Vulnerability Prediction Models: A case study on the Linux Kernel

Matthieu Jimenez Mike Papadakis Yves Le Traon

Jimenez et al. "Vulnerability Prediction Models: A Case Study on the Linux Kernel" SCAM'16

Slides: Matthieu Jimenez
Thème: Sébastien Mosser

Vulnerabilities ?

A vulnerability

"An information security '**vulnerability**' is a **mistake** in a software that can be directly used by a **hacker** to **gain access** to a **system** or network."

~ CVE - website ~

Vulnerabilities are special

More **Important** - Critical

There are **more bugs** than vulnerabilities

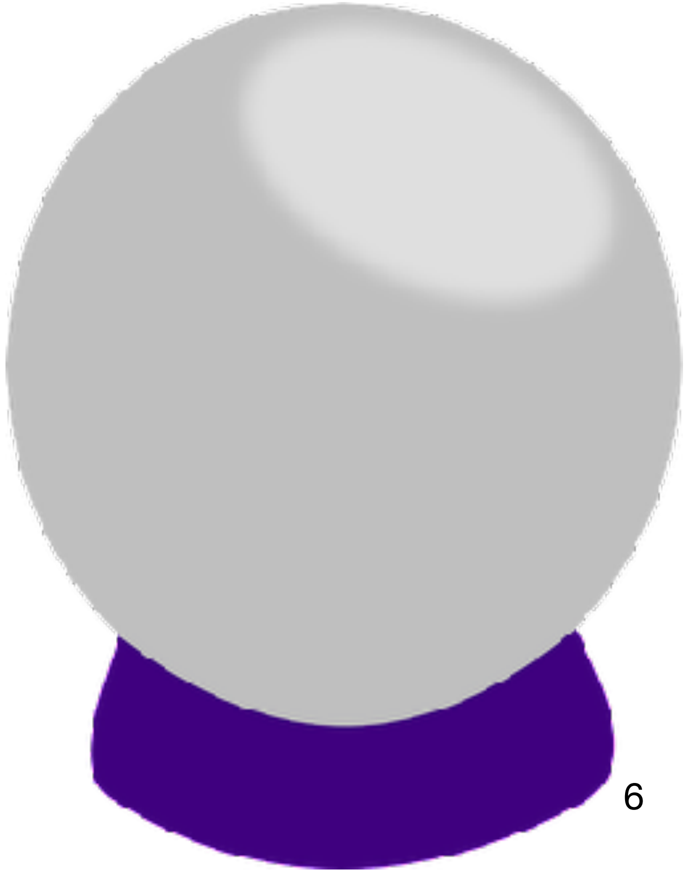
Uncovered differently - defects can be easily noticed, while vulnerabilities not.

Vulnerabilities are

Web server used to remotely control the **glassware-cleaning machine**

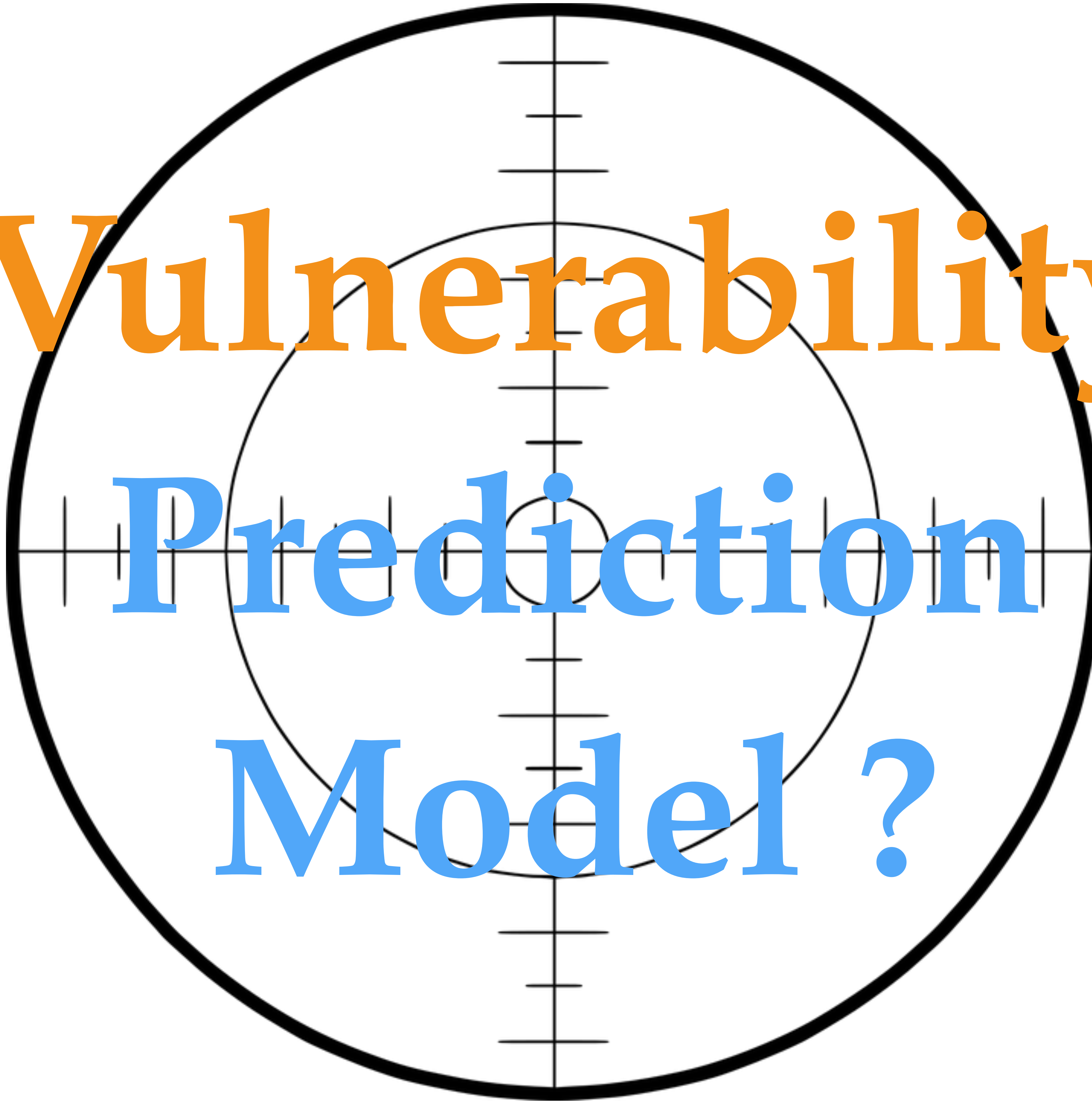
CVE for that…

Prediction Model



Prediction Models

Models analysing **current and historical events** to make **prediction** about the **future** and/or **unknown events** !



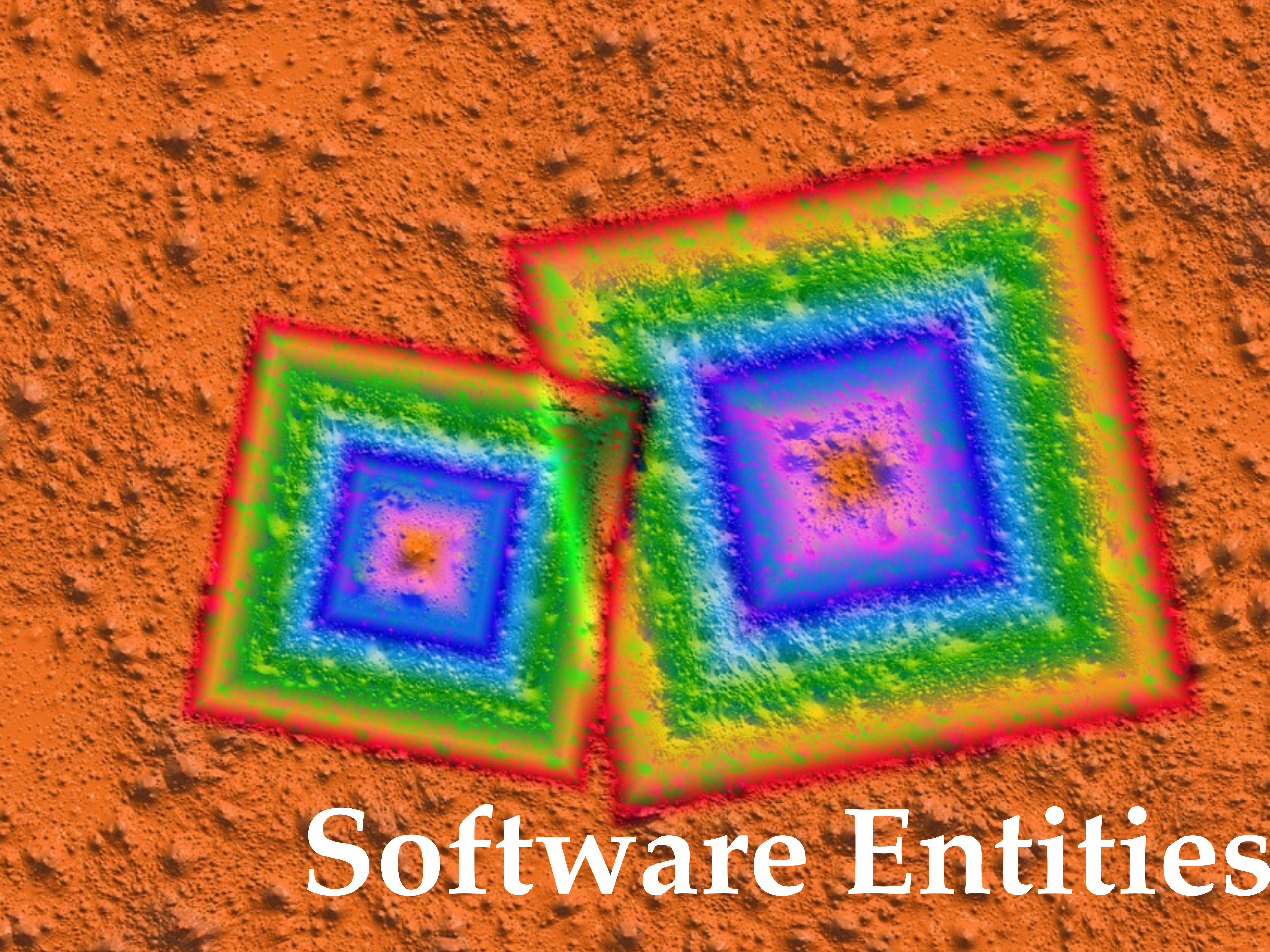
Vulnerability Prediction Model ?

Vulnerability Prediction

Take **advantage** of the **knowledge** on some **part** of a **software system** and / or **previous releases**

Vulnerability Prediction

to **automatically** classify **software entities** as **vulnerable** or **not** !



Software Entities

Granularity

Possibility to work at :

- **module** level
- **file** level
- **function** level
- …

In this **work**, we stay at the **file** level !

*Morrison et al. "Challenges with applying vulnerability prediction models," in *HotSoS'15*.

