Detection of Recurring Software Vulnerabilities

Nam H. Pham
nampham@iastate.edu

Tung Thanh Nguyen
tung@iastate.edu

Hoan Anh Nguyen
hoan@iastate.edu

Tien N. Nguyen
tien@iastate.edu

Electrical and Computer Engineering Department
Iowa State University, USA

ABSTRACT

Software security vulnerabilities are discovered on an almost daily basis and have caused substantial damage. Aiming at supporting early detection and resolution for them, we have conducted an empirical study on thousands of vulnerabilities and found that many of them are recurring due to software reuse. Based on the knowledge gained from the study, we developed SecureSync, an automatic tool to detect recurring software vulnerabilities on the systems that reuse source code or libraries. The core of SecureSync includes two techniques to represent and compute the similarity of vulnerable code across different systems. The evaluation for 60 vulnerabilities on 176 releases of 119 open-source software systems shows that SecureSync is able to detect recurring vulnerabilities with high accuracy and to identify 90 releases having potentially vulnerable code that are not reported or fixed yet, even in mature systems. A couple of cases were actually confirmed by their developers.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Design, Reliability

1. INTRODUCTION

New software security vulnerabilities are discovered on an almost daily basis and have caused substantial financial damage. Thus, 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 fixes. In current practice, known software security vulnerabilities and/or fixes are often reported in public databases (e.g. National Vulnerability Database (NVD) [4], Common Vulnerabilities and Exposures database (CVE) [3]), 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 [4], CVE [3], 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, 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 in the 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 (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 placing 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 3 types will be discussed in Section 2.

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 corresponding patched code. It is able to support detecting and resolving vulnerabilities in two following scenarios:

1. Given a vulnerability report in a system A with corresponding vulnerable and patched code, SecureSync analyzes the patch and stores the information in its knowledge base. Then, via Google Code Search, it searches for all other systems B that share source

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE'10, September 20–24, 2010, Antwerp, Belgium.
Copyright 2010 ACM 978-1-4503-0116-9/10/09 ...$10.00.

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 reported vulnerable code, SecureSync also provides 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 [23], 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 performance, SecureSync uses locality-sensitive hashing (LSH) [1] to perform fast searching for similar trees: only trees having the same hash code are compared to each other. It also uses set-based filtering to find the candidates of Type 2: only the graphs containing the nodes having the same 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 vulnerable code in the systems that share source code or libraries with the systems in its knowledge base. Interestingly, it detects 90 releases having potentially vulnerable code locations (i.e. 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.

The contribution of this paper 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 API/libraries;

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

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

Section 2 reports our empirical study on recurring vulnerabilities. Sections 3, 4, and 5 present the overview of our approach and detailed techniques. The empirical evaluation is in Section 6. Related work is discussed in Section 7 and conclusions appear last.

2. EMPIRICAL STUDY

2.1 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 paper, the similar properties are 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 APIs/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.

Terminology. A **system** is a software product, program, module, or library under investigation (e.g. Firefox, Thunderbird, etc). A **release** refers to a specific release/version of a software system (e.g. Firefox 3.0.5, Thunderbird 2.0.17). A **vulnerability** is an exploitable software fault occurring on specific release(s) of a system. For example, CVE-2008-5023 reported a vulnerability on Firefox 3.0.0 to 3.0.4. 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 Section 2.2.

Analysis Process. To confirm those two hypotheses, we considered around 3,000 vulnerability reports in several security databases: National Vulnerability Database (NVD [4]), Open Source Computer Emergency Response Team Advisories (oCERT) [28], Mozilla Foundation Security Advisories (MFSA [21]), and Apache Security Team (ASF) [2]. Generally, each report describes a vulnerability, thus, a recurring vulnerability would be described in multiple reports. 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. Fortunately, some of such groups really report recurring vulnerabilities. Sometimes, one report (such as in oCERT) describes more than one 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.

