MSR for Vulnerability Prediction

Mining Vulnerability-Contributing Commits

Emanuele Iannone

SeSa Lab @ University of Salerno, Italy

`eiannone@unisa.it`

Master Course “Cybersecurity Data Science”

Winter Semester 22/23

Who are you?


MOLISE
MAR ADRIATICO
LAZIO
Galluccio
MATESE
SANNIO
APPENNINO CAMPANO
Gaeta
PUGLIA
Benevento
Caserta
Volturno
Sabato
Calore
IRPINIA
Avellino
NAPOLI
Vesuvio
MONTI PICENTINI
Isola di Procida
Isola d'Ischia
Salerno
BASILICATA
Sorrento
Amalfi
Sele
Isola di Capri
PIANA DEL SELE
MAR TIRRENO
CILENTO
Punta Licosa
Vallo della Lucania
Sapri
CALABRIA

Pontecagnano Faiano


LAZIO
MOLISE
MAR
ADRIATICO
Galluccio
MATESE
SANNIO
APPENNINO
CAMPANO
PUGLIA
Gaeta
Benevento
Volturno
Caserta
Sabato
Calore
IRPINIA
Avellino
NAPOLI
Vesuvio
MONTI
PICENTINI
Isola di Procida
Isola d'Ischia
Salerno
BASILICATA
Sorrento
Ama
Sele
Isola di Capri
PIANA
MAR TI
ania
Sapri
CALABRIA


MOLISE
MAR ADRIATICO
LAZIO
Galluccio
MATESE
SANNIO
APPENNINO CAMPANO
PUGLIA
Gaeta
Benevento
Volturno
Caserta
Sabato
Calore
IRPINIA
Avellino
NAPOLI
Vesuvio
cida
GOMORRA
LA SERIE
Da un'idea di Roberto Saviano
BASILICATA
CILENTO
Vallo della Lucania
Sapri
CALABRIA


MOLISE
MAR ADRIATICO
LAZIO
Galluccio
MATESE
SANNIO
APPENNINO CAMPANO
PUGLIA
Gaeta
Benevento
Volturno
Caserta
Sabato
Calore
IRPINIA
Avellino
NAPOLI
Vesuvio
MONTI PICENTINI
Isola di Procida
Isola d'Ischia
Salerno
Sorrento
Sele
BASILICATA
a Lucania
Sapri
CALABRIA

Fisciano

DIPARTIMENTO DI INFORMATICA
UNISA.IT


DIPARTIMENTO DI INFORMATICA
UNISA.IT
F3
F

~$ whoami

Nationality 🌍 Italian

Emanuele Iannone

✉ eiannone@unisa.it

🌐 https://emaiannone.github.io/

@EmanueleIannon3

~$ whoami

Nationality	Italian
Age	26 y.o.

Emanuele Iannone

✉ eiannone@unisa.it

https://emaiannone.github.io/

@EmanueleIannon3

MSR for Vulnerability Prediction — About Me

~$ whoami

Emanuele Iannone

✉ eiannone@unisa.it

https://emaiannone.github.io/

@EmanueleIannon3

Nationality	Italian
Age	26 y.o.
Affiliation	University of Salerno Department of Computer Science Software Engineering (SeSa) Lab

~$ whoami

Emanuele Iannone

eiannone@unisa.it

https://emaiannone.github.io/

@EmanueleIannon3

Nationality	Italian	
Age	26 y.o.	
Affiliation	University of Salerno Department of Computer Science Software Engineering (SeSa) Lab	
Curriculum	B.Sc. in Computer Science, thesis on *Refactoring for Android Energy Consumption*	2015-2018

~$ whoami

Emanuele Iannone

✉ eiannone@unisa.it

https://emaiannone.github.io/

@EmanueleIannon3

Nationality	Italian	
Age	26 y.o.	
Affiliation	University of Salerno Department of Computer Science Software Engineering (SeSa) Lab	
Curriculum	B.Sc. in Computer Science, thesis on *Refactoring for Android Energy Consumption*	2015-2018
	M.Sc. in Computer Science, thesis on Test Generation for *Third-party Vulnerabilities*	2018-2020

~$ whoami

Emanuele Iannone

eiannone@unisa.it

https://emaiannone.github.io/

@EmanueleIannon3

Nationality	Italian	
Age	26 y.o.	
Affiliation	University of Salerno Department of Computer Science Software Engineering (SeSa) Lab	
Curriculum	B.Sc. in Computer Science, thesis on *Refactoring for Android Energy Consumption*	2015-2018
	M.Sc. in Computer Science, thesis on Test Generation for *Third-party Vulnerabilities*	2018-2020
	Ph.D. in Computer Science, researching on *Software Vulnerabilities Analysis, Prediction, and Assessment*	2020-2023

~$ whoami

Emanuele Iannone

✉ eiannone@unisa.it

https://emaiannone.github.io/

@EmanueleIannon3

Teaching

- **[BSc] Software Engineering**
- **[BSc] Fundamentals of AI**
- **[MSc] Software Maintenance & Evolution**
- **[MSc] Software Dependability**

15-2018

18-2020

20-2023

Software Vulnerabilities Analysis, Prediction, and Assessment

~$ whoami

Emanuele Iannone

✉ eiannone@unisa.it

https://emaiannone.github.io/

@EmanueleIannon3

My Research

Mining Software Repositories for Security

Machine Learning for Vulnerability Prediction

Evolutionary Algorithms for Vulnerability Assessment

Other topics of Empirical Software Engineering:

- Source Code Refactoring
- Program Comprehension
- Energy Consumption of Mobile Apps

Software Vulnerabilities Analysis, Prediction, and Assessment

Can we just start?

Vulnerability-contributing commit (VCC)

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

“A commit that contributed to the introduction of a post-release vulnerability.”

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

"A commit that contributed to the introduction of a post-release vulnerability."

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

> *“A commit that contributed to the introduction of a post-release vulnerability.”*

Code changes that “move” the code toward the state in which it contains the weakness.

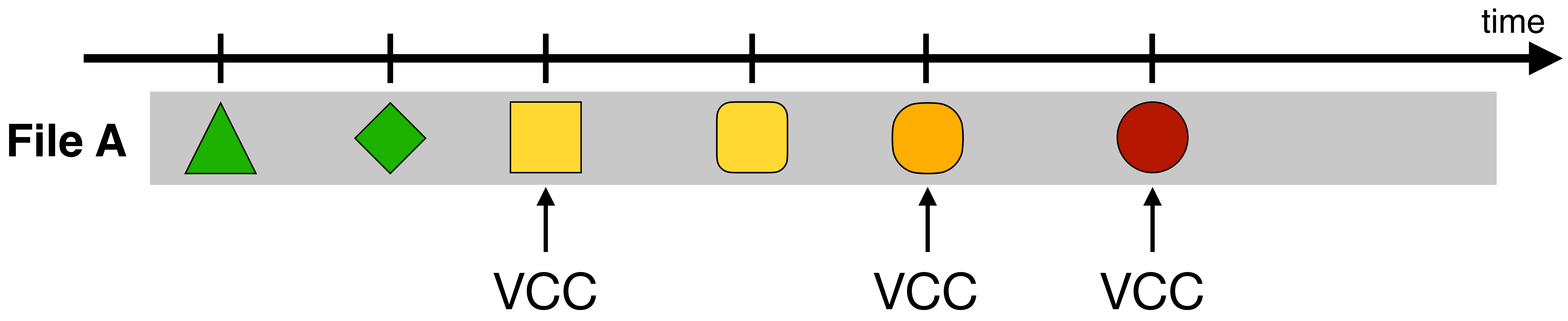
A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

> *“A commit that contributed to the introduction of a post-release vulnerability.”*

Code changes that “move” the code toward the state in which it contains the weakness.



A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

“A commit that contributed to the introduction of a post-release vulnerability.”

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

“A commit that contributed to the introduction of a post-release vulnerability.”

The concept is irrelevant for pre-release vulnerabilities, fixed the exposure to externals.

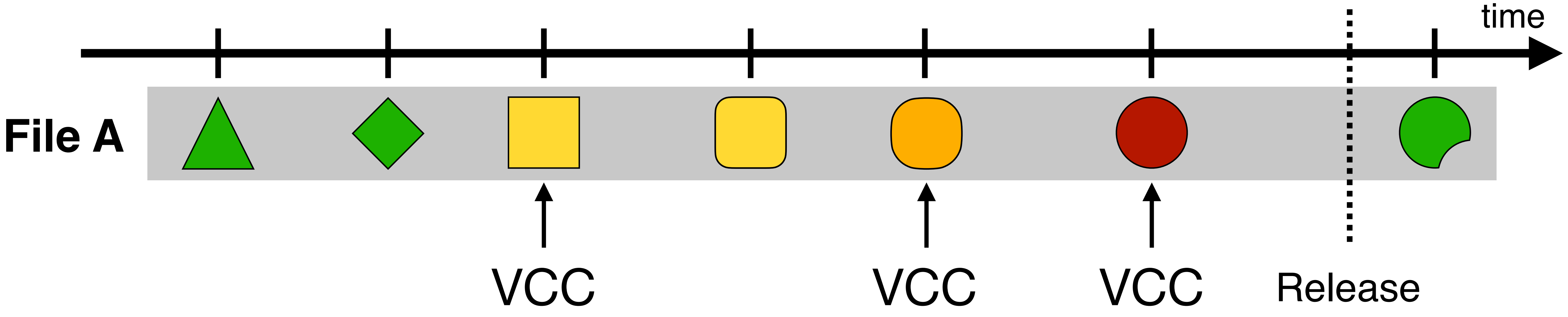
A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

"A commit that contributed to the introduction of a post-release vulnerability."

The concept is irrelevant for pre-release vulnerabilities, fixed the exposure to externals.



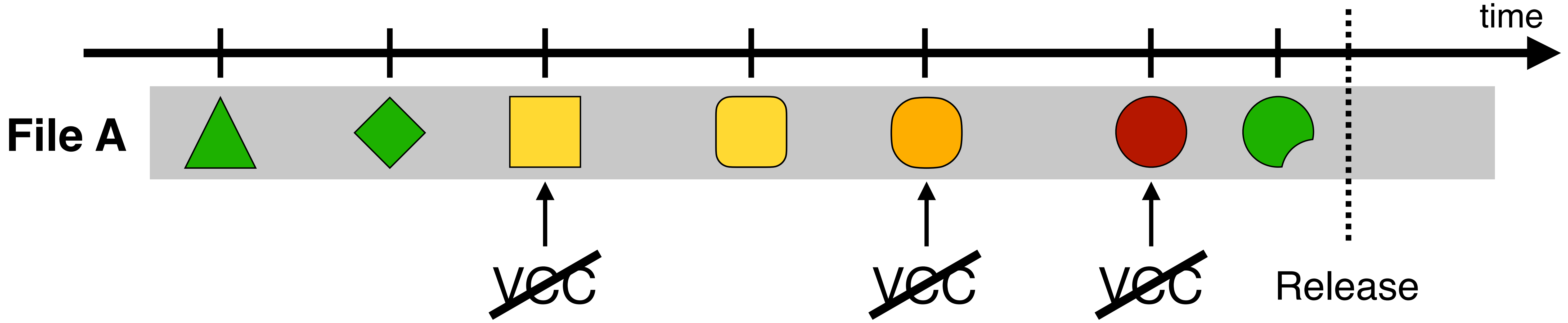
A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Vulnerability-contributing commit (VCC)

Andrew (Andy) Meneely

“A commit that contributed to the introduction of a post-release vulnerability.”

The concept is irrelevant for pre-release vulnerabilities, fixed the exposure to externals.



A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Example of a VCC

CVE-2019-11274

“Cloud Foundry UAA, versions prior to 74.0.0, is vulnerable to an XSS attack. A remote unauthenticated malicious attacker could craft a URL that contains a SCIM filter that contains malicious JavaScript, which older browsers may execute.”

Example of a VCC

CVE-2019-11274

“Cloud Foundry UAA, versions prior to 74.0.0, is vulnerable to an XSS attack. A remote unauthenticated malicious attacker could craft a URL that contains a SCIM filter that contains malicious JavaScript, which older browsers may execute.”

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Example of a VCC

CVE-2019-11274

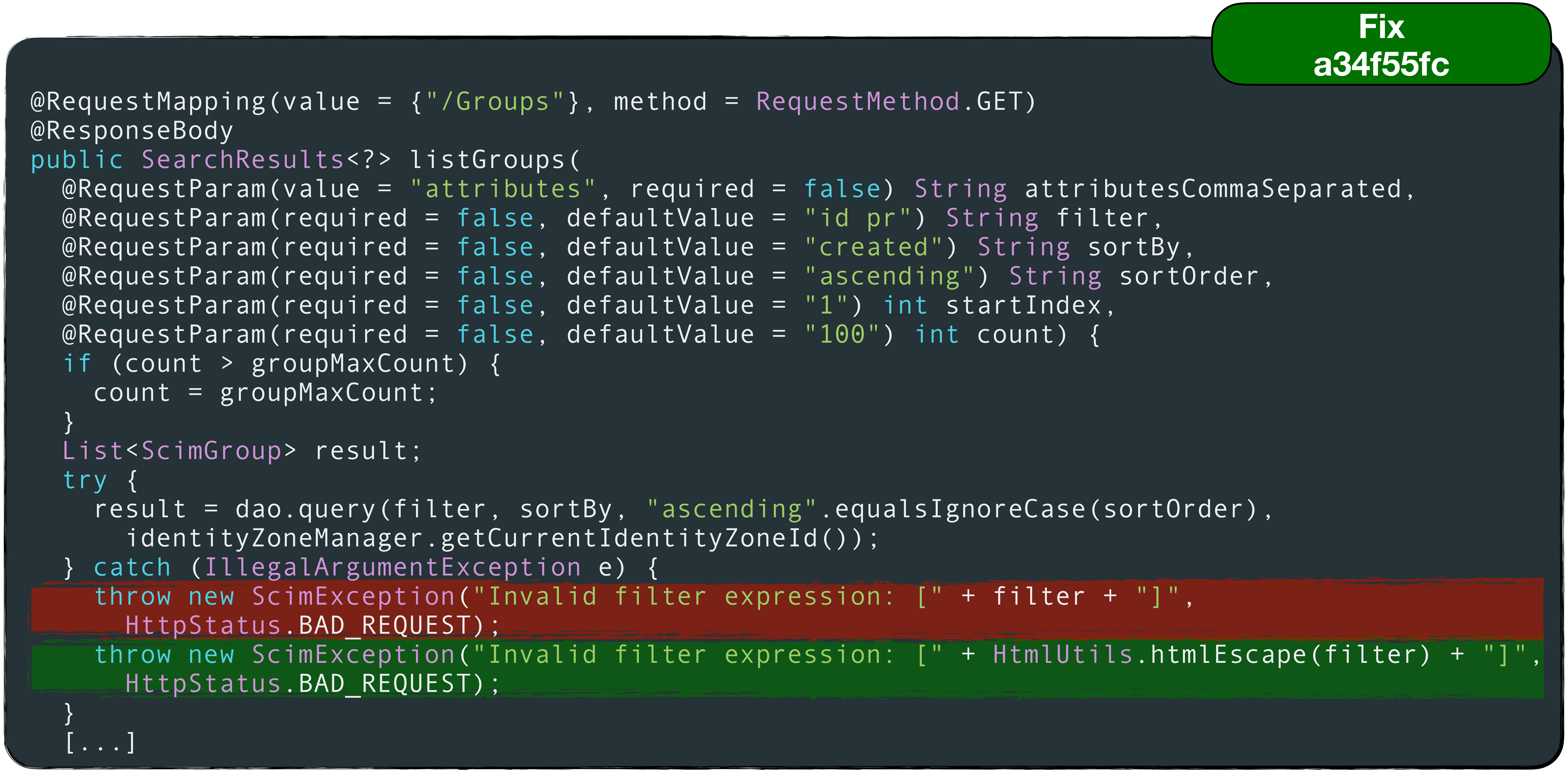
“Cloud Foundry UAA, versions prior to 74.0.0, is vulnerable to an XSS attack. A remote unauthenticated malicious attacker could craft a URL that contains a SCIM filter that contains malicious JavaScript, which older browsers may execute.”

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

We expect unescaped or unvalidated data supplied from the user via URL parameters that end up directly in the response.

Example of a VCC

CVE-2019-11274



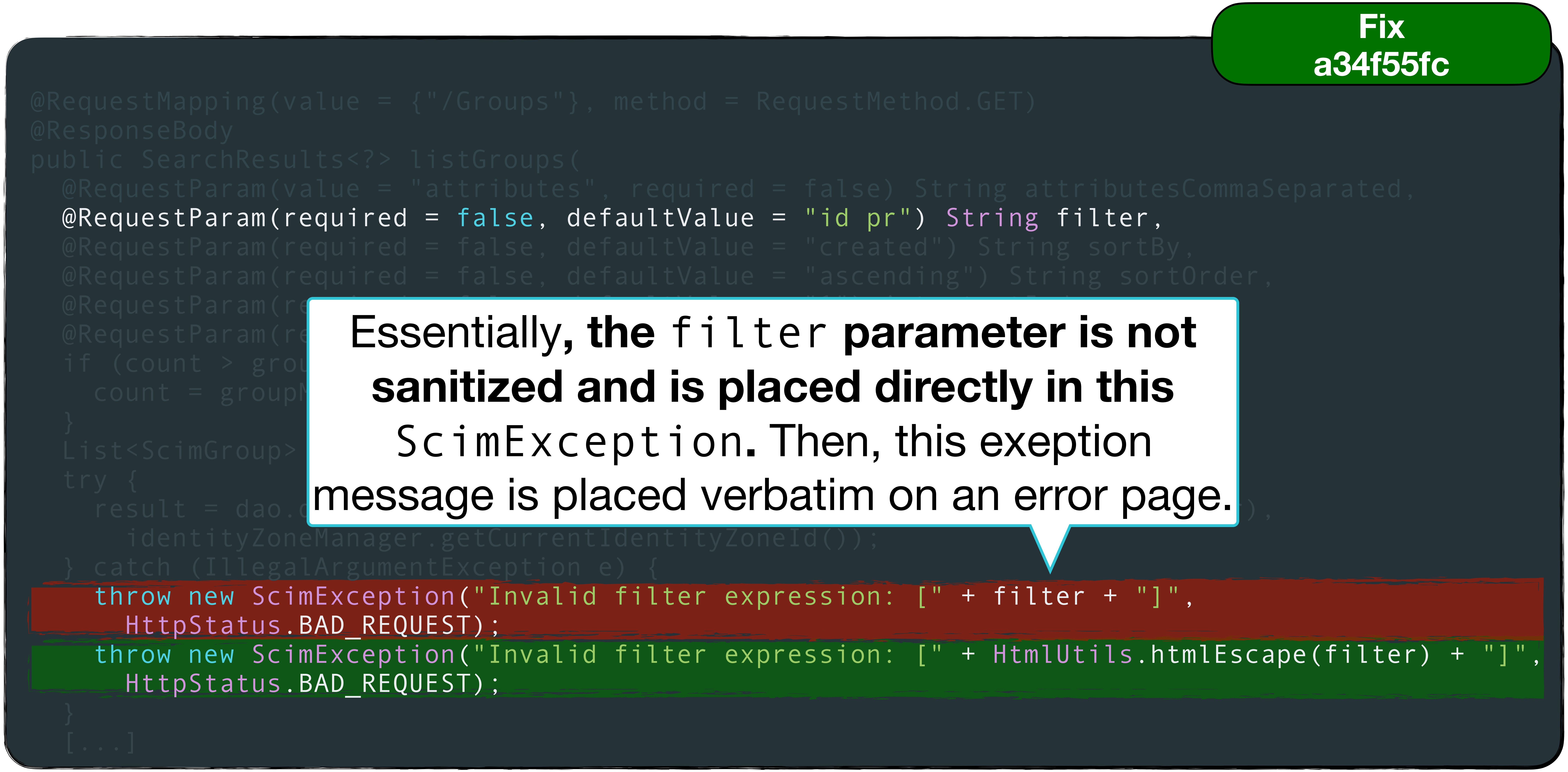
Fix
a34f55fc

```java
@RequestMapping(value = {"/Groups"}, method = RequestMethod.GET)
@ResponseBody
public SearchResults<?> listGroups(
  @RequestParam(value = "attributes", required = false) String attributesCommaSeparated,
  @RequestParam(required = false, defaultValue = "id pr") String filter,
  @RequestParam(required = false, defaultValue = "created") String sortBy,
  @RequestParam(required = false, defaultValue = "ascending") String sortOrder,
  @RequestParam(required = false, defaultValue = "1") int startIndex,
  @RequestParam(required = false, defaultValue = "100") int count) {
  if (count > groupMaxCount) {
    count = groupMaxCount;
  }
  List<ScimGroup> result;
  try {
    result = dao.query(filter, sortBy, "ascending".equalsIgnoreCase(sortOrder),
      identityZoneManager.getCurrentIdentityZoneId());
  } catch (IllegalArgumentException e) {
    throw new ScimException("Invalid filter expression: [" + filter + "]",
      HttpStatus.BAD_REQUEST);
    throw new ScimException("Invalid filter expression: [" + HtmlUtils.htmlEscape(filter) + "]",
      HttpStatus.BAD_REQUEST);
  }
  [...]
```

Example of a VCC

CVE-2019-11274

Fix
a34f55fc

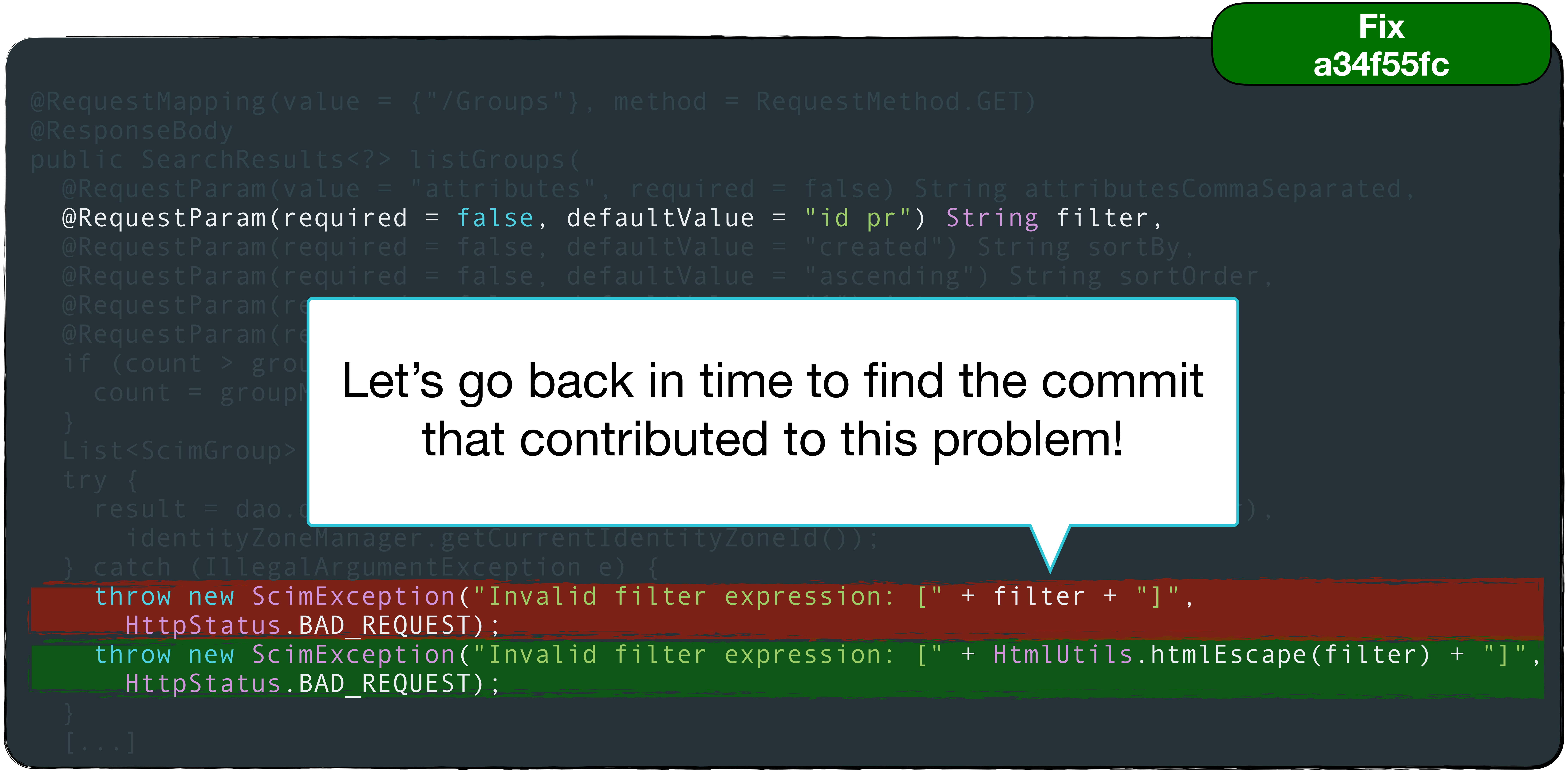


```
@RequestMapping(value = {"/Groups"}, method = RequestMethod.GET)
@ResponseBody
public SearchResults<?> listGroups(
  @RequestParam(value = "attributes", required = false) String attributesCommaSeparated,
  @RequestParam(required = false, defaultValue = "id pr") String filter,
  @RequestParam(required = false, defaultValue = "created") String sortBy,
  @RequestParam(required = false, defaultValue = "ascending") String sortOrder,
  @RequestParam(re
  @RequestParam(re
  if (count > grou
    count = groupM
  }
  List<ScimGroup>
  try {
    result = dao.c                                                           ),
      identityZoneManager.getCurrentIdentityZoneId());
  } catch (IllegalArgumentException e) {
    throw new ScimException("Invalid filter expression: [" + filter + "]",
      HttpStatus.BAD_REQUEST);
    throw new ScimException("Invalid filter expression: [" + HtmlUtils.htmlEscape(filter) + "]",
      HttpStatus.BAD_REQUEST);
  }
  [...]
```

Essentially, **the `filter` parameter is not sanitized and is placed directly in this `ScimException`.** Then, this exeption message is placed verbatim on an error page.

Example of a VCC

CVE-2019-11274



Fix
a34f55fc

```
@RequestMapping(value = {"/Groups"}, method = RequestMethod.GET)
@ResponseBody
public SearchResults<?> listGroups(
  @RequestParam(value = "attributes", required = false) String attributesCommaSeparated,
  @RequestParam(required = false, defaultValue = "id pr") String filter,
  @RequestParam(required = false, defaultValue = "created") String sortBy,
  @RequestParam(required = false, defaultValue = "ascending") String sortOrder,
  @RequestParam(re
  @RequestParam(re
  if (count > grou
    count = groupM
  }
  List<ScimGroup>
  try {
    result = dao.c                                                           ),
      identityZoneManager.getCurrentIdentityZoneId());
  } catch (IllegalArgumentException e) {
    throw new ScimException("Invalid filter expression: [" + filter + "]",
      HttpStatus.BAD_REQUEST);
    throw new ScimException("Invalid filter expression: [" + HtmlUtils.htmlEscape(filter) + "]",
      HttpStatus.BAD_REQUEST);
  }
  [...]
```

Let’s go back in time to find the commit that contributed to this problem!

Example of a VCC

CVE-2019-11274

VCC
bb8ff8f4

```
@RequestMapping(value = { "/Groups/External/list" }, method = RequestMethod.GET)
@ResponseBody
public SearchResults<?> listExternalGroups(
  @RequestParam(required = false, defaultValue = "1") int startIndex,
  @RequestParam(required = false, defaultValue = "100") int count) {
  String filter = "";
  List<ScimGroupExternalMember> result;
  try {
    result = externalMembershipManager.query(filter);
  } catch (IllegalArgumentException e) {
    throw new ScimException("Invalid filter expression: [" + filter + "]",
      HttpStatus.BAD_REQUEST);
  }
  [...]
```

Example of a VCC

CVE-2019-11274

VCC
bb8ff8f4

```
@RequestMapping(value = { "/Groups/External/list" }, method = RequestMethod.GET)
@ResponseBody
public SearchResults<?> listExternalGroups(
  @RequestParam(required = false, defaultValue = "1") int startIndex,
  @RequestParam(required = false, defaultValue = "100") int count) {
  String filter = "";
  List<ScimGroupExternalMember> result;
  try {
    result = externalMembershipManager.query(filter);
  } catch (IllegalArgumentException e) {
    throw new ScimException("Invalid filter expression: [" + filter + "]",
      HttpStatus.BAD_REQUEST);
  }
  [...]
```

This was the first revision where the `filter` parameters was put inside the exception message: the vulnerability was there since the method (with a different name) was born.

Terminology

2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement

When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits

Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa,
Alberto Rodriguez Tejeda, Matthew Mokary, Brian Spates
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
andy@se.rit.edu, {hxs8839, ajm661, acr921, mxm6060, bxs4361}@rit.edu

***Abstract*—Security is a harsh reality for software teams today. Developers must engineer secure software by preventing vulnerabilities, which are design and coding mistakes that have security consequences. Even in open source projects, vulnerable source code can remain unnoticed for years. In this paper, we traced 68 vulnerabilities in the Apache HTTP server back to the version control commits that contributed the vulnerable code originally. We manually found 124 Vulnerability-Contributing Commits (VCCs), spanning 17 years. In this exploratory study, we analyzed these VCCs quantitatively and qualitatively with the over-arching question: "What could developers have looked for to identify security concerns in this commit?" Specifically, we examined the size of the commit via code churn metrics, the amount developers overwrite each others' code via interactive churn metrics, exposure time between VCC and fix, and dissemination of the VCC to the development community via release notes and voting mechanisms. Our results show that VCCs are large: more than twice as much code churn on average than non-VCCs, even when normalized against lines of code. Furthermore, a commit was twice as likely to be a VCC when the author was a new developer to the source code. The insight from this study can help developers understand how vulnerabilities originate in a system so that security-related mistakes can be prevented or caught in the future.**

***Index Terms*— vulnerability, churn, socio-technical, empirical.**

I. INTRODUCTION

Security is a harsh reality for software teams today. Insecure software is not only expensive to maintain, but can cause immeasurable damage to a brand, or worse, to the livelihood of customers, patients, and citizens.

To software developers, the key to secure software lies in preventing vulnerabilities. Software vulnerabilities are special types of "faults that violate an [implicit or explicit] security policy" [1]. If developers want to find and fix vulnerabilities they must focus beyond making the system work as specified and prevent the system's functionality from being abused. According to security experts [2]–[4], finding vulnerabilities requires expertise in both the specific product and in software security in general.

The field of engineering secure software has a plethora of security practices for finding vulnerabilities, such as threat modeling, penetration testing, code inspections, misuse and abuse cases [5], and automated static analysis [2]–[4]. While these practices have been shown to be effective, they can also be inefficient. Development teams are then faced with the challenge of prioritizing their fortification efforts within the entire development process. Developers might know what is possible, but lack a firm grip on what is probable. As a result, an uninformed development team can easily focus on the wrong areas for fortification.

Fortunately, an historical, longitudinal analysis of how vulnerabilities originated in professional products can inform fortification prioritization. Understanding the specific trends of how vulnerabilities can arise in a software development product can help developers understand where to look and what to look for in their own product. Some of these trends have been quantified in vulnerability prediction [6]–[10] studies using metrics aggregated at the file level, but little has been done to explore the original coding mistakes that contributed the vulnerabilities in the first place. In this study, we have identified and analyzed original coding mistakes as Vulnerability-Contributing Commits (VCCs), or commits in the version control repository that contributed to the introduction of a post-release vulnerability.

A myriad of factors can lead to the introduction and lack of detection of vulnerabilities. A developer may make a single massive change to the system, leaving his peers with an overwhelmingly large review. Furthermore, a developer may make small, incremental changes, but his work might be affecting the work of many other developers. Or, a developer may forget to disseminate her work in the change notes and so the code may miss out on be reviewed entirely.