GOAL

Replicating and **comparing** the **main VPMs approaches** on the same **software system**

Replication …

Exact independent replication

Exact replication

procedures of an **experiment** are **followed as closely as possible**

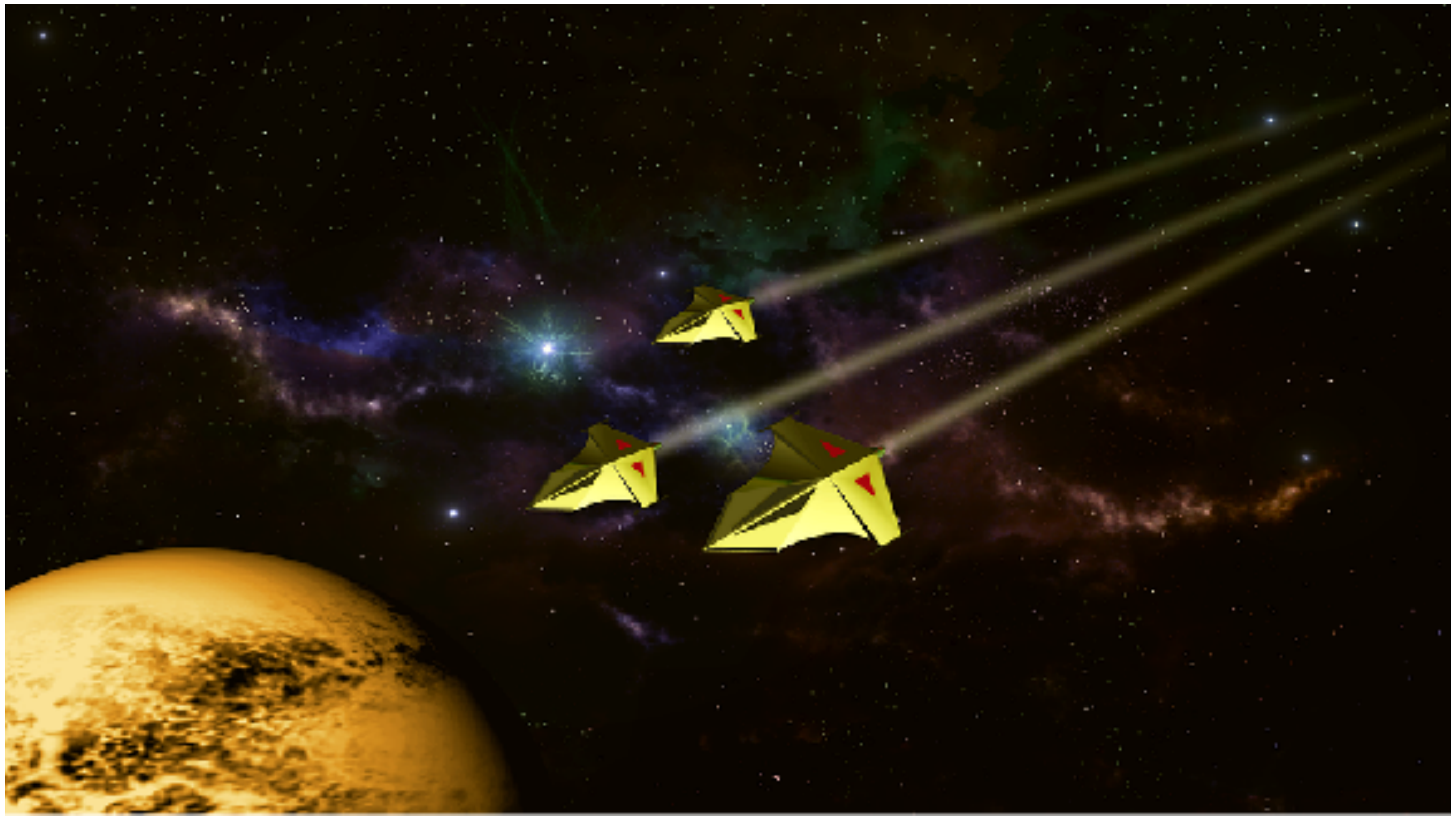
e.g. here we replicate using the same machine learning settings

Independent replication

deliberately **vary** **one** or more major **aspects** of the **conditions** of the **experiment**

e.g. we use our dataset

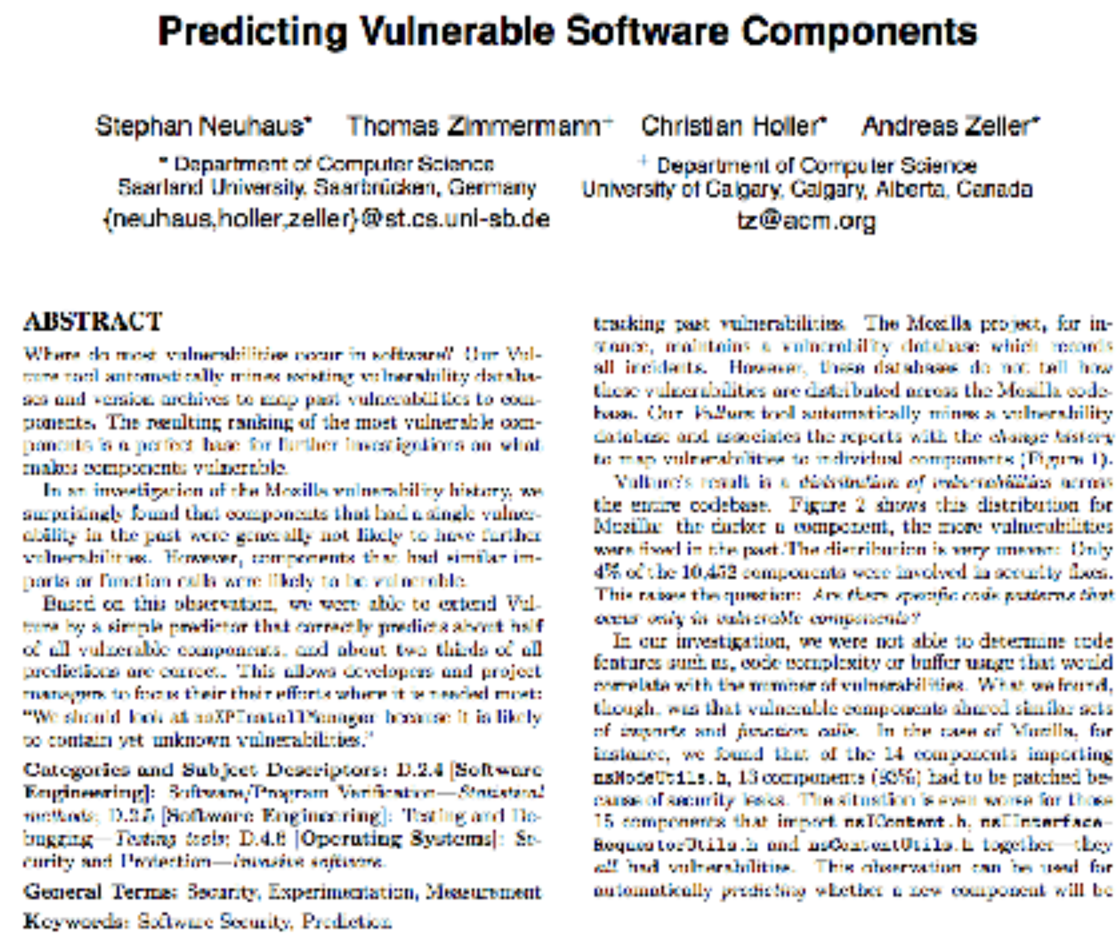
Approaches …



Include and f(n) calls

Include & Function calls

Introduced by Neuhaus et al. at CCS'07



Predicting Vulnerable Software Components

Stephan Neuhaus[*] Thomas Zimmermann[+] Christian Holler[*] Andreas Zeller[*]

[*] Department of Computer Science
Saarland University, Saarbrücken, Germany
{neuhaus,holler,zeller}@st.cs.uni-sb.de

[+] Department of Computer Science
University of Calgary, Calgary, Alberta, Canada
tz@acm.org

ABSTRACT

Where do most vulnerabilities occur in software? Our Vulture tool automatically mines existing vulnerability databases and version archives to map past vulnerabilities to components. The resulting ranking of the most vulnerable components is a perfect base for further investigations on what makes components vulnerable.

In an investigation of the Mozilla vulnerability history, we surprisingly found that components that had a single vulnerability in the past were generally not likely to have further vulnerabilities. However, components that had similar imports or function calls were likely to be vulnerable.

Based on this observation, we were able to extend Vulture by a simple predictor that correctly predicts about half of all vulnerable components, and about two thirds of all predictions are correct. This allows developers and project managers to focus their their efforts where it is needed most: "We should look at nsXPInstallManager because it is likely to contain yet unknown vulnerabilities."

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*.

General Terms: Security, Experimentation, Measurement

Keywords: Software Security, Prediction

tracking past vulnerabilities. The Mozilla project, for instance, maintains a vulnerability database which records all incidents. However, these databases do not tell how these vulnerabilities are distributed across the Mozilla codebase. Our Vulture tool automatically mines a vulnerability database and associates the reports with the change history to map vulnerabilities to individual components (Figure 1).

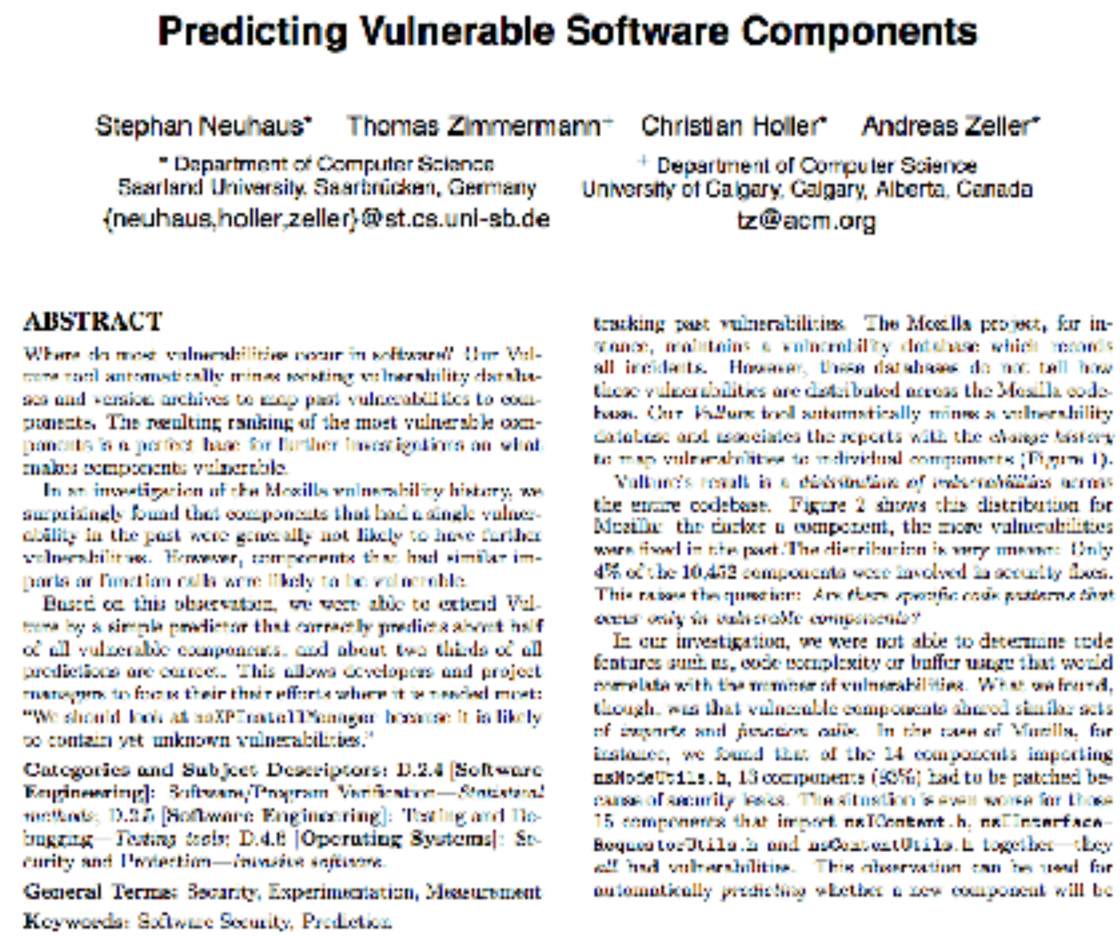
Vulture's result is a *distribution of vulnerabilities* across the entire codebase. Figure 2 shows this distribution for Mozilla: the darker a component, the more vulnerabilities were fixed in the past. The distribution is very uneven: Only 4% of the 10,452 components were involved in security fixes. This raises the question: *Are there specific code patterns that occur only in vulnerable components?*

