ss of A.

5.1.4 Origin Analysis

When reusing source code, developers could make modifications to the identifiers/names of the code entities. For example, the function `ap_proxy_send_dir_filter` in Apache HTTP Server 2.0.x was renamed to `proxy_send_dir_filter` in Apache HTTP Server 2.2.x. Because the features of xASTs rely on names, such renaming could affect the comparison of code fragments in different systems/versions. This problem is addressed by an origin analysis process that provides the name mapping between two versions of a system or two code-sharing systems. Using such mapping, when producing the features of xASTs for candidate code fragments, SecureSync uses the mapped names, instead of the names in the candidate code, thus, avoids the renaming problem. To map the names from such two versions, currently, SecureSync uses an origin analysis technique in our prior work, OperV [47]. OperV models each software system by a project tree where each of its nodes corresponds to a program element such as package, file, function, or statement. Then, it does origin analysis using a tree alignment algorithm that compares two project trees based on the similarity of the sub-trees and provides the

mapping of the nodes. Using OperV, SecureSync knows the mapped entities thus, is able to produce the name mapping that is used in feature extraction.

5.2 Type 2 Vulnerability Detection

5.2.1 Representation

Type 2 vulnerabilities are caused by the misuse or mishandling of APIs. Therefore, we emphasize on API usages to detect such recurring vulnerabilities. If a candidate code fragment B has similar API usages to a vulnerable code fragment A, B is likely to have a recurring vulnerability with A. In SecureSync, API usages are represented as graph-based models, in which nodes represent the usages of API function calls, data structures, and control structures, and edges represent the relations or control/data dependencies between them. Our graph-based representation for API usages, called ***Ex**tended **GR**aph-based **U**sage **M**odel* (xGRUM), is as follows:

Definition 4 (xGRUM) *Each extended graph-based usage model is a directed, labeled, acyclic graph in which:*

1. Each action node represents a function or method call;

2. Each data node represents a variable;

3. Each control node represents the branching point of a control structure (e.g. `if, for, while, switch`*);*

4. Each operator node represents an operator (e.g. `not, and, or`*);*

5. An edge connecting two nodes x and y represents the control and data dependencies between x and y; and

6. The label of an action, data, control, and operator node is the name, data type, or token of the corresponding function, variable, control structure, or operator, along with the type of the corresponding node.

The rationale behind this representation is as follows. The usage of an API function or data structure is represented as an action or data node. The order of two API function calls, e.g. x must be called before y, is represented by an edge connecting the action nodes corresponding to the calls to x and

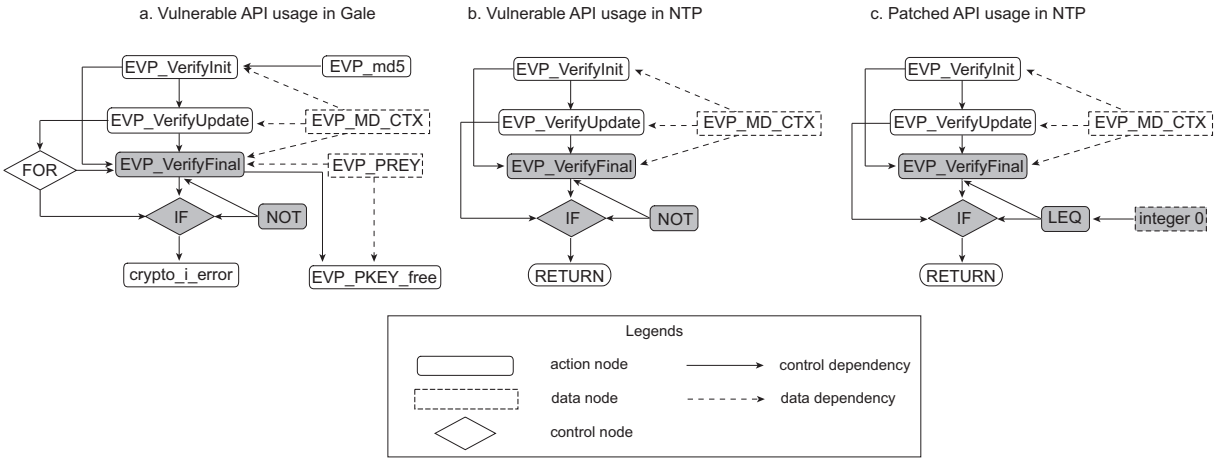

Figure 5.3 xGRUMs from Vulnerable and Patched Code in Figure 3.5

to y. The checking of input or output of API functions is modeled via the control and operator nodes surrounding the action nodes of those function calls and via the edges between such control, operator and action nodes.

Figure 5.3 partially shows three xGRUMs of two vulnerable code fragments in two figures 3.5 and 3.6 and one patched code fragment. (For better understanding, only the nodes/edges related to API usages and the vulnerabilities are drawn). The xGRUMs have the action nodes representing function calls `EVP_VerifyFinal`, `EVP_VerifyInit`, data node `EVP_MD_CTX`, control nodes `IF`, `FOR`, and operator nodes `NOT`, `LEQ`. An edge from `EVP_VerifyUpdate` to `EVP_VerifyFinal` represents both their *control dependency*, (i.e. `EVP_VerifyUpdate` is used before `EVP_VerifyFinal`), and the *data dependency*: those two nodes share data via data node `EVP_MD_CTX`. The edge between `EVP_MD_CTX` and `EVP_VerifyFinal` shows that the corresponding variable is used as an input for `EVP_VerifyFinal` (as well as an output, since the variable is a reference). The edge from action node `EVP_VerifyFinal` to control node `IF` shows the control dependency: `EVP_VerifyFinal` is called before the branching point of that `if` statement. That is, condition checking occurs after the call.