The objective of this research is to improve software security by analyzing the size, interactive churn, and community dissemination of VCCs. We conducted an empirical case study of the Apache HTTP Server project (HTTPD). Using a multi-researcher, cross-validating, semi-automated, semi-manual process, we identified the VCCs for each known post-release vulnerability in HTTPD. To explore **commit size,** we analyzed three code churn metrics. **Interactive churn** is a suite of five recently-developed [6] socio-technical variants of code churn metrics that measure the degree to which developers' changes overwrite each others' code at the line level. To explore **community dissemination,** we analyzed the

978-0-7695-5056-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ESEM.2013.19

The core idea behind VCCs is not new to the MSR world, and stems the from research on traditional bugs.

Terminology

2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement

When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits

Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa,
Alberto Rodríguez Tejeda, Matthew Mokary, Brian Spates
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
andy@se.rit.edu, {hxs8839, ajm661, acr921, mxm6060, bxs4361}@rit.edu

***Abstract*—Security is a harsh reality for software teams today. Developers must engineer secure software by preventing vulnerabilities, which are design and coding mistakes that have security consequences. Even in open source projects, vulnerable source code can remain unnoticed for years. In this paper, we traced 68 vulnerabilities in the Apache HTTP server back to the version control commits that contributed the vulnerable code originally. We manually found 124 Vulnerability-Contributing Commits (VCCs), spanning 17 years. In this exploratory study, we analyzed these VCCs quantitatively and qualitatively with the over-arching question: "What could developers have looked for to identify security concerns in this commit?" Specifically, we examined the size of the commit via code churn metrics, the amount developers overwrite each others' code via interactive churn metrics, exposure time between VCC and fix, and dissemination of the VCC to the development community via release notes and voting mechanisms. Our results show that VCCs are large: more than twice as much code churn on average than non-VCCs, even when normalized against lines of code. Furthermore, a commit was twice as likely to be a VCC when the author was a new developer to the source code. The insight from this study can help developers understand how vulnerabilities originate in a system so that security-related mistakes can be prevented or caught in the future.**

***Index Terms*— vulnerability, churn, socio-technical, empirical.**

I. INTRODUCTION

Security is a harsh reality for software teams today. Insecure software is not only expensive to maintain, but can cause immeasurable damage to a brand, or worse, to the livelihood of customers, patients, and citizens.

To software developers, the key to secure software lies in preventing vulnerabilities. Software vulnerabilities are special types of "faults that violate an [implicit or explicit] security policy" [1]. If developers want to find and fix vulnerabilities they must focus beyond making the system work as specified and prevent the system's functionality from being abused. According to security experts [2]–[4], finding vulnerabilities requires expertise in both the specific product and in software security in general.

The field of engineering secure software has a plethora of security practices for finding vulnerabilities, such as threat modeling, penetration testing, code inspections, misuse and abuse cases [5], and automated static analysis [2]–[4]. While these practices have been shown to be effective, they can also be inefficient. Development teams are then faced with the challenge of prioritizing their fortification efforts within the entire development process. Developers might know what is possible, but lack a firm grip on what is probable. As a result, an uninformed development team can easily focus on the wrong areas for fortification.

Fortunately, an historical, longitudinal analysis of how vulnerabilities originated in professional products can inform fortification prioritization. Understanding the specific trends of how vulnerabilities can arise in a software development product can help developers understand where to look and what to look for in their own product. Some of these trends have been quantified in vulnerability prediction [6]–[10] studies using metrics aggregated at the file level, but little has been done to explore the original coding mistakes that contributed the vulnerabilities in the first place. In this study, we have identified and analyzed original coding mistakes as Vulnerability-Contributing Commits (VCCs), or commits in the version control repository that contributed to the introduction of a post-release vulnerability.

A myriad of factors can lead to the introduction and lack of detection of vulnerabilities. A developer may make a single massive change to the system, leaving his peers with an overwhelmingly large review. Furthermore, a developer may make small, incremental changes, but his work might be affecting the work of many other developers. Or, a developer may forget to disseminate her work in the change notes and so the code may miss out on be reviewed entirely.

The objective of this research is to improve software security by analyzing the size, interactive churn, and community dissemination of VCCs. We conducted an empirical case study of the Apache HTTP Server project (HTTPD). Using a multi-researcher, cross-validating, semi-automated, semi-manual process, we identified the VCCs for each known post-release vulnerability in HTTPD. To explore **commit size,** we analyzed three code churn metrics. **Interactive churn** is a suite of five recently-developed [6] socio-technical variants of code churn metrics that measure the degree to which developers' changes overwrite each others' code at the line level. To explore **community dissemination,** we analyzed the

978-0-7695-5056-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ESEM.2013.19

The core idea behind VCCs is not new to the MSR world, and stems the from research on traditional bugs.

Fix-inducing Change

Bug-introducing Change

Bug-injecting Change

Bug-inducing Change

Terminology

2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement

When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits

Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa,
Alberto Rodríguez Tejeda, Matthew Mokary, Brian Spates
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
andy@se.rit.edu, {hxs8839, ajm661, acr921, mxm6060, bxs4361}@rit.edu

***Abstract*—Security is a harsh reality for software teams today. Developers must engineer secure software by preventing vulnerabilities, which are design and coding mistakes that have security consequences. Even in open source projects, vulnerable source code can remain unnoticed for years. In this paper, we traced 68 vulnerabilities in the Apache HTTP server back to the version control commits that contributed the vulnerable code originally. We manually found 124 Vulnerability-Contributing Commits (VCCs), spanning 17 years. In this exploratory study, we analyzed these VCCs quantitatively and qualitatively with the over-arching question: "What could developers have looked for to identify security concerns in this commit?" Specifically, we examined the size of the commit via code churn metrics, the amount developers overwrite each others' code via interactive churn metrics, exposure time between VCC and fix, and dissemination of the VCC to the development community via release notes and voting mechanisms. Our results show that VCCs are large: more than twice as much code churn on average than non-VCCs, even when normalized against lines of code. Furthermore, a commit was twice as likely to be a VCC when the author was a new developer to the source code. The insight from this study can help developers understand how vulnerabilities originate in a system so that security-related mistakes can be prevented or caught in the future.**

***Index Terms*— vulnerability, churn, socio-technical, empirical.**

I. INTRODUCTION

Security is a harsh reality for software teams today. Insecure software is not only expensive to maintain, but can cause immeasurable damage to a brand, or worse, to the livelihood of customers, patients, and citizens.

To software developers, the key to secure software lies in preventing vulnerabilities. Software vulnerabilities are special types of "faults that violate an [implicit or explicit] security policy" [1]. If developers want to find and fix vulnerabilities they must focus beyond making the system work as specified and prevent the system's functionality from being abused. According to security experts [2]–[4], finding vulnerabilities requires expertise in both the specific product and in software security in general.

The field of engineering secure software has a plethora of security practices for finding vulnerabilities, such as threat modeling, penetration testing, code inspections, misuse and abuse cases [5], and automated static analysis [2]–[4]. While these practices have been shown to be effective, they can also be inefficient. Development teams are then faced with the challenge of prioritizing their fortification efforts within the entire development process. Developers might know what is possible, but lack a firm grip on what is probable. As a result, an uninformed development team can easily focus on the wrong areas for fortification.

Fortunately, an historical, longitudinal analysis of how vulnerabilities originated in professional products can inform fortification prioritization. Understanding the specific trends of how vulnerabilities can arise in a software development product can help developers understand where to look and what to look for in their own product. Some of these trends have been quantified in vulnerability prediction [6]–[10] studies using metrics aggregated at the file level, but little has been done to explore the original coding mistakes that contributed the vulnerabilities in the first place. In this study, we have identified and analyzed original coding mistakes as Vulnerability-Contributing Commits (VCCs), or commits in the version control repository that contributed to the introduction of a post-release vulnerability.

A myriad of factors can lead to the introduction and lack of detection of vulnerabilities. A developer may make a single massive change to the system, leaving his peers with an overwhelmingly large review. Furthermore, a developer may make small, incremental changes, but his work might be affecting the work of many other developers. Or, a developer may forget to disseminate her work in the change notes and so the code may miss out on be reviewed entirely.

The objective of this research is to improve software security by analyzing the size, interactive churn, and community dissemination of VCCs. We conducted an empirical case study of the Apache HTTP Server project (HTTPD). Using a multi-researcher, cross-validating, semi-automated, semi-manual process, we identified the VCCs for each known post-release vulnerability in HTTPD. To explore **commit size,** we analyzed three code churn metrics. **Interactive churn** is a suite of five recently-developed [6] socio-technical variants of code churn metrics that measure the degree to which developers' changes overwrite each others' code at the line level. To explore **community dissemination,** we analyzed the

978-0-7695-5056-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ESEM.2013.19

The core idea behind VCCs is not new to the MSR world, and stems the from research on traditional bugs.

Fix-inducing Change

Bug-introducing Change

Bug-injecting Change

Bug-inducing Change

Meneely et al. argued about the term "fix-inducing", which can be translated into "persuade to fix (the bug)". In their view, a VCC does not persuade developers to fix the vulnerability… the vulnerability is fixed after its discovery, not because of a flawed commit!

Terminology

2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement

When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits

Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa,
Alberto Rodríguez Tejeda, Matthew Mokary, Brian Spates
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
andy@se.rit.edu, {hxs8839, ajm661, acr921, mxm6060, bxs4361}@rit.edu

***Abstract*—Security is a harsh reality for software teams today. Developers must engineer secure software by preventing vulnerabilities, which are design and coding mistakes that have security consequences. Even in open source projects, vulnerable source code can remain unnoticed for years. In this paper, we traced 68 vulnerabilities in the Apache HTTP server back to the version control commits that contributed the vulnerable code originally. We manually found 124 Vulnerability-Contributing Commits (VCCs), spanning 17 years. In this exploratory study, we analyzed these VCCs quantitatively and qualitatively with the over-arching question: "What could developers have looked for to identify security concerns in this commit?" Specifically, we examined the size of the commit via code churn metrics, the amount developers overwrite each others' code via interactive churn metrics, exposure time between VCC and fix, and dissemination of the VCC to the development community via release notes and voting mechanisms. Our results show that VCCs are large: more than twice as much code churn on average than non-VCCs, even when normalized against lines of code. Furthermore, a commit was twice as likely to be a VCC when the author was a new developer to the source code. The insight from this study can help developers understand how vulnerabilities originate in a system so that security-related mistakes can be prevented or caught in the future.**

***Index Terms*— vulnerability, churn, socio-technical, empirical.**

I. INTRODUCTION

Security is a harsh reality for software teams today. Insecure software is not only expensive to maintain, but can cause immeasurable damage to a brand, or worse, to the livelihood of customers, patients, and citizens.

To software developers, the key to secure software lies in preventing vulnerabilities. Software vulnerabilities are special types of "faults that violate an [implicit or explicit] security policy" [1]. If developers want to find and fix vulnerabilities they must focus beyond making the system work as specified and prevent the system's functionality from being abused. According to security experts [2]–[4], finding vulnerabilities requires expertise in both the specific product and in software security in general.

The field of engineering secure software has a plethora of security practices for finding vulnerabilities, such as threat modeling, penetration testing, code inspections, misuse and abuse cases [5], and automated static analysis [2]–[4]. While these practices have been shown to be effective, they can also be inefficient. Development teams are then faced with the challenge of prioritizing their fortification efforts within the entire development process. Developers might know what is possible, but lack a firm grip on what is probable. As a result, an uninformed development team can easily focus on the wrong areas for fortification.

Fortunately, an historical, longitudinal analysis of how vulnerabilities originated in professional products can inform fortification prioritization. Understanding the specific trends of how vulnerabilities can arise in a software development product can help developers understand where to look and what to look for in their own product. Some of these trends have been quantified in vulnerability prediction [6]–[10] studies using metrics aggregated at the file level, but little has been done to explore the original coding mistakes that contributed the vulnerabilities in the first place. In this study, we have identified and analyzed original coding mistakes as Vulnerability-Contributing Commits (VCCs), or commits in the version control repository that contributed to the introduction of a post-release vulnerability.

A myriad of factors can lead to the introduction and lack of detection of vulnerabilities. A developer may make a single massive change to the system, leaving his peers with an overwhelmingly large review. Furthermore, a developer may make small, incremental changes, but his work might be affecting the work of many other developers. Or, a developer may forget to disseminate her work in the change notes and so the code may miss out on be reviewed entirely.

The objective of this research is to improve software security by analyzing the size, interactive churn, and community dissemination of VCCs. We conducted an empirical case study of the Apache HTTP Server project (HTTPD). Using a multi-researcher, cross-validating, semi-automated, semi-manual process, we identified the VCCs for each known post-release vulnerability in HTTPD. To explore **commit size**, we analyzed three code churn metrics. **Interactive churn** is a suite of five recently-developed [6] socio-technical variants of code churn metrics that measure the degree to which developers' changes overwrite each others' code at the line level. To explore **community dissemination,** we analyzed the

978-0-7695-5056-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ESEM.2013.19

The core idea behind VCCs is not new to the MSR world, and stems the from research on traditional bugs.

Fix-inducing Change

Bug-introducing Change

Bug-injecting Change

Bug-inducing Change

Meneely et al. argued about the term "fix-inducing", which can be translated into "persuade to fix (the bug)". In their view, a VCC does not persuade developers to fix the vulnerability… the vulnerability is fixed after its discovery, not because of a flawed commit!

Long story short: as long as we all agree, it makes no (real) difference.

Main Uses of VCCs

Train Vulnerability Prediction Models

We can build a **just-in-time vulnerability prediction model** if the dataset is made of VCCs and non-VCCs.

Main Uses of VCCs

Train Vulnerability Prediction Models

We can build a **just-in-time vulnerability prediction model** if the dataset is made of VCCs and non-VCCs.

Recover Vulnerable Versions/Releases

VCCs can help understand which project releases are affected by the vulnerability.

Main Uses of VCCs

Train Vulnerability Prediction Models

We can build a **just-in-time vulnerability prediction model** if the dataset is made of VCCs and non-VCCs.

Recover Vulnerable Versions/Releases

VCCs can help understand which project releases are affected by the vulnerability.

Expand the Knowledge on Vulnerabilities

Understand how vulnerabilities are progressively introduced in the code, drawing out interesting facts.

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement

When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits

Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa,
Alberto Rodríguez Tejeda, Matthew Mokary, Brian Spates
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
andy@se.rit.edu, {hxs8839, ajm661, acr921, mxm6060, bxs4361}@rit.edu

***Abstract*—Security is a harsh reality for software teams today. Developers must engineer secure software by preventing vulnerabilities, which are design and coding mistakes that have security consequences. Even in open source projects, vulnerable source code can remain unnoticed for years. In this paper, we traced 68 vulnerabilities in the Apache HTTP server back to the version control commits that contributed the vulnerable code originally. We manually found 124 Vulnerability-Contributing Commits (VCCs), spanning 17 years. In this exploratory study, we analyzed these VCCs quantitatively and qualitatively with the over-arching question: "What could developers have looked for to identify security concerns in this commit?" Specifically, we examined the size of the commit via code churn metrics, the amount developers overwrite each others' code via interactive churn metrics, exposure time between VCC and fix, and dissemination of the VCC to the development community via release notes and voting mechanisms. Our results show that VCCs are large: more than twice as much code churn on average than non-VCCs, even when normalized against lines of code. Furthermore, a commit was twice as likely to be a VCC when the author was a new developer to the source code. The insight from this study can help developers understand how vulnerabilities originate in a system so that security-related mistakes can be prevented or caught in the future.**

***Index Terms*— vulnerability, churn, socio-technical, empirical.**

I. INTRODUCTION

Security is a harsh reality for software teams today. Insecure software is not only expensive to maintain, but can cause immeasurable damage to a brand, or worse, to the livelihood of customers, patients, and citizens.

To software developers, the key to secure software lies in preventing vulnerabilities. Software vulnerabilities are special types of "faults that violate an [implicit or explicit] security policy" [1]. If developers want to find and fix vulnerabilities they must focus beyond making the system work as specified and prevent the system's functionality from being abused. According to security experts [2]–[4], finding vulnerabilities requires expertise in both the specific product and in software security in general.

The field of engineering secure software has a plethora of security practices for finding vulnerabilities, such as threat modeling, penetration testing, code inspections, misuse and abuse cases [5], and automated static analysis [2]–[4]. While these practices have been shown to be effective, they can also be inefficient. Development teams are then faced with the challenge of prioritizing their fortification efforts within the entire development process. Developers might know what is possible, but lack a firm grip on what is probable. As a result, an uninformed development team can easily focus on the wrong areas for fortification.

Fortunately, an historical, longitudinal analysis of how vulnerabilities originated in professional products can inform fortification prioritization. Understanding the specific trends of how vulnerabilities can arise in a software development product can help developers understand where to look and what to look for in their own product. Some of these trends have been quantified in vulnerability prediction [6]–[10] studies using metrics aggregated at the file level, but little has been done to explore the original coding mistakes that contributed the vulnerabilities in the first place. In this study, we have identified and analyzed original coding mistakes as Vulnerability-Contributing Commits (VCCs), or commits in the version control repository that contributed to the introduction of a post-release vulnerability.

A myriad of factors can lead to the introduction and lack of detection of vulnerabilities. A developer may make a single massive change to the system, leaving his peers with an overwhelmingly large review. Furthermore, a developer may make small, incremental changes, but his work might be affecting the work of many other developers. Or, a developer may forget to disseminate her work in the change notes and so the code may miss out on be reviewed entirely.

The objective of this research is to improve software security by analyzing the size, interactive churn, and community dissemination of VCCs. We conducted an empirical case study of the Apache HTTP Server project (HTTPD). Using a multi-researcher, cross-validating, semi-automated, semi-manual process, we identified the VCCs for each known post-release vulnerability in HTTPD. To explore **commit size,** we analyzed three code churn metrics. **Interactive churn** is a suite of five recently-developed [6] socio-technical variants of code churn metrics that measure the degree to which developers' changes overwrite each others' code at the line level. To explore **community dissemination,** we analyzed the

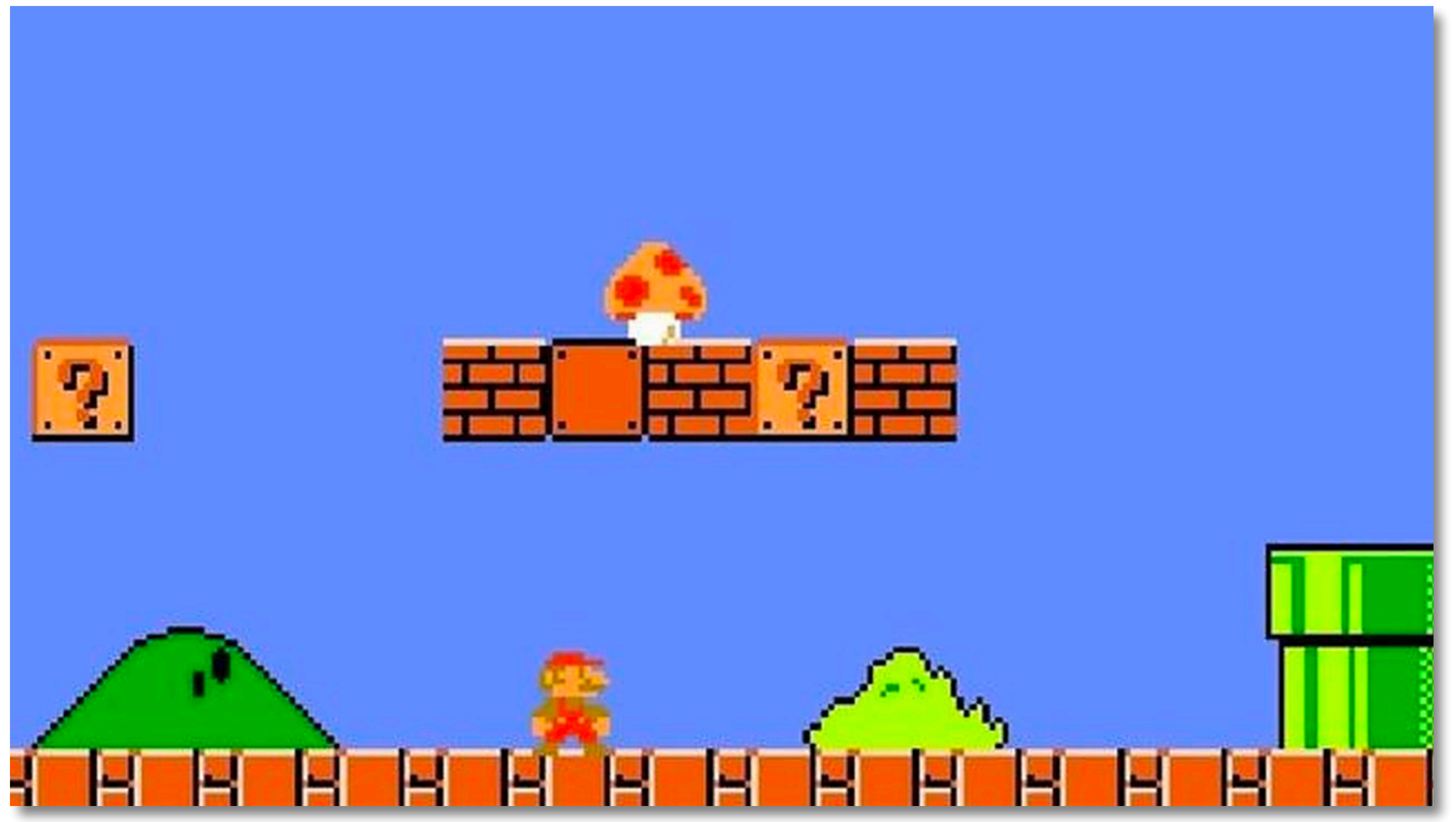
978-0-7695-5056-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ESEM.2013.19

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.



Size matters

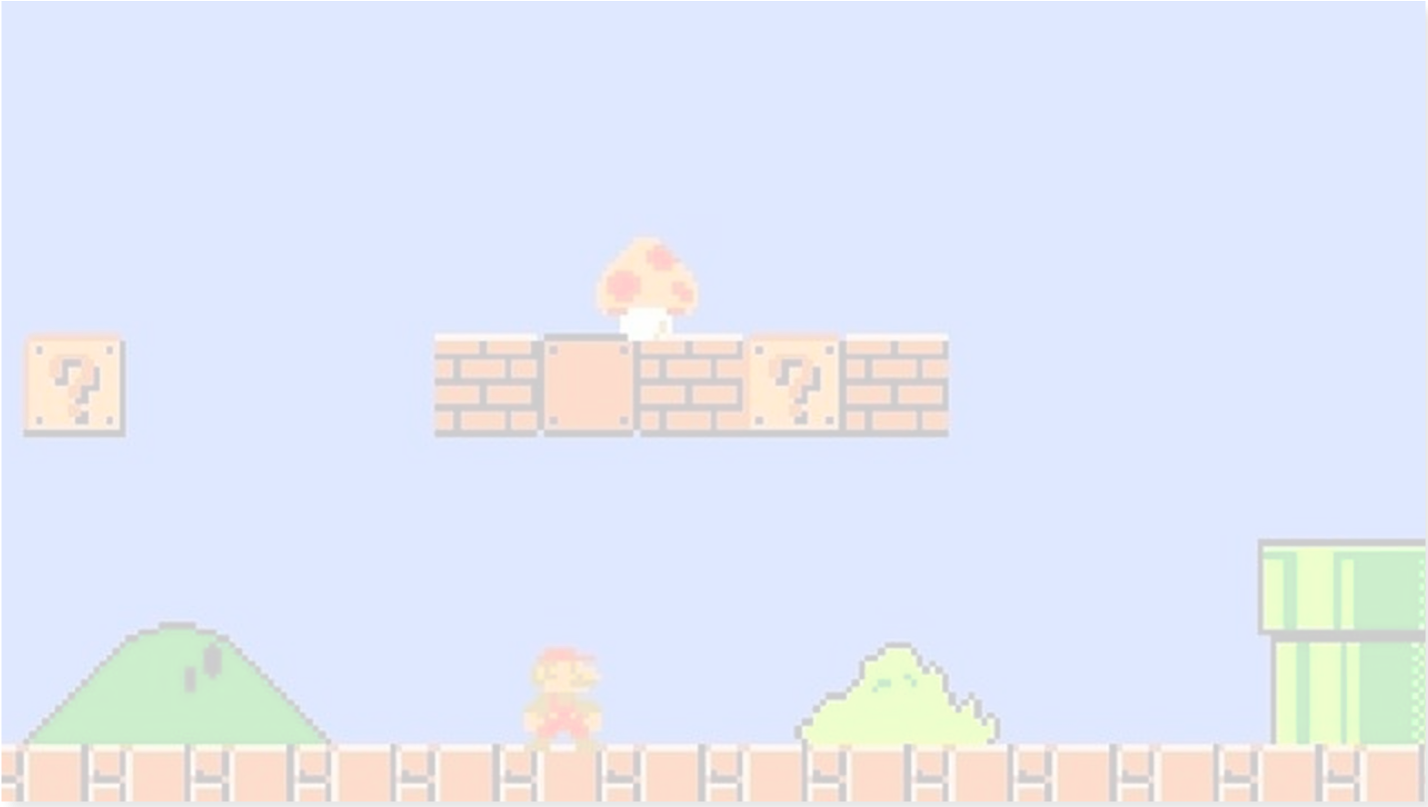
VCCs change **x10 more lines of code** than non-VCCs.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.



Size matters

Don't step on someone's toes

VCCs are made by **new authors in 15% more cases** than non-VCCs.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

Size matters

Don’t step on someone’s toes

A leopard CAN change its spots

VCCs affect **existing files in 87% of the cases** rather than new files.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

Large commits might increase the chance of contributing to a vulnerability.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

Large commits might increase the chance of contributing to a vulnerability.

Changing **other developers' code** might increase the chance of contributing to a vulnerability.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

Large commits might increase the chance of contributing to a vulnerability.

Changing **other developers' code** might increase the chance of contributing to a vulnerability.

Vulnerabilities are more likely to be added when **modifying existing files** rather than creating new files.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Yeah, cool.
How can we
mine them?

Mining VCCs: A First Approach

Now let’s see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

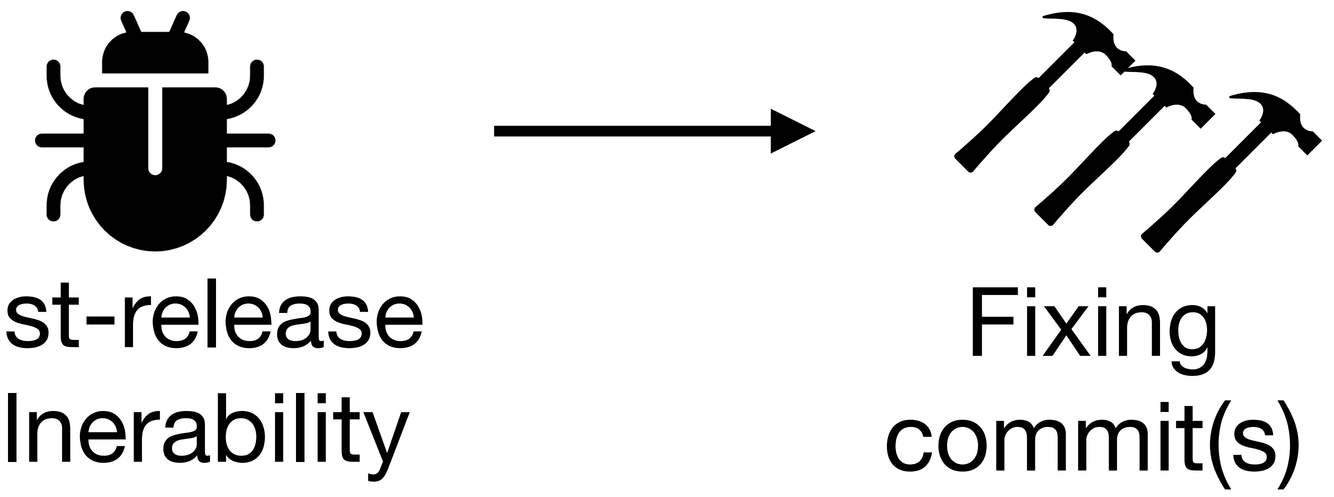
Post-release vulnerability

Can be a known vulnerability from NVD or another source, it is the same.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



We assume the vulnerability is already mapped to its fixing commits.

Mining VCCs: A First Approach

Now let’s see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

Post-release vulnerability

Fixing commit(s)

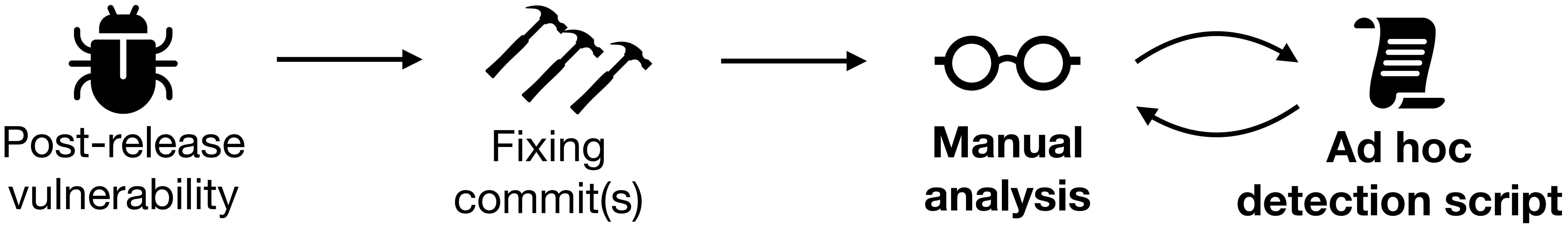
Manual analysis

One (or more) inspectors examine(s) the patch and its context to find the **vulnerable code elements** (statements). All the fixing commits are analyzed as one single big commit.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



Post-release vulnerability

Fixing commit(s)

Manual analysis

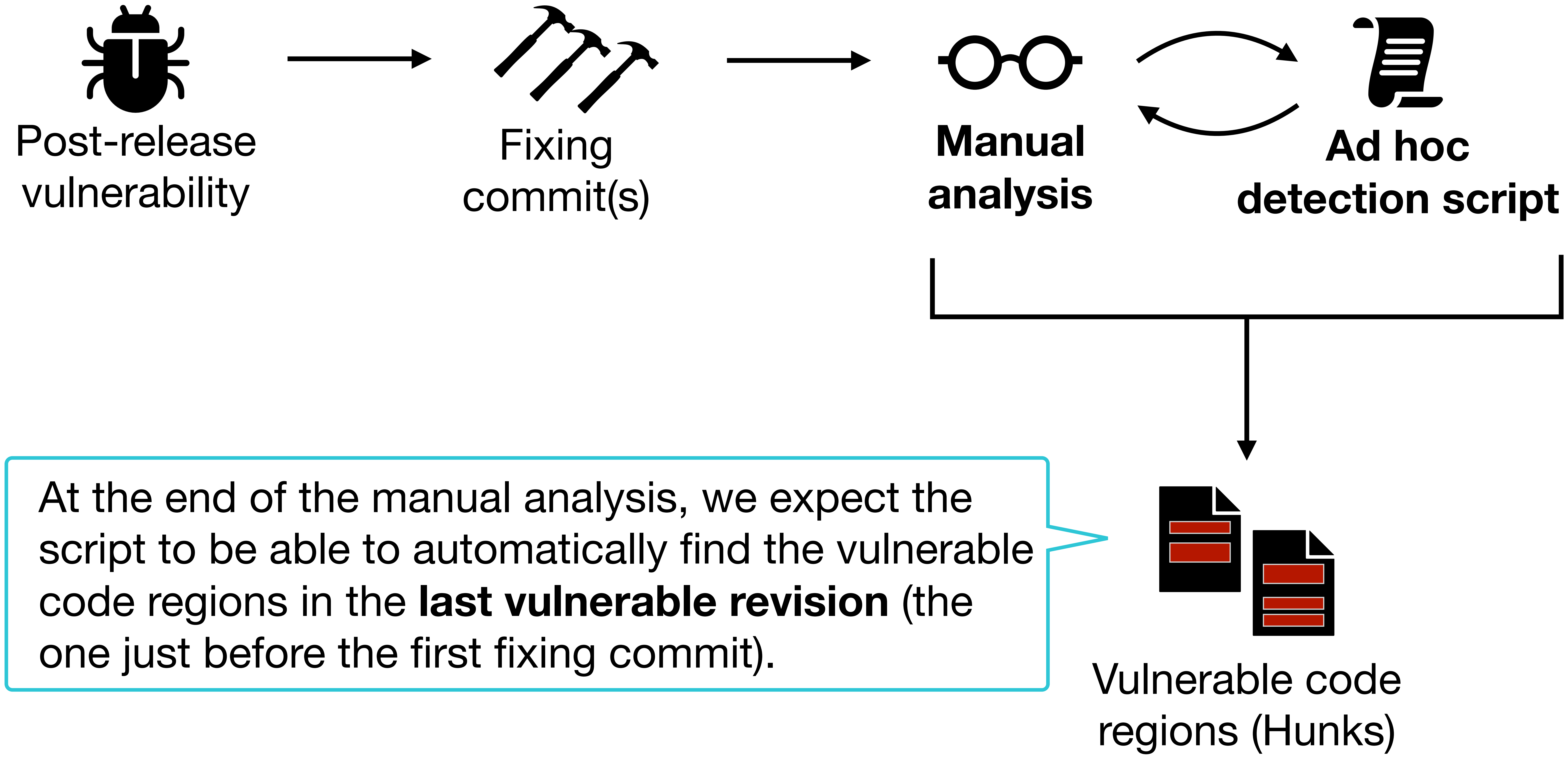
Ad hoc detection script

Detecting the vulnerable code elements is supported by a **regex-based string search** crafted by the inspector(s). This script is continuously updated until the vulnerability is fully understood.