In our investigation, we were not able to determine code features such as, code complexity or buffer usage that would correlate with the number of vulnerabilities. What we found, though, was that vulnerable components shared similar sets of *imports* and *function calls*. In the case of Mozilla, for instance, we found that of the 14 components importing `nsNodeUtils.h`, 13 components (93%) had to be patched because of security leaks. The situation is even worse for those 15 components that import `nsIContent.h`, `nsIInterfaceRequestorUtils.h` and `nsContentUtils.h` together—they *all* had vulnerabilities. This observation can be used for automatically predicting whether a new component will be

Include & Function calls

Introduced by Neuhaus et al. at CCS'07

Intuition : vulnerable files **share** similar set of **imports and function calls**



Predicting Vulnerable Software Components

Stephan Neuhaus* Thomas Zimmermann+ Christian Holler* Andreas Zeller*

* Department of Computer Science
Saarland University, Saarbrücken, Germany
{neuhaus,holler,zeller}@st.cs.uni-sb.de

+ Department of Computer Science
University of Calgary, Calgary, Alberta, Canada
tz@acm.org

ABSTRACT

Where do most vulnerabilities occur in software? Our Vulture tool automatically mines existing vulnerability databases and version archives to map past vulnerabilities to components. The resulting ranking of the most vulnerable components is a perfect base for further investigations on what makes components vulnerable.

In an investigation of the Mozilla vulnerability history, we surprisingly found that components that had a single vulnerability in the past were generally not likely to have further vulnerabilities. However, components that had similar imports or function calls were likely to be vulnerable.

Based on this observation, we were able to extend Vulture by a simple predictor that correctly predicts about half of all vulnerable components, and about two thirds of all predictions are correct. This allows developers and project managers to focus their their efforts where it is needed most: "We should look at nsXPInstallManager because it is likely to contain yet unknown vulnerabilities."

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*.

General Terms: Security, Experimentation, Measurement

Keywords: Software Security, Prediction

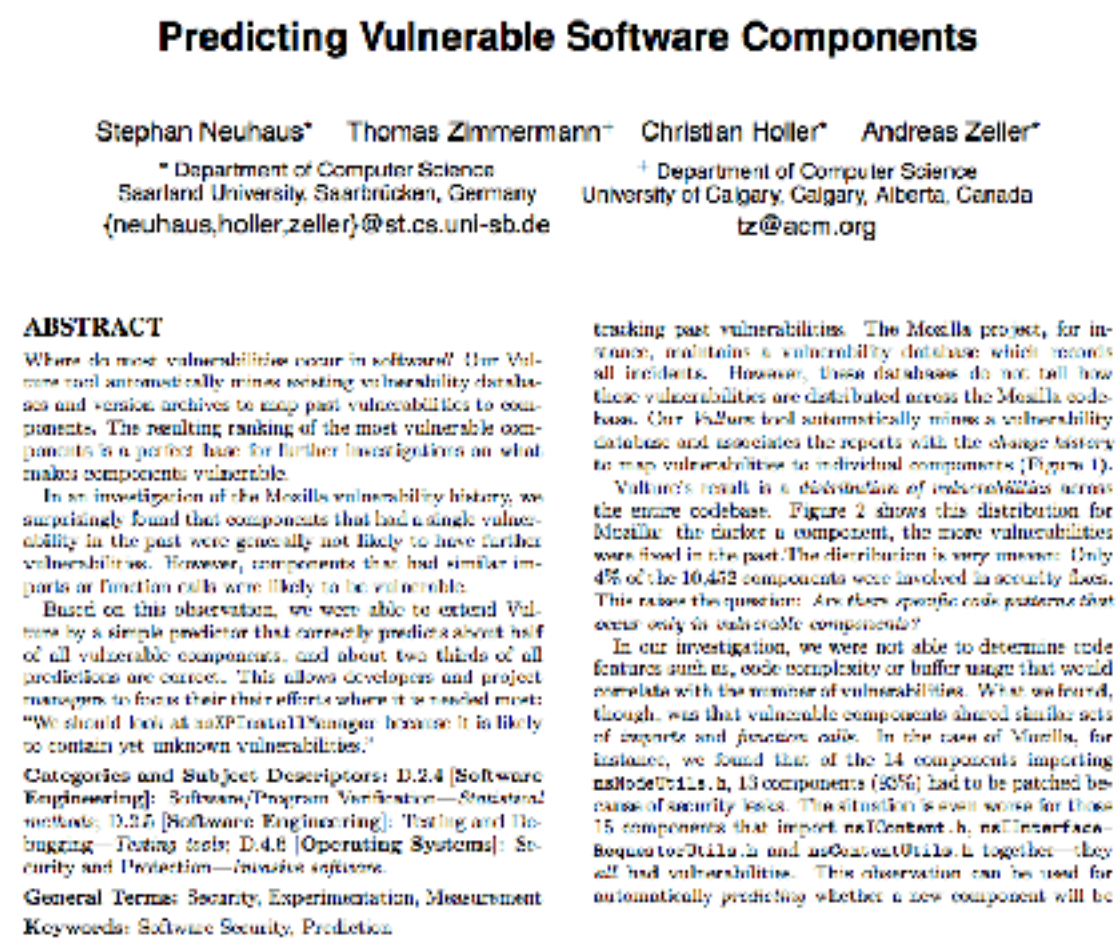
tracking past vulnerabilities. The Mozilla project, for instance, maintains a vulnerability database which records all incidents. However, these databases do not tell how these vulnerabilities are distributed across the Mozilla codebase. Our Vulture tool automatically mines a vulnerability database and associates the reports with the change history to map vulnerabilities to individual components (Figure 1).

Vulture's result is a *distribution of vulnerabilities* across the entire codebase. Figure 2 shows this distribution for Mozilla: the darker a component, the more vulnerabilities were fixed in the past. The distribution is very uneven: Only 4% of the 10,452 components were involved in security fixes. This raises the question: *Are there specific code patterns that occur only in vulnerable components?*

In our investigation, we were not able to determine code features such as, code complexity or buffer usage that would correlate with the number of vulnerabilities. What we found, though, was that vulnerable components shared similar sets of *imports* and *function calls*. In the case of Mozilla, for instance, we found that of the 14 components importing `nsNodeUtils.h`, 13 components (93%) had to be patched because of security leaks. The situation is even worse for those 15 components that import `nsIContent.h`, `nsIInterfaceRequestorUtils.h` and `nsContentUtils.h` together—they *all* had vulnerabilities. This observation can be used for automatically *predicting* whether a new component will be

Include & Function calls

Introduced by Neuhaus et al. at CCS'07



Predicting Vulnerable Software Components

Stephan Neuhaus* Thomas Zimmermann+ Christian Holler* Andreas Zeller*
* Department of Computer Science
Saarland University, Saarbrücken, Germany
{neuhaus,holler,zeller}@st.cs.uni-sb.de
+ Department of Computer Science
University of Calgary, Calgary, Alberta, Canada
tz@acm.org

ABSTRACT

Where do most vulnerabilities occur in software? Our Vulture tool automatically mines existing vulnerability databases and version archives to map past vulnerabilities to components. The resulting ranking of the most vulnerable components is a perfect base for further investigations on what makes components vulnerable.

In an investigation of the Mozilla vulnerability history, we surprisingly found that components that had a single vulnerability in the past were generally not likely to have further vulnerabilities. However, components that had similar imports or function calls were likely to be vulnerable.

Based on this observation, we were able to extend Vulture by a simple predictor that correctly predicts about half of all vulnerable components, and about two thirds of all predictions are correct. This allows developers and project managers to focus their their efforts where it is needed most: "We should look at nsXPInstallManager because it is likely to contain yet unknown vulnerabilities."

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification—Statistical methods; D.2.5 [Software Engineering]: Testing and Debugging—Testing tools; D.4.6 [Operating Systems]: Security and Protection—Invasive software.

General Terms: Security, Experimentation, Measurement

Keywords: Software Security, Prediction

tracking past vulnerabilities. The Mozilla project, for instance, maintains a vulnerability database which records all incidents. However, these databases do not tell how these vulnerabilities are distributed across the Mozilla codebase. Our Vulture tool automatically mines a vulnerability database and associates the reports with the change history to map vulnerabilities to individual components (Figure 1).

Vulture's result is a *distribution of vulnerabilities* across the entire codebase. Figure 2 shows this distribution for Mozilla: the darker a component, the more vulnerabilities were fixed in the past. The distribution is very uneven: Only 4% of the 10,452 components were involved in security fixes. This raises the question: *Are there specific code patterns that occur only in vulnerable components?*

In our investigation, we were not able to determine code features such as, code complexity or buffer usage that would correlate with the number of vulnerabilities. What we found, though, was that vulnerable components shared similar sets of *imports* and *function calls*. In the case of Mozilla, for instance, we found that of the 14 components importing nsNodeUtils.h, 13 components (93%) had to be patched because of security leaks. The situation is even worse for those 15 components that import nsIContent.h, nsIInterfaceRequestorUtils.h and nsContentUtils.h together—they *all* had vulnerabilities. This observation can be used for automatically *predicting* whether a new component will be

Intuition : vulnerable files **share** similar set of **imports and function calls**

build a **model** based on either **includes or function calls** of a **file**.