Especially, operator node `NOT` represents the operator in the control expression `!EVP_VerifyFinal(...)`. It has control dependencies with two nodes `EVP_VerifyFinal` and `IF`. In Figure 5.3c, the control expression is modified into `EVP_VerifyFinal(...)<=`. Thus, that operator node `NOT` is replaced by the operator `LEQ` and the data node `integer 0` is added for the literal value zero. A literal is

modeled as a special data node, and its label is formed by its type and value. SecureSync models only the literals of supported primitive data types (e.g. `integer`, `float`, `char`) and special values (e.g. 0, 1, -1, null, empty string). Prior work [24, 34] showed that bugs often occur at condition checking points of such special values. Currently, SecureSync uses intra-procedural data analysis to find the data dependencies between graph nodes. For example, the data dependency between `EVP_VerifyInit`, `EVP_VerifyUpdate`, and `EVP_VerifyFinal` are found via their connections to the data node `EVP_MD_CTX`.

5.2.2 Feature Extraction and Similarity Measure

Each vulnerable code fragment A is represented by an xGRUM G. Extracted features of A represent the nodes of G that are relevant to misused APIs. Note that, not all nodes in G is relevant to the misused API functions. For example, in Example 2, only `EVP_VerifyFinal` is misused, `EVP_VerifyInit` and `EVP_VerifyUpdate` are correctly used. Thus, the feature of the vulnerability should emphasize on the action node `EVP_VerifyFinal`, operator node `NOT`, control node `IF`, and data node `EVP_MD_CTX`, and of course, on the edges, i.e. the control and data dependencies between such nodes.

For RV2s, features are extracted from the comparison between two xGRUMs representing the vulnerable and patched code, respectively. SecureSync finds the nodes related to misused APIs based on the idea that: if some program entities are related to the bug, they should be changed/affected by the fix. Since SecureSync represents program entities and dependencies via labels and edges, changed/affected entities are represented by the nodes having different labels or neighborhoods, or being added/deleted. That is, the *unchanged nodes* between two xGRUMs of the vulnerable and patched code represent the entities *irrelevant* to API misuse. Thus, *sub-graphs* containing changed nodes of those two xGRUMs are considered as *features* of the corresponding vulnerability. To find the changed and unchanged nodes, SecureSync uses the following approximate graph alignment algorithm Figure 5.4.

```
1  function Align(G, G', μ) //align and differ two usage models
2    for all u ∈ G, v ∈ G' //calculate similarity of all nodes.
3      if label(u) = label(v)
4        sim(u, v) = 1 − |N(u) − N(v)|/(|N(u)| + |N(v)|)
5    M = MaximumWeightedMatching(U, U', sim) //matching
6    for each (u, v) ∈ M:
7      if sim(u, v) < μ then M.remove((u, v)) //remove too low matches
8  return M
```

Figure 5.4 Graph Alignment Algorithm

5.2.2.1 Graph Alignment Algorithm

This algorithm aligns (i.e. maps) the nodes between two xGRUMs G and G' based on their labels and neighborhoods, then the aligned nodes could be considered as unchanged nodes and not affected by the patch. The detailed algorithm is shown in Figure 5.4. For each node u into a graph, SecureSync extracts an Exas vector $N(u)$ to represent the neighborhood of u. The similarity of two nodes $u \in G$ and $v \in G'$, $sim(u, v)$, is calculated based on the vector distance of $N(u)$ and $N(v)$ as in Formula 5.1 if they have the same label (see lines 2-4), otherwise they have zero similarity. Then, the maximum weighted matching with such similarity as weights is computed (line 5). Only matched nodes with sufficiently high similarity are kept (lines 6-7) and returned as aligned nodes (line 8).

5.2.2.2 Feature Extraction and Similarity Measure

Using that algorithm, SecureSync extracts features as follows. It first parses the vulnerable and corresponding patched code fragments A and A' into two xGRUMs G and G'. Then, it runs the graph alignment algorithm to find the aligned nodes and considered them as unchanged. Unaligned nodes are considered as changed, and the subgraphs formed by such nodes in G and G' are put into the feature sets $F(A)$ and $F(A')$ for the current vulnerability.

Let us examine the code in Figure 3.5 and 3.6. Figure 5.3b and Figure 5.3c display the xGRUMs G and G' of vulnerable code and patched code fragments in NTP. The neighborhood structures of two nodes labeled `EVP_VerifyInit` in two graphs are identical, thus, they are 100% similar. The similarity of two nodes labeled `EVP_VerifyFinal` is less because they have different neighborhood structures (one has a neighbor node `NOT`, one has `LEQ`). Therefore, after maximum matching, those

two `EVP_VerifyInit` nodes are aligned, however, the nodes `EVP_VerifyFinal` and other nodes representing operators `NOT`, `LEQ` and literal `integer 0` are considered as *changed* nodes. Then, each feature set $F(A)$ and $F(A')$ contains the corresponding sub-graph with those changed nodes in gray color in Figures 5.3b and 5.3c.

Similarity Measure. Given a code fragment B with the corresponding xGRUM H. SecureSync measures the similarity of B against A in the database based on the usages of API functions that are (mis)used in A and (re)used in B. To find such API usages, SecureSync aligns H and $F(A)$ which contains the changed nodes representing the entities related to the misused API functions in A. This alignment also uses the aforementioned graph alignment algorithm with a smaller similarity threshold μ because the difference between B and A might be larger than that of A' and A.