Mining VCCs: A First Approach

Now let’s see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

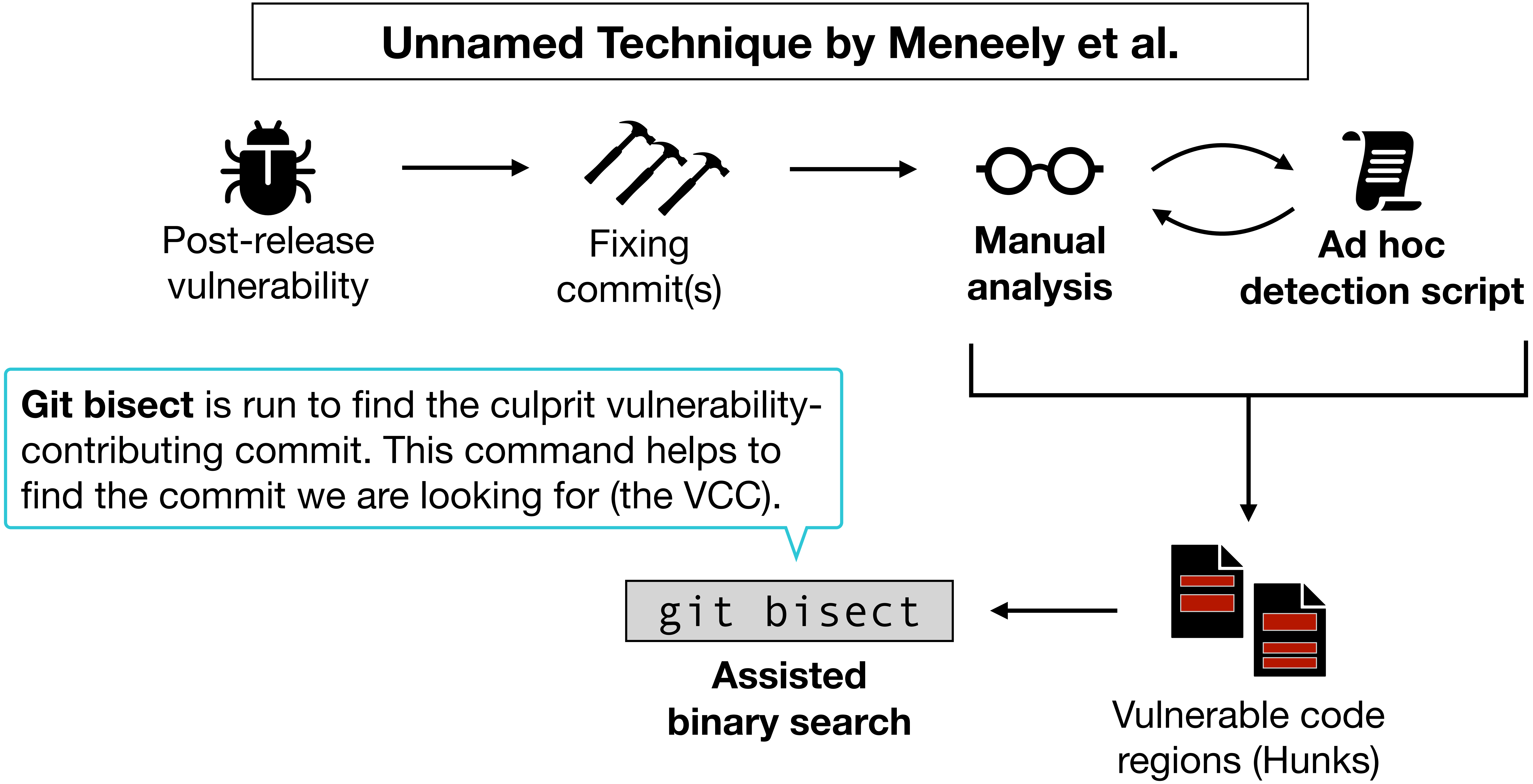


Post-release vulnerability → Fixing commit(s) → **Manual analysis** ⇄ **Ad hoc detection script** → Vulnerable code regions (Hunks)

At the end of the manual analysis, we expect the script to be able to automatically find the vulnerable code regions in the **last vulnerable revision** (the one just before the first fixing commit).

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.



Unnamed Technique by Meneely et al.

Post-release vulnerability

Fixing commit(s)

Manual analysis

Ad hoc detection script

Vulnerable code regions (Hunks)

`git bisect`

Assisted binary search

Git bisect is run to find the culprit vulnerability-contributing commit. This command helps to find the commit we are looking for (the VCC).

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

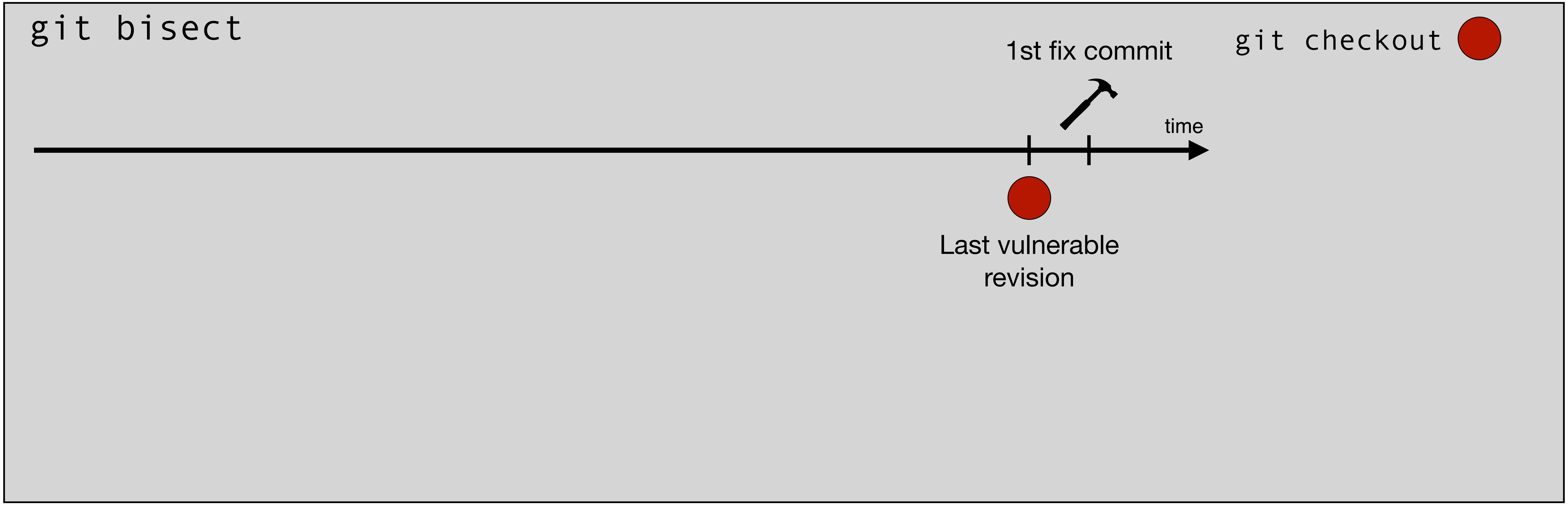
Unnamed Technique by Meneely et al.

`git bisect`

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

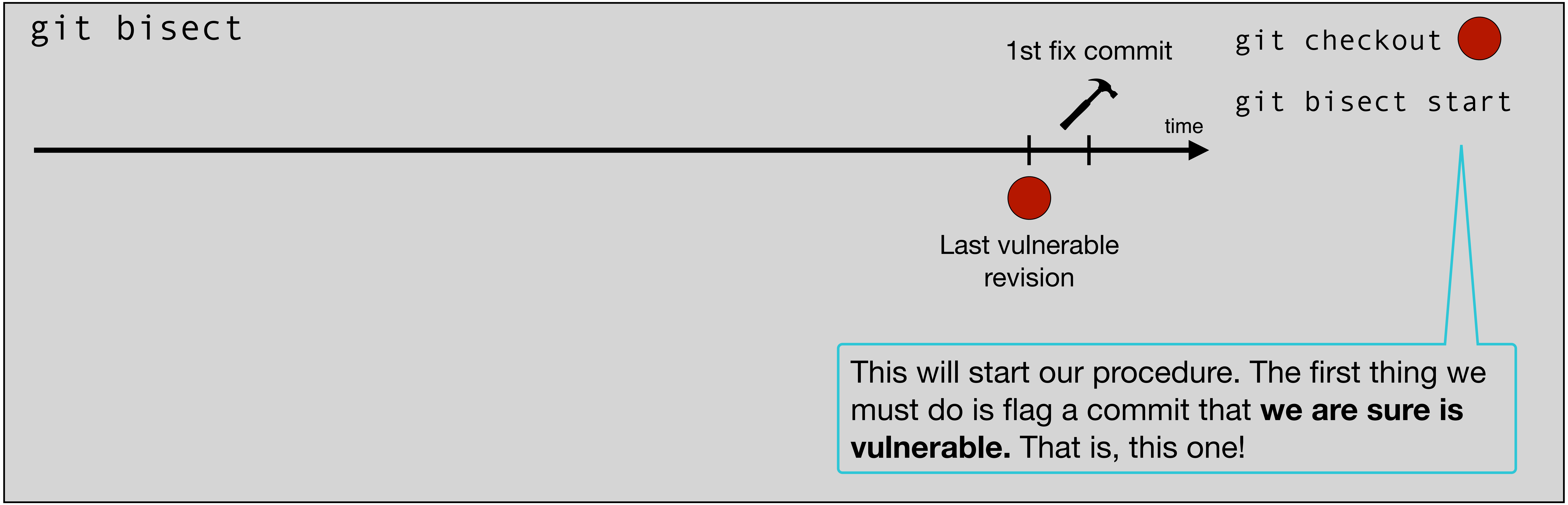
Unnamed Technique by Meneely et al.



Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



`git bisect`

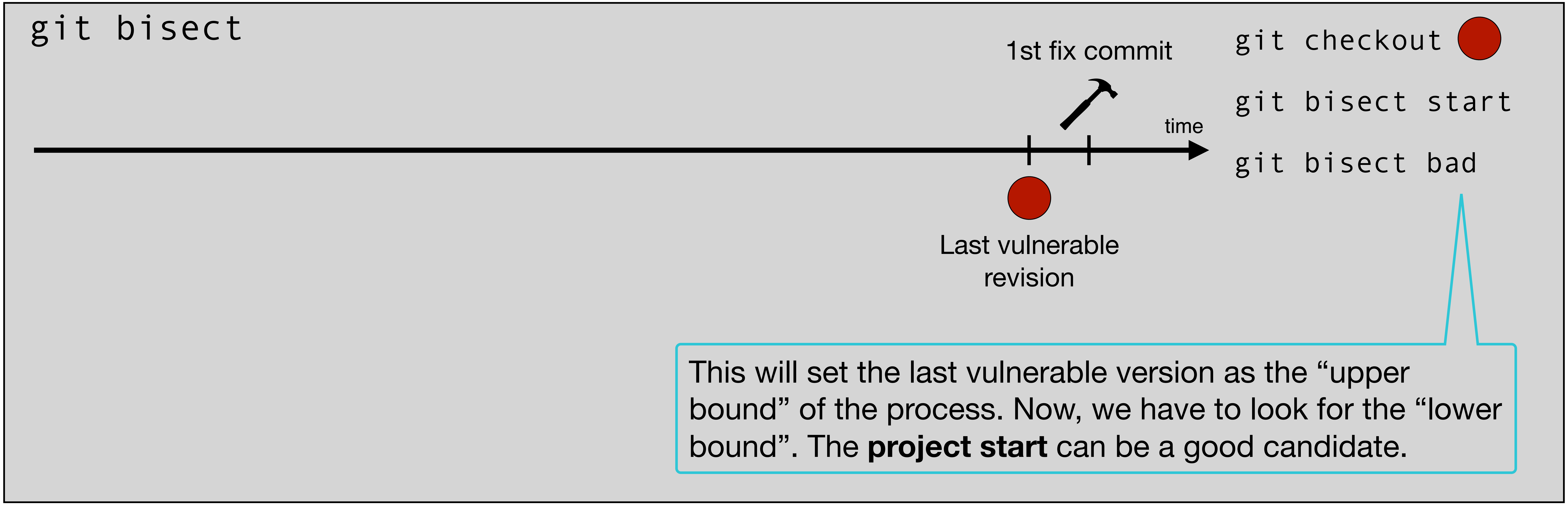
```
git checkout ●
git bisect start
```

This will start our procedure. The first thing we must do is flag a commit that **we are sure is vulnerable.** That is, this one!

Mining VCCs: A First Approach

Now let’s see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



```
git checkout ●
git bisect start
git bisect bad
```

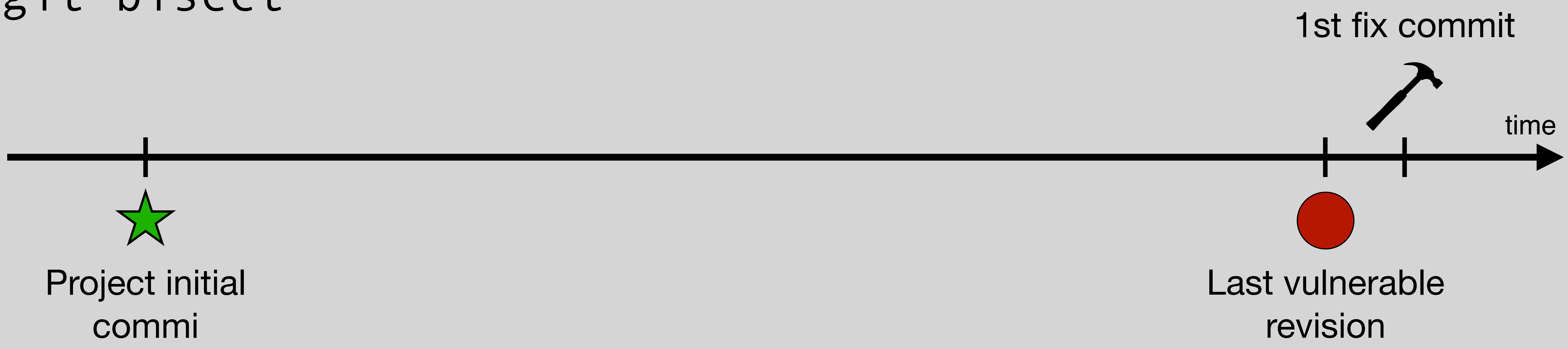
This will set the last vulnerable version as the “upper bound” of the process. Now, we have to look for the “lower bound”. The **project start** can be a good candidate.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

`git bisect`



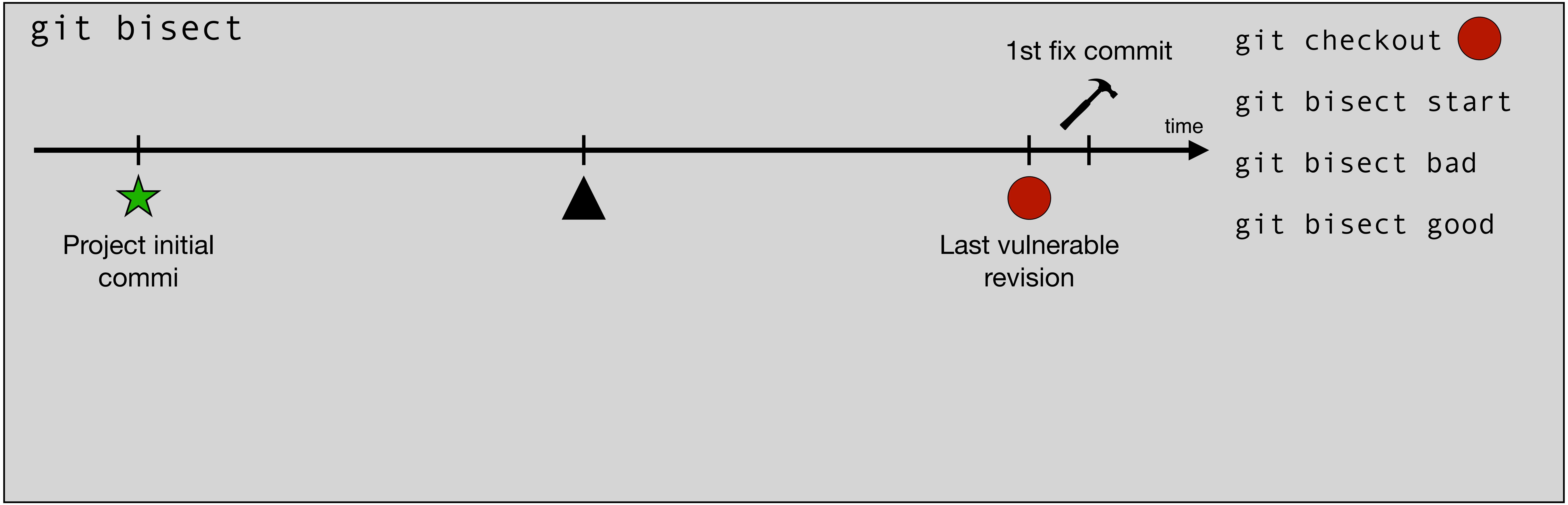
```
git checkout ●
git bisect start
git bisect bad
git bisect good
```

The main process starts now. Git will **select a commit in the middle**, on which we are automatically checked out so that we can inspect it.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



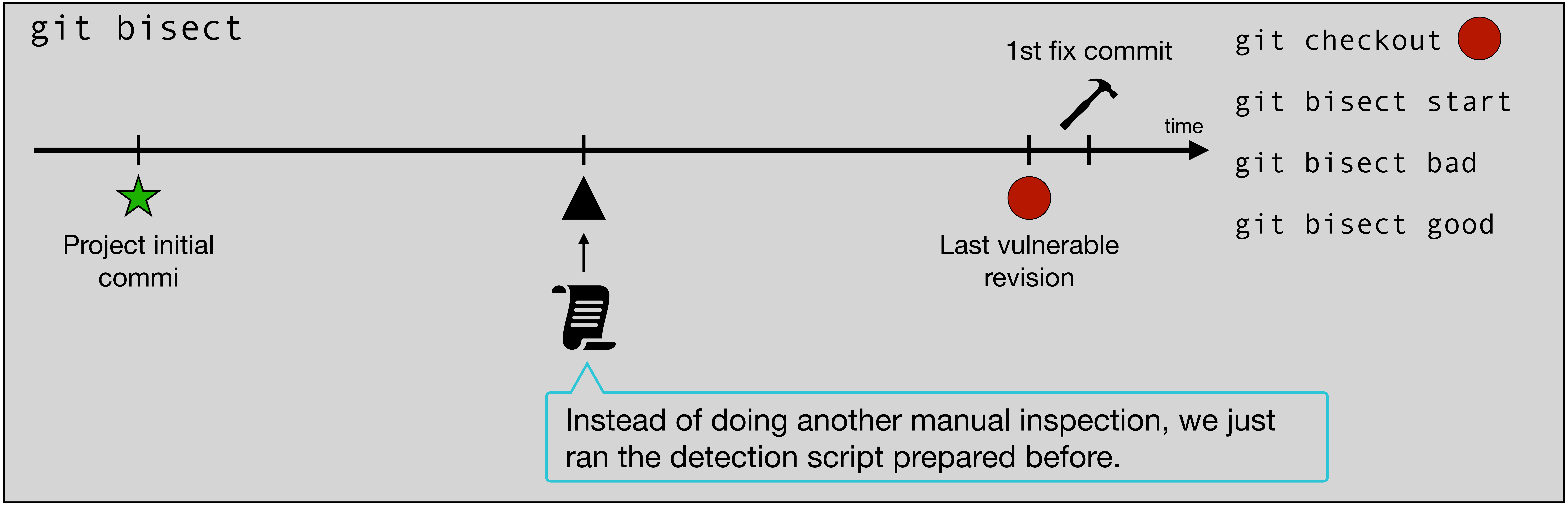
`git bisect`

```
git checkout ●
git bisect start
git bisect bad
git bisect good
```

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



```
git checkout ●
git bisect start
git bisect bad
git bisect good
```

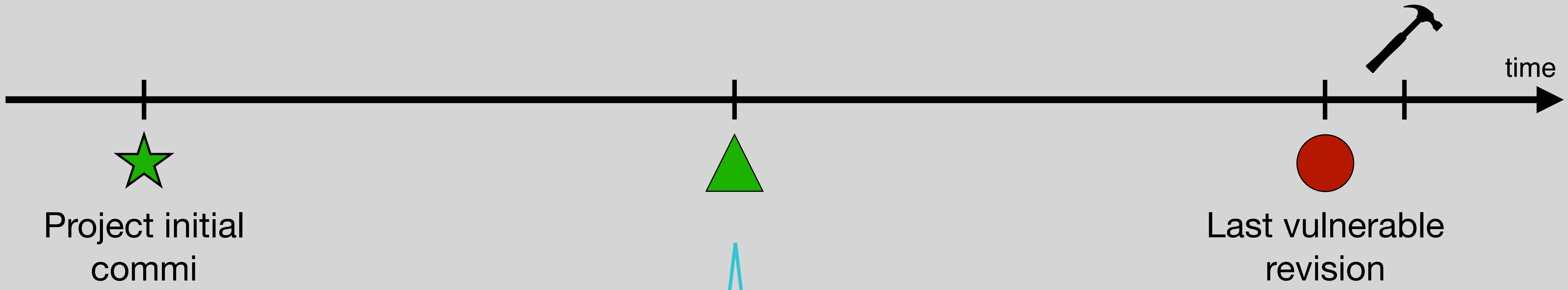
Instead of doing another manual inspection, we just ran the detection script prepared before.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

`git bisect`



```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
```

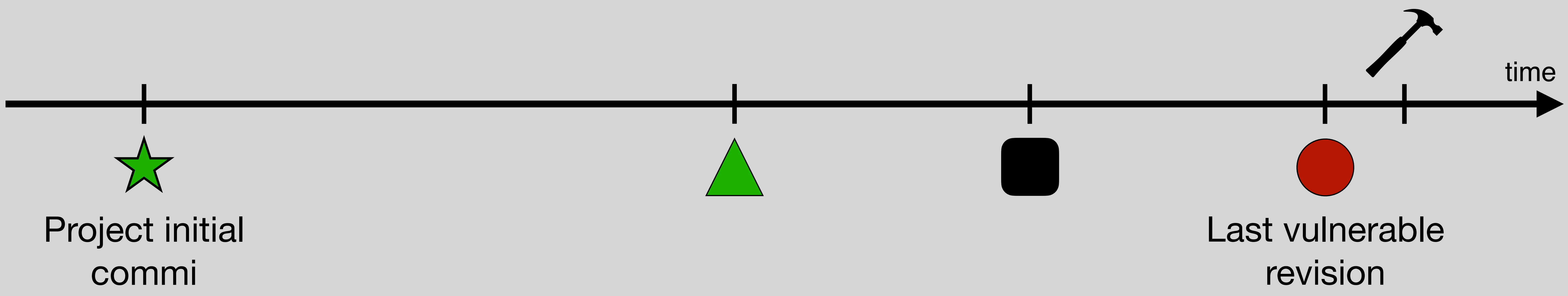
Let's assume the script said this commit does not have the vulnerability: this is a "good" commit.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

`git bisect`



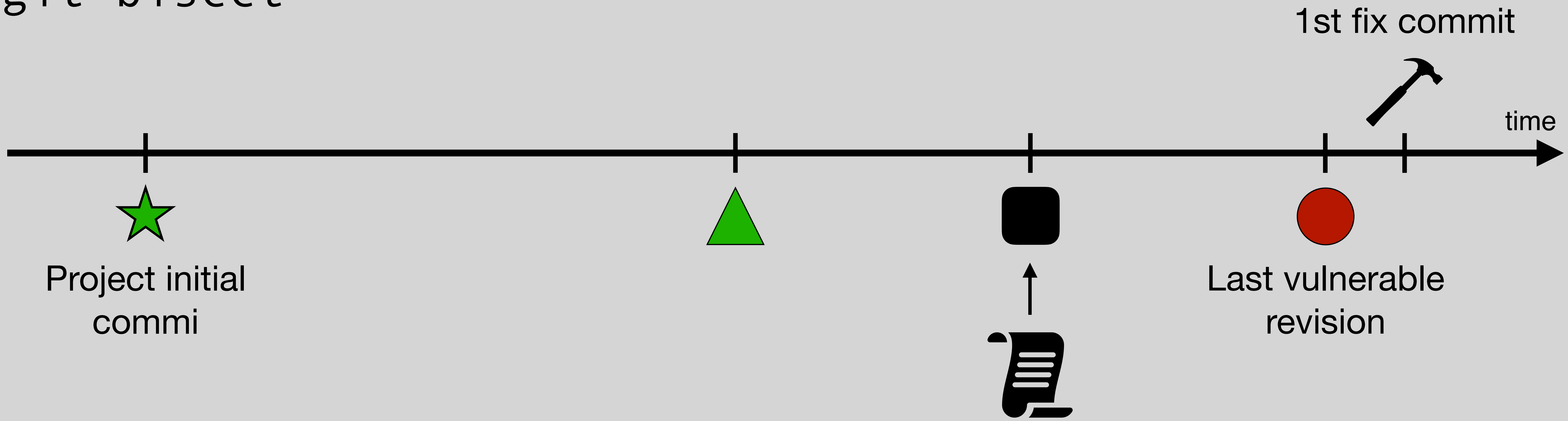
```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
```

Mining VCCs: A First Approach

Now let’s see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

`git bisect`



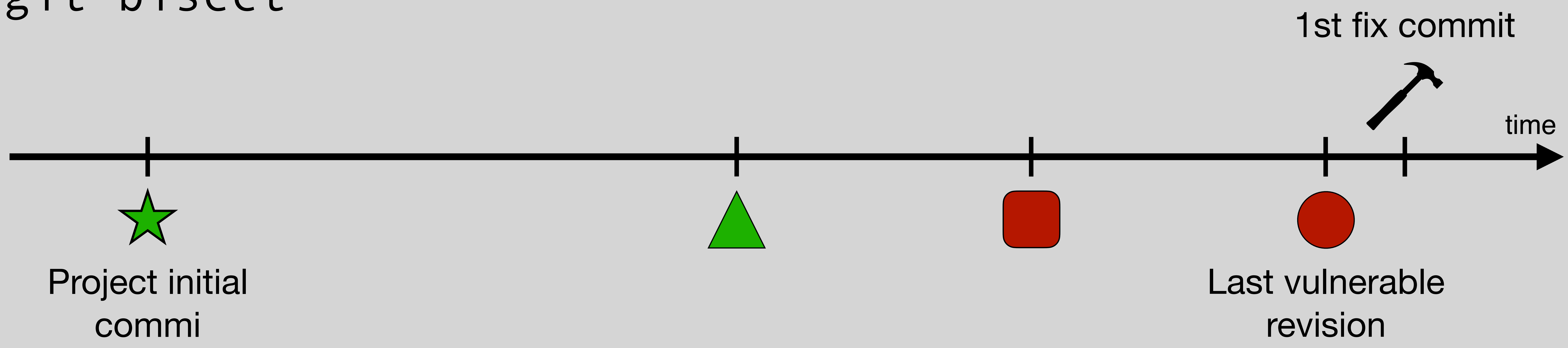
```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
```

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

`git bisect`



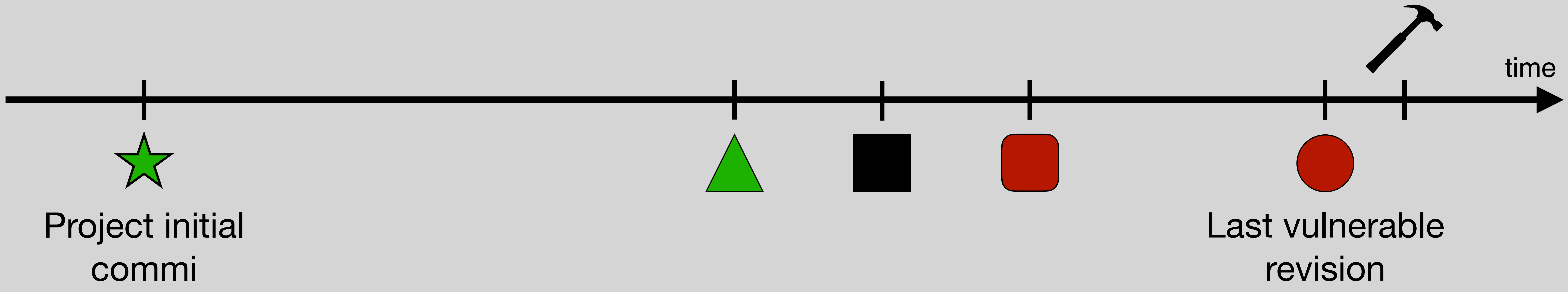
```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
git bisect bad
```

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

`git bisect`

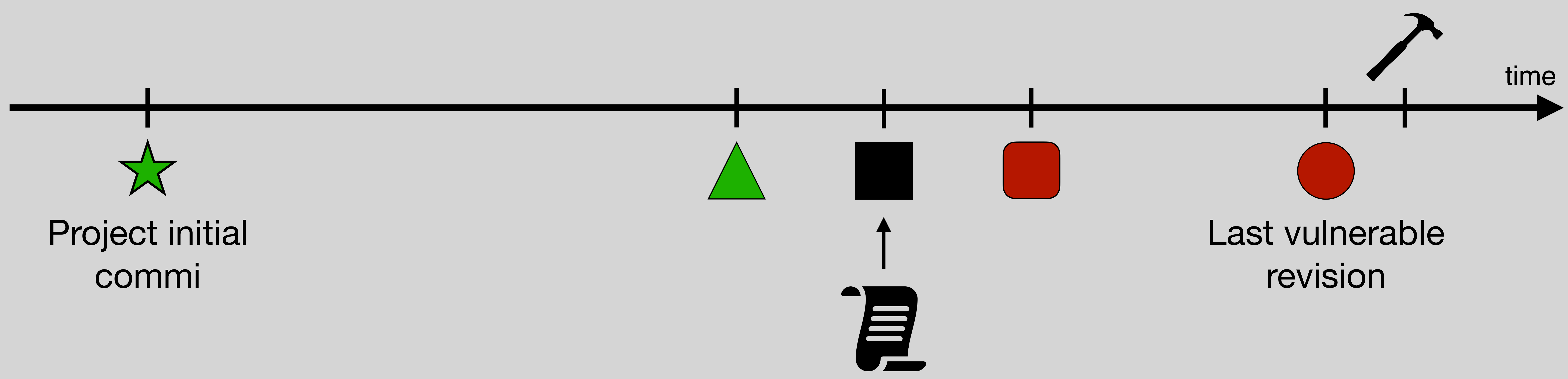


```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
git bisect bad
```