Overview

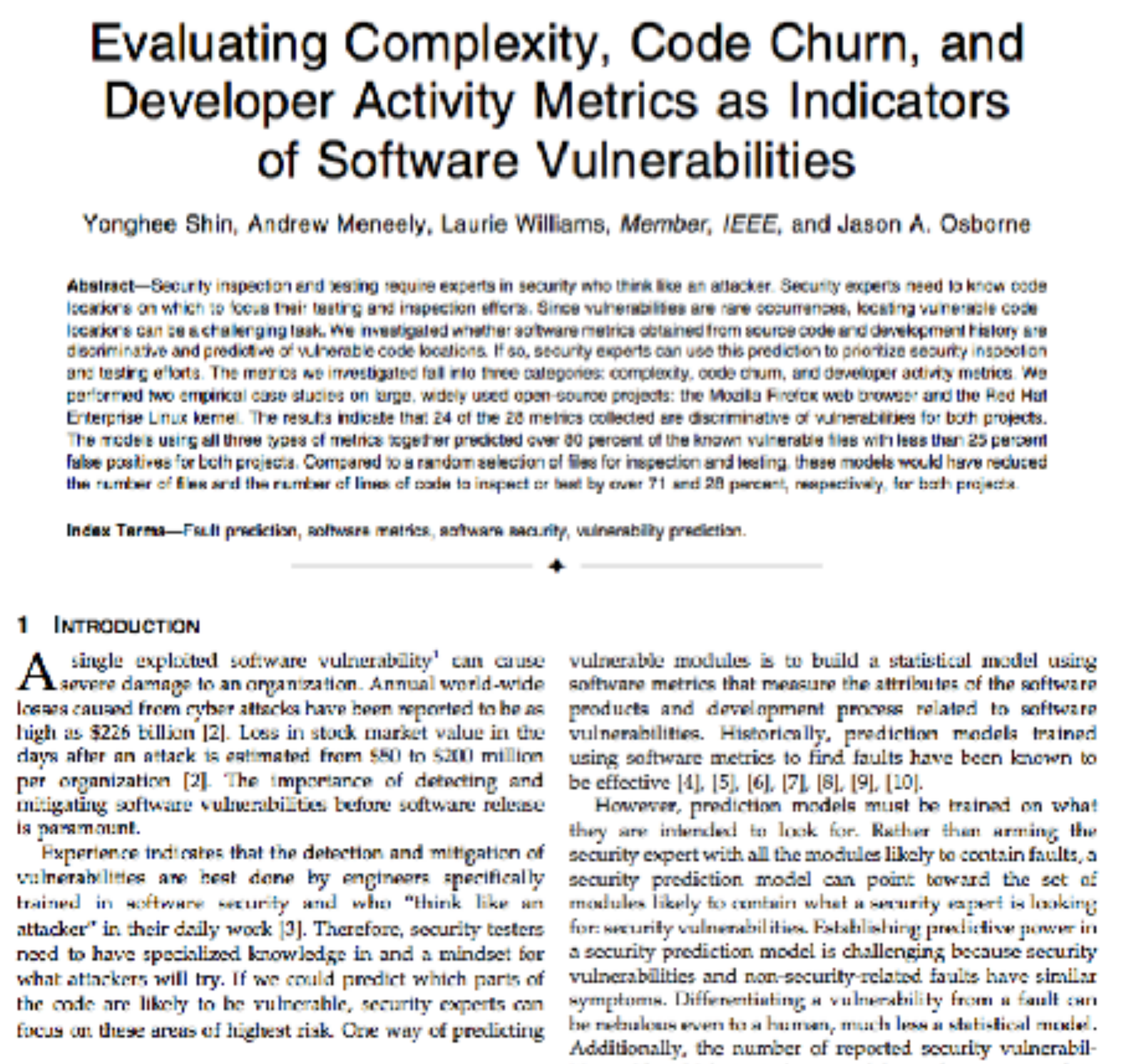
	Preprocessing	Learning
Include & function calls	Retrieve all include and function calls of a file	SVM with a linear kernel

2 models are build

Software Metrics

Software Metrics

Several **works** on using **metrics** to predict **vulnerabilities**, mostly by Shin et al.



Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities

Yonghee Shin, Andrew Meneely, Laurie Williams, *Member, IEEE*, and Jason A. Osborne

Abstract—Security inspection and testing require experts in security who think like an attacker. Security experts need to know code locations on which to focus their testing and inspection efforts. Since vulnerabilities are rare occurrences, locating vulnerable code locations can be a challenging task. We investigated whether software metrics obtained from source code and development history are discriminative and predictive of vulnerable code locations. If so, security experts can use this prediction to prioritize security inspection and testing efforts. The metrics we investigated fall into three categories: complexity, code churn, and developer activity metrics. We performed two empirical case studies on large, widely used open-source projects: the Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. The results indicate that 24 of the 28 metrics collected are discriminative of vulnerabilities for both projects. The models using all three types of metrics together predicted over 80 percent of the known vulnerable files with less than 25 percent false positives for both projects. Compared to a random selection of files for inspection and testing, these models would have reduced the number of files and the number of lines of code to inspect or test by over 71 and 28 percent, respectively, for both projects.

Index Terms—Fault prediction, software metrics, software security, vulnerability prediction.

1 Introduction

A single exploited software vulnerability[1] can cause severe damage to an organization. Annual world-wide losses caused from cyber attacks have been reported to be as high as $226 billion [2]. Loss in stock market value in the days after an attack is estimated from $50 to $200 million per organization [2]. The importance of detecting and mitigating software vulnerabilities before software release is paramount.

Experience indicates that the detection and mitigation of vulnerabilities are best done by engineers specifically trained in software security and who "think like an attacker" in their daily work [3]. Therefore, security testers need to have specialized knowledge in and a mindset for what attackers will try. If we could predict which parts of the code are likely to be vulnerable, security experts can focus on these areas of highest risk. One way of predicting vulnerable modules is to build a statistical model using software metrics that measure the attributes of the software products and development process related to software vulnerabilities. Historically, prediction models trained using software metrics to find faults have been known to be effective [4], [5], [6], [7], [8], [9], [10].

However, prediction models must be trained on what they are intended to look for. Rather than arming the security expert with all the modules likely to contain faults, a security prediction model can point toward the set of modules likely to contain what a security expert is looking for: security vulnerabilities. Establishing predictive power in a security prediction model is challenging because security vulnerabilities and non-security-related faults have similar symptoms. Differentiating a vulnerability from a fault can be nebulous even to a human, much less a statistical model. Additionally, the number of reported security vulnerabil-

Software Metrics

Several **works** on using **metrics** to predict **vulnerabilities**, mostly by Shin et al.

Software metrics are **used** in **defect prediction**

Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities

Yonghee Shin, Andrew Meneely, Laurie Williams, *Member, IEEE*, and Jason A. Osborne

Abstract—Security inspection and testing require experts in security who think like an attacker. Security experts need to know code locations on which to focus their testing and inspection efforts. Since vulnerabilities are rare occurrences, locating vulnerable code locations can be a challenging task. We investigated whether software metrics obtained from source code and development history are discriminative and predictive of vulnerable code locations. If so, security experts can use this prediction to prioritize security inspection and testing efforts. The metrics we investigated fall into three categories: complexity, code churn, and developer activity metrics. We performed two empirical case studies on large, widely used open-source projects: the Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. The results indicate that 24 of the 28 metrics collected are discriminative of vulnerabilities for both projects. The models using all three types of metrics together predicted over 80 percent of the known vulnerable files with less than 25 percent false positives for both projects. Compared to a random selection of files for inspection and testing, these models would have reduced the number of files and the number of lines of code to inspect or test by over 71 and 28 percent, respectively, for both projects.

Index Terms—Fault prediction, software metrics, software security, vulnerability prediction.

1 Introduction

A single exploited software vulnerability[1] can cause severe damage to an organization. Annual world-wide losses caused from cyber attacks have been reported to be as high as \$226 billion [2]. Loss in stock market value in the days after an attack is estimated from \$50 to \$200 million per organization [2]. The importance of detecting and mitigating software vulnerabilities before software release is paramount.

Experience indicates that the detection and mitigation of vulnerabilities are best done by engineers specifically trained in software security and who "think like an attacker" in their daily work [3]. Therefore, security testers need to have specialized knowledge in and a mindset for what attackers will try. If we could predict which parts of the code are likely to be vulnerable, security experts can focus on these areas of highest risk. One way of predicting vulnerable modules is to build a statistical model using software metrics that measure the attributes of the software products and development process related to software vulnerabilities. Historically, prediction models trained using software metrics to find faults have been known to be effective [4], [5], [6], [7], [8], [9], [10].

However, prediction models must be trained on what they are intended to look for. Rather than arming the security expert with all the modules likely to contain faults, a security prediction model can point toward the set of modules likely to contain what a security expert is looking for: security vulnerabilities. Establishing predictive power in a security prediction model is challenging because security vulnerabilities and non-security-related faults have similar symptoms. Differentiating a vulnerability from a fault can be nebulous even to a human, much less a statistical model. Additionally, the number of reported security vulnerabil-

build a model based software metrics (complexity, code churn, …)