Assume that the sets of aligned nodes are $M(A)$ and $M(B)$. SecureSync builds two xGRUMs $U(A)$ and $U(B)$ containing the nodes in $M(A)$ and $M(B)$ as well as their dependent nodes and edges in G and H, respectively. Since $M(A)$ and $M(B)$ contain the nodes related to API functions that are (mis)used in A and are (re)used in B, $U(A)$ and $U(B)$ will represent the corresponding API usages in A and in B. Then, the similarity of A and B are measured based on the similarity of $U(A)$ and $U(B)$:

$$Sim(A, B) = 1 - \frac{|Ex(U(A)) - Ex(U(B))|}{|Ex(U(A))| + |Ex(U(B))|} \tag{5.2}$$

This formula is in fact similar to Formula 5.1. The only different is that $Ex(U(A))$ and $Ex(U(B))$ are Exas vectors of two xGRUMs, not xASTs. In Figures 5.3a and 5.3b, $M(A)$ and $M(B)$ will contain the action node `EVP_VerifyFinal`, operator node `NOT` and control node `IF`. Then, $U(A)$ and $U(B)$ will be formed from them and their data/control-dependent nodes, such as `EVP_VerifyInit`, `EVP_VerifyUpdate`, and `EVP_MD_CTX`. In Figure 5.3a, nodes `EVP_PKEY_free`, `FOR`, `EVP_PKEY`, and `crypto_i_error` are also included in $U(A)$. Their similarity calculated based on Formula 5.2 is 90%.

5.2.3 Candidate Searching

Similarly to the detection of RV1s, SecureSync uses origin analysis to find renaming of API functions between systems and versions. Then, it also uses text-based filtering and set-based filtering to keep only source files and xGRUMs that contain tokens and names similar to misused API functions stored as the features in its database. After such filterings, SecureSync has a set of xGRUMs that

potentially contain similarly misused API functions with some xGRUMs in the vulnerable code in its database. Then, the candidate xGRUMs of code fragments are compared to xGRUMs of vulnerable and patched code fragments in the database, using Definition 2 and Formula 5.2. Matched candidates are recommended for patching.

CHAPTER 6. EMPIRICAL EVALUATION

This chapter presents our evaluation of SecureSync on real-world software systems and vulnerabilities. The evaluation is separated into two experiments for the detection of type 1 and type 2 vulnerabilities. Each experiment has three steps: 1) selecting of vulnerabilities and systems for building knowledge base, 2) investigating and running, and 3) analyzing results. Let us describe the details of each experiment.

6.1 Evaluation of Type 1 Vulnerability Detection

Selecting. We chose three Mozilla-based open-source systems FireFox, Thunderbird and SeaMonkey for the evaluation of type 1 vulnerabilities because they are actively developed and maintained, and have available source code, security reports, forums, and discussions. First, we contacted and obtained the release and branch history of those three systems from Mozilla security team. For each system, we chose a range of releases that are currently maintained and supported on security updates, with the total of 51 releases for three systems. The numbers of releases of each system is shown in column `Release` of Table 6.1.

We aimed to evaluate how SecureSync uses the knowledge base of vulnerabilities built from the reports in some releases of FireFox to detect the recurring ones in Thunderbird and SeaMonkey, and also in different FireFox's release branches in which those vulnerabilities are not reported yet. Thus, we selected 48 vulnerabilities reported in the chosen releases of FireFox with publicly available vulnerable code and corresponding patched code to build the knowledge base for SecureSync. In the cases that a vulnerability occurred and was patched in several releases of FireFox, i.e., there were several pairs of vulnerable and patched code fragments, we chose only one pair to build its features in the database.

Running. With the knowledge base of those 48 vulnerabilities, we run SecureSync on 51 chosen

Systems	Release	DB report	SS report	✓ in DB	✓ new	X	Miss in DB
ThunderBird	12	21	33	19	11	3	2
SeaMonkey	10	28	39	26	10	3	2
FireFox	29	14	22	14	5	3	0
TOTAL	51	63	94	59	26	9	4

Table 6.1 Recurring Vulnerability Type 1 Detection Evaluation

releases. For each release, SecureSync reported the locations of vulnerable code (if any). We analyzed those results and considered a vulnerability v to be *correctly detected* in a release r if either 1) v is officially reported about r; or 2) the code locations of r reported by SecureSync have the same or highly similar programming flaws to the vulnerable code of v. We also sent the reports from SecureSync to Mozilla security team for their confirmation. We did not count the cases when a vulnerability is reported on the release or branch from which the vulnerable and patched code are used in the database since a vulnerability is considered recurring if it occurs on different release branches.

Analyzing. Table 6.1 shows the analysis result. There are 21 vulnerabilities which had been officially reported by MFSA [8] and verified by us as truly recurrings on Thunderbird (see column `DB report`). However, SecureSync reports 33 RV1s (column `SS report`). The manual analysis confirms that 19 of them (see ✓`in DB`) were in fact officially reported (i.e. coverage of 19/21 = 90%) and that 11 RV1s are *not-yet-reported* and *newly discovered* ones (see ✓`new`). Thus, three cases are incorrectly reported (column `X`) and two are missed (`Miss in DB`), given the precision of 91% (30/33). The results on SeaMonkey are even better: coverage of 93% (26/28) and precision of 92% (36/39). The detection of RV1 on different branches of FireFox is also quite good: coverage of 100% (14/14) and precision of 86% (19/22).

The result shows that SecureSync is able to detect RV1s with high accuracy. Most importantly, it is able to correctly detect the total of 26 *not-yet-reported* vulnerabilities in three subject systems. Figure 6.1 shows a vulnerable code fragment in Thunderbird 2.0.17 as an example of such not-yet reported and patched vulnerabilities. Note that, this one is the same vulnerability presented in Example 1. However, it is reported in CVE-2008-5023 for only FireFox and SeaMonkey and now it is revealed by SecureSync on Thunderbird. Based on the recommendation from our tool, we had produced a patch

```
PRBool nsXBLBinding::AllowScripts() {
...
  JSContext* cx = (JSContext*) context->GetNativeContext();
  nsCOMPtr<nsIDocument> ourDocument;
  mPrototypeBinding->XBLDocumentInfo()->GetDocument(getter_AddRefs(ourDocument));

  nsIPrincipal* principal = ourDocument->GetPrincipal();
  if (!principal) return PR_FALSE;

  PRBool canExecute;
  nsresult rv = mgr->CanExecuteScripts(cx, principal, &canExecute);
  return NS_SUCCEEDED(rv) && canExecute; ...
}
```

Figure 6.1 Vulnerable Code in Thunderbird 2.0.17

and sent it to Mozilla security team. They had kindly confirmed this programming flaw and our provided patch. We are waiting for their confirmation on other vulnerabilities reported by SecureSync and corresponding patches that we built based on its fixing recommendation.