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



`git bisect`

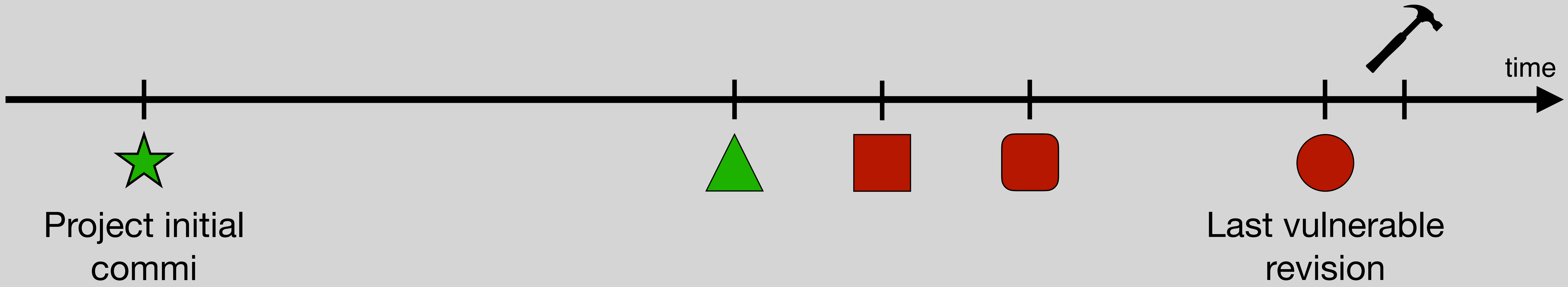
```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
git bisect bad
```

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

`git bisect`

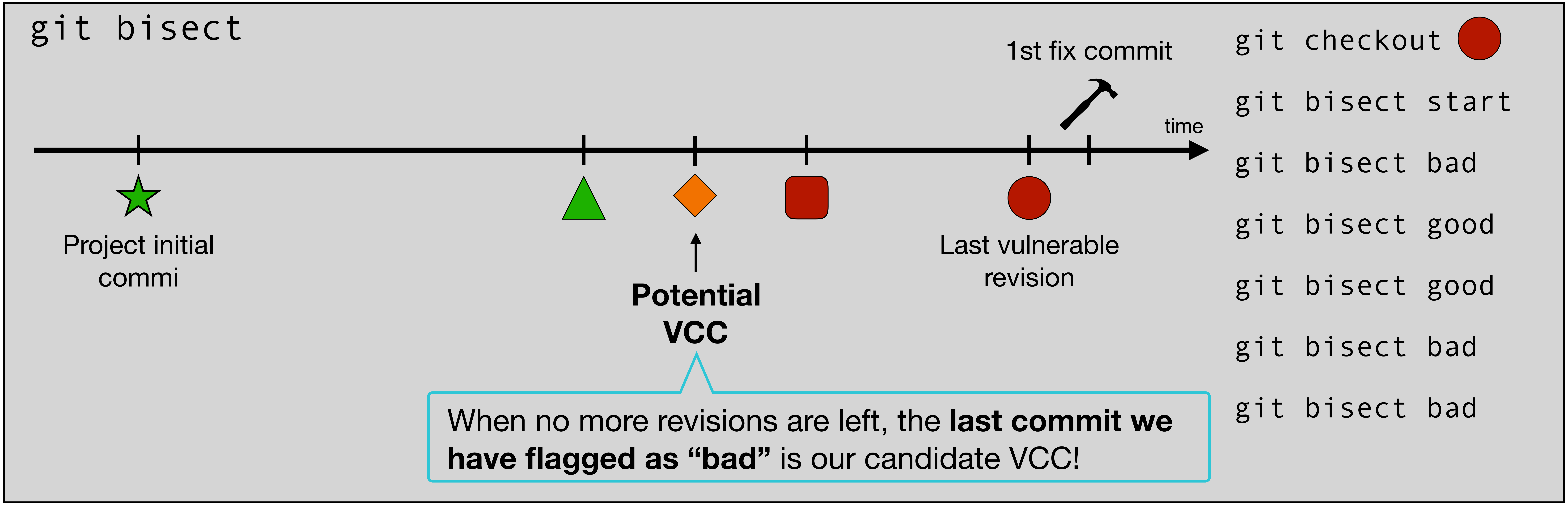


```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
git bisect bad
git bisect bad
```

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



`git bisect`

1st fix commit

time

Project initial commi

Potential VCC

Last vulnerable revision

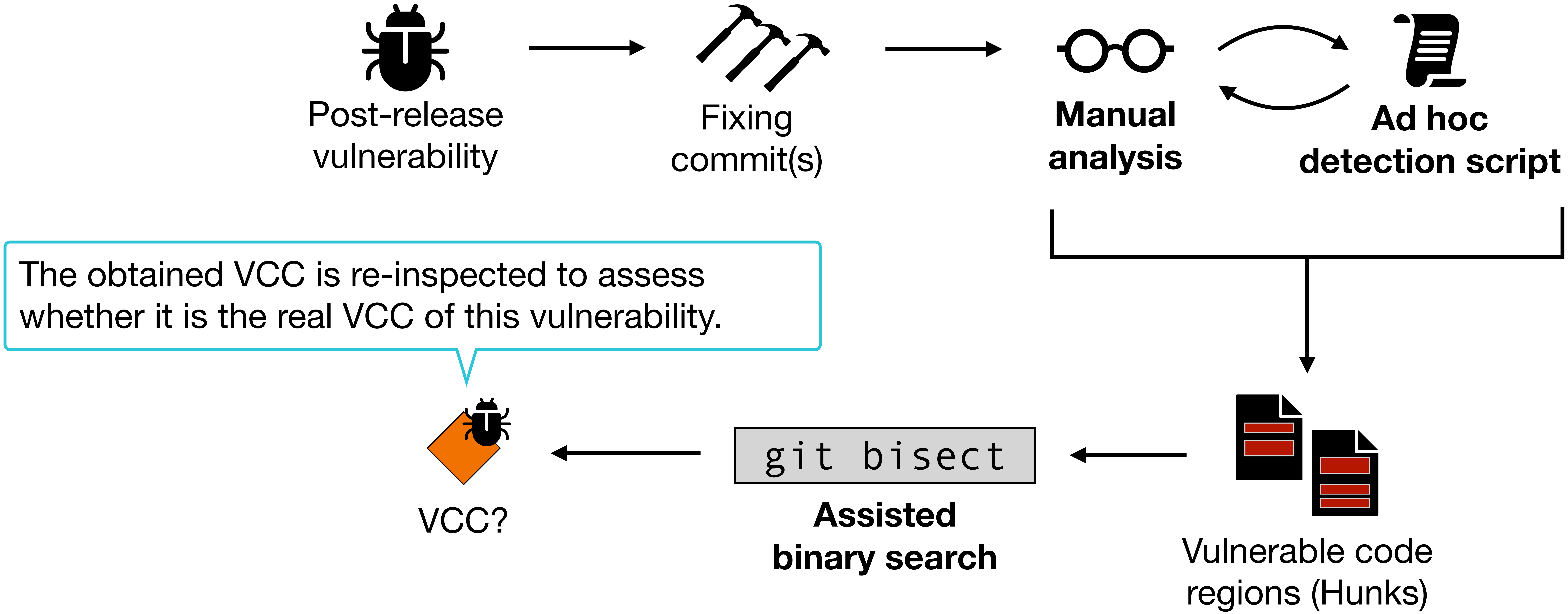
When no more revisions are left, the **last commit we have flagged as "bad"** is our candidate VCC!

```
git checkout ●
git bisect start
git bisect bad
git bisect good
git bisect good
git bisect bad
git bisect bad
```

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



Post-release vulnerability

Fixing commit(s)

Manual analysis

Ad hoc detection script

Vulnerable code regions (Hunks)

`git bisect`

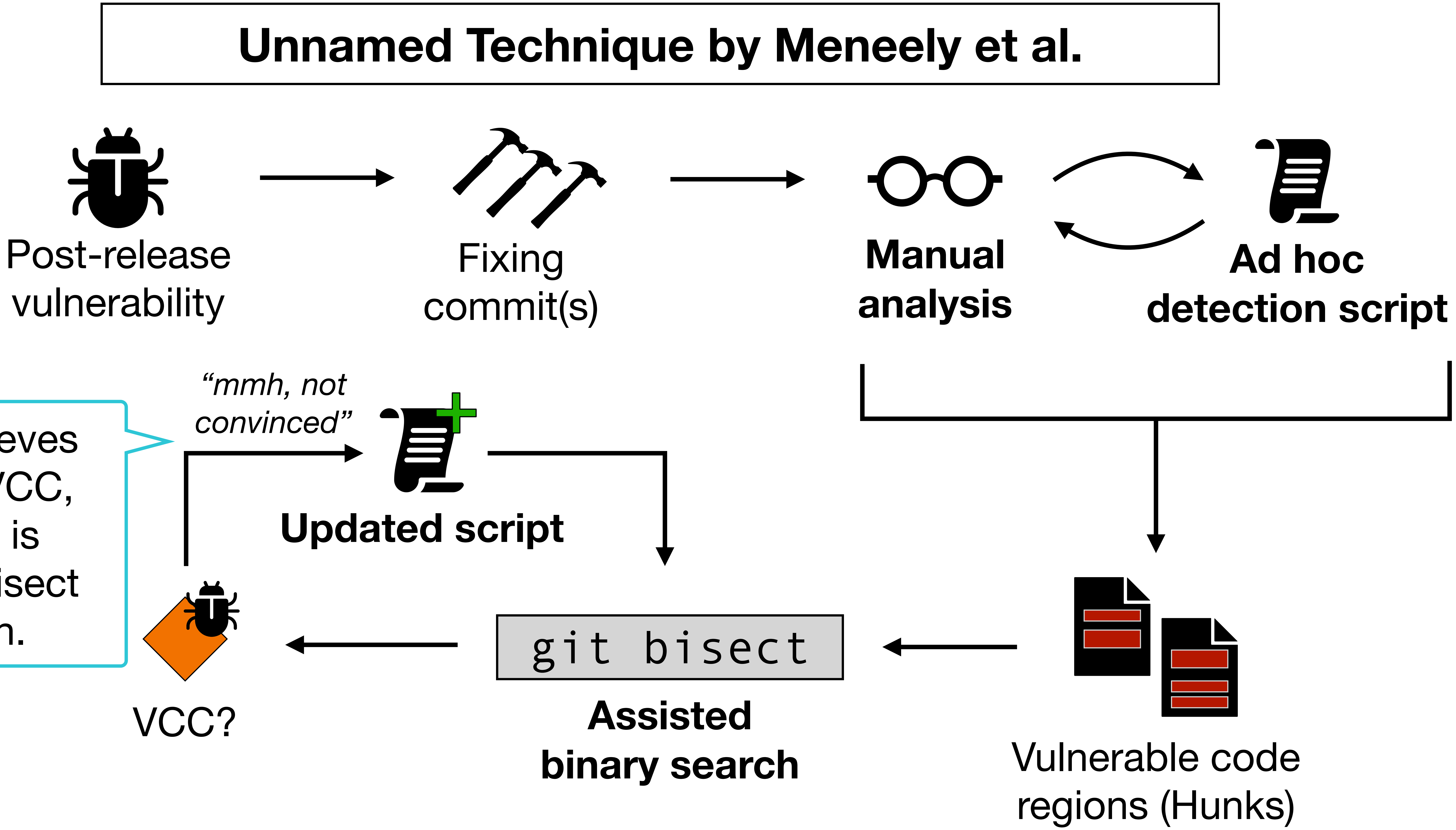
Assisted binary search

VCC?

The obtained VCC is re-inspected to assess whether it is the real VCC of this vulnerability.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.



Unnamed Technique by Meneely et al.

Post-release vulnerability

Fixing commit(s)

Manual analysis

Ad hoc detection script

Vulnerable code regions (Hunks)

`git bisect`

Assisted binary search

VCC?

"mmh, not convinced"

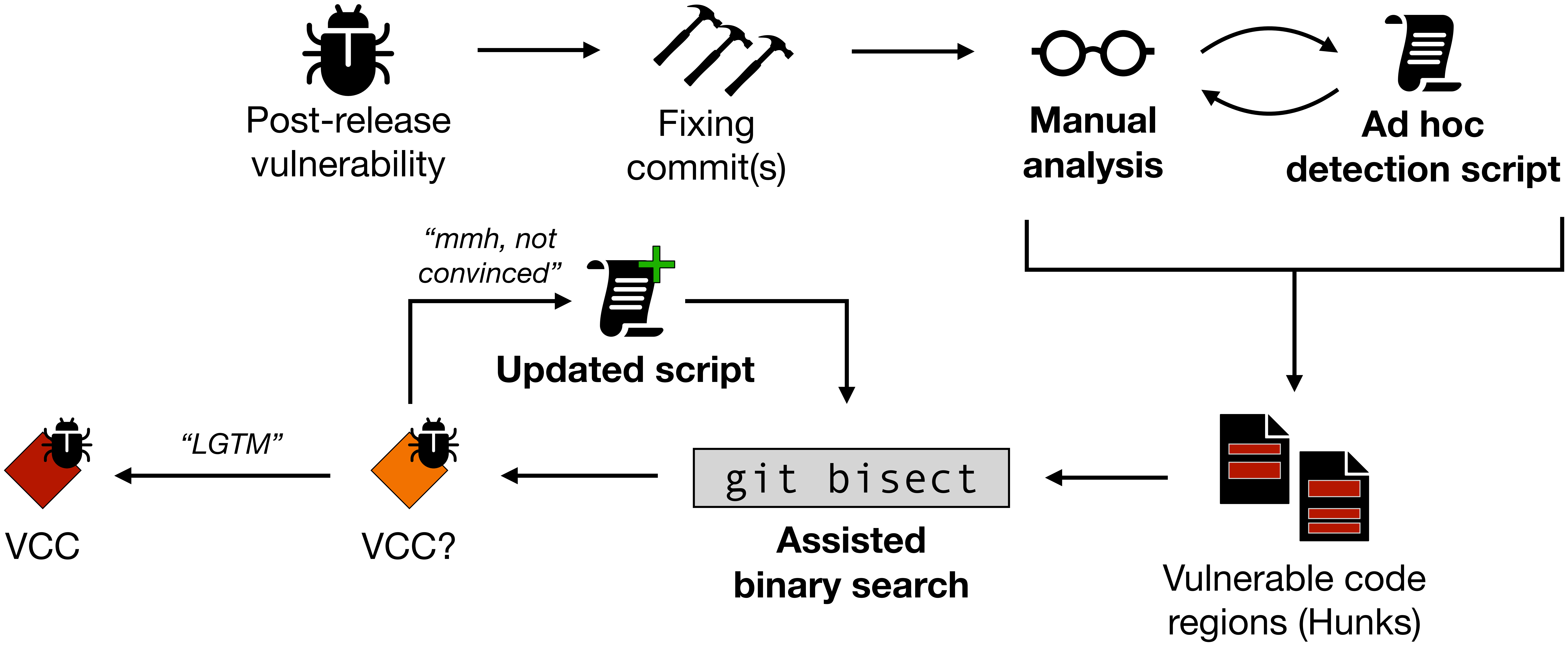
Updated script

If the inspector believes this is not the real VCC, the detection script is updated, and the bisect process starts again.

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.



Mining VCCs: Borrowing from the Bug World

Meneely et al.’s technique doesn’t scale: it’s manual and time-consuming. We need a fully-automated solution.
Let’s go back a couple of years: 2005!

Mining VCCs: Borrowing from the Bug World

Meneely et al.’s technique doesn’t scale: it’s manual and time-consuming. We need a fully-automated solution. Let’s go back a couple of years: 2005!

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution. Let's go back a couple of years: 2005!

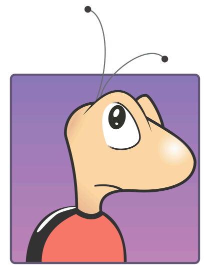
Śliwerski, Zimmermann, Zeller (SZZ)

J. Śliwerski, T. Zimmermann, and A. Zeller. 2005. When do changes induce fixes? In Proceedings of the 2005 international workshop on Mining software repositories (MSR '05). Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1083142.1083147

Mining VCCs: Borrowing from the Bug World

Meneely et al.’s technique doesn’t scale: it’s manual and time-consuming. We need a fully-automated solution.
Let’s go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)



Project Bug Tracker

The original approach relies on *Bugzilla*, but we can mine any bug tracker or similar database.

Mining VCCs: Borrowing from the Bug World

Meneely et al.’s technique doesn’t scale: it’s manual and time-consuming. We need a fully-automated solution. Let’s go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)

Project Bug Tracker

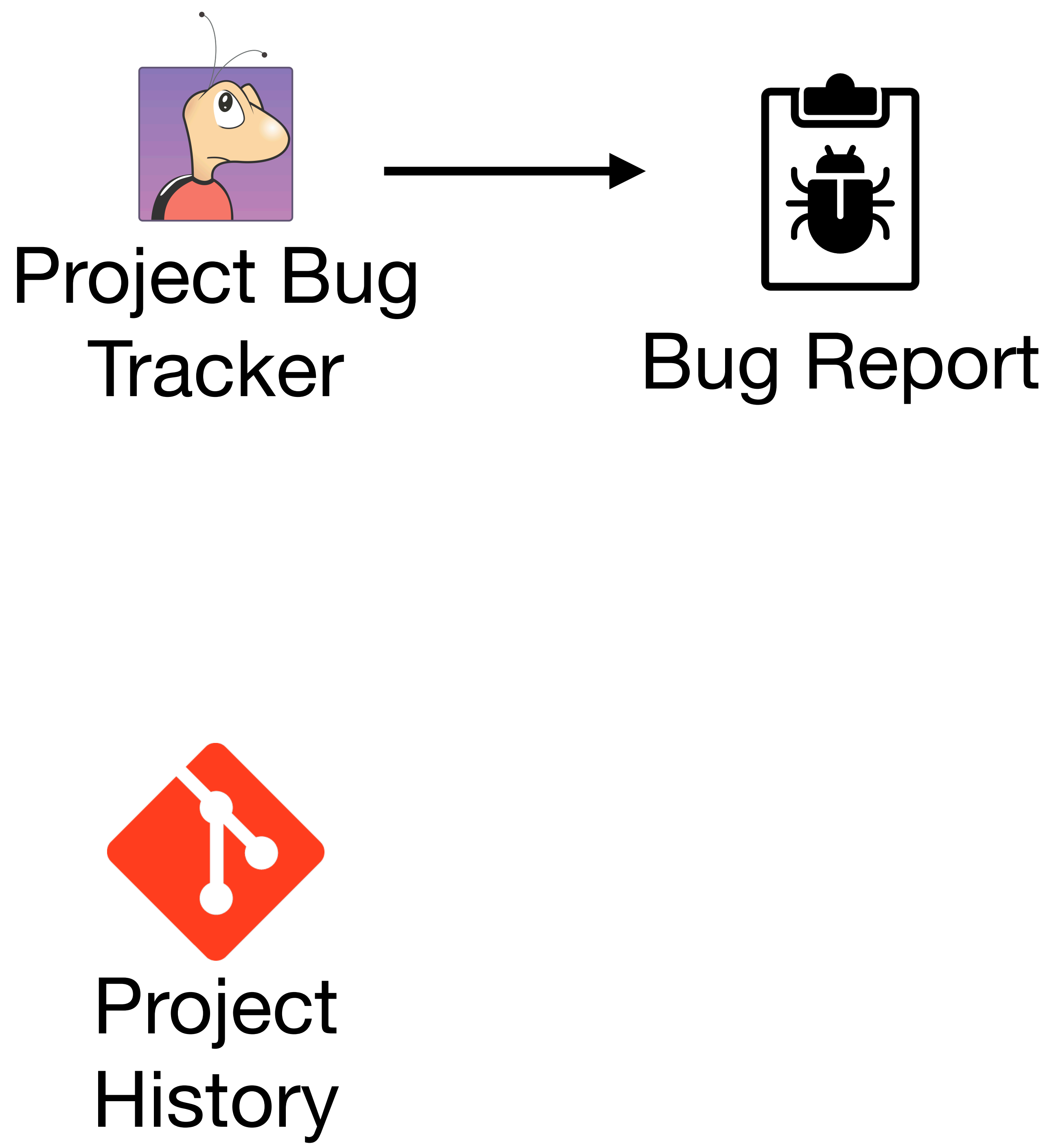
Project History

The original approach relies on *CVS (Concurrent Versioning System)*, but here we consider *git*.

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution. Let's go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)

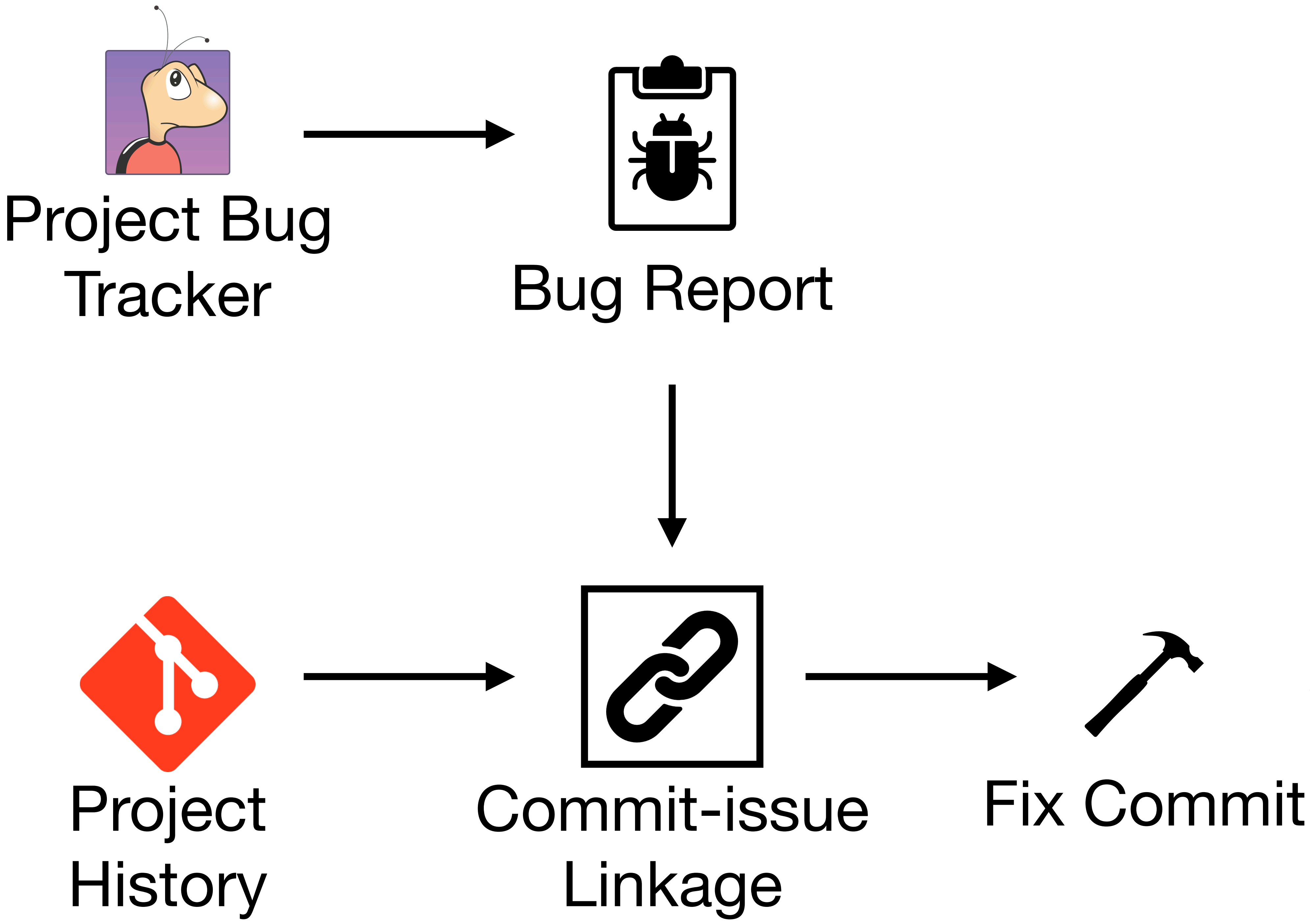


We pick a bug report for which we want to know its *bug-inducing commits* (BICs).

Mining VCCs: Borrowing from the Bug World

Meneely et al.’s technique doesn’t scale: it’s manual and time-consuming. We need a fully-automated solution. Let’s go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)

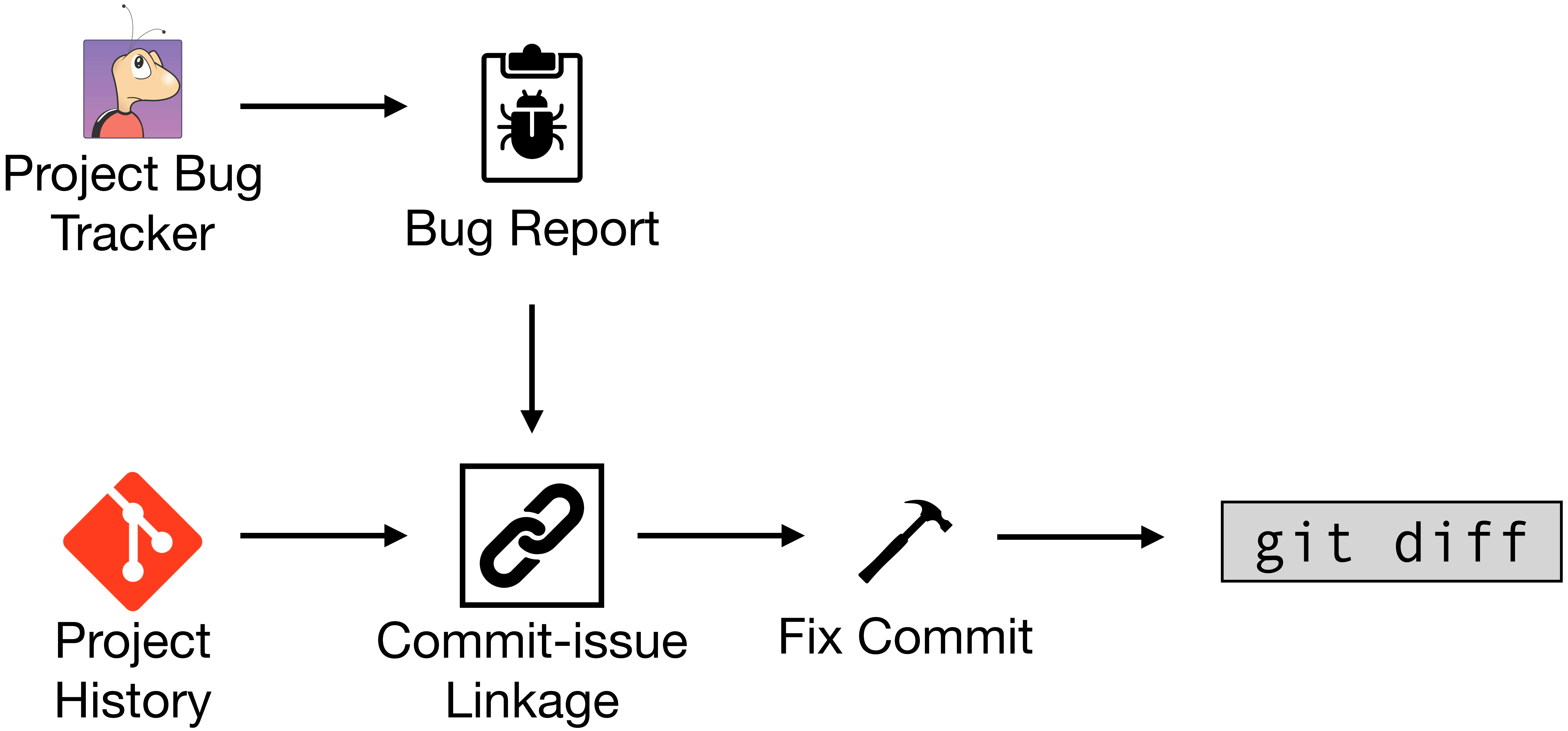


We can run any commit-issue link algorithm we want. The original approach uses a **pattern-based search**, looking for the bug ID (a number) inside the commit messages. In any case, we just want the *bug-fixing commit*.

Mining VCCs: Borrowing from the Bug World

Meneely et al.’s technique doesn’t scale: it’s manual and time-consuming. We need a fully-automated solution. Let’s go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)



Project Bug Tracker

Bug Report

Project History

Commit-issue Linkage

Fix Commit

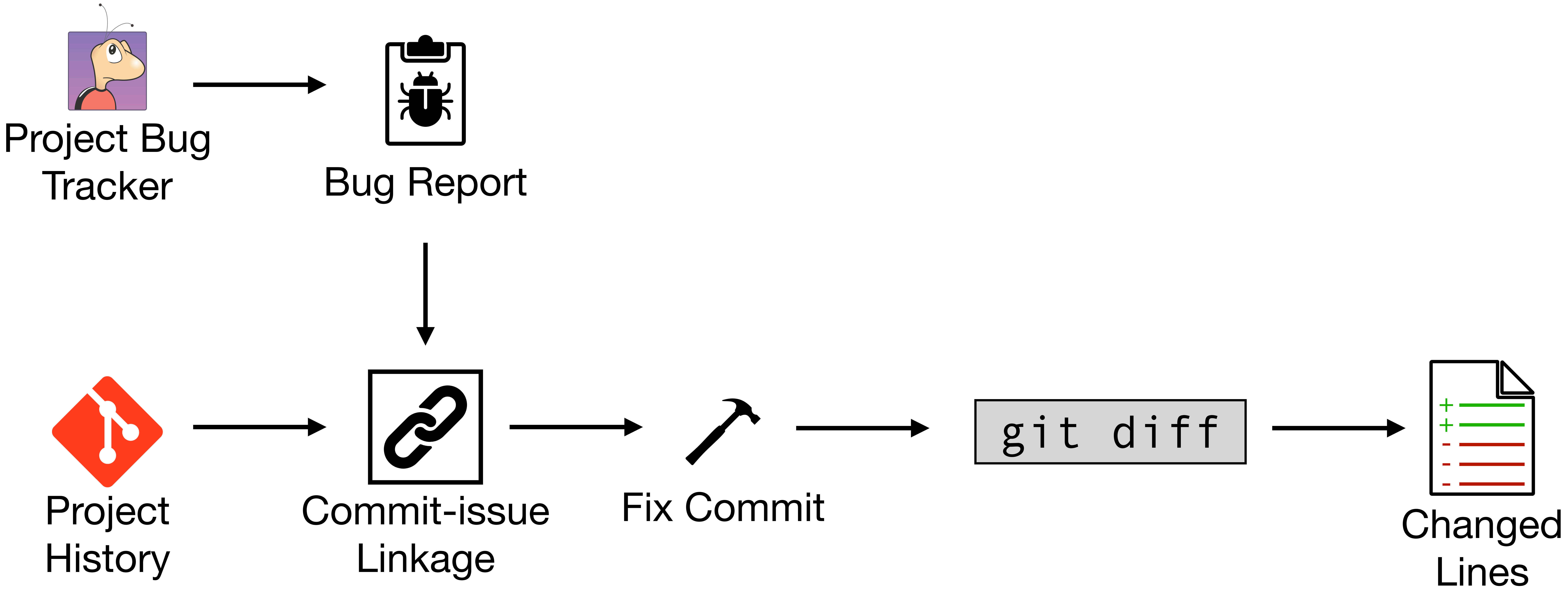
`git diff`

The *git diff* allows the retrieval of the **lines changed** (added and deleted) in the files modified in the fixing commit.