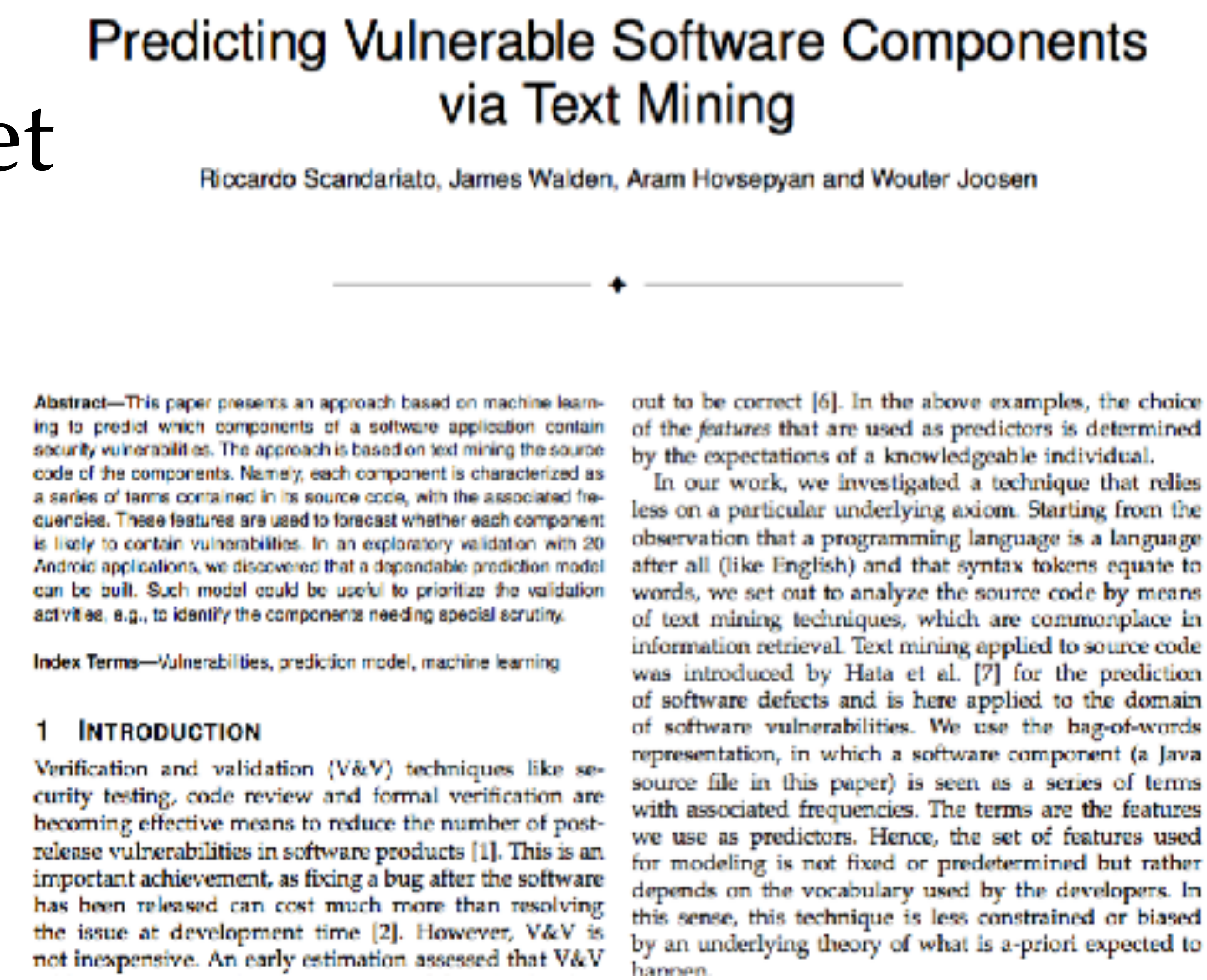
Overview

	Preprocessing	Learning
Software Metrics	Compute complexity metrics of each function (keeping sum, avg and max) code churn and the number of authors of every files.	Logistic regression

Text Mining

Text Mining

suggested by Scandariato et al. in 2014.



Predicting Vulnerable Software Components via Text Mining

Riccardo Scandariato, James Walden, Aram Hovsepyan and Wouter Joosen

Abstract—This paper presents an approach based on machine learning to predict which components of a software application contain security vulnerabilities. The approach is based on text mining the source code of the components. Namely, each component is characterized as a series of terms contained in its source code, with the associated frequencies. These features are used to forecast whether each component is likely to contain vulnerabilities. In an exploratory validation with 20 Android applications, we discovered that a dependable prediction model can be built. Such model could be useful to prioritize the validation activities, e.g., to identify the components needing special scrutiny.

Index Terms—Vulnerabilities, prediction model, machine learning

1 Introduction

Verification and validation (V&V) techniques like security testing, code review and formal verification are becoming effective means to reduce the number of post-release vulnerabilities in software products [1]. This is an important achievement, as fixing a bug after the software has been released can cost much more than resolving the issue at development time [2]. However, V&V is not inexpensive. An early estimation assessed that V&V

out to be correct [6]. In the above examples, the choice of the *features* that are used as predictors is determined by the expectations of a knowledgeable individual.

In our work, we investigated a technique that relies less on a particular underlying axiom. Starting from the observation that a programming language is a language after all (like English) and that syntax tokens equate to words, we set out to analyze the source code by means of text mining techniques, which are commonplace in information retrieval. Text mining applied to source code was introduced by Hata et al. [7] for the prediction of software defects and is here applied to the domain of software vulnerabilities. We use the bag-of-words representation, in which a software component (a Java source file in this paper) is seen as a series of terms with associated frequencies. The terms are the features we use as predictors. Hence, the set of features used for modeling is not fixed or predetermined but rather depends on the vocabulary used by the developers. In this sense, this technique is less constrained or biased by an underlying theory of what is a-priori expected to happen.

Text Mining

suggested by Scandariato et al. in 2014.

Aim : building a **model** requiring **no human intuition** for **feature selection**

Predicting Vulnerable Software Components via Text Mining

Riccardo Scandariato, James Walden, Aram Hovsepyan and Wouter Joosen

Abstract—This paper presents an approach based on machine learning to predict which components of a software application contain security vulnerabilities. The approach is based on text mining the source code of the components. Namely, each component is characterized as a series of terms contained in its source code, with the associated frequencies. These features are used to forecast whether each component is likely to contain vulnerabilities. In an exploratory validation with 20 Android applications, we discovered that a dependable prediction model can be built. Such model could be useful to prioritize the validation activities, e.g., to identify the components needing special scrutiny.

Index Terms—Vulnerabilities, prediction model, machine learning

1 Introduction

Verification and validation (V&V) techniques like security testing, code review and formal verification are becoming effective means to reduce the number of post-release vulnerabilities in software products [1]. This is an important achievement, as fixing a bug after the software has been released can cost much more than resolving the issue at development time [2]. However, V&V is not inexpensive. An early estimation assessed that V&V out to be correct [6]. In the above examples, the choice of the *features* that are used as predictors is determined by the expectations of a knowledgeable individual.

In our work, we investigated a technique that relies less on a particular underlying axiom. Starting from the observation that a programming language is a language after all (like English) and that syntax tokens equate to words, we set out to analyze the source code by means of text mining techniques, which are commonplace in information retrieval. Text mining applied to source code was introduced by Hata et al. [7] for the prediction of software defects and is here applied to the domain of software vulnerabilities. We use the bag-of-words representation, in which a software component (a Java source file in this paper) is seen as a series of terms with associated frequencies. The terms are the features we use as predictors. Hence, the set of features used for modeling is not fixed or predetermined but rather depends on the vocabulary used by the developers. In this sense, this technique is less constrained or biased by an underlying theory of what is a-priori expected to happen.

Text Mining

suggested by Scandariato et al. in 2014.

Aim : building a **model** requiring **no human intuition** for **feature selection**

Predicting Vulnerable Software Components via Text Mining

Riccardo Scandariato, James Walden, Aram Hovsepyan and Wouter Joosen

Abstract—This paper presents an approach based on machine learning to predict which components of a software application contain security vulnerabilities. The approach is based on text mining the source code of the components. Namely, each component is characterized as a series of terms contained in its source code, with the associated frequencies. These features are used to forecast whether each component is likely to contain vulnerabilities. In an exploratory validation with 20 Android applications, we discovered that a dependable prediction model can be built. Such model could be useful to prioritize the validation activities, e.g., to identify the components needing special scrutiny.

Index Terms—Vulnerabilities, prediction model, machine learning

1 Introduction

Verification and validation (V&V) techniques like security testing, code review and formal verification are becoming effective means to reduce the number of post-release vulnerabilities in software products [1]. This is an important achievement, as fixing a bug after the software has been released can cost much more than resolving the issue at development time [2]. However, V&V is not inexpensive. An early estimation assessed that V&V

out to be correct [6]. In the above examples, the choice of the *features* that are used as predictors is determined by the expectations of a knowledgeable individual.

In our work, we investigated a technique that relies less on a particular underlying axiom. Starting from the observation that a programming language is a language after all (like English) and that syntax tokens equate to words, we set out to analyze the source code by means of text mining techniques, which are commonplace in information retrieval. Text mining applied to source code was introduced by Hata et al. [7] for the prediction of software defects and is here applied to the domain of software vulnerabilities. We use the bag-of-words representation, in which a software component (a Java source file in this paper) is seen as a series of terms with associated frequencies. The terms are the features we use as predictors. Hence, the set of features used for modeling is not fixed or predetermined but rather depends on the vocabulary used by the developers. In this sense, this technique is less constrained or biased by an underlying theory of what is a-priori expected to happen.

build a model based on a **bag of word** extracted from a **file**

Overview

	Preprocessing	Learning
Text mining	Creating a bag of word (splitting the code according to the language grammar) for every files	• Discretisation of the features (making them boolean) • Remove of all features considered useless • Random Forest with 100 trees

Dataset

Introducing the dataset

based on **commit** and not **release**

Introducing the dataset

- **CVE-NVD database** as a **source** of **vulnerabilities**
- **Bugzilla** as a **source** of **bugs**

Introducing the dataset

- build **automatically**
- with the **latest data available**
- on the **Linux Kernel**

Overall dataset statistics

2006-June 2016

- **1,640 vulnerable files**, accounting for **743 vulnerabilities**
- **4,900 buggy files** related to **3,400 bug reports**
- **more than 50,000 files** in tot

Research Questions

- **RQ1**. Can we **distinguish** between **buggy** and **vulnerable files**?

Research Questions

- **RQ1**. Can we **distinguish** between **buggy** and **vulnerable files**?

- **RQ2**. Can we **distinguish** between **vulnerable** and **non vulnerable files**?

Research Questions

- **RQ1**. Can we **distinguish** between **buggy** and **vulnerable files**?
- **RQ2**. Can we **distinguish** between **vulnerable** and **non vulnerable files**?
- **RQ3**. Can we **predict future vulnerable** when **using past data**?

Research Questions

- **RQ1**. Can we **distinguish** between **buggy** and **vulnerable files**?

- **RQ2**. Can we **distinguish** between **vulnerable** and **non vulnerable files**?

- **RQ3**. Can we **predict future vulnerable** when **using past data**?
    - ✦ **Distinguish** between **buggy** and **vulnerable files**
    - ✦ **Distinguish** between **vulnerable** and **non vulnerable files**?

Experimental Dataset

*Buggy vs Vulnerable files

Experimental dataset

Can we **distinguish** between **buggy** and **vulnerable files**?