6.2 Evaluation of Type 2 Vulnerability Detection

Selecting. Out of 50 RV2s identified in our empirical study, some have no publicly available source code (e.g. commercial software), and some have no available patches. We found available patches (vulnerable and corresponding patched code) for only 12 RV2s and used all of them to build the knowledge base for SecureSync in this experiment. For each of those 12 RV2s, if it is related to an API function m, we used Google Code Search to find all systems using m and randomly chose 1-2 releases of each system from the result returned by Google Code Search (it could return several systems using m, and several releases for each system). Some of those releases have been officially reported to have the RV2s in knowledge base, and some have not. However, we did not select the releases containing the vulnerable and patched code that we already used for building the knowledge base. Thus, in total, we selected 12 RV2s, 116 different systems, with 125 releases for this experiment.

Running and Analyzing. The running and analyzing is similar to the experiment for RV1s. Table 6.2 shows the analysis result. For example, there is an RV2 related to the misuse of two functions `seteuid` and `setuid` in ftpd and ksu programs which "...do not check return codes for setuid calls, which might allow local users to gain privileges by causing setuid to fail to drop privileges" (CVE-

API function related to vulnerability	System	Release	DB report	SS report	✓ in DB	✓ new	X	Miss in DB
seteuid/setuid	42	46	4	28	3	20	5	1
ftpd	21	23	0	19	0	12	7	0
gmalloc	10	10	3	10	3	7	0	0
ObjectStream	7	8	3	6	3	3	0	0
EVP_VerifyFinal	7	8	5	7	5	2	0	0
DSA_verify	7	8	3	8	3	4	1	0
libcurl	7	7	3	7	3	4	0	0
RSA_public_decrypt	5	5	1	5	1	4	0	0
ReadSetOfCurves	4	4	1	4	1	3	0	0
DSA_do_verify	3	3	1	2	0	2	0	1
ECDSA_verify	2	2	0	2	0	2	0	0
ECDSA_do_verify	1	1	0	1	0	1	0	0
TOTAL	116	125	24	99	22	64	13	2

Table 6.2 Recurring Vulnerability Type 2 Detection Evaluation

2006-3084). We found 46 releases of 42 different systems using those two functions. 4 out of 46 are officially reported in CVE-2006-3083 and CVE-2006-3084 (column `DB report`). In the experiment, SecureSync reports 28 out of 46 releases vulnerable. Manually checking confirms 23/28 to be correct and 5 are incorrect (giving precision of 82%). In the 23 correctly reported vulnerabilities, 3 are officially reported and 20 others are *not-yet-reported*. SecureSync missed only one officially reported case. Similarly, for the RV2 related to API `ftpd`, it correctly detected 12 unreported releases and wrongly reported on 7 releases. For other RV2s, it detects correctly in almost all releases.

Manual analyzing of all the cases that SecureSync missed, we found that they are due to the data analysis. Currently, the implementation of data analysis in SecureSync is restricted to intra-procedural. Therefore, it misses the cases when checking/handling of inputs/outputs for API function calls is processed in different functions. For the cases that SecureSync incorrectly detected, we found the problem is mostly due to the chosen threshold. In this experiment, we chose $\sigma = 0.8$. When we chose $\sigma = 0.9$, for the RV2 related to `ftpd`, the number of wrongly detected cases reduces from 7 to 3, however, the number of correctly detected cases also reduces from 12 to 10. However, the results still show that SecureSync is useful, and it could be improved with more powerful data analysis.

Interesting Examples. Here are some interesting cases on which SecureSync correctly detected

```
static int ssl3_get_key_exchange(s){
...
 if (pkey->type == EVP_PKEY_DSA){
  /* lets do DSS */
  EVP_VerifyInit(&md_ctx,EVP_dss1());
  EVP_VerifyUpdate(&md_ctx,&(s->s3->client_random[0]),SSL3_RANDOM_SIZE);
  EVP_VerifyUpdate(&md_ctx,&(s->s3->server_random[0]),SSL3_RANDOM_SIZE);
  EVP_VerifyUpdate(&md_ctx,param,param_len);
  if (!EVP_VerifyFinal(&md_ctx,p,(int)n,pkey)){
    /* bad signature */
    al=SSL3_AD_ILLEGAL_PARAMETER;
    SSLerr(SSL_F_SSL3_GET_KEY_EXCHANGE,SSL_R_BAD_SIGNATURE);
    goto f_err;
  }
 }
}
```

Figure 6.2 Vulnerable Code in Arronwork 1.2

```
gchar *g_base64_encode (const guchar *data, gsize len) {
  gchar *out;
  gint state = 0, save = 0, outlen;
  g_return_val_if_fail (data != NULL, NULL);
  g_return_val_if_fail (len > 0, NULL);

- g_malloc (len * 4 / 3 + 4);

+ if (len >= ((G_MAXSIZE - 1) / 4 - 1) * 3)
+   g_error("Input␣too␣large␣for␣Base64␣encoding␣"...);

+ out = g_malloc ((len / 3 + 1) * 4 + 1);

  outlen = g_base64_encode_step (data, len, FALSE, out, &state, &save);
  outlen += g_base64_encode_close (FALSE, out + outlen, &state, &save);
  out[outlen] = '\0';
  return (gchar *) out;
}
```