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution. Let's go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)



Project Bug Tracker

Bug Report

Project History

Commit-issue Linkage

Fix Commit

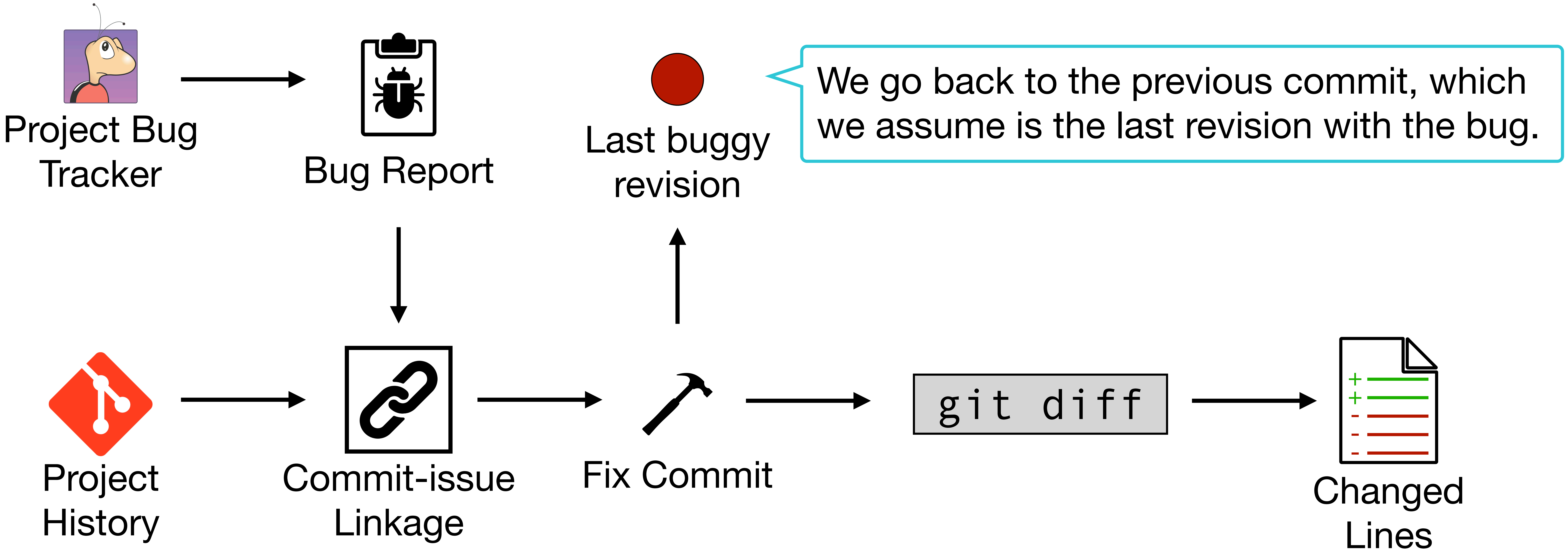
`git diff`

Changed Lines

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution. Let's go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)



Project Bug Tracker

Bug Report

Last buggy revision

We go back to the previous commit, which we assume is the last revision with the bug.

Project History

Commit-issue Linkage

Fix Commit

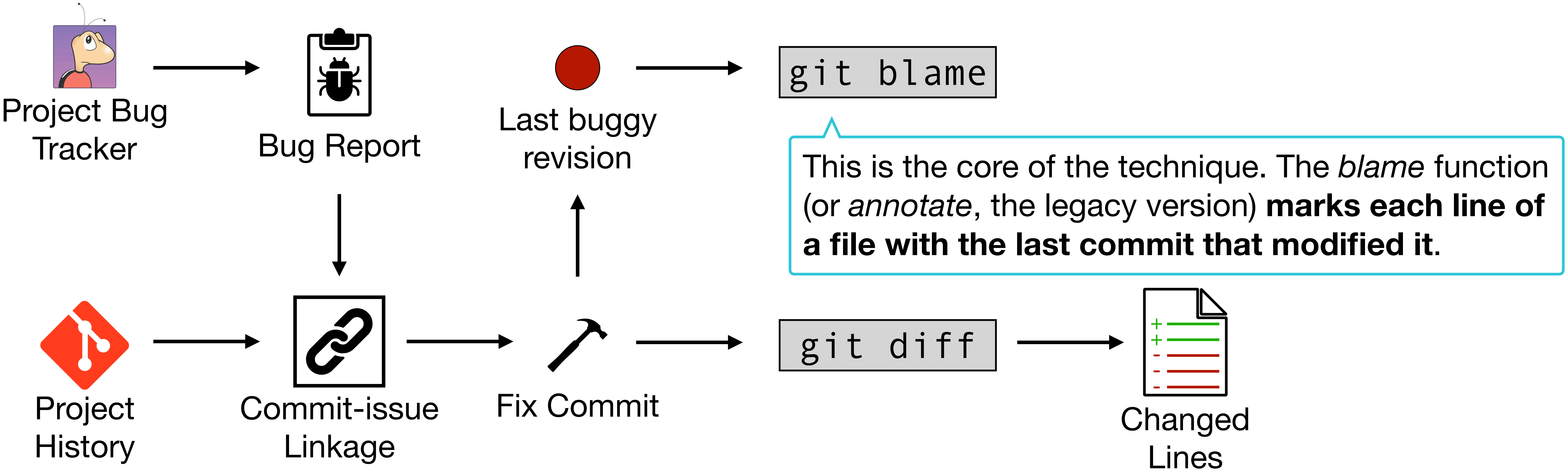
`git diff`

Changed Lines

Mining VCCs: Borrowing from the Bug World

Meneely et al.’s technique doesn’t scale: it’s manual and time-consuming. We need a fully-automated solution. Let’s go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)



Project Bug Tracker

Bug Report

Last buggy revision

`git blame`

This is the core of the technique. The *blame* function (or *annotate*, the legacy version) **marks each line of a file with the last commit that modified it**.

Project History

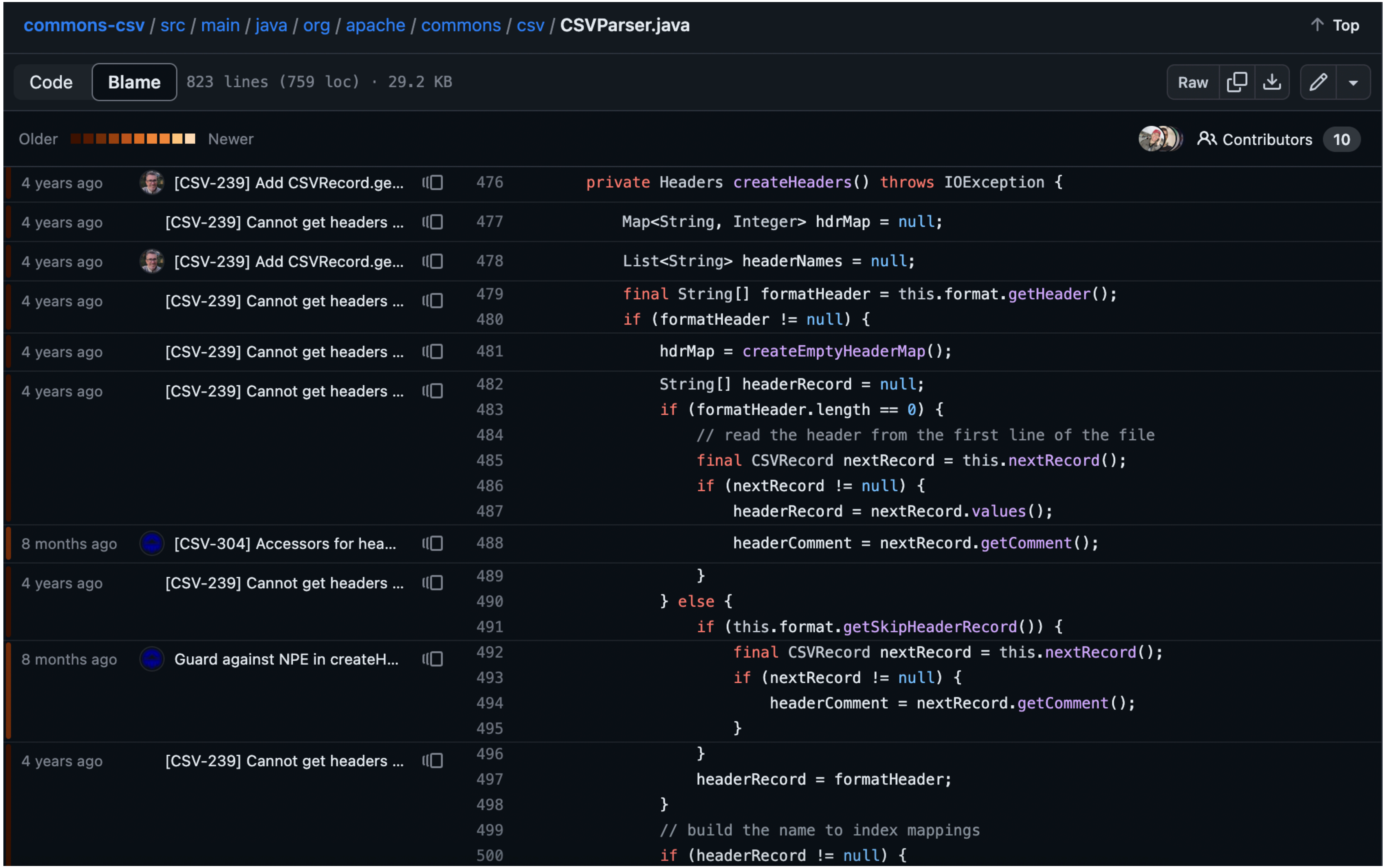
Commit-issue Linkage

Fix Commit

`git diff`

Changed Lines

MSR for Vulnerability Prediction — Mining VCCs



commons-csv / src / main / java / org / apache / commons / csv / **CSVParser.java**

823 lines (759 loc) · 29.2 KB

4 years ago [CSV-239] Add CSVRecord.ge...

4 years ago [CSV-239] Cannot get headers ...

4 years ago [CSV-239] Add CSVRecord.ge...

4 years ago [CSV-239] Cannot get headers ...

4 years ago [CSV-239] Cannot get headers ...

4 years ago [CSV-239] Cannot get headers ...

8 months ago [CSV-304] Accessors for hea...

4 years ago [CSV-239] Cannot get headers ...

8 months ago Guard against NPE in createH...

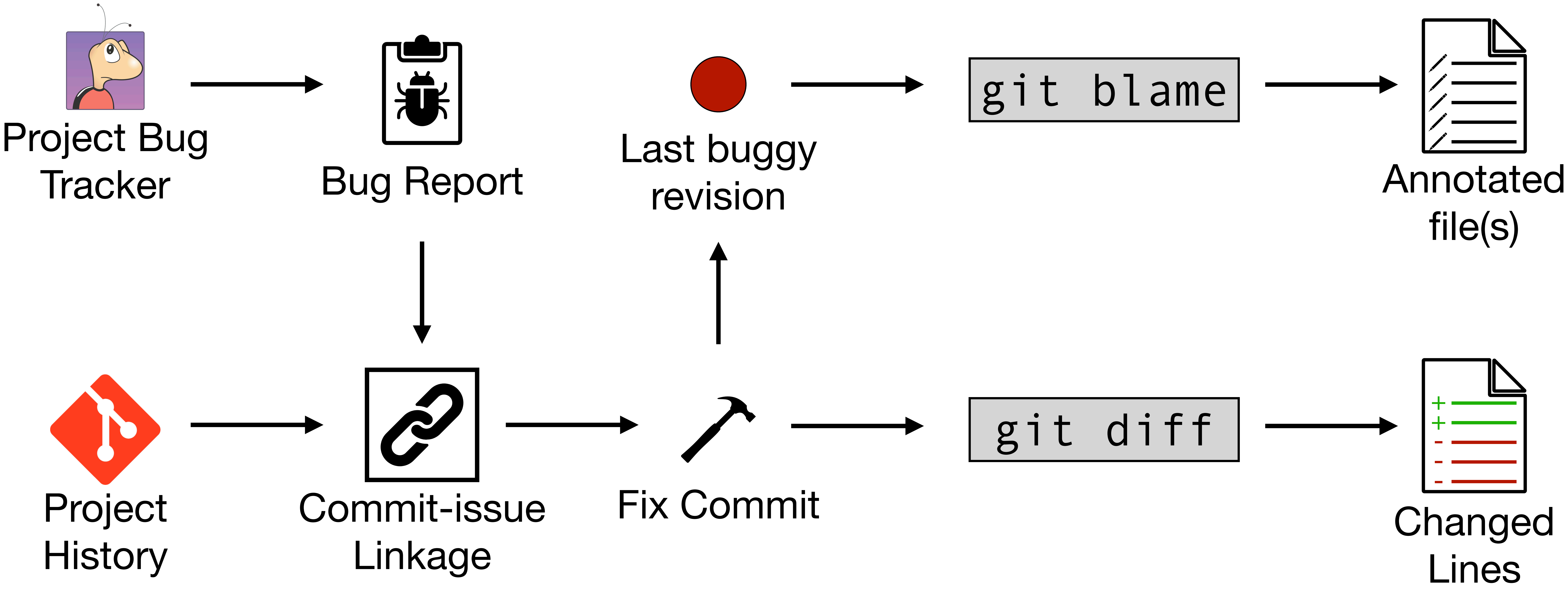
4 years ago [CSV-239] Cannot get headers ...

```java
    private Headers createHeaders() throws IOException {
        Map<String, Integer> hdrMap = null;
        List<String> headerNames = null;
        final String[] formatHeader = this.format.getHeader();
        if (formatHeader != null) {
            hdrMap = createEmptyHeaderMap();
            String[] headerRecord = null;
            if (formatHeader.length == 0) {
                // read the header from the first line of the file
                final CSVRecord nextRecord = this.nextRecord();
                if (nextRecord != null) {
                    headerRecord = nextRecord.values();
                    headerComment = nextRecord.getComment();
                }
            } else {
                if (this.format.getSkipHeaderRecord()) {
                    final CSVRecord nextRecord = this.nextRecord();
                    if (nextRecord != null) {
                        headerComment = nextRecord.getComment();
                    }
                }
                headerRecord = formatHeader;
            }
            // build the name to index mappings
            if (headerRecord != null) {
```

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution. Let's go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)

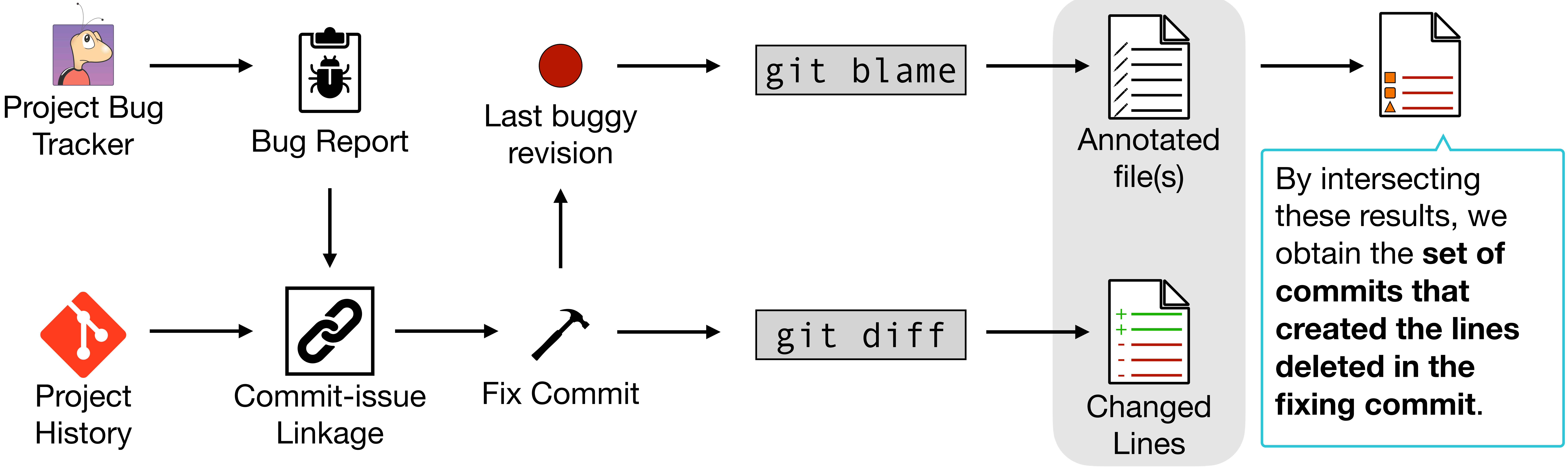


Here, git blame is run on **all the files modified in the fixing commit.** Let's assume it was only just one.

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution.
Let's go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)

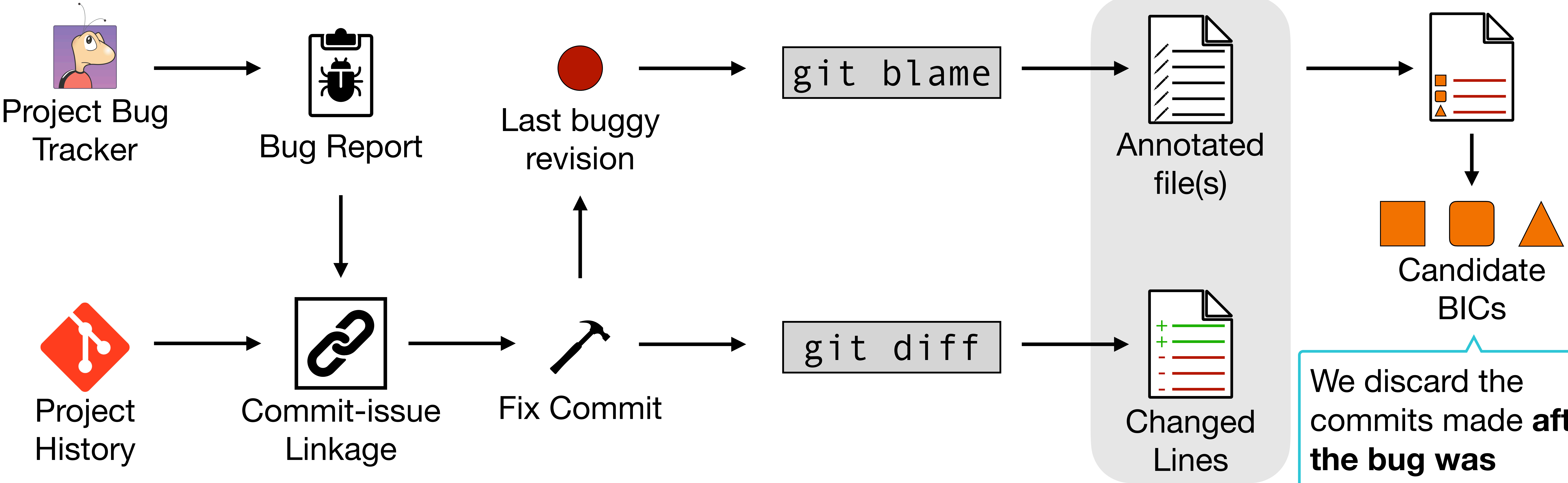


By intersecting these results, we obtain the **set of commits that created the lines deleted in the fixing commit.**

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution.
Let's go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)

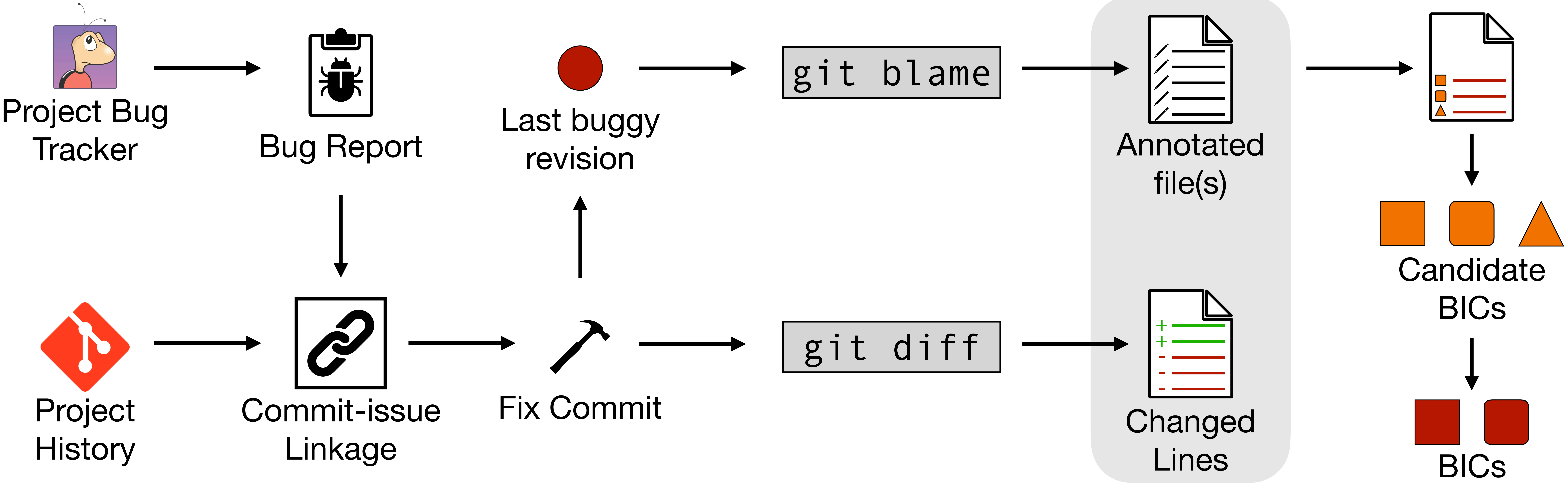


We discard the commits made **after the bug was reported**.

Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution. Let's go back a couple of years: 2005!

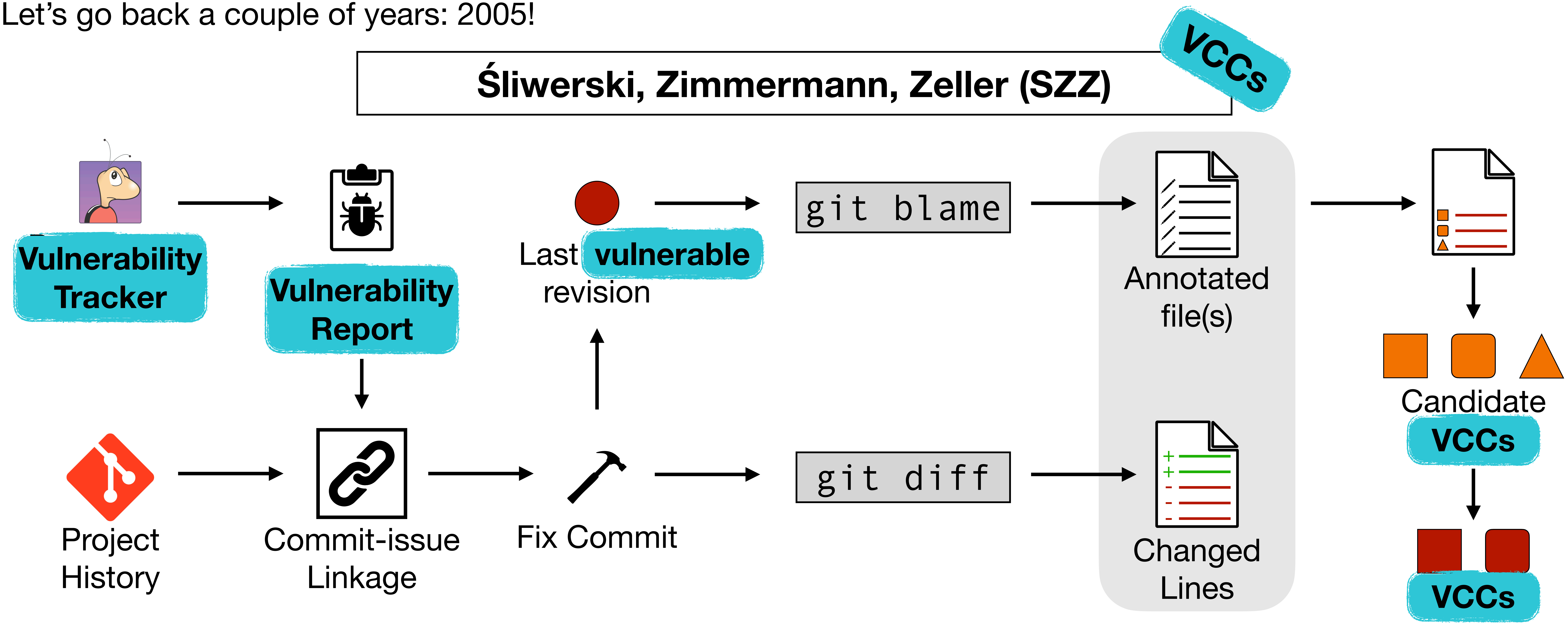
Śliwerski, Zimmermann, Zeller (SZZ)



Mining VCCs: Borrowing from the Bug World

Meneely et al.'s technique doesn't scale: it's manual and time-consuming. We need a fully-automated solution. Let's go back a couple of years: 2005!

Śliwerski, Zimmermann, Zeller (SZZ)



Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Comments and Blank Lines

If the fixing commit also **modified an existing comment** or **removed a blank line**, the BICs (or VCCs) resulting from blaming these lines would be false positives: they made no real contribution to the bug.

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Comments and Blank Lines

If the fixing commit also **modified an existing comment** or **removed a blank line**, the BICs (or VCCs) resulting from blaming these lines would be false positives: they made no real contribution to the bug.

🔴 Last buggy/vulnerable

```
public void foo() {
  // print report
  if (report == null)
  {
    println(report);

  }
```

🟢 Fixed Revision

```
public void foo() {
  // print out report
  if (report != null)
  {
    println(report);
  }
```

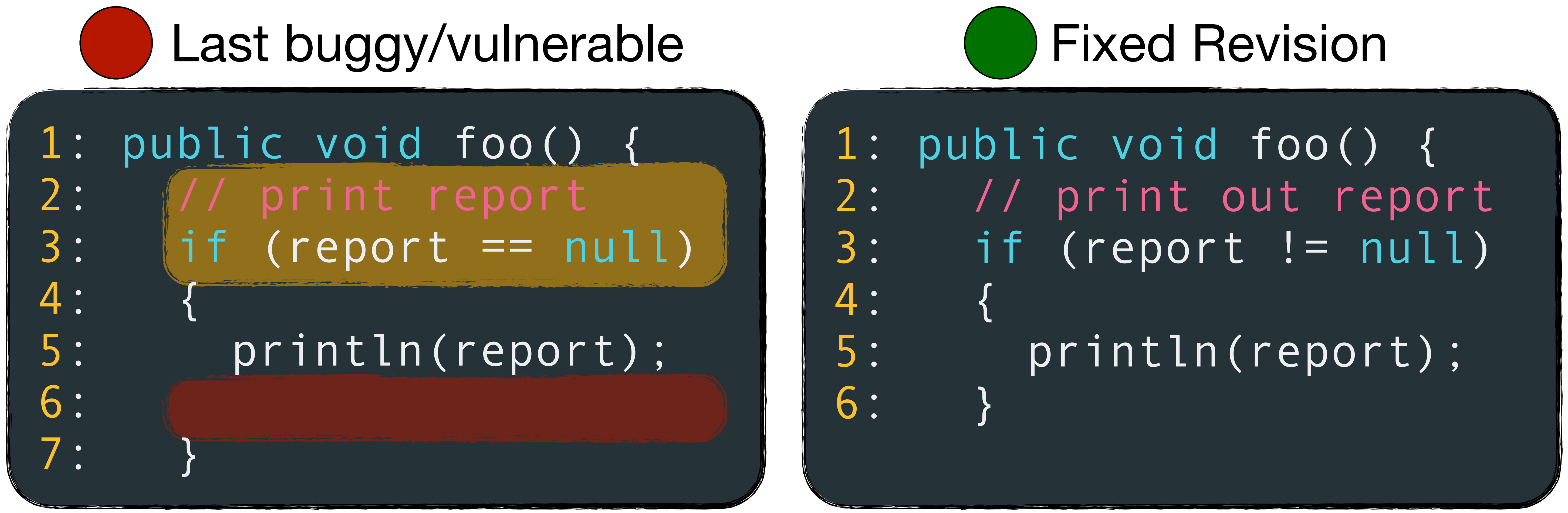
Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Comments and Blank Lines

If the fixing commit also **modified an existing comment** or **removed a blank line**, the BICs (or VCCs) resulting from blaming these lines would be false positives: they made no real contribution to the bug.

Two lines changed, one was just deleted.



● Last buggy/vulnerable

```
public void foo() {
  // print report
  if (report == null)
  {
    println(report);

  }
```

● Fixed Revision

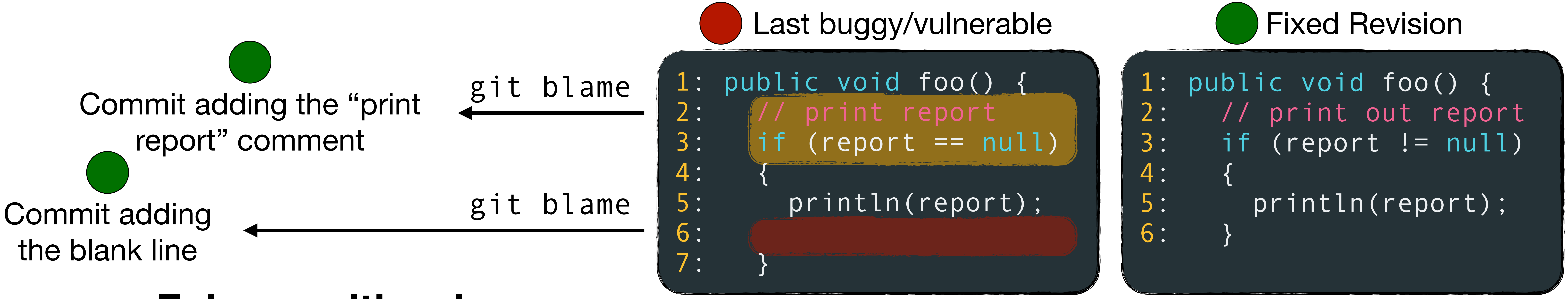
```
public void foo() {
  // print out report
  if (report != null)
  {
    println(report);
  }
```

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Comments and Blank Lines

If the fixing commit also **modified an existing comment** or **removed a blank line**, the BICs (or VCCs) resulting from blaming these lines would be <u>false positives</u>: they made no real contribution to the bug.



🟢 Commit adding the "print report" comment

🟢 Commit adding the blank line

🔴 Last buggy/vulnerable

```
public void foo() {
  // print report
  if (report == null)
  {
    println(report);

  }
```

🟢 Fixed Revision

```
public void foo() {
  // print out report
  if (report != null)
  {
    println(report);
  }
```

False positives!

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Format/Aesthetic Changes

If the fixing commit **modified a line that underwent at least one format change after the bug was introduced**, the BICs (or VCCs) resulting from blaming these lines would be <u>false positives</u>, and the <u>real BICs (VCCs) will be false negatives</u>.

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Format/Aesthetic Changes

If the fixing commit **modified a line that underwent at least one format change after the bug was introduced**, the BICs (or VCCs) resulting from blaming these lines would be <u>false positives</u>, and the <u>real BICs (VCCs) will be false negatives</u>.