- files related to **bug report patches** **vs** files from **vulnerability patches**

- **ratio** 3.3 : 1

Realistic Dataset

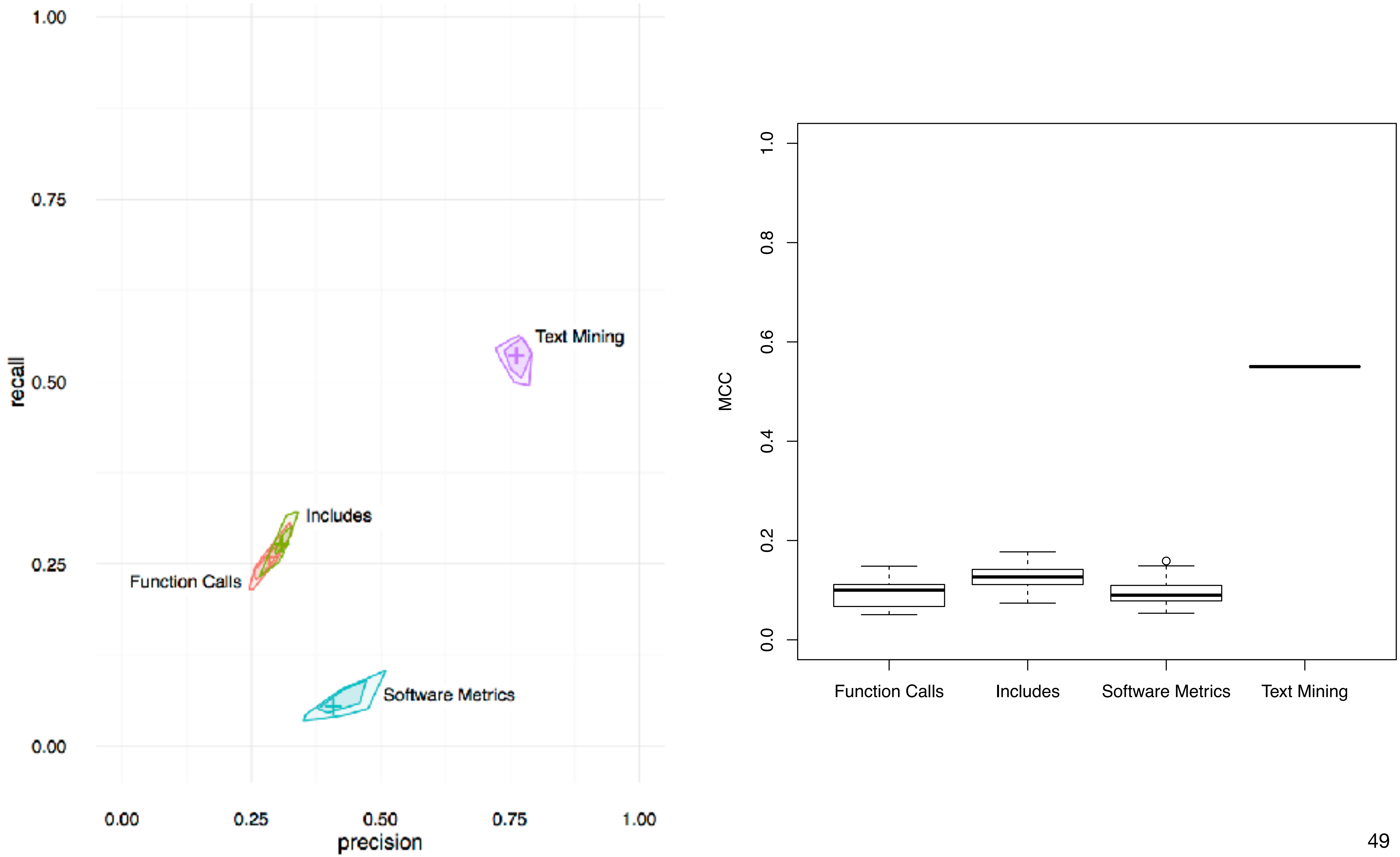
*Vulnerable vs Non-Vulnerable files

Realistic dataset

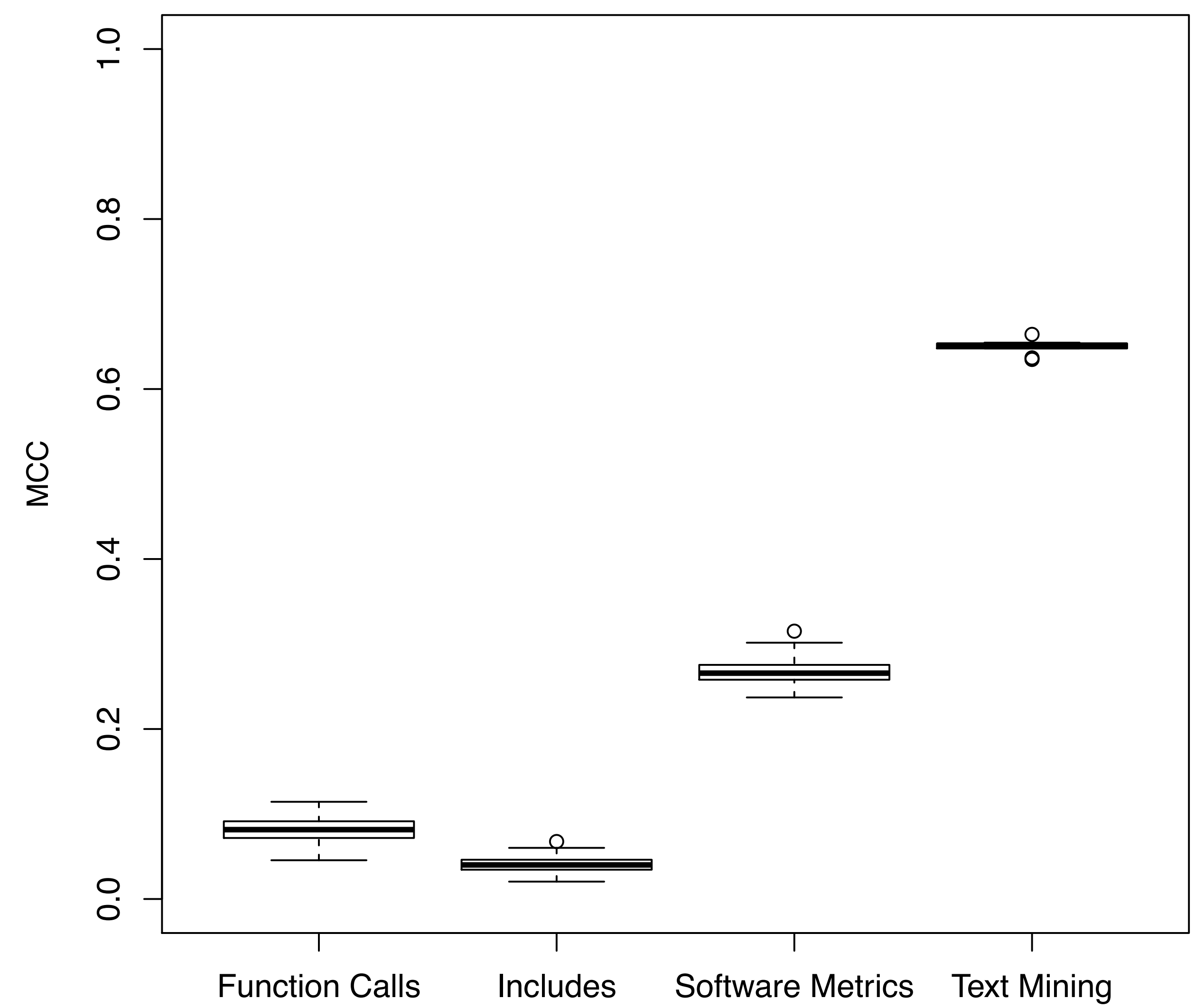
- Can we **distinguish** between **Vulnerable** and **Non-Vulnerable** files?
  - Reproduce observed **ratio** between different categories of files
    - **3%** of (likely) vulnerable files
    - **47%** of (likely) buggy files
    - **50%** of clear files