2.2 Representative Examples

Example 1. In CVE-2008-5023, it is reported that a vulnerability *"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 1 shows the vulnerable code fragments extracted from Firefox 3.0.3 and SeaMonkey 1.1.12. Figure 2 shows the corresponding patched code in Firefox 3.0.4. As we could see, the vul-

a) Vulnerable code in Firefox 3.0.3

```
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;
```

b) Vulnerable code in SeaMonkey 1.1.12

```
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 1: Recurring Vulnerability in Firefox and SeaMonkey due to Reusing Vulnerable Source Code

nerable code was patched by adding more checking mechanisms via two functions GetHasCertificate and Subsumes. (The vulnerable code in SeaMonkey is identically patched, however, due to the space limit, it is not shown here). Analyzing deeper, we figure out that the vulnerability is recurring due to the *reuse of source code*. In Figure 1, the vulnerable code fragments are highly similar. That is, because Firefox and SeaMonkey are developed from the common framework Mozilla, they largely share source code, including the vulnerable code in those fragments.

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. That is, a code fragment in one system has some undetected flaws and is reused in another system. Then, when the flaws are exploited as a vulnerability, the vulnerability is recurring in both systems. Generally, 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. As we could see, due to 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 are copied from earlier versions, thus, 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 special kind of patching is referred to as *backporting* [35].

Example 2. OpenSSL, an open source implementation of SSL and TLS protocols, provides a 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. They thought that, EVP_VerifyFinal would return only **two** values: 1 and 0 for correct and incorrect verification. Therefore, in several systems, the flaw statement 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 is false as in the case when 1 is returned. That is, the program would behave as if the verification is correct, i.e. it is vulnerable to this exploitation.

Patched code in Firefox 3.0.4

```
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 2: Patched Code of Vulnerability in Figure 1

From CVE-2009-0021 and CVE-2009-0047, this programming flaw appeared in two systems NTP and Gale using EVP, 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 shows 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 example into Type 2, i.e. API-shared/reused recurring vulnerability (denoted by **RV2**). The vulnerabilities of this type occur in 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 an API function call 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 value. 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 this type. According to CVE-2006-4439 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. Unfortu-

Vulnerable code in NTP 4.2.5

```
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);


...
```

Vulnerable code in Gale 0.99

```
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: Recurring Vulnerability in NTP and Gale due to Similar Misuse of API

nately, 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 be recognized, localized, or matched in those systems due to the wide varieties of implementation choices and differences among systems in design, architecture, programming language, etc.

2.3 Results and Implications

Table 1 summarizes the result of our study. Column Report shows the total number of 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. That is, 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-shared) 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. Thus, 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 in implementing an algorithm might be less than the chance that they create a flaw in shared code or misuse libraries in similar ways.

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 one 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. Type 1 vulnerable code is generally similar in texts and structure, while 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.

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 1: Recurring Software Vulnerabilities

3. 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 in a given system the vulnerable code fragments that are similar to the ones in its knowledge base.

3.1 Problem Formulation

To build SecureSync, there are 2 core issues: how to represent and measure the similarity of RV1s and RV2s, and how to localize the recurring ones in different systems. SecureSync represents vulnerabilities via the *features* extracted from their vulnerable and patched code, and calculates 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*