🔴 Last buggy/vulnerable

```
public void foo() {
  if (folder == null)
    return;
```

🟢 Fixed Revision

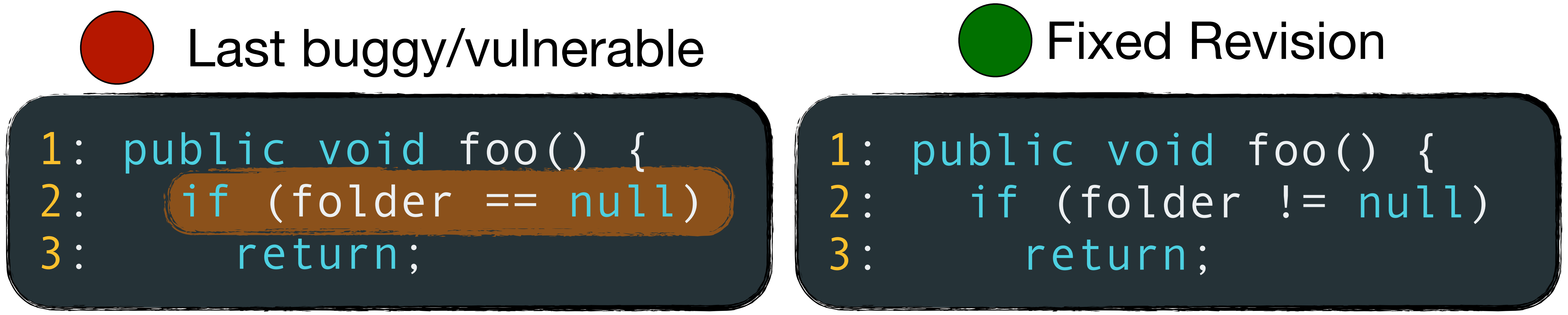
```
public void foo() {
  if (folder != null)
    return;
```

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Format/Aesthetic Changes

If the fixing commit **modified a line that underwent at least one format change after the bug was introduced**, the BICs (or VCCs) resulting from blaming these lines would be false positives, and the real BICs (VCCs) will be false negatives.



Last buggy/vulnerable

```
public void foo() {
  if (folder == null)
     return;
```

Fixed Revision

```
public void foo() {
   if (folder != null)
     return;
```

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Format/Aesthetic Changes

If the fixing commit **modified a line that underwent at least one format change after the bug was introduced**, the BICs (or VCCs) resulting from blaming these lines would be false positives, and the real BICs (VCCs) will be false negatives.

Revision A

```
public void foo() {
   if (folder != null) return;
```

Revision B

```
public void foo() {
   if (folder == null) return;
```

Revision C

```
public void foo() {
   if (folder == null)
      return;
```

Last buggy/vulnerable

```
public void foo() {
   if (folder == null)
      return;
```

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Format/Aesthetic Changes

If the fixing commit **modified a line that underwent at least one format change after the bug was introduced**, the BICs (or VCCs) resulting from blaming these lines would be false positives, and the real BICs (VCCs) will be false negatives.

The commit that brought A to B is adding the bug/ vulnerability!

Revision A

```
public void foo() {
  if (folder != null) return;
```

Revision B

```
public void foo() {
  if (folder == null) return;
```

Revision C

```
public void foo() {
  if (folder == null)
    return;
```

Last buggy/vulnerable

```
public void foo() {
  if (folder == null)
    return;
```

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

Format/Aesthetic Changes

If the fixing commit **modified a line that underwent at least one format change after the bug was introduced**, the BICs (or VCCs) resulting from blaming these lines would be <u>false positives</u>, and the <u>real BICs (VCCs) will be false negatives</u>.

C is the last commit that changed line 2 (false positive), shadowing B (false negative)!

● Revision A

```
1:  public void foo() {
2:     if (folder != null) return;
```

● Revision B

```
1:  public void foo() {
2:     if (folder == null) return;
```

● Revision C

```
1:  public void foo() {
2:     if (folder == null)
3:       return;
```

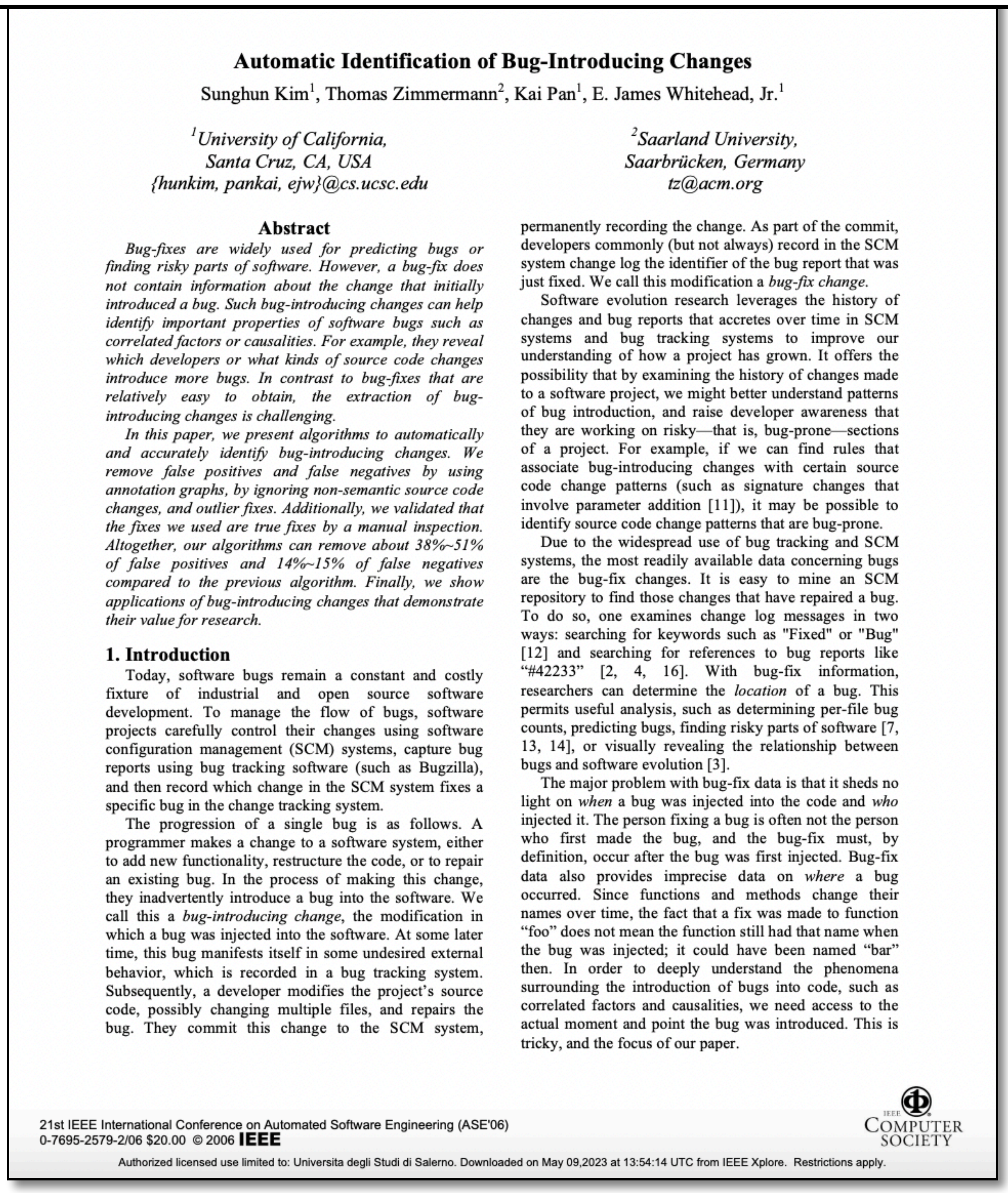
● Last buggy/vulnerable

```
1:  public void foo() {
2:     if (folder == null)
3:       return;
```

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

SZZ by Kim et al.



Automatic Identification of Bug-Introducing Changes

Sunghun Kim[1], Thomas Zimmermann[2], Kai Pan[1], E. James Whitehead, Jr.[1]

[1]*University of California,*
Santa Cruz, CA, USA
{hunkim, pankai, ejw}@cs.ucsc.edu

[2]*Saarland University,*
Saarbrücken, Germany
tz@acm.org

Abstract

Bug-fixes are widely used for predicting bugs or finding risky parts of software. However, a bug-fix does not contain information about the change that initially introduced a bug. Such bug-introducing changes can help identify important properties of software bugs such as correlated factors or causalities. For example, they reveal which developers or what kinds of source code changes introduce more bugs. In contrast to bug-fixes that are relatively easy to obtain, the extraction of bug-introducing changes is challenging.

In this paper, we present algorithms to automatically and accurately identify bug-introducing changes. We remove false positives and false negatives by using annotation graphs, by ignoring non-semantic source code changes, and outlier fixes. Additionally, we validated that the fixes we used are true fixes by a manual inspection. Altogether, our algorithms can remove about 38%~51% of false positives and 14%~15% of false negatives compared to the previous algorithm. Finally, we show applications of bug-introducing changes that demonstrate their value for research.

1. Introduction

Today, software bugs remain a constant and costly fixture of industrial and open source software development. To manage the flow of bugs, software projects carefully control their changes using software configuration management (SCM) systems, capture bug reports using bug tracking software (such as Bugzilla), and then record which change in the SCM system fixes a specific bug in the change tracking system.

The progression of a single bug is as follows. A programmer makes a change to a software system, either to add new functionality, restructure the code, or to repair an existing bug. In the process of making this change, they inadvertently introduce a bug into the software. We call this a *bug-introducing change*, the modification in which a bug was injected into the software. At some later time, this bug manifests itself in some undesired external behavior, which is recorded in a bug tracking system. Subsequently, a developer modifies the project's source code, possibly changing multiple files, and repairs the bug. They commit this change to the SCM system, permanently recording the change. As part of the commit, developers commonly (but not always) record in the SCM system change log the identifier of the bug report that was just fixed. We call this modification a *bug-fix change*.

Software evolution research leverages the history of changes and bug reports that accretes over time in SCM systems and bug tracking systems to improve our understanding of how a project has grown. It offers the possibility that by examining the history of changes made to a software project, we might better understand patterns of bug introduction, and raise developer awareness that they are working on risky—that is, bug-prone—sections of a project. For example, if we can find rules that associate bug-introducing changes with certain source code change patterns (such as signature changes that involve parameter addition [11]), it may be possible to identify source code change patterns that are bug-prone.

Due to the widespread use of bug tracking and SCM systems, the most readily available data concerning bugs are the bug-fix changes. It is easy to mine an SCM repository to find those changes that have repaired a bug. To do so, one examines change log messages in two ways: searching for keywords such as "Fixed" or "Bug" [12] and searching for references to bug reports like "#42233" [2, 4, 16]. With bug-fix information, researchers can determine the *location* of a bug. This permits useful analysis, such as determining per-file bug counts, predicting bugs, finding risky parts of software [7, 13, 14], or visually revealing the relationship between bugs and software evolution [3].

The major problem with bug-fix data is that it sheds no light on *when* a bug was injected into the code and *who* injected it. The person fixing a bug is often not the person who first made the bug, and the bug-fix must, by definition, occur after the bug was first injected. Bug-fix data also provides imprecise data on *where* a bug occurred. Since functions and methods change their names over time, the fact that a fix was made to function "foo" does not mean the function still had that name when the bug was injected; it could have been named "bar" then. In order to deeply understand the phenomena surrounding the introduction of bugs into code, such as correlated factors and causalities, we need access to the actual moment and point the bug was introduced. This is tricky, and the focus of our paper.

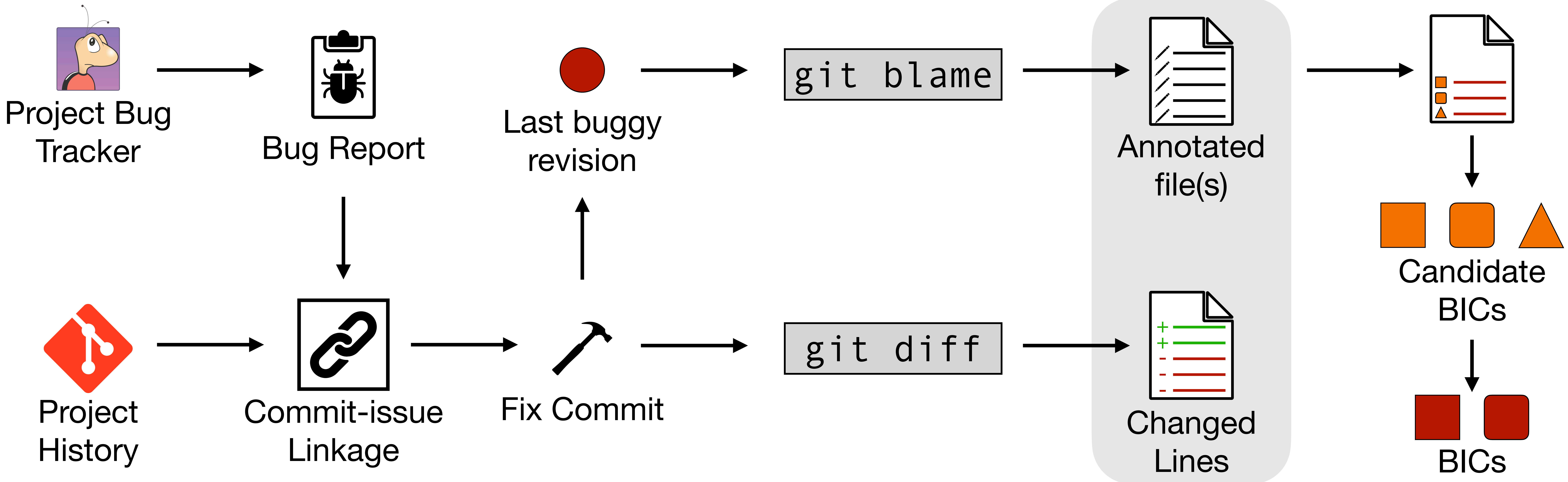
21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00 © 2006 IEEE

S. Kim, T. Zimmermann, K. Pan and E. J. Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, Tokyo, Japan, 2006, pp. 81-90, doi: 10.1109/ASE.2006.23.

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

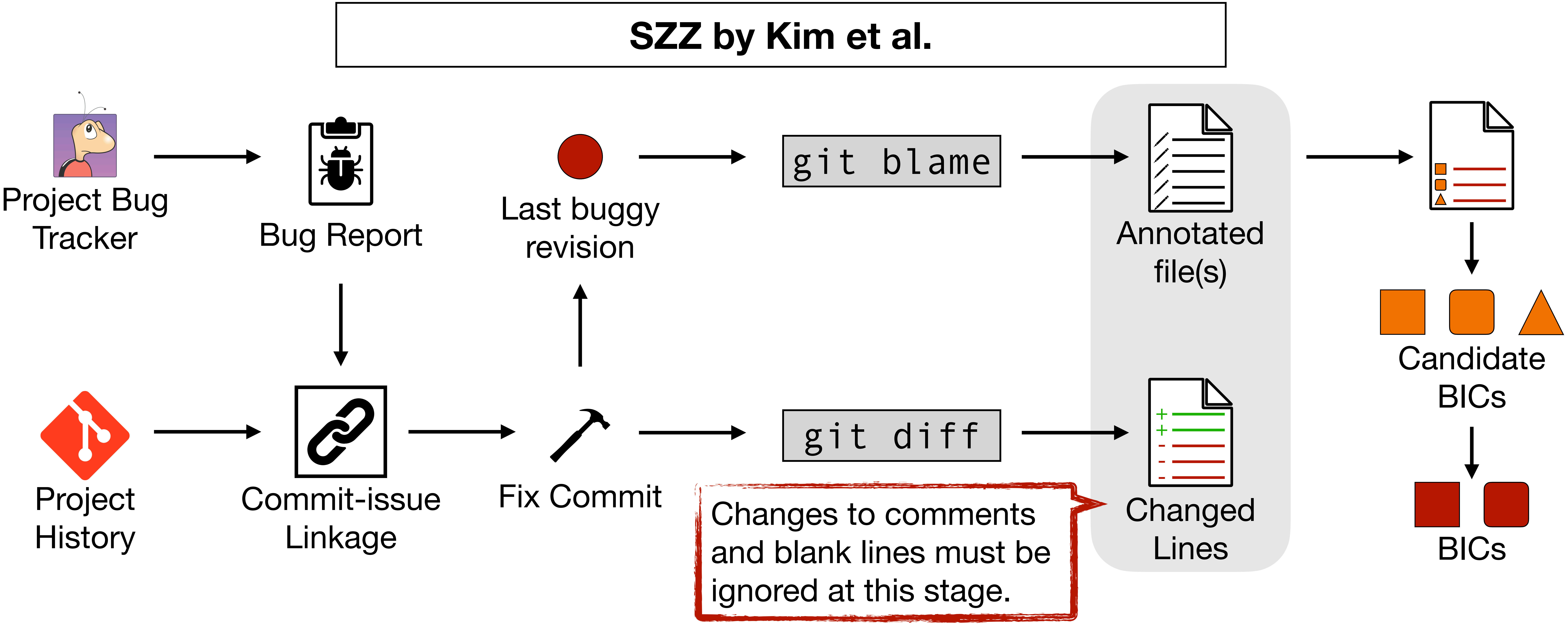
SZZ by Kim et al.



Let’s go back to the original SZZ…

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.



SZZ by Kim et al.

Project Bug Tracker

Bug Report

Last buggy revision

`git blame`

Annotated file(s)

Candidate BICs

BICs

Project History

Commit-issue Linkage

Fix Commit

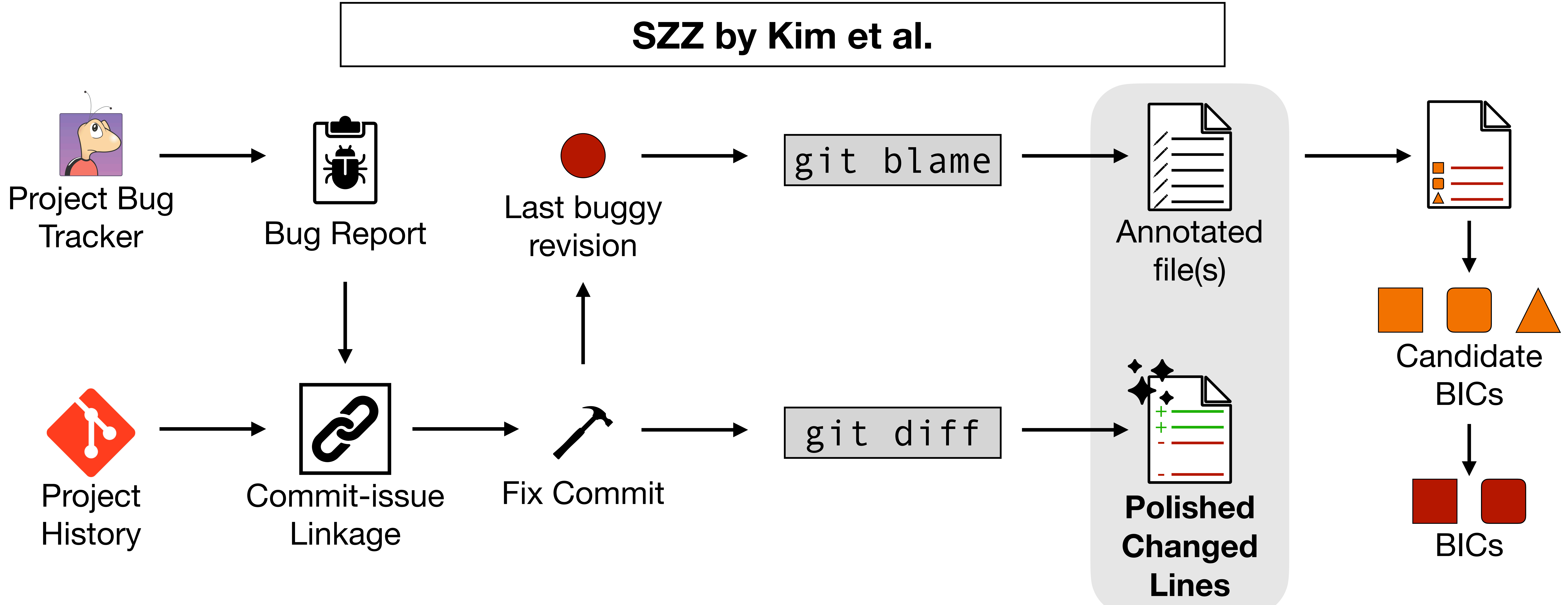
`git diff`

Changed Lines

Changes to comments and blank lines must be ignored at this stage.

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

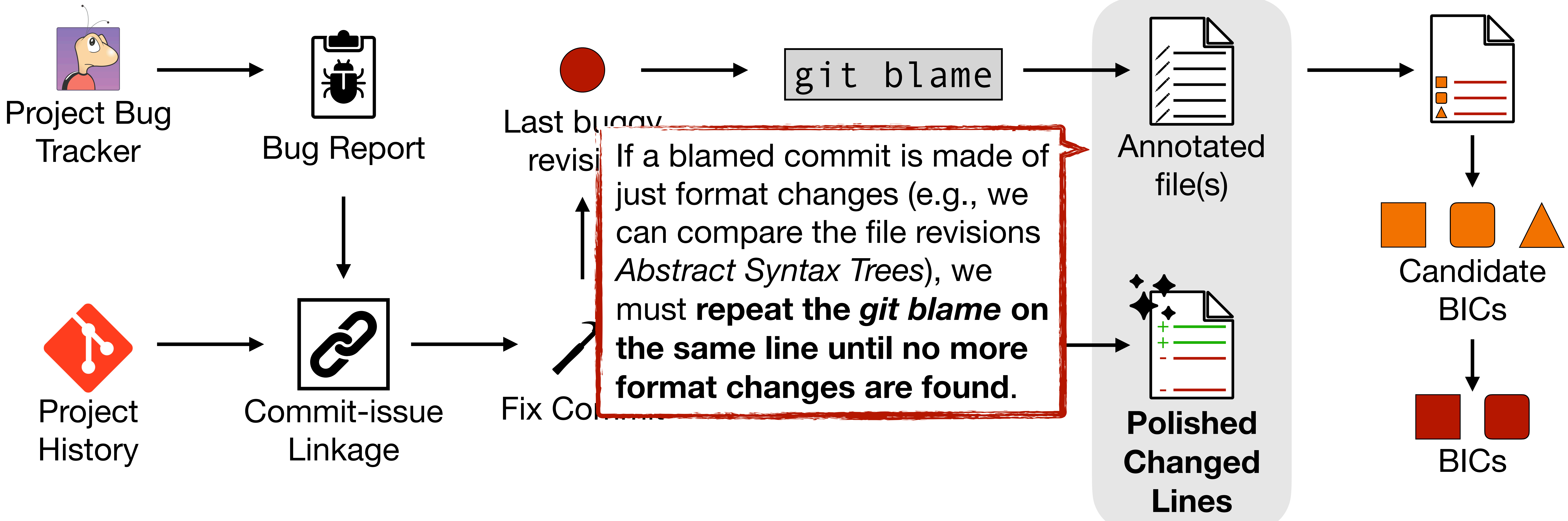


SZZ by Kim et al.

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

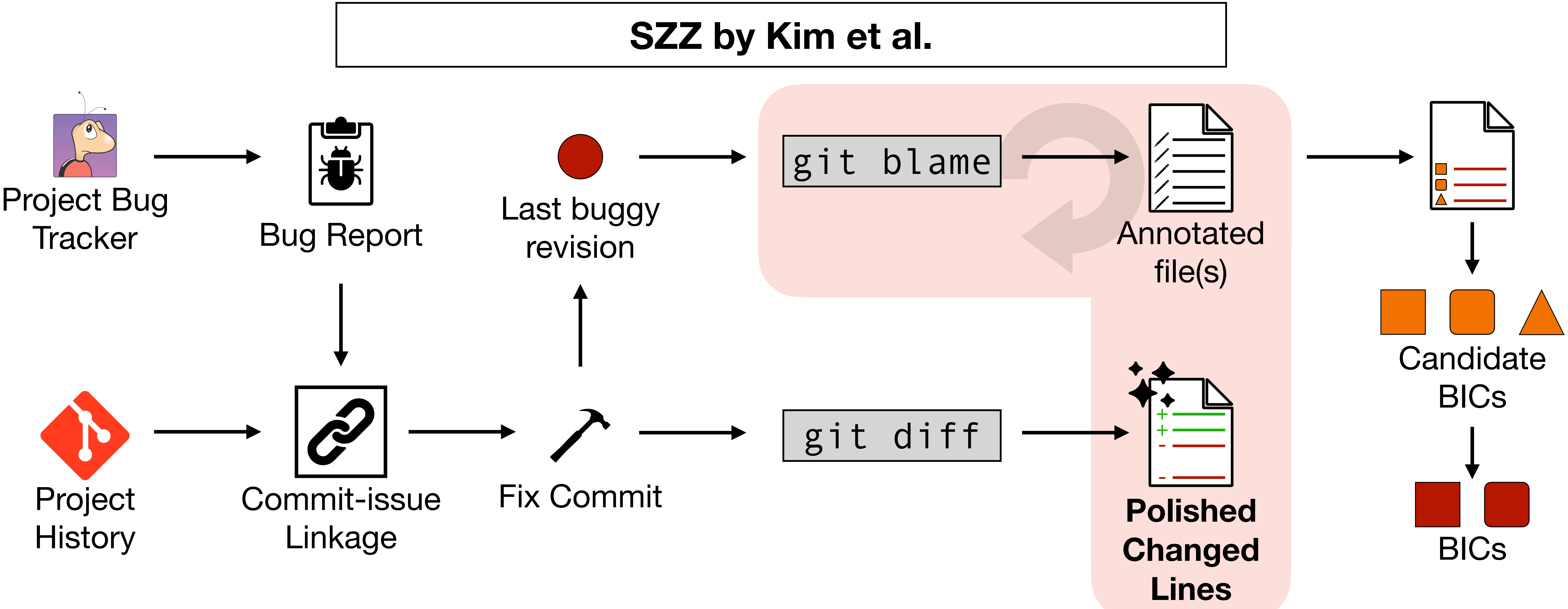
SZZ by Kim et al.



If a blamed commit is made of just format changes (e.g., we can compare the file revisions *Abstract Syntax Trees*), we must **repeat the *git blame* on the same line until no more format changes are found.**

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.



SZZ by Kim et al.

Project Bug Tracker

Bug Report

Last buggy revision

`git blame`

Annotated file(s)

Candidate BICs

BICs

Project History

Commit-issue Linkage

Fix Commit

`git diff`

Polished Changed Lines

Mining VCCs: Borrowing from the Bug World

This is surely a good improvement, but there are still some more problems…

Mining VCCs: Borrowing from the Bug World

This is surely a good improvement, but there are still some more problems…

Meta-changes

The set of candidate BICs/VCCs might be made of commits that do not really modify the source code, e.g., **merge commits**, which incorporate commits from one branch into another.

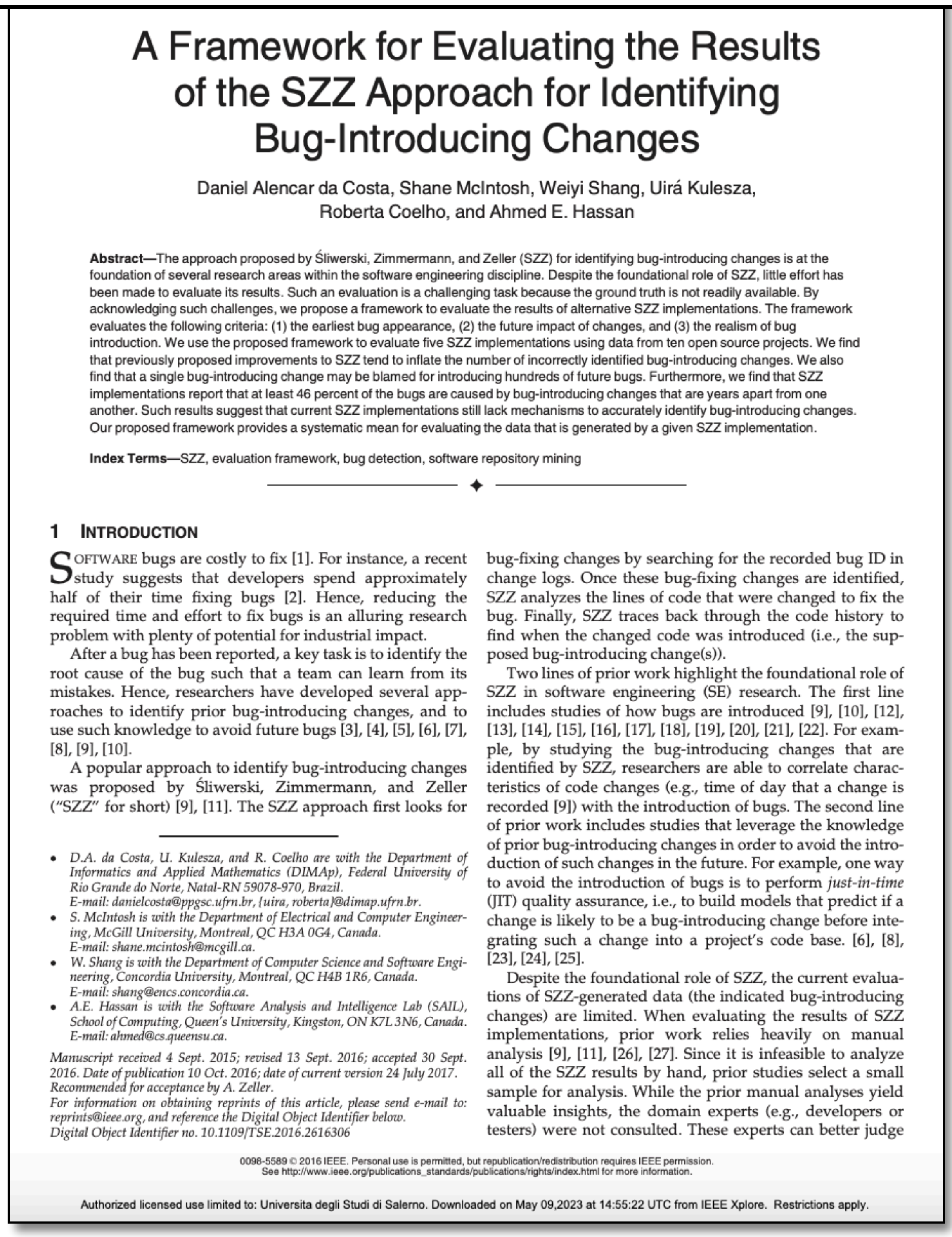
Mining VCCs: Borrowing from the Bug World

This is surely a good improvement, but there are still some more problems…

Meta-changes

The set of candidate BICs/VCCs might be made of commits that do not really modify the source code, e.g., **merge commits**, which incorporate commits from one branch into another.

SZZ by da Costa et al.



A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza,
Roberta Coelho, and Ahmed E. Hassan

Abstract—The approach proposed by Śliwerski, Zimmermann, and Zeller (SZZ) for identifying bug-introducing changes is at the foundation of several research areas within the software engineering discipline. Despite the foundational role of SZZ, little effort has been made to evaluate its results. Such an evaluation is a challenging task because the ground truth is not readily available. By acknowledging such challenges, we propose a framework to evaluate the results of alternative SZZ implementations. The framework evaluates the following criteria: (1) the earliest bug appearance, (2) the future impact of changes, and (3) the realism of bug introduction. We use the proposed framework to evaluate five SZZ implementations using data from ten open source projects. We find that previously proposed improvements to SZZ tend to inflate the number of incorrectly identified bug-introducing changes. We also find that a single bug-introducing change may be blamed for introducing hundreds of future bugs. Furthermore, we find that SZZ implementations report that at least 46 percent of the bugs are caused by bug-introducing changes that are years apart from one another. Such results suggest that current SZZ implementations still lack mechanisms to accurately identify bug-introducing changes. Our proposed framework provides a systematic mean for evaluating the data that is generated by a given SZZ implementation.