Figure 6.3 Vulnerable and Patched Code in GLib 2.12.3

not-yet-reported RV2s. Figure 6.2 illustrates a code fragment in Arronwork having the same vulnerability related to the incorrect usage of `EVP_VerifyFinal` function as described in Chapter 3, and to the best of our knowledge, it has not been reported anywhere. The code in Arronwork has different details from the code in NTP (which we chose in building knowledge base). For example, there are different variables and function calls, and `EVP_VerifyUpdate` is called three times, instead of one. However, it uses the same EVP protocol and has the same flaw. Using the recommendation from SecureSync to change the operator and expression related to the function call to `EVP_VerifyFinal`, we derived a patch for it and reported this case to Arron's developers.

```
guchar * seahorse_base64_decode (const gchar *text, gsize *out_len) {
  guchar *ret;
  gint inlen, state = 0, save = 0;
  inlen = strlen (text);
  ret = g_malloc0 (inlen * 3 / 4);
  *out_len = seahorse_base64_decode_step (text, inlen, ret, &state, &save);
  return ret;
}
```

```
gchar * seahorse_base64_encode (const guchar *data, gsize len) {
  gchar *out;
  gint state = 0, outlen, save = 0;
  out = g_malloc (len * 4 / 3 + 4);
  outlen = seahorse_base64_encode_step (data, len, FALSE, out, &state, &save);
  outlen += seahorse_base64_encode_close (FALSE, out + outlen, &state, &save);
  out[outlen] = '\0';
  return (gchar *) out;
}
```

Figure 6.4 Vulnerable Code in SeaHorse 1.0.1

Here is another interesting example. CVE-2008-4316 reported *"Multiple integer overflows in glib/gbase64.c in GLib before 2.20 allow context-dependent attackers to execute arbitrary code via a long string that is converted either (1) from or (2) to a base64 representation"*. The vulnerable and patched code in GLib is in Figure 6.3, the new and removed code are marked with symbols "+" and "-", respectively. This vulnerability is related to the misuse of function `g_malloc` for memory allocation with a parameter that is *unchecked* against the amount of available memory (`len>=((G_MAXSIZE-1)/4-1)*3)`), and against an integer overflow in the expression (`len*4/3+4`).

Using this patch, SecureSync is able to detect a similar flaw in SeaHorse system in which two functions `base64` encoder and decoder incorrectly use `g_malloc` and `g_malloc0` (see Figure 6.4). The interesting point is that, the API function names in two systems are just similar, but not identical (e.g. `g_malloc0` and `g_malloc`, `g_base64_*` and `seahorse_base64_*`). Thus, the origin analysis information SecureSync uses is helpful for this correct detection. Using the alignment between `g_malloc` and `g_malloc0` when comparing the graphs of two methods in Figure 6.4 with that of the method in Figure 6.3, SecureSync correctly suggests the fixing by adding the `if` statement before the calls to `g_malloc` and `g_malloc0` functions, respectively.

API-related	Systems	Releases	SS report	Correct	Incorrect	Precision
seteuid/setuid	42	46	28	23	5	82
gmalloc	10	10	12	11	1	92
ftpd	21	23	19	12	7	63
ObjectStream	7	8	6	6	0	100
EVP_VerifyFinal	7	8	7	7	0	100
DSA_verify	7	8	8	7	1	88
libcurl	7	7	7	7	0	100
RSA_public_decrypt	5	5	5	5	0	100
ReadSetOfCurves	4	4	8	8	0	100
DSA_do_verify	3	3	2	2	0	100
ECDSA_verify	2	2	2	2	0	100
ECDSA_do_verify	1	1	1	1	0	100
Total	116	125	105	91	14	

Table 6.3 Recurring Vulnerability Type 2 Recommendation

6.3 Patching Recommendation

SecureSync could suggest developers fixing code by applying the tree edit operations to transform the xAST of buggy code into the patched one. As in Example 1 Section 3.2, after SecureSync detects the recurring vulnerability in `AllowScripts` function in Thunderbird system, it compares two xASTs of buggy and patched code to detect that the patch adds the function calls for privilege checking and a change in the return statement. Therefore, it suggests to add `GetHasCertificate()` and `Subsumes`, and to replace the return variable `canExecute` of the return statement with `subsumes` variable. For Type 2, SecureSync provides the operations related to API function calls for developers to fix API misuses. For example, in Figure 6.3 and Figure 6.4, SecureSync detects the changes in `g_malloc` and `g_malloc0` functions when comparing the graphs of two methods `seahorse_base64_decode` and `seahorse_base64_encode` with that of `g_base64_encode` method. It correctly suggests fixing by adding the `if` statement before calling `g_malloc` and `g_malloc0` functions.

Table 6.3 shows the recommendation result for type 2 vulnerabilities. For each testing system, SecureSync not only checks whether it is vulnerable, but also point out the locations of vulnerable code with proposed patches. For example, there is a vulnerability related to the misuse of API ReadSetOfCurves. Among 4 releases of testing systems, SecureSync detect 8 locations (column `SS`

`report`) containing vulnerable code. Manually checking confirms all of them correct (column `Correct`), thus giving the precision 100% (column `Precision`).

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

This thesis reports an empirical study on recurring software vulnerabilities. The study shows that there exist many vulnerabilities recurring in different systems due to the reuse of source code, APIs, and artifacts at higher levels of abstraction (e.g. specifications). We also introduce an automatic tool to detect such recurring vulnerabilities on different systems. The core of SecureSync includes two techniques for modeling and matching vulnerable code across different systems. The evaluation on real-world software vulnerabilities and systems shows that SecureSync is able to detect recurring vulnerabilities with high accuracy and to identify several vulnerable code locations that are not yet reported or fixed even in mature systems. A couple of detected ones were confirmed by developers.