```
function Detect($P, K, \sigma$) //detect recurring vulnerability
  $C$ = Candidates($P, K$) //produce candidates
  for each fragment $B \in C$: //check against knowledge base for recurring
    if $\exists(A, A') \in K : Sim(B, A) \geq \sigma \wedge Sim(B, A') < Sim(B, A)$
      ReportAndRecommend($B$)
```

Figure 4: Detection of Recurring Vulnerabilities

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(s)* B_i*(s) that are recurring vulnerable code of some fragment* A_j.

3.2 Algorithmic Solution and Techniques

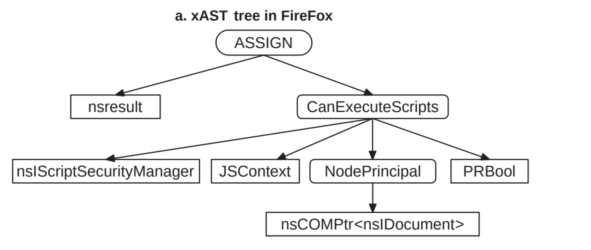
The general process for detection of RV1s and RV2s is illustrated in Figure 4. First, SecureSync produces candidate fragments from the program P under investigation (line 2). Then, each candidate is compared against vulnerable and patched code of the vulnerabilities in the knowledge base K to find the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

Feature Extraction and Similarity Measure. For RV1s, the tool 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 [23]. For Type 2, 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 inputs or handling of outputs of function calls. Then, the similarity of code fragments is measured by the similarity of those xGRUMs based on their aligned nodes (see Section 5).

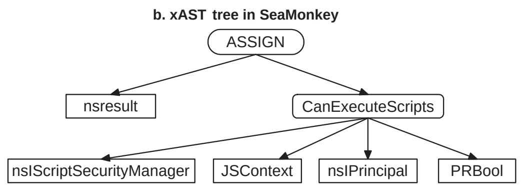
Building Knowledge Base of Reported Vulnerabilities. We build the knowledge base for SecureSync using a semi-automated method. First, we access to vulnerability databases and manually analyze each report to choose vulnerabilities. Then, using code search, we find the corresponding vulnerable and patched 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 in 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 graph alignment algorithm, SecureSync suggests the addition of missed function calls, or the checking of inputs/outputs before/after the calls, etc.



nsresult rv = mgr->CanExecuteScripts(cx, ourDocument->NodePrincipal(), &canExecute);



nsresult rv = mgr->CanExecuteScripts(cx, principal, &canExecute);

Figure 5: xAST from Code in Figure 1

4. TYPE 1 VULNERABILITY DETECTION

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 with different labels would represent different calls). Figure 5 illustrates two xASTs of similar vulnerable statements in Figure 1. Two trees have mostly similar structures and nodes' labels, e.g. the nodes representing the function calls CanExecuteScripts, the variables of data type nsresult, etc. (For simplicity, the node types or parameter lists are not drawn).

4.1 Feature Extraction and Similarity Measure

SecureSync extracts from a code fragment A a feature set $F(A)$, which is defined as a set of xASTs, each represents a statement in A. For instance, vulnerable code fragments in Figure 1 have feature sets of 6 and 8 xASTs, respectively. The similarity of two fragments is measured via the similarity of corresponding feature sets.

SecureSync uses Exas [23] 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 its structural features $Ex(T)$. For a tree, a structural feature is a sequence of labels of the nodes along a limited length path. For example, both trees in Figure 5 have a feature [ASSIGN]-[nsresult] and a feature [ASSIGN]-[CanExecuteScripts]-[PRBool].

The similarity of 2 fragments is measured based on the Manhattan distance of 2 corresponding Exas vectors of their feature sets:

$$Sim(A, B) = 1 - \frac{|Ex(F(A)) - Ex(F(B))|}{|Ex(F(A))| + |Ex(F(B))|} \qquad (4.1)$$

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.

4.2 Candidate Searching

A vulnerable code fragment of Type 1 generally scatters in several *non-consecutive* statements (see Figure 1). Thus, traditional

P中每条语句与A和A'的每条语句比较，忽略了语句的顺序？

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.

根据function name和变量名过滤可能导致漏报，重命名。

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 keeps only the files that contain the tokens and 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 a program 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) [1]. 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 [1]. 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 codes of $Ex(S)$ with those of $Ex(T)$. If $Ex(S)$ and $Ex(T)$ have some common LSH 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 the previous step, 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 within a method. Then, candidate code fragments will be compared to vulnerable and patched code fragments in the knowledge base, using Definition 2 and Formula 4.1. Based on the LSH 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 codes with the statements Ss in B.

只可处理跨版本的，不能处理同版本内的clone时的rename

4.3 Detection of Renaming

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 2.0.x was renamed to proxy_send_dir_filter in Apache 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 [26]. OperV models a software system by a project tree, i.e. a tree-based structure of program elements. 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.

5. TYPE 2 VULNERABILITY DETECTION

5.1 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, an API usage is represented as a graph-based model, 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 node type.

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 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 6 partially shows 3 xGRUMs of two vulnerable code fragments in Figure 3 and one patched code fragment. (Due to the space limit, 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, the condition checking with that if statement 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 6c, 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 the data type and value. SecureSync models only the literals of supported primitive data types (e.g. integer, float, string) and special values (e.g. 0, 1, -1, null, empty string). Prior work [7,14] showed that bugs often occur at the 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 depen-

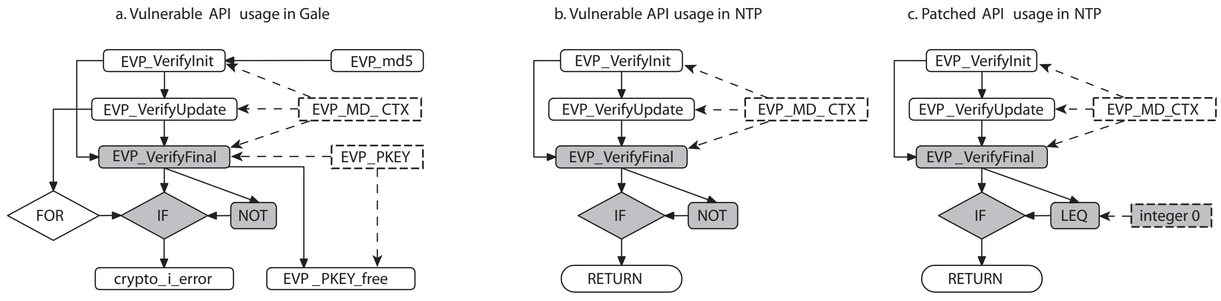


Figure 6: xGRUMs from Vulnerable and Patched Code in Figure 3

dency between EVP_VerifyInit, EVP_VerifyUpdate, and EVP_VerifyFinal are found via their connections to the data node EVP_MD_CTX.

5.2 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 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 vulnerable and patched code are considered to represent the entities *irrelevant* to API misuse. Thus, the *sub-graphs* containing changed nodes in those two xGRUMs are considered as the *features* of the corresponding vulnerability. To find the changed and unchanged nodes, SecureSync uses the following approximate graph alignment algorithm.

5.2.1 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 7. 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 4.1 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 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,