Index Terms—SZZ, evaluation framework, bug detection, software repository mining

◆

1 INTRODUCTION

SOFTWARE bugs are costly to fix [1]. For instance, a recent study suggests that developers spend approximately half of their time fixing bugs [2]. Hence, reducing the required time and effort to fix bugs is an alluring research problem with plenty of potential for industrial impact.

After a bug has been reported, a key task is to identify the root cause of the bug such that a team can learn from its mistakes. Hence, researchers have developed several approaches to identify prior bug-introducing changes, and to use such knowledge to avoid future bugs [3], [4], [5], [6], [7], [8], [9], [10].

A popular approach to identify bug-introducing changes was proposed by Śliwerski, Zimmermann, and Zeller ("SZZ" for short) [9], [11]. The SZZ approach first looks for bug-fixing changes by searching for the recorded bug ID in change logs. Once these bug-fixing changes are identified, SZZ analyzes the lines of code that were changed to fix the bug. Finally, SZZ traces back through the code history to find when the changed code was introduced (i.e., the supposed bug-introducing change(s)).

Two lines of prior work highlight the foundational role of SZZ in software engineering (SE) research. The first line includes studies of how bugs are introduced [9], [10], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. For example, by studying the bug-introducing changes that are identified by SZZ, researchers are able to correlate characteristics of code changes (e.g., time of day that a change is recorded [9]) with the introduction of bugs. The second line of prior work includes studies that leverage the knowledge of prior bug-introducing changes in order to avoid the introduction of such changes in the future. For example, one way to avoid the introduction of bugs is to perform *just-in-time* (JIT) quality assurance, i.e., to build models that predict if a change is likely to be a bug-introducing change before integrating such a change into a project's code base. [6], [8], [23], [24], [25].

Despite the foundational role of SZZ, the current evaluations of SZZ-generated data (the indicated bug-introducing changes) are limited. When evaluating the results of SZZ implementations, prior work relies heavily on manual analysis [9], [11], [26], [27]. Since it is infeasible to analyze all of the SZZ results by hand, prior studies select a small sample for analysis. While the prior manual analyses yield valuable insights, the domain experts (e.g., developers or testers) were not consulted. These experts can better judge

---

- *D.A. da Costa, U. Kulesza, and R. Coelho are with the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte, Natal-RN 59078-970, Brazil.*
  E-mail: danielcosta@ppgsc.ufrn.br, {uira, roberta}@dimap.ufrn.br.
- *S. McIntosh is with the Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 0G4, Canada.*
  E-mail: shane.mcintosh@mcgill.ca.
- *W. Shang is with the Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H4B 1R6, Canada.*
  E-mail: shang@encs.concordia.ca.
- *A.E. Hassan is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada.*
  E-mail: ahmed@cs.queensu.ca.

Manuscript received 4 Sept. 2015; revised 13 Sept. 2016; accepted 30 Sept. 2016. Date of publication 10 Oct. 2016; date of current version 24 July 2017.
Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2616306

D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho and A. E. Hassan, "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes," in IEEE Transactions on Software Engineering, vol. 43, no. 7, pp. 641-657, 1 July 2017, doi: 10.1109/TSE.2016.2616306.

Mining VCCs: Borrowing from the Bug World

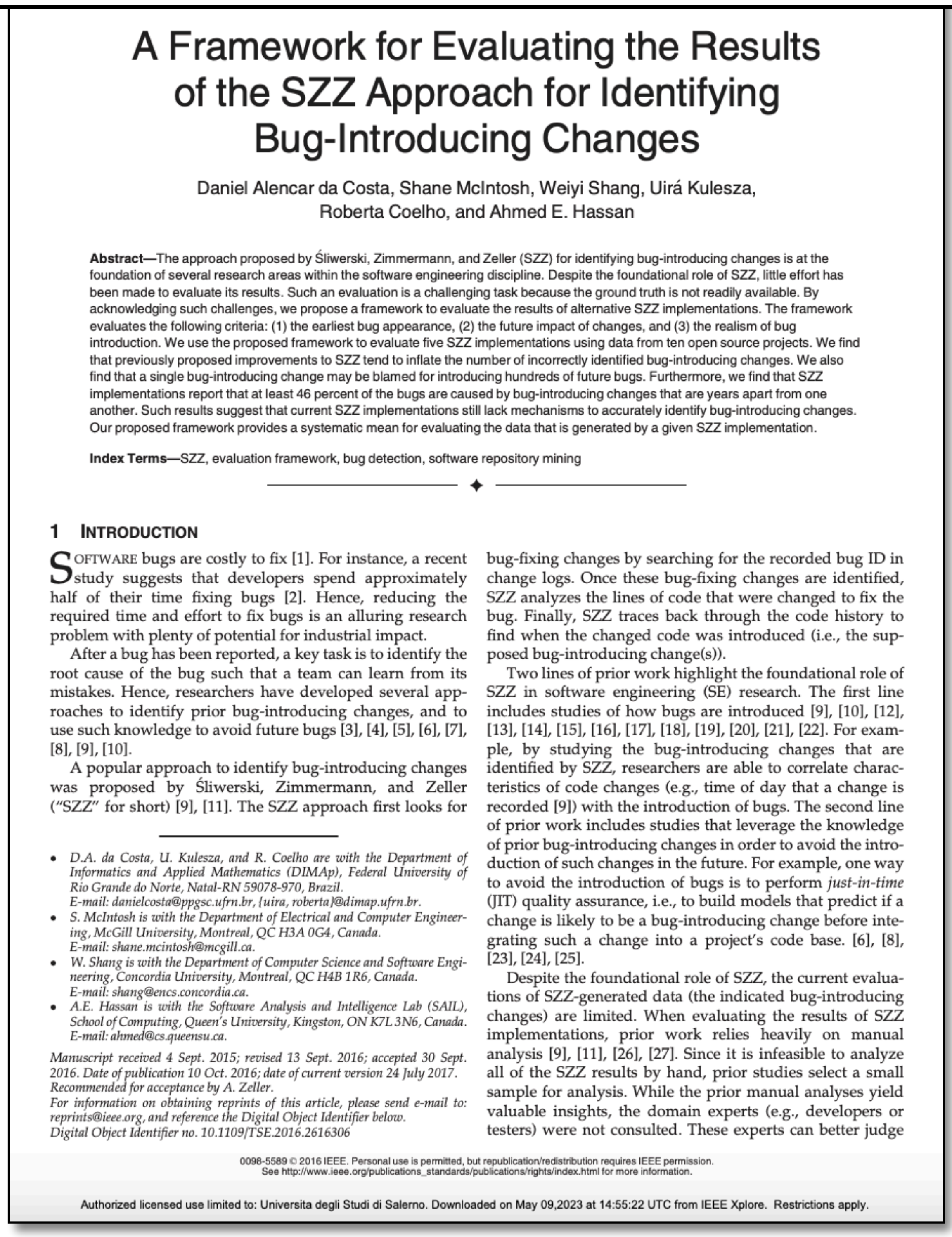
This is surely a good improvement, but there are still some more problems…

Meta-changes

The set of candidate BICs/VCCs might be made of commits that do not really modify the source code, e.g., **merge commits**, which incorporate commits from one branch into another.

Basically, it's a variant of the SZZ by Kim et al. that **ignores merge commits while traversing the history with the repeated *git blames*.**

SZZ by da Costa et al.



A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza,
Roberta Coelho, and Ahmed E. Hassan

Abstract—The approach proposed by Śliwerski, Zimmermann, and Zeller (SZZ) for identifying bug-introducing changes is at the foundation of several research areas within the software engineering discipline. Despite the foundational role of SZZ, little effort has been made to evaluate its results. Such an evaluation is a challenging task because the ground truth is not readily available. By acknowledging such challenges, we propose a framework to evaluate the results of alternative SZZ implementations. The framework evaluates the following criteria: (1) the earliest bug appearance, (2) the future impact of changes, and (3) the realism of bug introduction. We use the proposed framework to evaluate five SZZ implementations using data from ten open source projects. We find that previously proposed improvements to SZZ tend to inflate the number of incorrectly identified bug-introducing changes. We also find that a single bug-introducing change may be blamed for introducing hundreds of future bugs. Furthermore, we find that SZZ implementations report that at least 46 percent of the bugs are caused by bug-introducing changes that are years apart from one another. Such results suggest that current SZZ implementations still lack mechanisms to accurately identify bug-introducing changes. Our proposed framework provides a systematic mean for evaluating the data that is generated by a given SZZ implementation.

Index Terms—SZZ, evaluation framework, bug detection, software repository mining

◆

1 INTRODUCTION

SOFTWARE bugs are costly to fix [1]. For instance, a recent study suggests that developers spend approximately half of their time fixing bugs [2]. Hence, reducing the required time and effort to fix bugs is an alluring research problem with plenty of potential for industrial impact.

After a bug has been reported, a key task is to identify the root cause of the bug such that a team can learn from its mistakes. Hence, researchers have developed several approaches to identify prior bug-introducing changes, and to use such knowledge to avoid future bugs [3], [4], [5], [6], [7], [8], [9], [10].

A popular approach to identify bug-introducing changes was proposed by Śliwerski, Zimmermann, and Zeller ("SZZ" for short) [9], [11]. The SZZ approach first looks for bug-fixing changes by searching for the recorded bug ID in change logs. Once these bug-fixing changes are identified, SZZ analyzes the lines of code that were changed to fix the bug. Finally, SZZ traces back through the code history to find when the changed code was introduced (i.e., the supposed bug-introducing change(s)).

Two lines of prior work highlight the foundational role of SZZ in software engineering (SE) research. The first line includes studies of how bugs are introduced [9], [10], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. For example, by studying the bug-introducing changes that are identified by SZZ, researchers are able to correlate characteristics of code changes (e.g., time of day that a change is recorded [9]) with the introduction of bugs. The second line of prior work includes studies that leverage the knowledge of prior bug-introducing changes in order to avoid the introduction of such changes in the future. For example, one way to avoid the introduction of bugs is to perform *just-in-time* (JIT) quality assurance, i.e., to build models that predict if a change is likely to be a bug-introducing change before integrating such a change into a project's code base. [6], [8], [23], [24], [25].

Despite the foundational role of SZZ, the current evaluations of SZZ-generated data (the indicated bug-introducing changes) are limited. When evaluating the results of SZZ implementations, prior work relies heavily on manual analysis [9], [11], [26], [27]. Since it is infeasible to analyze all of the SZZ results by hand, prior studies select a small sample for analysis. While the prior manual analyses yield valuable insights, the domain experts (e.g., developers or testers) were not consulted. These experts can better judge

---

- *D.A. da Costa, U. Kulesza, and R. Coelho are with the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte, Natal-RN 59078-970, Brazil. E-mail: danielcosta@ppgsc.ufrn.br, {uira, roberta}@dimap.ufrn.br.*
- *S. McIntosh is with the Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 0G4, Canada. E-mail: shane.mcintosh@mcgill.ca.*
- *W. Shang is with the Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H4B 1R6, Canada. E-mail: shang@encs.concordia.ca.*
- *A.E. Hassan is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: ahmed@cs.queensu.ca.*

Manuscript received 4 Sept. 2015; revised 13 Sept. 2016; accepted 30 Sept. 2016. Date of publication 10 Oct. 2016; date of current version 24 July 2017.
Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2616306

D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho and A. E. Hassan, "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes," in IEEE Transactions on Software Engineering, vol. 43, no. 7, pp. 641-657, 1 July 2017, doi: 10.1109/TSE.2016.2616306.

Mining VCCs: Borrowing from the Bug World

Many SZZ variants have been proposed over the years. It is difficult to remember them all or understand which is better. Luckily, some studies put things in order.

Mining VCCs: Borrowing from the Bug World

Many SZZ variants have been proposed over the years. It is difficult to remember them all or understand which is better. Luckily, some studies put things in order.

2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)

Evaluating SZZ Implementations Through a Developer-informed Oracle

Giovanni Rosa*, Luca Pascarella†, Simone Scalabrino*, Rosalia Tufano†, Gabriele Bavota†,
Michele Lanza†, and Rocco Oliveto*
*University of Molise, Italy
†Software Institute @ USI Università della Svizzera italiana, Switzerland

***Abstract*—The SZZ algorithm for identifying bug-inducing changes has been widely used to evaluate defect prediction techniques and to empirically investigate when, how, and by whom bugs are introduced. Over the years, researchers have proposed several heuristics to improve the SZZ accuracy, providing various implementations of SZZ. However, fairly evaluating those implementations on a reliable oracle is an open problem: SZZ evaluations usually rely on (i) the manual analysis of the SZZ output to classify the identified bug-inducing commits as true or false positives; or (ii) a golden set linking bug-fixing and bug-inducing commits. In both cases, these manual evaluations are performed by researchers with limited knowledge of the studied subject systems. Ideally, there should be a golden set created by the original developers of the studied systems.**

We propose a methodology to build a "developer-informed" oracle for the evaluation of SZZ variants. We use Natural Language Processing (NLP) to identify bug-fixing commits in which developers explicitly reference the commit(s) that introduced a fixed bug. This was followed by a manual filtering step aimed at ensuring the quality and accuracy of the oracle. Once built, we used the oracle to evaluate several variants of the SZZ algorithm in terms of their accuracy. Our evaluation helped us to distill a set of lessons learned to further improve the SZZ algorithm.

***Index Terms*—SZZ, Defect Prediction, Empirical Study**

I. INTRODUCTION

The SZZ algorithm, proposed by Śliwerski, Zimmermann, and Zeller [1] at MSR 2005, identifies, given a bug-fixing commit C_{BF}, the commits that likely introduced the bug fixed in C_{BF}. These commits are termed "bug-inducing" commits. In essence, given C_{BF} as input, SZZ identifies the last change (commit) to each source code line changed in C_{BF} (*i.e.,* changed to fix the bug). This is done by relying on the annotation/blame feature of versioning systems. The identified commits are considered as the ones that later on triggered the bug-fixing commit C_{BF}.

SZZ has been widely adopted to (i) design and evaluate defect prediction techniques [2]–[6], and to (ii) run empirical studies aimed at investigating under which circumstances bugs are introduced [7]–[10]. The relevance of the SZZ algorithm was recognized a decade later with a MIP (Most Influential Paper award) presented at the 12th Working Conference on Mining Software Repositories (MSR 2015).

Several researchers have proposed variants of the original algorithm, with the goal of boosting its accuracy [11]–[16].

For example, one issue with the basic SZZ implementation is that it considers changes to code comments and whitespaces like any other change.

This means that if a comment is modified in C_{BF}, the latest change to that comment is mistakenly considered as a bug-inducing commit. An improvement by Kim *et al.* [11] was therefore to ignore changes to code comments and blank lines as candidate bug-inducing commits.

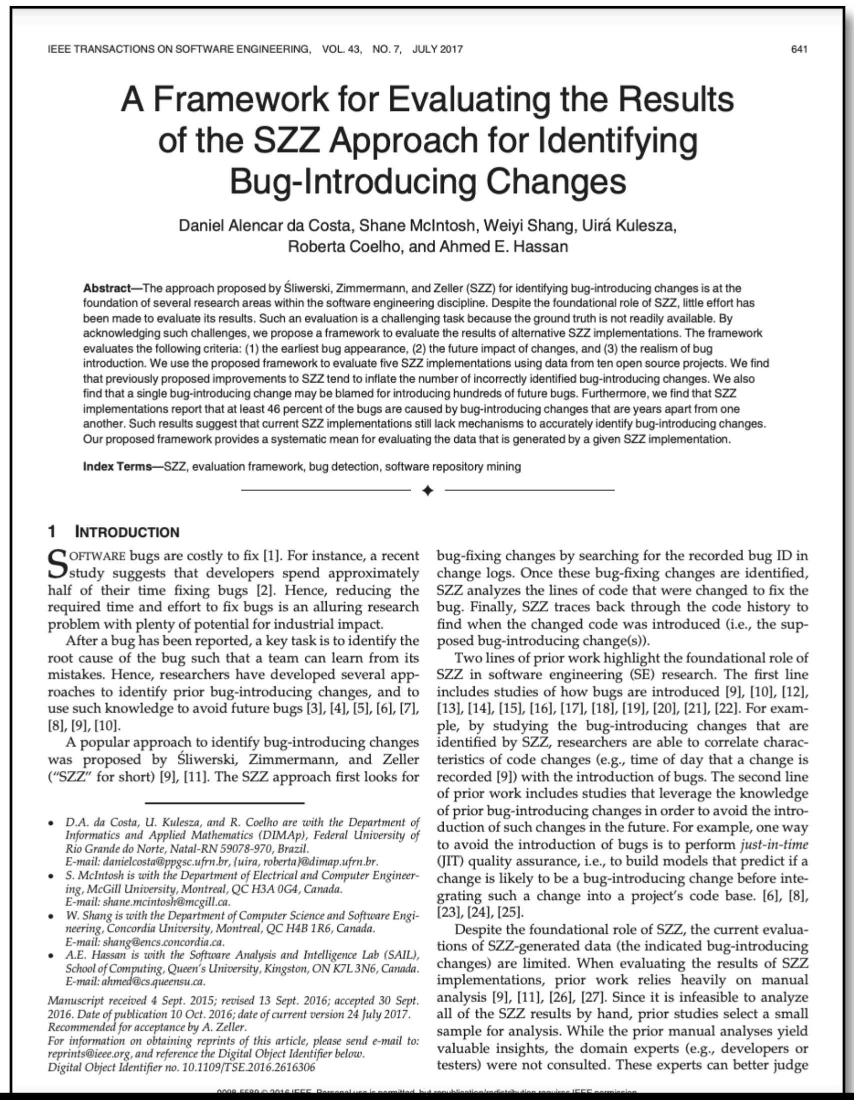
Despite the major advances made on the accuracy of SZZ, Alencar da Costa *et al.* [14] highlighted the major difficulties in fairly evaluating and comparing the SZZ variants proposed in the literature. They observed that the studies presenting and evaluating SZZ variants mostly rely on manual analysis of a small sample of SZZ results [1], [11]–[13], with the goal of evaluating its accuracy. Such an evaluation is usually performed by the researchers who—not being the original developers of the studied systems—do not always have the knowledge needed to correctly identify the bug introducing commit. Also, due to the high cost of such a manual analysis, it is usually performed on a small sample of the identified bug-inducing commits. Other researchers built instead a ground truth to evaluate the performance of the SZZ algorithm [16]. However, also in these cases, the ground truth is produced by the researchers. Alencar da Costa *et al.* [14] called for evaluations performed with "*domain experts (*e.g., *developers or testers)*" reporting however that "*such an analysis is impractical*" since "*the experts would need to verify a large sample of bug-introducing changes, which is difficult to scale up to the size of modern defect datasets*" [14].

We present a methodology to build a "developer-informed" oracle for the evaluation of SZZ implementations. To explain its idea, let us take as example commit `a8a97bd` from the `apache/thrift` GitHub project, accompanied by a commit message saying: "*THRIFT-4513: fix bug in comparator introduced by e58f75d*". The developer fixing the bug is explicitly documenting the commit that introduced such a bug. Based on this observation, we defined a number of strict NLP-based heuristics to automatically identify notes in bug-fixing commits in which developers explicitly reference the commit(s) that introduced the fixed bug. We applied these heuristics to a total of 19,603,736 mined through GH Archive [39], which archives all public events on GitHub.

Our goal with the above described process is not to be exhaustive, *i.e.,* we do not want to identify all bug-fixing commits in which developers indicated the bug-inducing commit(s), but rather to obtain a high-quality dataset of commits that were certainly of the bug-inducing kind.

Rosa et al.

Comparison of nine SZZ variants on 123 OSS projects.



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 43, NO. 7, JULY 2017 641

A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza,
Roberta Coelho, and Ahmed E. Hassan

Abstract—The approach proposed by Śliwerski, Zimmermann, and Zeller (SZZ) for identifying bug-introducing changes is at the foundation of several research areas within the software engineering discipline. Despite the foundational role of SZZ, little effort has been made to evaluate its results. Such an evaluation is a challenging task because the ground truth is not readily available. By acknowledging such challenges, we propose a framework to evaluate the results of alternative SZZ implementations. The framework evaluates the following criteria: (1) the earliest bug appearance, (2) the future impact of changes, and (3) the realism of bug introduction. We use the proposed framework to evaluate five SZZ implementations using data from ten open source projects. We find that previously proposed improvements to SZZ tend to inflate the number of incorrectly identified bug-introducing changes. We also find that a single bug-introducing change may be blamed for introducing hundreds of future bugs. Furthermore, we find that SZZ implementations report that at least 46 percent of the bugs are caused by bug-introducing changes that are years apart from one another. Such results suggest that current SZZ implementations still lack mechanisms to accurately identify bug-introducing changes. Our proposed framework provides a systematic mean for evaluating the data that is generated by a given SZZ implementation.

Index Terms—SZZ, evaluation framework, bug detection, software repository mining

✦

1 INTRODUCTION

SOFTWARE bugs are costly to fix [1]. For instance, a recent study suggests that developers spend approximately half of their time fixing bugs [2]. Hence, reducing the required time and effort to fix bugs is an alluring research problem with plenty of potential for industrial impact.

After a bug has been reported, a key task is to identify the root cause of the bug such that a team can learn from its mistakes. Hence, researchers have developed several approaches to identify prior bug-introducing changes, and to use such knowledge to avoid future bugs [3], [4], [5], [6], [7], [8], [9], [10].

A popular approach to identify bug-introducing changes was proposed by Śliwerski, Zimmermann, and Zeller ("SZZ" for short) [9], [11]. The SZZ approach first looks for bug-fixing changes by searching for the recorded bug ID in change logs. Once these bug-fixing changes are identified, SZZ analyzes the lines of code that were changed to fix the bug. Finally, SZZ traces back through the code history to find when the changed code was introduced (i.e., the supposed bug-introducing change(s)).

Two lines of prior work highlight the foundational role of SZZ in software engineering (SE) research. The first line includes studies of how bugs are introduced [9], [10], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. For example, by studying the bug-introducing changes that are identified by SZZ, researchers are able to correlate characteristics of code changes (e.g., time of day that a change is recorded [9]) with the introduction of bugs. The second line of prior work includes studies that leverage the knowledge of prior bug-introducing changes in order to avoid the introduction of such changes in the future. For example, one way to avoid the introduction of bugs is to perform *just-in-time* (JIT) quality assurance, i.e., to build models that predict if a change is likely to be a bug-introducing change before integrating such a change into a project's code base. [6], [8], [23], [24], [25].

Despite the foundational role of SZZ, the current evaluations of SZZ-generated data (the indicated bug-introducing changes) are limited. When evaluating the results of SZZ implementations, prior work relies heavily on manual analysis [9], [11], [26], [27]. Since it is infeasible to analyze all of the SZZ results by hand, prior studies select a small sample for analysis. While the prior manual analyses yield valuable insights, the domain experts (e.g., developers or testers) were not consulted. These experts can better judge

---

- *D.A. da Costa, U. Kulesza, and R. Coelho are with the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte, Natal-RN 59078-970, Brazil. E-mail: danielcosta@ppgsc.ufrn.br, {uira, roberta}@dimap.ufrn.br.*
- *S. McIntosh is with the Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 0G4, Canada. E-mail: shane.mcintosh@mcgill.ca.*
- *W. Shang is with the Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H4B 1R6, Canada. E-mail: shang@encs.concordia.ca.*
- *A.E. Hassan is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: ahmed@cs.queensu.ca.*

Manuscript received 4 Sept. 2015; revised 13 Sept. 2016; accepted 30 Sept. 2016. Date of publication 10 Oct. 2016; date of current version 24 July 2017. Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2616306

da Costa et al.

Comparison of five SZZ variants on ten OSS projects.

Empirical Software Engineering (2020) 25:1294–1340
https://doi.org/10.1007/s10664-019-09781-y

How bugs are born: a model to identify how bugs are introduced in software components

Gema Rodríguez-Pérez[1] · Gregorio Robles[2] · Alexander Serebrenik[3] · Andy Zaidman[4] · Daniel M. Germán[5] · Jesus M. Gonzalez-Barahona[2]

Published online: 4 February 2020
© The Author(s) 2020

Abstract

When identifying the origin of software bugs, many studies assume that "a bug was introduced by the lines of code that were modified to fix it". However, this assumption does not always hold and at least in some cases, these modified lines are not responsible for introducing the bug. For example, when the bug was caused by a change in an external API. The lack of empirical evidence makes it impossible to assess how important these cases are and therefore, to which extent the assumption is valid. To advance in this direction, and better understand how bugs "are born", we propose a model for defining criteria to identify the first snapshot of an evolving software system that exhibits a bug. This model, based on the *perfect test* idea, decides whether a bug is observed after a change to the software. Furthermore, we studied the model's criteria by carefully analyzing how 116 bugs were introduced in two different open source software projects. The manual analysis helped classify the root cause of those bugs and created manually curated datasets with bug-introducing changes and with bugs that were not introduced by any change in the source code. Finally, we used these datasets to evaluate the performance of four existing SZZ-based algorithms for detecting bug-introducing changes. We found that SZZ-based algorithms are not very accurate, especially when multiple commits are found; the F-Score varies from 0.44 to 0.77, while the percentage of true positives does not exceed 63%. Our results show empirical evidence that the prevalent assumption, "a bug was introduced by the lines of code that were modified to fix it", is just one case of how bugs are introduced in a software system. Finding what introduced a bug is not trivial: bugs can be introduced by the developers and be in the code, or be created irrespective of the code. Thus, further research towards a better understanding of the origin of bugs in software projects could help to improve design integration tests and to design other procedures to make software development more robust.

Keywords Bug origins · Bug-introducing changes · First-failing change · SZZ algorithm · Extrinsic bugs · Intrinsic bugs

---

Communicated by: Per Runeson

✉ Gregorio Robles
grex@gsyc.urjc.es

Extended author information available on the last page of the article.

Rodríguez-Pérez et al.

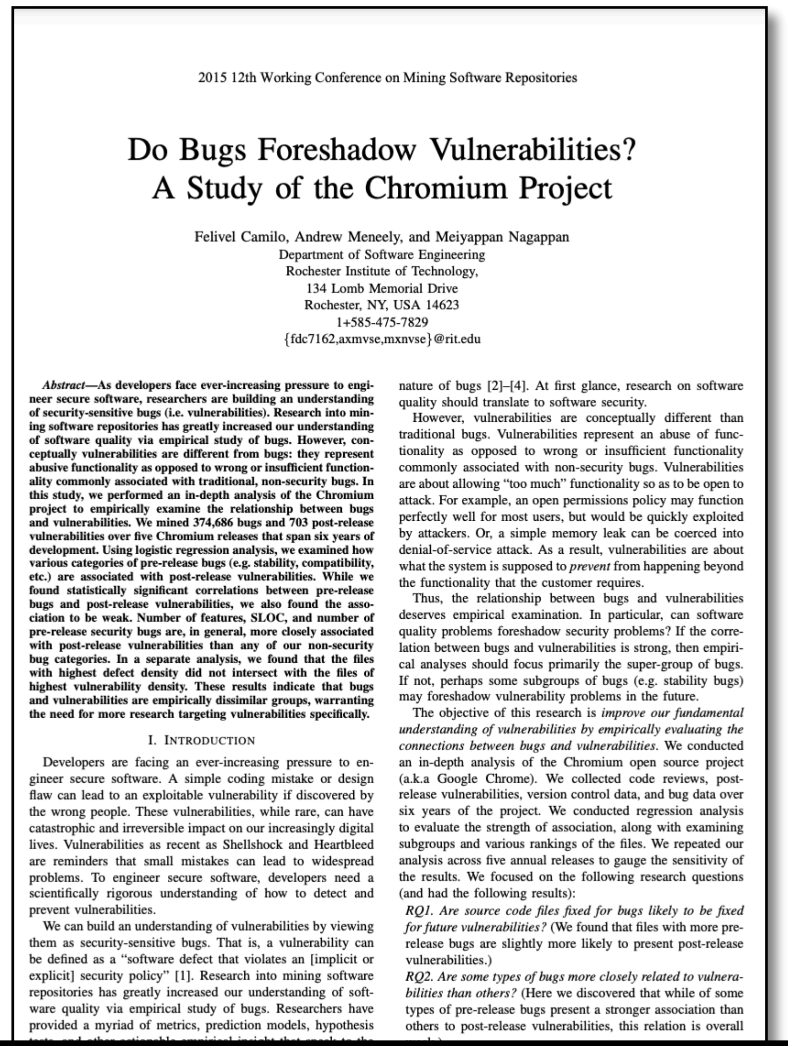
Comparison of four SZZ variants on two OSS projects.

Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs.
Indeed, there are studies explaining how bugs and vulnerabilities differ.

Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs.
Indeed, there are studies explaining how bugs and vulnerabilities differ.



2015 12th Working Conference on Mining Software Repositories

Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project

Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan
Department of Software Engineering
Rochester Institute of Technology,
134 Lomb Memorial Drive
Rochester, NY, USA 14623
1+585-475-7829
{fdc7162,axmvse,mxnvse}@rit.edu

***Abstract*—As developers face ever-increasing pressure to engineer secure software, researchers are building an understanding of security-sensitive bugs (i.e. vulnerabilities). Research into mining software repositories has greatly increased our understanding of software quality via empirical study of bugs. However, conceptually vulnerabilities are different from bugs: they represent abusive functionality as opposed to wrong or insufficient functionality commonly associated with traditional, non-security bugs. In this study, we performed an in-depth analysis of the Chromium project to empirically examine the relationship between bugs and vulnerabilities. We mined 374,686 bugs and 703 post-release vulnerabilities over five Chromium releases that span six years of development. Using logistic regression analysis, we examined how various categories of pre-release bugs (e.g. stability, compatibility, etc.) are associated with post-release vulnerabilities. While we found statistically significant correlations between pre-release bugs and post-release vulnerabilities, we also found the association to be weak. Number of features, SLOC, and number of pre-release security bugs are, in general, more closely associated with post-release vulnerabilities than any of our non-security bug categories. In a separate analysis, we found that the files with highest defect density did not intersect with the files of highest vulnerability density. These results indicate that bugs and vulnerabilities are empirically dissimilar groups, warranting the need for more research targeting vulnerabilities specifically.**

I. INTRODUCTION

Developers are facing an ever-increasing pressure to engineer secure software. A simple coding mistake or design flaw can lead to an exploitable vulnerability if discovered by the wrong people. These vulnerabilities, while rare, can have catastrophic and irreversible impact on our increasingly digital lives. Vulnerabilities as recent as Shellshock and Heartbleed are reminders that small mistakes can lead to widespread problems. To engineer secure software, developers need a scientifically rigorous understanding of how to detect and prevent vulnerabilities.

We can build an understanding of vulnerabilities by viewing them as security-sensitive bugs. That is, a vulnerability can be defined as a "software defect that violates an [implicit or explicit] security policy" [1]. Research into mining software repositories has greatly increased our understanding of software quality via empirical study of bugs. Researchers have provided a myriad of metrics, prediction models, hypothesis nature of bugs [2]–[4]. At first glance, research on software quality should translate to software security.

However, vulnerabilities are conceptually different than traditional bugs. Vulnerabilities represent an abuse of functionality as opposed to wrong or insufficient functionality commonly associated with non-security bugs. Vulnerabilities are about allowing "too much" functionality so as to be open to attack. For example, an open permissions policy may function perfectly well for most users, but would be quickly exploited by attackers. Or, a simple memory leak can be coerced into denial-of-service attack. As a result, vulnerabilities are about what the system is supposed to *prevent* from happening beyond the functionality that the customer requires.

Thus, the relationship between bugs and vulnerabilities deserves empirical examination. In particular, can software quality problems foreshadow security problems? If the correlation between bugs and vulnerabilities is strong, then empirical analyses should focus primarily the super-group of bugs. If not, perhaps some subgroups of bugs (e.g. stability bugs) may foreshadow vulnerability problems in the future.