Future Work. We want to extend SecureSync approach to build a framework that incorporates the knowledge from vulnerability reports and vulnerable source code to better detect recurring vulnerabilities. In detail, the core of SecureSync will include a usage model and a mapping algorithm for matching vulnerable code across different systems, a model for the comparison of vulnerability reports, and a tracing technique from a report to corresponding source code [50]. In other words, we will extend SecureSync that:

1. Represents and compares the vulnerability reports to identify the ones that report the recurring/similar vulnerabilities,

2. Traces from a vulnerability report to the corresponding source code fragment(s) in the codebase,

3. Represents and compares code fragments to find the ones that are similar due to code reuse or similar in API library usages.

Figure 7.1 illustrates our framework. Given a system S_1 with source code C_1 and a known security vulnerability reported by R_1. The framework can support two following scenarios:

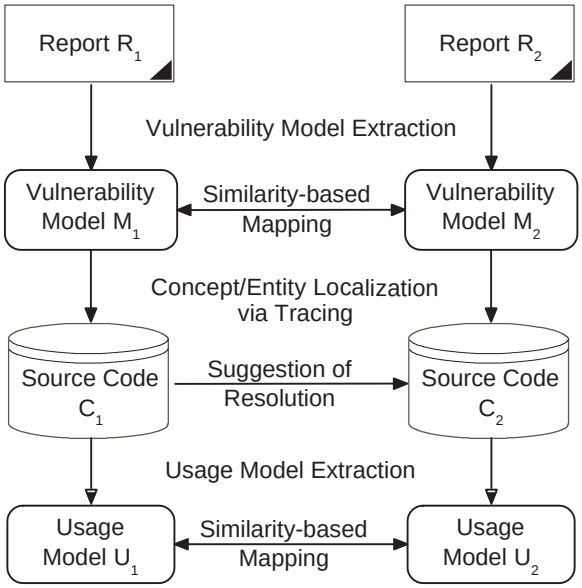

Figure 7.1 The SecureSync Framework

Scenario 1. Given a system S_2, one needs to determines whether S_2 potentially has a recurring/similar vulnerability as S_1 and point out the potential buggy code C_2. In the case that S_2 is a different version of S_1, the problem is referred to as *back-porting*. In general, due the difference in the usage contexts in two systems S_1 and S_2, the buggy code C_1 and C_2 might be different.

From R_1, a vulnerability model M_1 is built to describe the vulnerability of S_1. Then, the trace from R_1 will help to find the corresponding source code fragments C_1, which are used to extract the usage model U_1. If the tracing link is not available, SecureSync extends a traceability link recovery method, called incremental Latent Semantic Indexing (iLSI) [33], that we developed in prior work. From usage model U_1, SecureSync uses its usage clone detection algorithm (will be discussed later) to find code fragments C_2 with the usage U_2 similar to U_1. Those C_2 fragments are considered as potential buggy code that could cause a recurring/similar vulnerability as in S_1. The suggested patch for code in S_2 is derived from the comparison between U_1 and its patched U_1'. The change from U_1 to U_1' will be applied to U_2 (which is similar to U_1). Then, the concrete code will be derived to suggest the fix to C_2.

Scenario 2.Provided that R_2 is reported on S_2, SecureSync compares vulnerability models extracted from security reports. First, SecureSync extracts M_2 from R_2 and then searches for a vulnerability model M_1 in the security database that is similar to M_2. If such M_1 exists, SecureSync will

identify the corresponding system S_1, the patch, and then map the code fragments and recommend the patch in the similar manner as in scenario 1.

APPENDIX A. ADDITIONAL TECHNIQUES USED IN SECURESYNC

There are two techniques SecureSync used to calculate graph similarity and improve its perfomance. It is Exas - an approach previously developed by us to approximate and capture structure information of labeled graphs by vectors and measure the similarity of such graphs via vector distance. SecureSync also uses the hashing technique called Locality Sensitive Hashing to filter labeled trees with similar structure.

Exas: A Structural Characteristic Feature Extraction Approach

Structure-oriented Representation. In our structure-oriented representation approach, a software artifact is modeled as a *labeled, directed graph* (tree is a special case of graph), denoted as $G = (V, E, L)$. V is the set of nodes in which a node represents an element within an artifact. E is the set of edges in which each edge between two nodes models their relationship. L is a function that maps each node/edge to a label that describes its attributes. For example, for ASTs, node types could be used as nodes' labels. For Simulink models, the label of a node could be the type of its corresponding block. Other attributes could also be encoded within labels. In existing clone detection approaches, labels for edges are rarely explored. However, for general applicability, Exas supports the labels for both nodes and edges.

Figure A.1 shows an illustrated example of a Simulink model, its representation graph and two cloned fragments A and B.

Structural Feature Selection. Exas focuses on two kinds of *patterns* of structural information of the graph, called (p, q)*-node* and n*-path*.

A (p, q)**-node** is a node having p incoming and q outgoing edges. The values of p and q associated to a certain node might be different in different examined fragments. For example, node 9 in Figure A.1

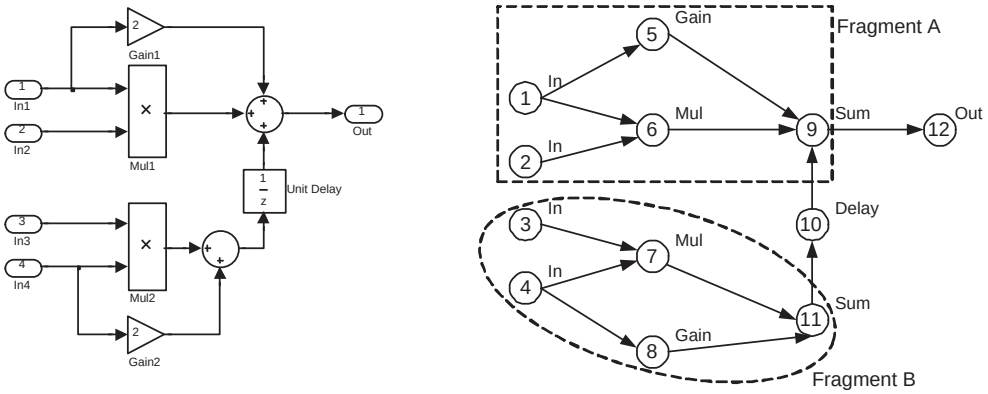

Figure A.1 The Simulink Model and Graph Representation

is a (3,1)-node if entire graph is currently considered as a fragment, but is a (2,0)-node if fragment A is examined.