Evaluation

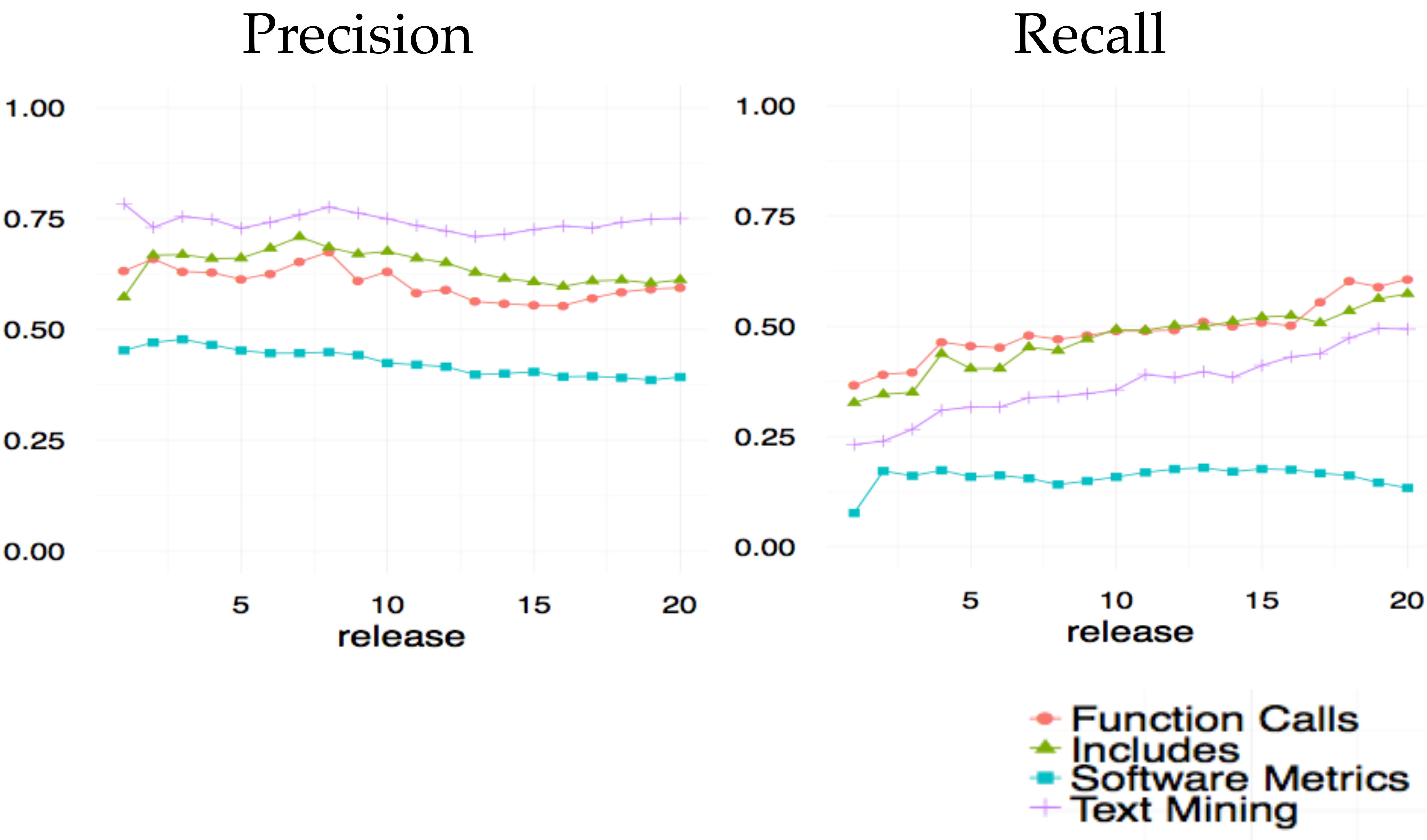
RQ1 - Bugs vs Vulnerabilities



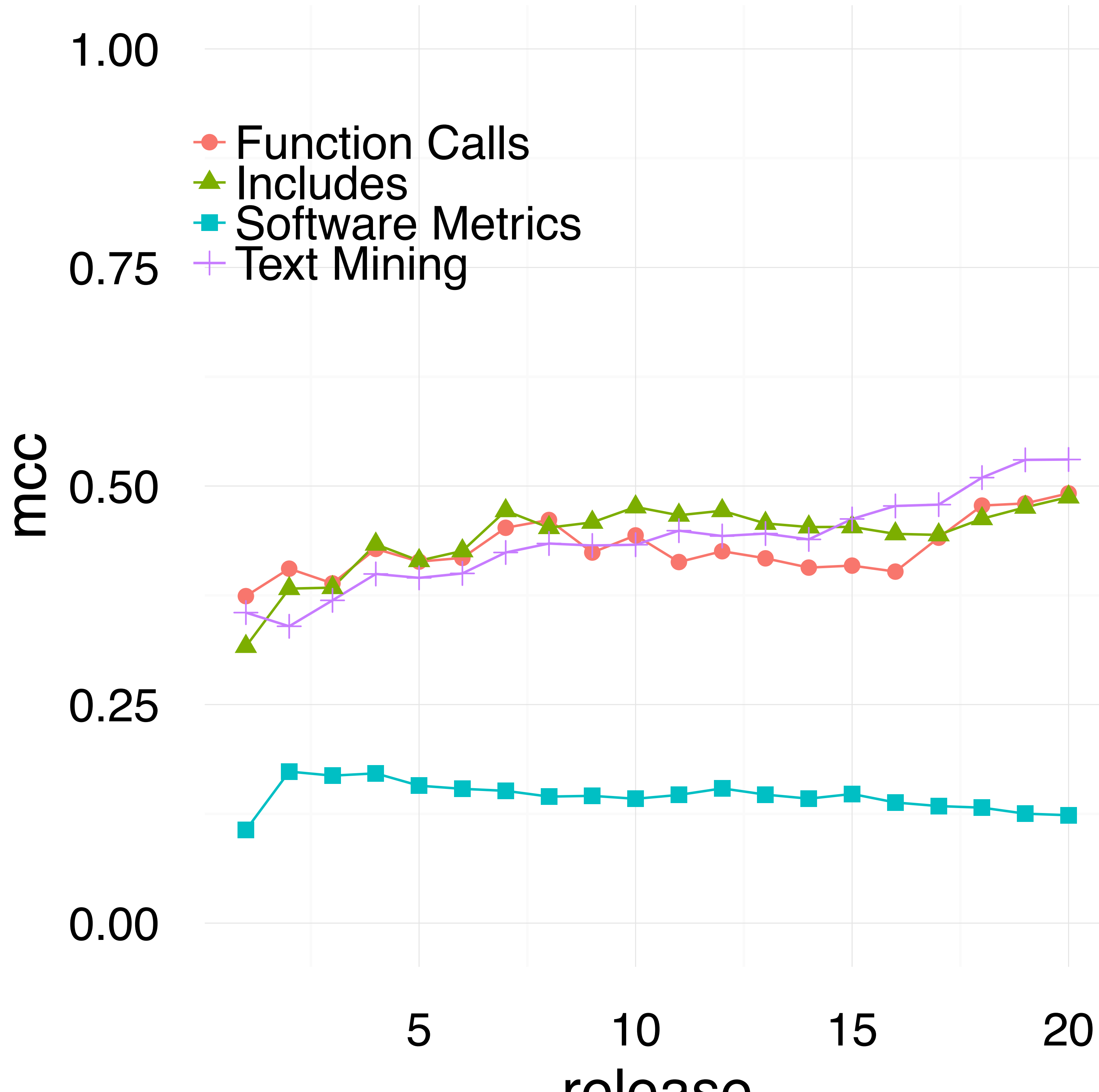
RQ2 - Vulnerable vs Non-



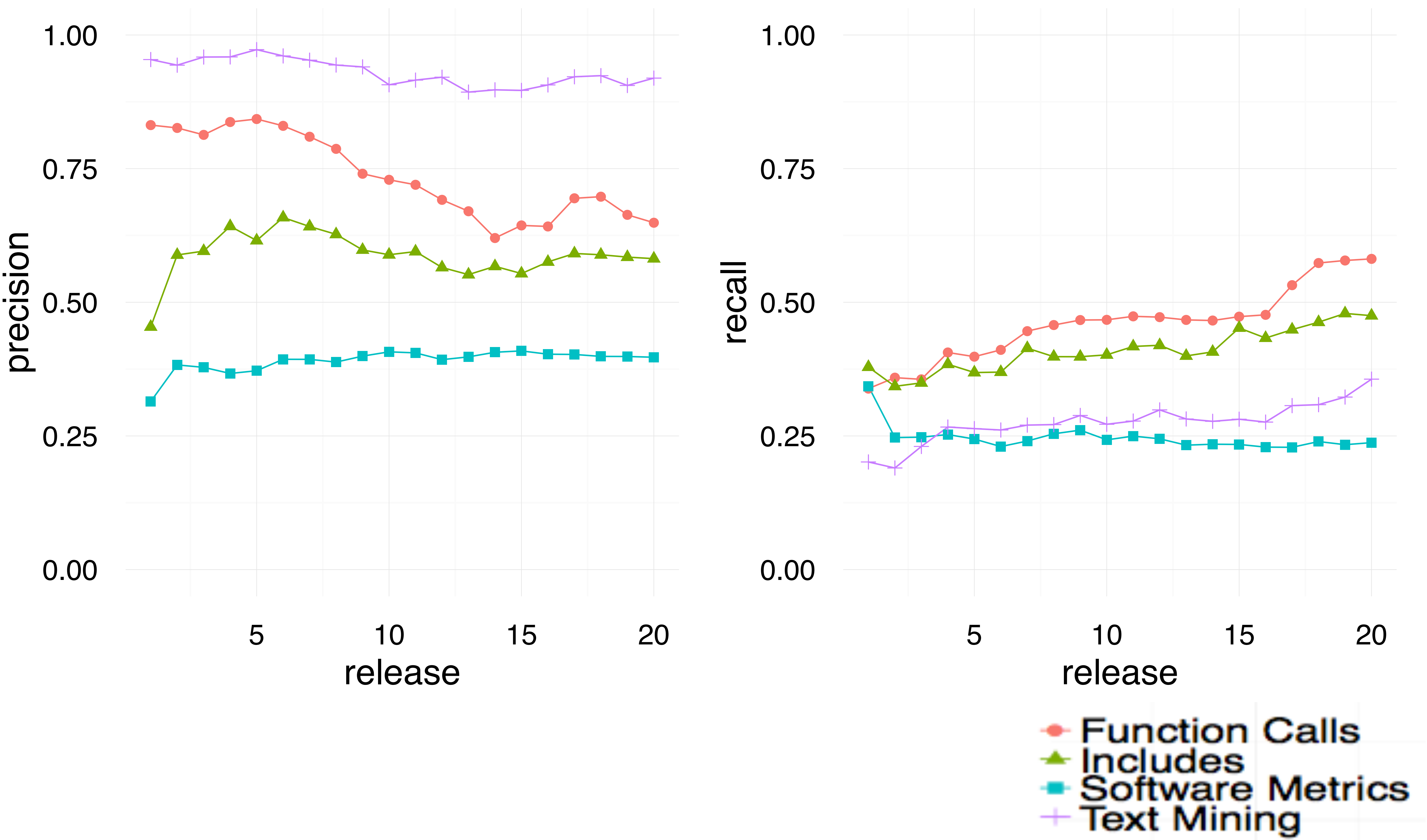
RQ3 Time - Bugs vs



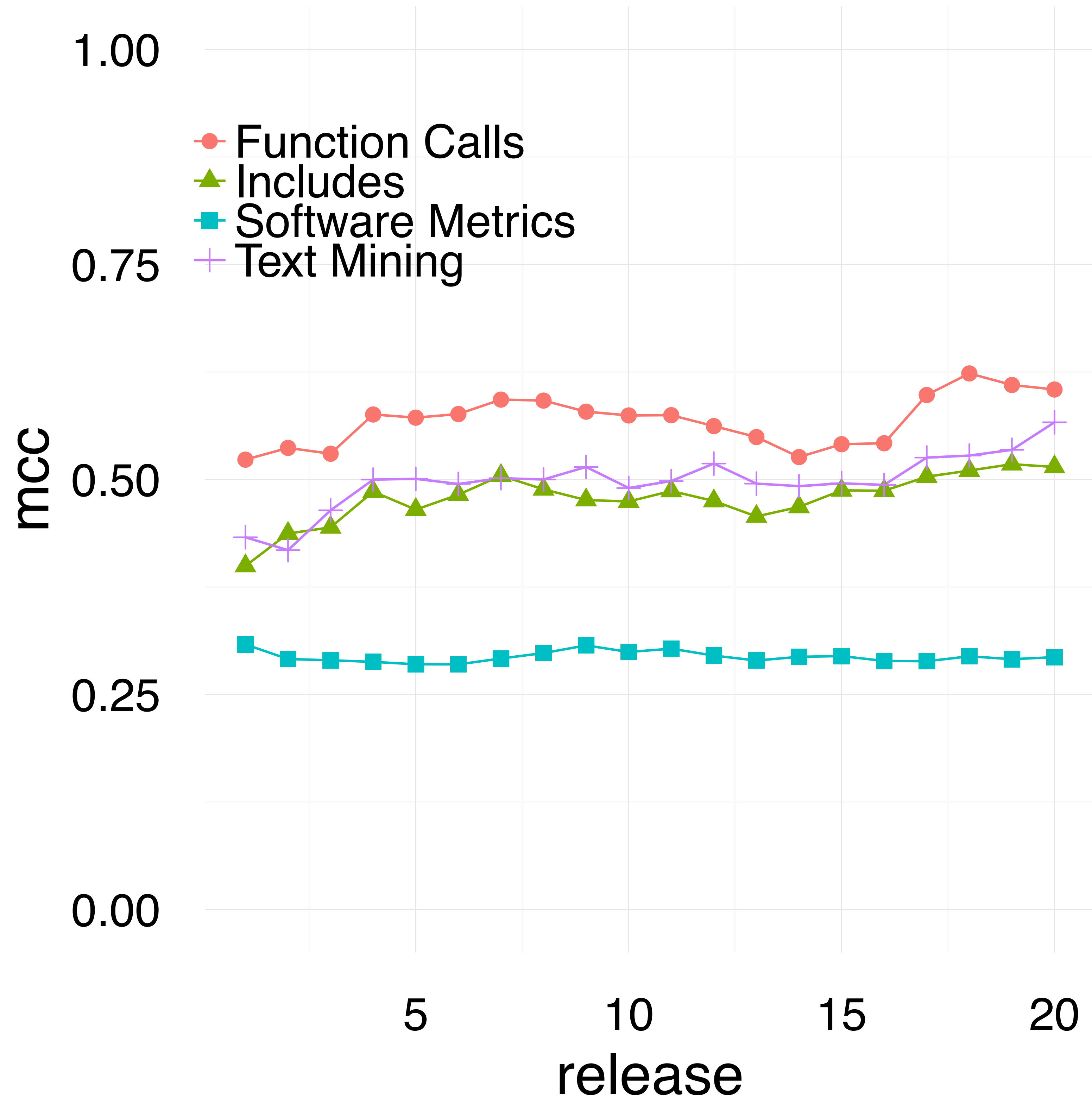
RQ3 Time - Bugs vs



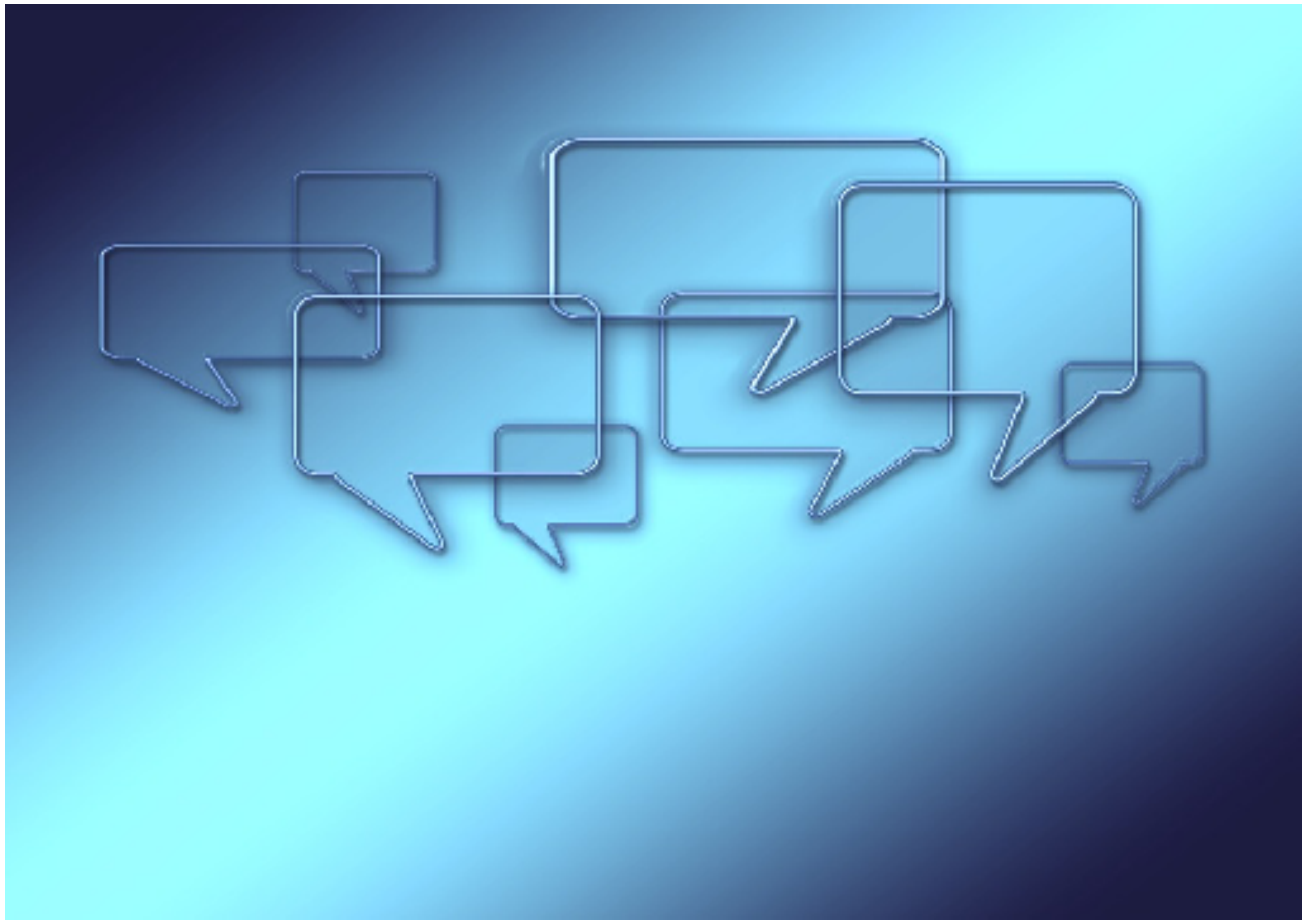
RQ3 Time - Vulnerable vs Non-



RQ3 Time - **Vulnerable** vs



Discussion - Findings



1

VPM's are working **well** with **historical data**

2

Good precision observed even with **unbalanced data**

In the **practical case**, the **best trade off** is in favour of **include and function calls**

3

In the **general case**, or favouring **precision** the best one is **text mining**.

4

Previous studies

Include and Function calls

There is no comparison with Metrics or Text Mining

There are no results related to time

In the context of Linux
we have similar results...

Reported	We found
Precision 70%	Precision 70%
Recall 45%	Recall 64%

Neuhaus et al. "Predicting vulnerable software components" *CCS'07*.

Previous studies

Software Metrics

Reported 10 fold cross validation **We found**

Precision 3-5, 9, 2-52% Precision 65%

Recall 87-90, 91, 66-79% Recall 22%