```
1 function Align($G, G', \mu$) //align and compare two usage models
2   for all $u \in G, v \in G'$ //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 $(u, v) \in M$:
7     if sim$(u, v) < \mu$ then $M$.remove($(u, v)$) //remove too low matches
8 return M
```

Figure 7: Graph Alignment Algorithm

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. Figure 6b and Figure 6c 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 identical. The similarity of two nodes labeled EVP_VerifyFinal is less similar 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 subgraph with those changed nodes in gray color in Figures 6b and 6c.

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 between A and B is measured based on the similarity between $U(A)$ 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 4.1. The only different is that $Ex(U(A))$ and $Ex(U(B))$ are Exas vectors of two xGRUMs, not xASTs. In Figures 6a and 6b, $M(A)$ 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 6a, 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%.

5.3 Candidate Searching

Similarly to the detection of RV1s, SecureSync uses origin analysis to find renamed API functions between systems and versions. Then, it also uses text-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 filtering, 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.

6. EVALUATION

This section presents our evaluation of SecureSync on real-world software systems and vulnerabilities. The evaluation is separated into two experiments for the detection of RV1s and RV2s. 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 RV1 Detection Evaluation

Selecting. We chose 3 Mozilla-based open-source systems FireFox, Thunderbird and SeaMonkey for the evaluation of RV1 because they are actively developed and maintained, and have available source code, security reports, forums, and discussions. First, we contacted and obtained the release history of those 3 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 3 systems. The numbers of releases of each system is shown in column Release of Table 2.

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 case 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 ran SecureSync on 51 chosen 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.

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 2: RV1 Detection Evaluation

```
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 8: Vulnerable Code in Thunderbird 2.0.17

Analyzing. Table 2 shows the analysis result. 21 vulnerabilities had been officially reported by MFSA [21] and verified by us as truly recurring 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, 3 cases are incorrectly reported (column X) and 2 are missed (Miss in DB), giving the precision of 30/33 = 91%. 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 3 subject systems. Figure 8 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 was 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 and sent it to Mozilla security team. They had kindly confirmed this programming flaw and our provided patch.

6.2 RV2 Detection Evaluation

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 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 (Google 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 execution and analysis is similar to the experiment for RV1s. Table 3 shows the analysis result. For example, there is an RV2 related to the misuse of two functions

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	75	22	64	13	2

Table 3: RV2 Detection Evaluation

seteuid and setuid. 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 as vulnerable. Manually checking confirms 23/28 to be correct and 5 are incorrect (giving precision of 82%). Among 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 generally 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 methods. 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 $\sigma = 0.9$, 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 could be improved with more powerful data analysis.

Interesting Examples. Here are some interesting cases on which SecureSync correctly detected not-yet-reported RV2s. Figure 9 illustrates a code fragment in Arronwork having the same vulnerability related to the incorrect usage of EVP_VerifyFinal function as described in Section 2, 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 Arronwork's developers.

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 is in Figure 10, 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 11). 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 11 with that of the method in Figure 10, SecureSync correctly suggests the fixing by adding the if statement before the calls to g_malloc and g_malloc0 functions, respectively.

```
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 9: 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 10: Vulnerable and Patched Code in GLib 2.12.3

7. RELATED WORK

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 correct code patterns via mining frequent code/usages [33], via mining rules from existing code [32], or predefined rules [13, 31]. 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 is 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* [13, 18, 32–34]. Sun *et al.* [31] 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 [8, 13]. In contrast to those pattern-based approaches, SecureSync is based on our general graph-based repre-

```
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 11: Vulnerable Code in SeaHorse 1.0.1

sentation for API-related code and its feature extraction that could support any vulnerable and corresponding patched code.