An n**-path** is a directed path of n nodes, i.e. a sequence of n nodes in which any two consecutive nodes are connected by a directed edge in the graph. A special case is 1-path which contains only one node.

Structural feature of a (p, q)-node is the label of the node along with two numbers p and q. For example, node 6 in fragment A is $(2, 1)$-node and gives the feature `mul-2-1`. *Structural feature* of an n-path is a sequence of labels of nodes and edges in the path. For example, the 3-path 1-5-9 gives the feature `in-gain-sum`. Table A.1 lists all patterns and features extracted from A and B. It shows that both fragments have the same feature set and the same number of each feature. Later, we will show that it holds for all isomorphic fragments.

Characteristic Vectors. An efficient way to express the property *"having the same or similar features"* is the use of vectors. The characteristic vector of a fragment is the occurrence-count vector of its features. That is, each position in the vector is indexed for a feature and the value at that position is the number of occurrences of that feature in the fragment. Table A.2 shows the indexes of the features, which are global across all vectors, and their occurrence counts in fragment A.

Two fragments having the same feature sets and occurrence counts will have the same vectors and vice versa. The vector similarity can be measured by an appreciably chosen vector distance such as

Pattern	Features of fragment A					Features of fragment B				
1-path	1 in	2 in	5 gain	6 mul	9 sum	4 in	3 in	8 gain	7 mul	11 sum
2-path	1-5 in-gain	1-6 in-mul	2-6 in-mul	6-9 mul-sum	5-9 gain-sum	4-8 in-gain	4-7 in-mul	3-7 in-mul	7-11 mul-sum	8-11 gain-sum
3-path	1-5-9 in-gain-sum		1-6-9 in-mul-sum		2-6-9 in-mul-sum	4-8-11 in-gain-sum		4-7-11 in-mul-sum		3-7-11 in-mul-sum
(p,q)-node	1 in-0-2		2 in-0-1		5 gain-1-1	4 in-0-2		3 in-0-1		8 gain-1-1
(p,q)-node (continued)	6 mul-2-1		9 sum-2-0			7 mul-2-1		11 sum-2-0		

Table A.1 Extracted Patterns and Features

Feature	Index	Counts	Feature	Index	Counts	Feature	Index	Counts	Feature	Index	Counts
in	1	**2**	in-gain	5	**1**	in-gain-sum	9	**1**	gain-1-1	13	**1**
gain	2	**1**	in-mul	6	**2**	in-mul-sum	10	**2**	mul-2-1	14	**1**
mul	3	**1**	gain-sum	7	**1**	in-0-1	11	**1**	sum-2-0	15	**1**
sum	4	**1**	mul-sum	8	**1**	in-0-2	12	**1**			

Table A.2 Feature Indexing and Occurrence Count

1-norm distance.

In the Table A.2, based on the occurrence counts of features in fragment A, the vector for A is (2,1,1,1,1,2,1,1,1,2,1,1,1,1,1).

LSH: Locality Sensitive Hashing

A locality-sensitive hashing (LSH) function is a hash function for vectors such that the probability that two vectors having a same hash code is a *strictly decreasing* function of their corresponding *distance*. In other words, vectors having smaller distance will have higher probability to have the same hash code, and vice versa. Then, if we use locality-sensitive hash functions to hash the fragments into buckets based on the hash codes of their vectors, fragments having similar vectors tend to be hashed into same buckets, and the other ones are less likely to be so.

The vector distance used in SecureSync for similarity measure is Manhattan distance. Therefore, it uses locality-sensitive hash functions for l_1 norm. The following family H of hash functions was proved to be locality-sensitive for Manhattan distance:

$$h(u) = \lfloor \frac{a.u + b}{w} \rfloor$$

In this formula, a is a vector whose elements are drawn from Cauchy distribution; w is a fixed positive real number; and b is a random number in $[0, w]$. Common implementations choose $w = 4$.

If $\|u - v\| = l$ then

$$Pr(l) = Pr[h(u) = h(v)] = \int_0^w \frac{2.e^{-(\frac{x}{l})^2}}{l.\sqrt{2\pi}}(1 - \frac{x}{w})dx$$

$Pr(l)$ is proved to be a decreasing function of l [20]. Then, for $l \leq \delta$, we have $Pr(l) \leq p = Pr(\delta)$. Therefore, for any two vectors u, v that $\|u - v\| \leq \delta$, $Pr[h(u) = h(v)] \leq p$. That means, they have a chance at least p to be hashed into a same bucket.

However, two distant points also have a chance at most p to be hashed into a same bucket. To reduce that odds, we could use more than one hash functions. Each hash function **h** used in SecureSync is a tuple of k independent hash functions of H: $\mathbf{h} = (h_1, h_2, ..., h_k)$. That means hashcode of each vector u will be a vector of integers $\mathbf{h}(u) = (h_1(u), h_2(u), ..., h_k(u))$, with each corresponding integer index for such a vector hashcode is calculated as follows:

$$\mathbf{h}(u) = \sum_{i=1}^{k} r_i.h_i(u) \ mod \ P$$

where each r_i is a randomly chosen integer and P is a very large prime number. In SecureSync, we use a 24 bit prime number.