In the context of Linux
there are significant differences…

Reported results based on time **We found**

Precision 3% Precision 42 : 39%

Recall 79-85% Recall 16 : 24%

Shin et al. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities" *TSE*'11.
Shinand et al. "Cantraditionalfaultpredictionmodelsbeused for vulnerability prediction?" *ESE*'13.
Walden et al. "Predicting Vulnerable Components: Software Metrics vs Text Mining" *ISSRE'14*.

Previous studies

Text Mining

Reported 10 fold cross validation **We found**

Precision 90, 2-57%

Recall 77, 74-81%

Precision 76%

Recall 58%

In the context of Linux there are again significant differences

Reported results based on time **We found**

Precision 86%

Recall 77%

Precision 74 : 93%

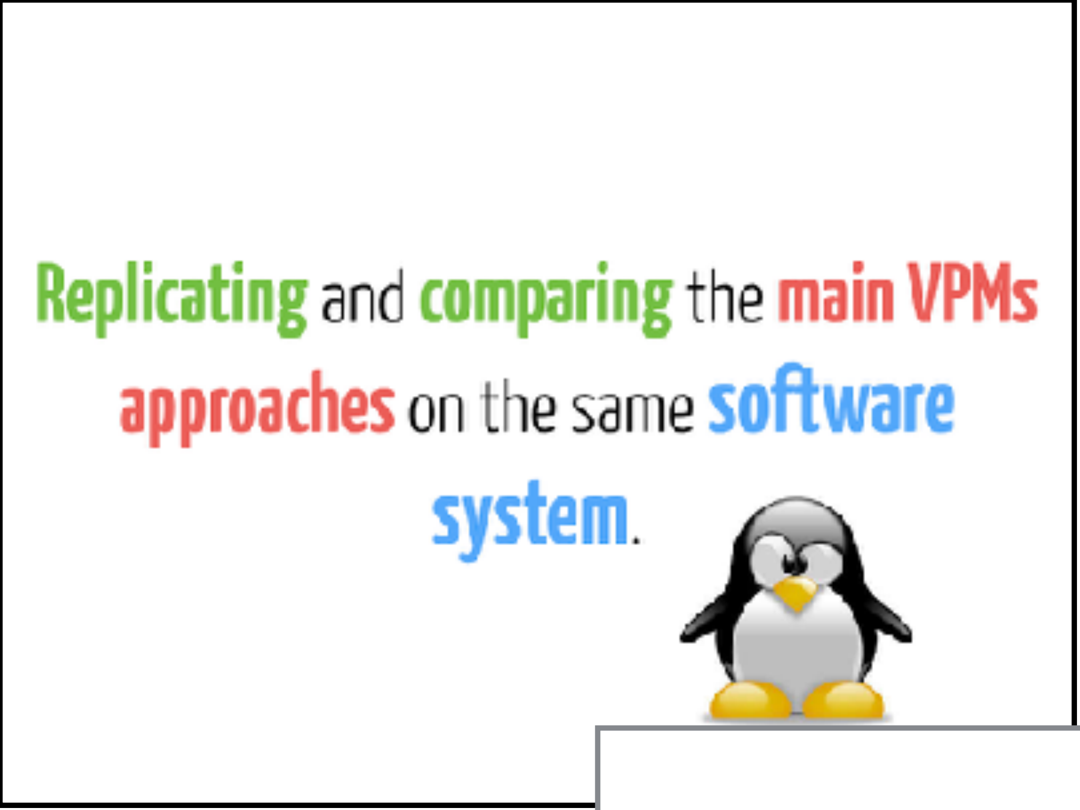
Recall 37 : 27%

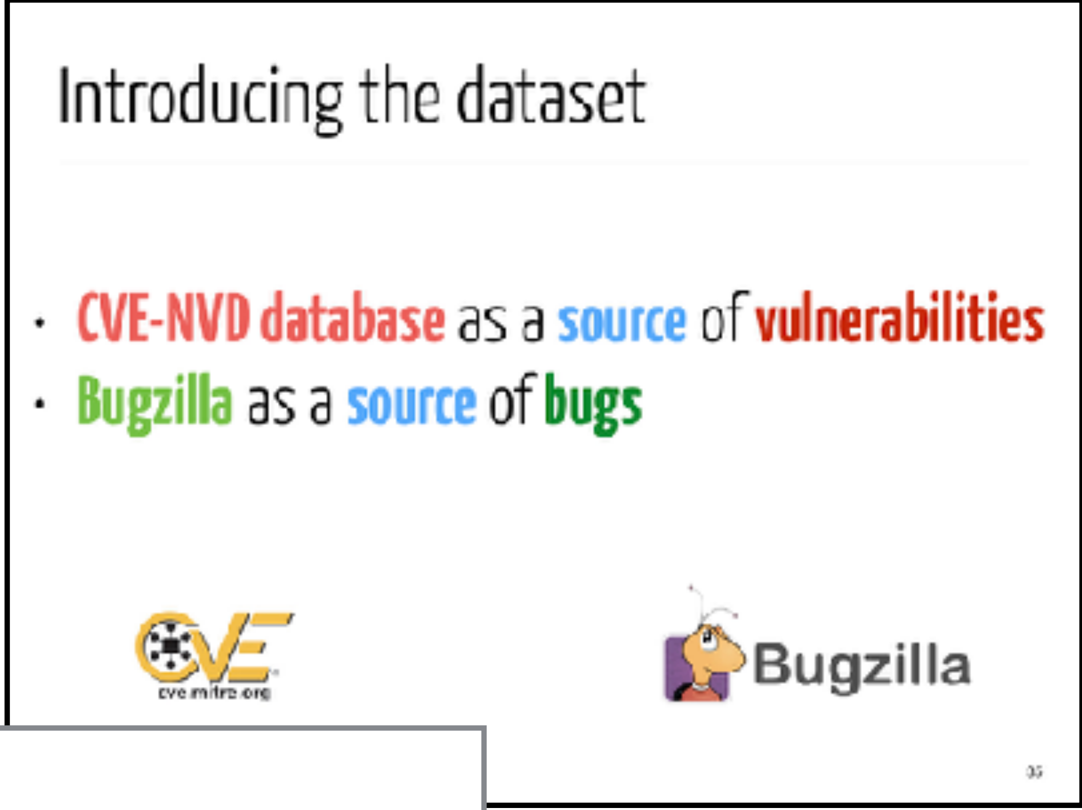
Scandariato et al."Predicting Vulnerable Software Components via Text Mining" *TSE*'14.

Walden et al. "Predicting Vulnerable Components: Software Metrics vs Text Mining" *ISSRE'14*.

DataSet and Replication package and **additional results** will be available soon…

Please contact Matthieu Jimenez (Matthieu.Jimenez@uni.lu)





Thank you for your attention !