JADET [33] and GrouMiner [24] perform mining object usage patterns and detect violations as potential bugs. Several approaches mine usage patterns in term of the orders of pairs of method calls [5, 18, 34], or association rules [30, 32]. Error-handling bugs are detected via mining sequence association rules [32]. Chang *et al.* [7] find patterns on condition nodes on program dependence graphs and detect bugs involving neglected conditions. Hipikat [9] extracts *lexical* information while building the project's memories and recommends relevant artifacts. Song *et al.* [30] detect association rules between 6 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 [33], error-handling [32], condition checking [7]). BugMem [15] 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 types of more general API-related recurring vulnerabilities than specific sets of bug patterns.

FixWizard [25] 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 [29] finds all code snapshots with similar snippets (i.e. cloned code) to the fragment that was fixed. CP-Miner [17] mines frequent subsequences of tokens to detect bugs caused by inconsistent editing to cloned code. Jiang *et al.* [14] 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 [10]. Compared to clone-related bug detection approaches [14, 17], our detection for code-reused vulnerabilities could also handle *non-continuous* clones (see Figure 1). Importantly, our graph-based representation and feature extraction are more specialized toward finding fragments with *similar API usages*, and more flexible to support the detection of API-related bugs *across systems*. The preliminary study on recurring vulnerabilities was reported in our prior work [27].

Several approaches have been proposed to help localize buggy code using project's change history [12,16], complexity metrics [19, 20], etc. Vulnerability prediction approaches that rely on projects' components and history include [11, 22]. In software security, other researchers study the characteristics of vulnerabilities [6]. However, none of existing software security approaches has studied the relation of recurring vulnerabilities and software reuse.

8. CONCLUSIONS

This paper 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 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.

9. REFERENCES

[1] A. Andoni and P. Indyk. E2LSH 0.1 User manual. http://web.mit.edu/andoni/www/LSH/manual.pdf.

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

[3] Common vulnerabilities and exposures. http://cve.mitre.org/.

[4] US-CERT bulletins. http://www.us-cert.gov/.

[5] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE'07*.

[6] O. Alhazmi, Y. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers and Security*, 26, 2007.

[7] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Trans. Softw. Eng.*, 34(5), 2008.

[8] T. Copeland. *PMD Applied*. Centennial Books, 2005.

[9] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6), 2005.

[10] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE'07*, pages 158–167. IEEE CS, 2007.

[11] M. Gegick, L. Williams, J. Osborne, and M. Vouk. Prioritizing software security fortification through code-level metrics. *Proceedings of 4th ACM workshop on Quality of protection*, pages 31–38, 2008.

[12] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *ICSM'05*, pages 263–272. IEEE CS, 2005.

[13] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12), 2004.

[14] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE'07*, pages 55–64. ACM, 2007.

[15] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *FSE'06*, pages 35–45. ACM, 2006.

[16] S. Kim, T. Zimmermann, E. E. J. Whitehead, Jr., and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498. IEEE CS, 2007.

[17] Z. Li, S. Lu, and S. Myagmar. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[18] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *FSE'05*, pages, 296–305, ACM, 2005.

[19] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, 2007.

[20] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*.

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

[22] S. Neuhaus and T. Zimmermann. The beauty and the beast: Vulnerabilities in red hat's packages. In *USENIX Annual Technical Conference*, June 2009.

[23] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and Efficient Structural Characteristic Feature Extraction Method for Clone Detection. In *FASE'09*. Springer-Verlag, 2009.

[24] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *FSE'09*, ACM.

[25] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Recurring Bug Fixes in Object-Oriented Programs. In *ICSE'10*.

[26] T. T. Nguyen, H. A. Nguyen, N. H. Pham, T. N. Nguyen. Operation-based, Fine-grained Version Control Model for Tree-based Representation. *FASE'10*.

[27] N. H. Pham, T. T. Nguyen, H. A. Nguyen, X. Wang, A. T. Nguyen, and T. N. Nguyen. Detecting Recurring and Similar Software Vulnerabilities. In *ICSE'10, NIER Track*, ACM Press, 2010.

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

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

[30] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE TSE*, 32(2):69–82, 2006.

[31] B. Sun, R.-Y. Chang, X. Chen, and A. Podgurski. Automated support for propagating bug fixes. In *ISSRE'08*, pages 187–196. IEEE CS, 2008.

[32] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *ICSE'09*, pages 496–506. IEEE CS, 2009.

[33] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE'07*, pages 35–44. ACM, 2007.

[34] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE TSE*, 31(6), 2005.

[35] Backporting - Wikipedia. http://en.wikipedia.org/wiki/Backporting.