We call this kind of hash functions as k-line functions. Then, two distant vectors having the same vector hashcode if all of the member hashcodes are the same, and the probability of this event is $\mathbf{q} \leq p^k$. The corresponding probability for two similar vectors is $\mathbf{p} \geq p^k$.

Since the chance for similar vectors be hashed to the same buckets reduces, SecureSync uses N independent k-hash functions, and each vector is hashed to N corresponding buckets. Then, if u and v are missed by a hash function, they still have chances from the others. Indeed, the probability that u and v are missed by all those N functions, i.e. having all different hash codes is $(1 - \mathbf{p})^N \leq (1 - p^k)^N$. If N is large enough, this probability approaches to zero, i.e. u and v are hashed into at least the same bucket with a high probability.

BIBLIOGRAPHY

[1] ASF Security Team. http://www.apache.org/security/.

[2] Common Configuration Enumeration. http://cce.mitre.org/.

[3] Common Platform Enumeration. http://cpe.mitre.org/.

[4] Common Vulnerabilities and Exposures. http://cve.mitre.org/.

[5] Common Vulnerability Scoring System. http://www.first.org/cvss/.

[6] Google Code Search. http://www.google.com/codesearch.

[7] IBM Internet Security Systems. http://www.iss.net/.

[8] Mozilla Foundation Security Advisories. http://www.mozilla.org/security/.

[9] Open Source Computer Emergency Response Team. http://www.ocert.org/.

[10] Open Vulnerability and Assessment Language. http://oval.mitre.org/.

[11] Patch (computing). http://en.wikipedia.org/wiki/Patch_(computing).

[12] Pattern Insight. http://patterninsight.com/solutions/find-once.php.

[13] Software Bug. http://en.wikipedia.org/wiki/Software_bug.

[14] The eXtensible Configuration Checklist Description Format. http://scap.nist.gov/specifications/xccdf.

[15] The Open Source Vulnerability Database. http://osvdb.org/.

[16] The Security Content Automation Protocol. www.nvd.nist.gov/scap/docs/SCAP.doc.

[17] US-CERT Bulletins. http://www.us-cert.gov/.

[18] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, pages 25–34, September 2007.

[19] O.H. Alhazmi, Y.K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219 – 228, 2007.

[20] Alexandr Andoni and Piotr Indyk. E2LSH 0.1 User manual. http://www.mit.edu/ andoni/LSH/-manual.pdf.

[21] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a java legacy system. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17. ACM, 2006.

[22] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35. ACM, 2008.

[23] Barry Boem. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, 1981.

[24] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Trans. Softw. Eng.*, 34(5):579–596, 2008.

[25] Omar Alhazmi Colorado and Omar H. Alhazmi. Quantitative vulnerability assessment of systems software. In *Proc. Annual Reliability and Maintainability Symposium*, pages 615–620, 2005.

[26] Tom Copeland. *PMD Applied*. Centennial Books, 2005.

[27] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31:446–465, 2005.

[28] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing software security fortification throughcode-level metrics. In *QoP '08: Proceedings of the 4th ACM workshop on Quality of protection*, pages 31–38, New York, NY, USA, 2008. ACM.

[30] Computer Security Institute. http://gocsi.com/survey.

[31] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[32] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[33] Hsin-Yi Jiang, T. N. Nguyen, Ing-Xiang Chen, H. Jaygarl, and C. K. Chang. Incremental latent semantic indexing for automatic traceability link evolution management. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 59–68, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, New York, NY, USA, 2007. ACM.

[35] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, New York, NY, USA, 2006. ACM.

[36] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[37] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[38] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[39] T. Longstaff. Update: Cert/cc vulnerability. knowledge base. Technical report, Technical presentation at a DARPA workshop in Savannah, Georgia, 1997.

[40] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, 2007.

[41] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 181–190, New York, NY, USA, 2008. ACM.

[42] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[43] Stephan Neuhaus and Thomas Zimmermann. The beauty and the beast: Vulnerabilities in red hat's packages. In *Proceedings of the 2009 USENIX Annual Technical Conference*, June 2009.

[44] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 440–455, Berlin, Heidelberg, 2009. Springer-Verlag.

[45] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *32nd International Conference on Software Engineering (ICSE 2010)*.

[46] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392, New York, NY, USA, 2009. ACM.

[47] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, and Tien N. Nguyen. Operv: Operation-based, fine-grained version control model for tree-based representation. In *FASE' 10: The 13th International Conference on Fundamental Approaches to Software Engineering*.

[48] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. volume 31, pages 340–355. IEEE CS, 2005.

[49] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[50] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, Xinying Wang, Anh Tuan Nguyen, and Tien N. Nguyen. Detecting recurring and similar software vulnerabilities. In *32nd International Conference on Software Engineering (ICSE 2010 NIER Track)*.

[51] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12, New York, NY, USA, 2008. ACM.

[52] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: raising risk awareness. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 107–110. ACM, 2005.

[53] Qinbao Song, Martin Shepperd, Michelle Cartwright, and Carolyn Mair. Software defect association mining and defect correction effort prediction. *IEEE Trans. Softw. Eng.*, 32(2):69–82, 2006.

[54] Boya Sun, Ray-Yaung Chang, Xianghao Chen, and Andy Podgurski. Automated support for propagating bug fixes. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 187–196, Washington, DC, USA, 2008. IEEE Computer Society.

[55] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 496–506, Washington, DC, USA, 2009. IEEE Computer Society.

[56] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44, New York, NY, USA, 2007. ACM.

[57] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. volume 31, pages 466–480, Piscataway, NJ, USA, 2005. IEEE Press.

[58] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 1–11. IEEE CS, 2009.