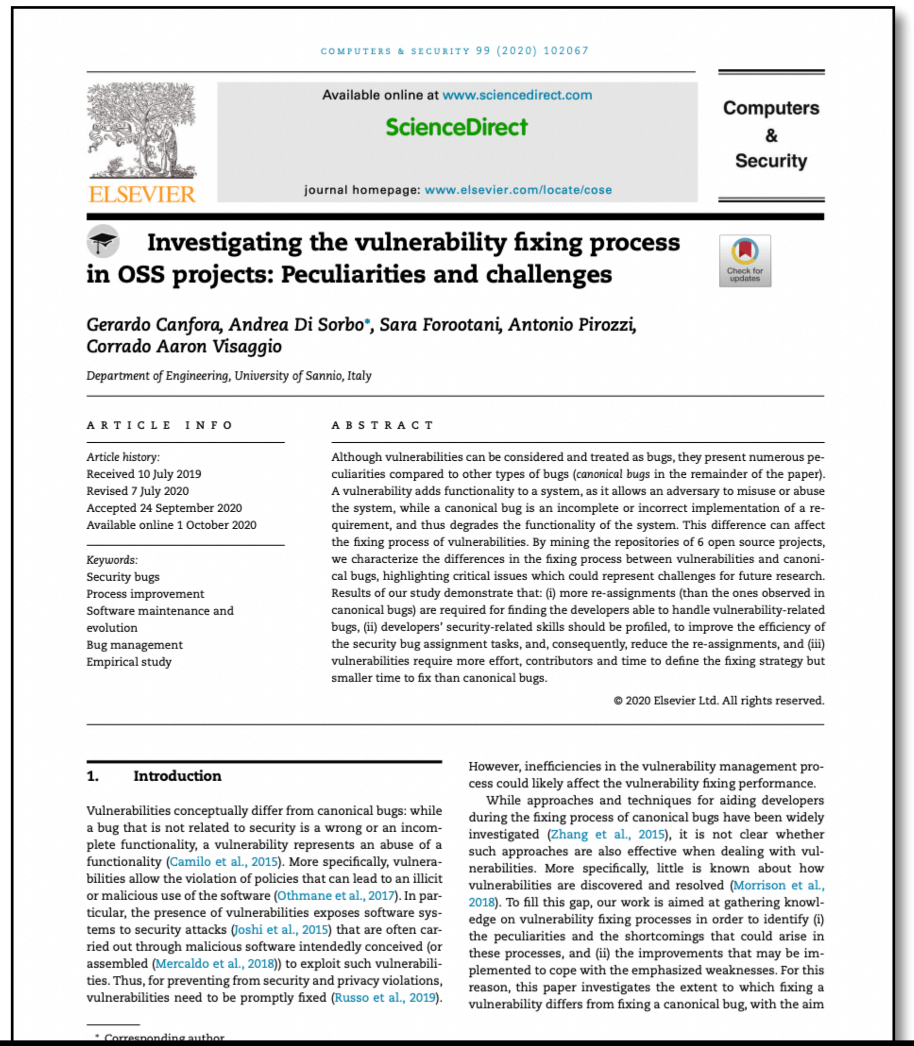
The objective of this research is *improve our fundamental understanding of vulnerabilities by empirically evaluating the connections between bugs and vulnerabilities*. We conducted an in-depth analysis of the Chromium open source project (a.k.a Google Chrome). We collected code reviews, post-release vulnerabilities, version control data, and bug data over six years of the project. We conducted regression analysis to evaluate the strength of association, along with examining subgroups and various rankings of the files. We repeated our analysis across five annual releases to gauge the sensitivity of the results. We focused on the following research questions (and had the following results):

RQ1. Are source code files fixed for bugs likely to be fixed for future vulnerabilities? (We found that files with more pre-release bugs are slightly more likely to present post-release vulnerabilities.)

RQ2. Are some types of bugs more closely related to vulnerabilities than others? (Here we discovered that while of some types of pre-release bugs present a stronger association than others to post-release vulnerabilities, this relation is overall

Camilo et al.

Comparison of pre-release bugs and post-release vulnerabilities in Chromium.



COMPUTERS & SECURITY 99 (2020) 102067

Available online at www.sciencedirect.com
ScienceDirect
journal homepage: www.elsevier.com/locate/cose

Computers & Security

Investigating the vulnerability fixing process in OSS projects: Peculiarities and challenges

Gerardo Canfora, Andrea Di Sorbo, Sara Forootani, Antonio Pirozzi, Corrado Aaron Visaggio*
Department of Engineering, University of Sannio, Italy

ARTICLE INFO

Article history:
Received 10 July 2019
Revised 7 July 2020
Accepted 24 September 2020
Available online 1 October 2020

Keywords:
Security bugs
Process improvement
Software maintenance and evolution
Bug management
Empirical study

ABSTRACT

Although vulnerabilities can be considered and treated as bugs, they present numerous peculiarities compared to other types of bugs (*canonical bugs* in the remainder of the paper). A vulnerability adds functionality to a system, as it allows an adversary to misuse or abuse the system, while a canonical bug is an incomplete or incorrect implementation of a requirement, and thus degrades the functionality of the system. This difference can affect the fixing process of vulnerabilities. By mining the repositories of 6 open source projects, we characterize the differences in the fixing process between vulnerabilities and canonical bugs, highlighting critical issues which could represent challenges for future research. Results of our study demonstrate that: (i) more re-assignments (than the ones observed in canonical bugs) are required for finding the developers able to handle vulnerability-related bugs, (ii) developers' security-related skills should be profiled, to improve the efficiency of the security bug assignment tasks, and, consequently, reduce the re-assignments, and (iii) vulnerabilities require more effort, contributors and time to define the fixing strategy but smaller time to fix than canonical bugs.

1. Introduction

Vulnerabilities conceptually differ from canonical bugs: while a bug that is not related to security is a wrong or an incomplete functionality, a vulnerability represents an abuse of a functionality (Camilo et al., 2015). More specifically, vulnerabilities allow the violation of policies that can lead to an illicit or malicious use of the software (Othmane et al., 2017). In particular, the presence of vulnerabilities exposes software systems to security attacks (Joshi et al., 2015) that are often carried out through malicious software intendedly conceived (or assembled (Mercaldo et al., 2018)) to exploit such vulnerabilities. Thus, for preventing from security and privacy violations, vulnerabilities need to be promptly fixed (Russo et al., 2019). However, inefficiencies in the vulnerability management process could likely affect the vulnerability fixing performance.

While approaches and techniques for aiding developers during the fixing process of canonical bugs have been widely investigated (Zhang et al., 2015), it is not clear whether such approaches are also effective when dealing with vulnerabilities. More specifically, little is known about how vulnerabilities are discovered and resolved (Morrison et al., 2018). To fill this gap, our work is aimed at gathering knowledge on vulnerability fixing processes in order to identify (i) the peculiarities and the shortcomings that could arise in these processes, and (ii) the improvements that may be implemented to cope with the emphasized weaknesses. For this reason, this paper investigates the extent to which fixing a vulnerability differs from fixing a canonical bug, with the aim

---

* Corresponding author.

Canfora et al.

Comparison of bug and vulnerability fixing commits in six OSS projects.

Information and Software Technology 142 (2022) 106745

Contents lists available at ScienceDirect
Information and Software Technology
journal homepage: www.elsevier.com/locate/infsof

Patchworking: Exploring the code changes induced by vulnerability fixing activities

Gerardo Canfora[a], Andrea Di Sorbo[a], Sara Forootani[a,*], Matias Martinez[b], Corrado A. Visaggio[a]
[a] *Department of Engineering, University of Sannio, Italy*
[b] *Université Polytechnique Hauts-de-France, France*

ARTICLE INFO

Keywords:
Software vulnerabilities
Software maintenance
Empirical study

ABSTRACT

Context: Identifying and repairing vulnerable code is a critical software maintenance task. Change impact analysis plays an important role during software maintenance, as it helps software maintainers to figure out the potential effects of a change before it is applied. However, while the software engineering community has extensively studied techniques and tools for performing impact analysis of change requests, there are no approaches for estimating the impact when the change involves the resolution of a vulnerability bug.
Objective: We hypothesize that similar vulnerabilities may present similar strategies for patching. More specifically, our work aims at understanding whether the class of the vulnerability to fix may determine the type of impact on the system to repair.
Method: To verify our conjecture, in this paper, we examine 524 security patches applied to vulnerabilities belonging to ten different weakness categories and extracted from 98 different open-source projects written in Java.
Results: We obtain empirical evidence that vulnerabilities of the same types are often resolved by applying similar code transformations, and, thus, produce almost the same impact on the codebase.
Conclusion: On the one hand, our findings open the way to better management of software maintenance activities when dealing with software vulnerabilities. Indeed, vulnerability class information could be exploited to better predict how much code will be affected by the fixing, how the structural properties of the code (i.e., complexity, coupling, cohesion, size) will change, and the effort required for the fix. On the other hand, our results can be leveraged for improving automated strategies supporting developers when they have to deal with security flaws.

1. Introduction

Software maintenance involves critical tasks, as fixing code defects and evolving the software according to the emerging needs of users [1]. In this context, change impact analysis aims at *"identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change"* [2]. In particular, change impact analysis plays an important role in software maintenance phase, as it allows estimating and identifying (i) the effort required to implement a change [2], (ii) the software artifacts that require to be changed [3], or (iii) test cases that need to be re-executed [4], *before* applying the change itself. It also enables developers and project leaders to evaluate alternative solutions to address the change, without the need of implementing them. While techniques and tools for performing impact analysis of change requests proposed by end-users on the existing software artifacts have been extensively explored [5], to the best of the authors' knowledge, there are no approaches for estimating the impact on the system when the changes involve vulnerability fixing activities. According to the definition provided by the MITRE corporation, a security vulnerability is a *"flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components"*.[1] Each software vulnerability is caused by a weakness in the code that could have specific effects on the system

---

* Corresponding author.
E-mail addresses: canfora@unisannio.it (G. Canfora), disorbo@unisannio.it (A. Di Sorbo), forootani@unisannio.it (S. Forootani), matias.martinez@uphf.fr (M. Martinez), visaggio@unisannio.it (C.A. Visaggio).

[1] https://cve.mitre.org/about/terminology.html.
[2] https://cwe.mitre.org/.
[3] https://www.cvedetails.com/vulnerabilities-by-types.php.
[4] https://nvd.nist.gov/vuln/detail/CVE-2018-1000656.

https://doi.org/10.1016/j.infsof.2021.106745

Canfora et al.

In-depth analysis of the changes made in vulnerability fixing commits in 98 Java projects.

Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.

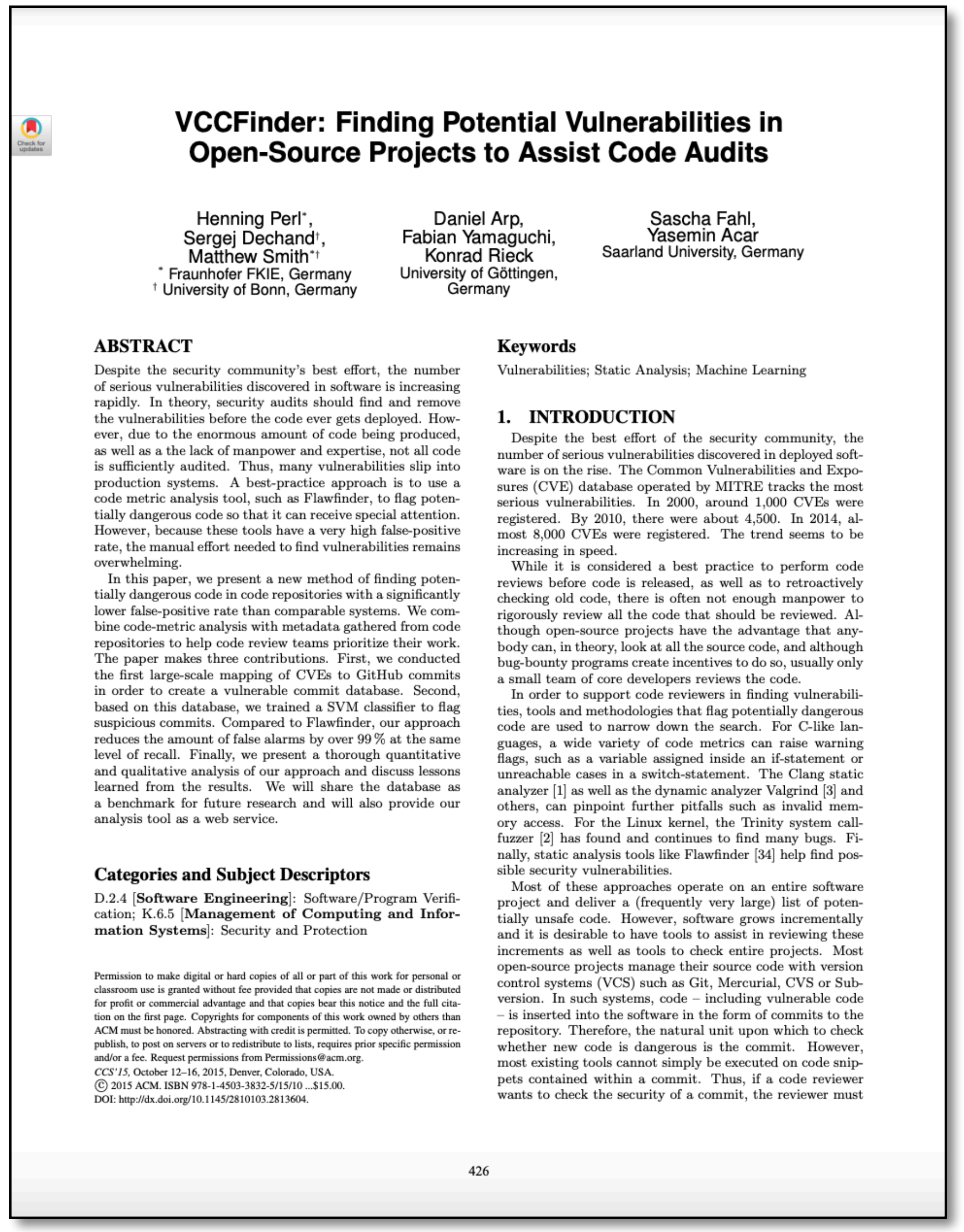
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs.
Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits

Henning Perl*, Sergej Dechand†, Matthew Smith*†
* Fraunhofer FKIE, Germany
† University of Bonn, Germany

Daniel Arp, Fabian Yamaguchi, Konrad Rieck
University of Göttingen, Germany

Sascha Fahl, Yasemin Acar
Saarland University, Germany

ABSTRACT

Despite the security community's best effort, the number of serious vulnerabilities discovered in software is increasing rapidly. In theory, security audits should find and remove the vulnerabilities before the code ever gets deployed. However, due to the enormous amount of code being produced, as well as a the lack of manpower and expertise, not all code is sufficiently audited. Thus, many vulnerabilities slip into production systems. A best-practice approach is to use a code metric analysis tool, such as Flawfinder, to flag potentially dangerous code so that it can receive special attention. However, because these tools have a very high false-positive rate, the manual effort needed to find vulnerabilities remains overwhelming.

In this paper, we present a new method of finding potentially dangerous code in code repositories with a significantly lower false-positive rate than comparable systems. We combine code-metric analysis with metadata gathered from code repositories to help code review teams prioritize their work. The paper makes three contributions. First, we conducted the first large-scale mapping of CVEs to GitHub commits in order to create a vulnerable commit database. Second, based on this database, we trained a SVM classifier to flag suspicious commits. Compared to Flawfinder, our approach reduces the amount of false alarms by over 99 % at the same level of recall. Finally, we present a thorough quantitative and qualitative analysis of our approach and discuss lessons learned from the results. We will share the database as a benchmark for future research and will also provide our analysis tool as a web service.

Categories and Subject Descriptors

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

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813604.

Keywords

Vulnerabilities; Static Analysis; Machine Learning

1. INTRODUCTION

Despite the best effort of the security community, the number of serious vulnerabilities discovered in deployed software is on the rise. The Common Vulnerabilities and Exposures (CVE) database operated by MITRE tracks the most serious vulnerabilities. In 2000, around 1,000 CVEs were registered. By 2010, there were about 4,500. In 2014, almost 8,000 CVEs were registered. The trend seems to be increasing in speed.

While it is considered a best practice to perform code reviews before code is released, as well as to retroactively checking old code, there is often not enough manpower to rigorously review all the code that should be reviewed. Although open-source projects have the advantage that anybody can, in theory, look at all the source code, and although bug-bounty programs create incentives to do so, usually only a small team of core developers reviews the code.

In order to support code reviewers in finding vulnerabilities, tools and methodologies that flag potentially dangerous code are used to narrow down the search. For C-like languages, a wide variety of code metrics can raise warning flags, such as a variable assigned inside an if-statement or unreachable cases in a switch-statement. The Clang static analyzer [1] as well as the dynamic analyzer Valgrind [3] and others, can pinpoint further pitfalls such as invalid memory access. For the Linux kernel, the Trinity system call-fuzzer [2] has found and continues to find many bugs. Finally, static analysis tools like Flawfinder [34] help find possible security vulnerabilities.

Most of these approaches operate on an entire software project and deliver a (frequently very large) list of potentially unsafe code. However, software grows incrementally and it is desirable to have tools to assist in reviewing these increments as well as tools to check entire projects. Most open-source projects manage their source code with version control systems (VCS) such as Git, Mercurial, CVS or Subversion. In such systems, code – including vulnerable code – is inserted into the software in the form of commits to the repository. Therefore, the natural unit upon which to check whether new code is dangerous is the commit. However, most existing tools cannot simply be executed on code snippets contained within a commit. Thus, if a code reviewer wants to check the security of a commit, the reviewer must

426

A modified version of the **original SZZ** but:

H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). https://doi.org/10.1145/2810103.2813604

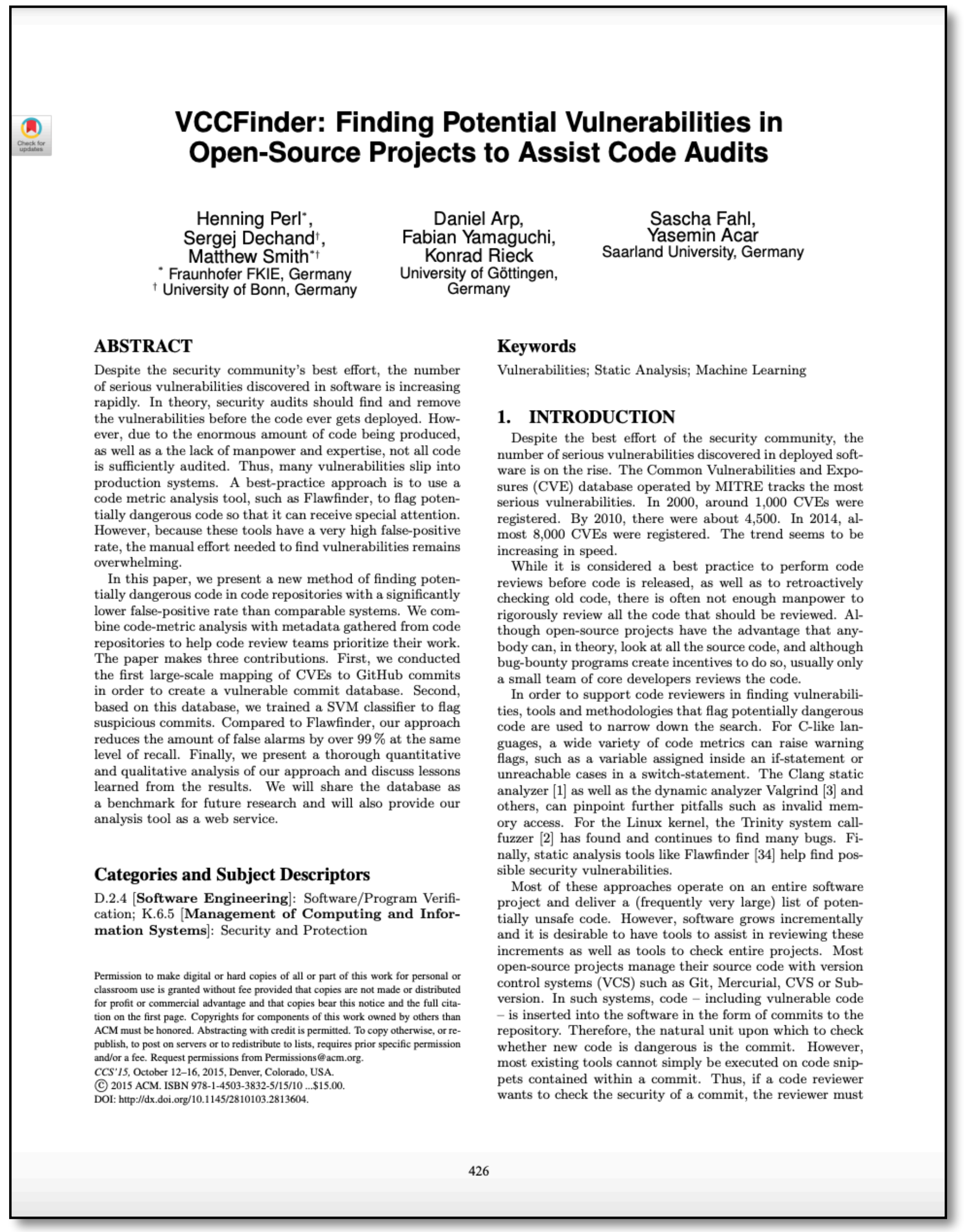
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs.
Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits

Henning Perl[*],
Sergej Dechand[†],
Matthew Smith[*†]
[*] Fraunhofer FKIE, Germany
[†] University of Bonn, Germany

Daniel Arp,
Fabian Yamaguchi,
Konrad Rieck
University of Göttingen,
Germany

Sascha Fahl,
Yasemin Acar
Saarland University, Germany

ABSTRACT

Despite the security community's best effort, the number of serious vulnerabilities discovered in software is increasing rapidly. In theory, security audits should find and remove the vulnerabilities before the code ever gets deployed. However, due to the enormous amount of code being produced, as well as a the lack of manpower and expertise, not all code is sufficiently audited. Thus, many vulnerabilities slip into production systems. A best-practice approach is to use a code metric analysis tool, such as Flawfinder, to flag potentially dangerous code so that it can receive special attention. However, because these tools have a very high false-positive rate, the manual effort needed to find vulnerabilities remains overwhelming.

In this paper, we present a new method of finding potentially dangerous code in code repositories with a significantly lower false-positive rate than comparable systems. We combine code-metric analysis with metadata gathered from code repositories to help code review teams prioritize their work. The paper makes three contributions. First, we conducted the first large-scale mapping of CVEs to GitHub commits in order to create a vulnerable commit database. Second, based on this database, we trained a SVM classifier to flag suspicious commits. Compared to Flawfinder, our approach reduces the amount of false alarms by over 99 % at the same level of recall. Finally, we present a thorough quantitative and qualitative analysis of our approach and discuss lessons learned from the results. We will share the database as a benchmark for future research and will also provide our analysis tool as a web service.

Categories and Subject Descriptors

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

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813604.

Keywords

Vulnerabilities; Static Analysis; Machine Learning

1. INTRODUCTION

Despite the best effort of the security community, the number of serious vulnerabilities discovered in deployed software is on the rise. The Common Vulnerabilities and Exposures (CVE) database operated by MITRE tracks the most serious vulnerabilities. In 2000, around 1,000 CVEs were registered. By 2010, there were about 4,500. In 2014, almost 8,000 CVEs were registered. The trend seems to be increasing in speed.

While it is considered a best practice to perform code reviews before code is released, as well as to retroactively checking old code, there is often not enough manpower to rigorously review all the code that should be reviewed. Although open-source projects have the advantage that anybody can, in theory, look at all the source code, and although bug-bounty programs create incentives to do so, usually only a small team of core developers reviews the code.

In order to support code reviewers in finding vulnerabilities, tools and methodologies that flag potentially dangerous code are used to narrow down the search. For C-like languages, a wide variety of code metrics can raise warning flags, such as a variable assigned inside an if-statement or unreachable cases in a switch-statement. The Clang static analyzer [1] as well as the dynamic analyzer Valgrind [3] and others, can pinpoint further pitfalls such as invalid memory access. For the Linux kernel, the Trinity system call-fuzzer [2] has found and continues to find many bugs. Finally, static analysis tools like Flawfinder [34] help find possible security vulnerabilities.

Most of these approaches operate on an entire software project and deliver a (frequently very large) list of potentially unsafe code. However, software grows incrementally and it is desirable to have tools to assist in reviewing these increments as well as tools to check entire projects. Most open-source projects manage their source code with version control systems (VCS) such as Git, Mercurial, CVS or Subversion. In such systems, code – including vulnerable code – is inserted into the software in the form of commits to the repository. Therefore, the natural unit upon which to check whether new code is dangerous is the commit. However, most existing tools cannot simply be executed on code snippets contained within a commit. Thus, if a code reviewer wants to check the security of a commit, the reviewer must

A modified version of the **original SZZ** but:

☛ Documentation files (e.g., README) are ignored.

H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS ’15). https://doi.org/10.1145/2810103.2813604

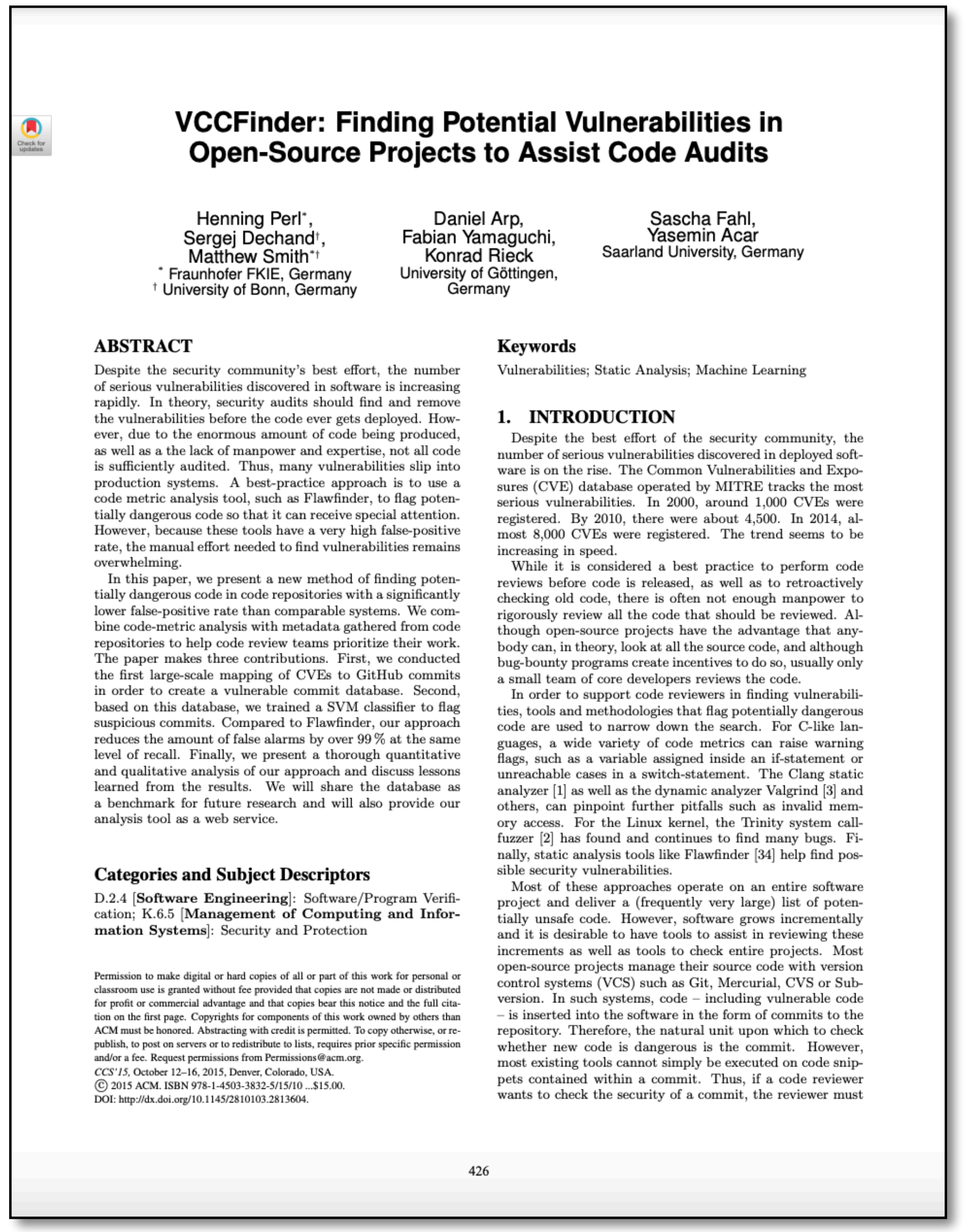
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits

Henning Perl*,
Sergej Dechand†,
Matthew Smith*†
* Fraunhofer FKIE, Germany
† University of Bonn, Germany

Daniel Arp,
Fabian Yamaguchi,
Konrad Rieck
University of Göttingen,
Germany

Sascha Fahl,
Yasemin Acar
Saarland University, Germany

ABSTRACT

Despite the security community's best effort, the number of serious vulnerabilities discovered in software is increasing rapidly. In theory, security audits should find and remove the vulnerabilities before the code ever gets deployed. However, due to the enormous amount of code being produced, as well as a the lack of manpower and expertise, not all code is sufficiently audited. Thus, many vulnerabilities slip into production systems. A best-practice approach is to use a code metric analysis tool, such as Flawfinder, to flag potentially dangerous code so that it can receive special attention. However, because these tools have a very high false-positive rate, the manual effort needed to find vulnerabilities remains overwhelming.

In this paper, we present a new method of finding potentially dangerous code in code repositories with a significantly lower false-positive rate than comparable systems. We combine code-metric analysis with metadata gathered from code repositories to help code review teams prioritize their work. The paper makes three contributions. First, we conducted the first large-scale mapping of CVEs to GitHub commits in order to create a vulnerable commit database. Second, based on this database, we trained a SVM classifier to flag suspicious commits. Compared to Flawfinder, our approach reduces the amount of false alarms by over 99 % at the same level of recall. Finally, we present a thorough quantitative and qualitative analysis of our approach and discuss lessons learned from the results. We will share the database as a benchmark for future research and will also provide our analysis tool as a web service.

Categories and Subject Descriptors

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

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813604.

Keywords

Vulnerabilities; Static Analysis; Machine Learning

1. INTRODUCTION

Despite the best effort of the security community, the number of serious vulnerabilities discovered in deployed software is on the rise. The Common Vulnerabilities and Exposures (CVE) database operated by MITRE tracks the most serious vulnerabilities. In 2000, around 1,000 CVEs were registered. By 2010, there were about 4,500. In 2014, almost 8,000 CVEs were registered. The trend seems to be increasing in speed.

While it is considered a best practice to perform code reviews before code is released, as well as to retroactively checking old code, there is often not enough manpower to rigorously review all the code that should be reviewed. Although open-source projects have the advantage that anybody can, in theory, look at all the source code, and although bug-bounty programs create incentives to do so, usually only a small team of core developers reviews the code.

In order to support code reviewers in finding vulnerabilities, tools and methodologies that flag potentially dangerous code are used to narrow down the search. For C-like languages, a wide variety of code metrics can raise warning flags, such as a variable assigned inside an if-statement or unreachable cases in a switch-statement. The Clang static analyzer [1] as well as the dynamic analyzer Valgrind [3] and others, can pinpoint further pitfalls such as invalid memory access. For the Linux kernel, the Trinity system call-fuzzer [2] has found and continues to find many bugs. Finally, static analysis tools like Flawfinder [34] help find possible security vulnerabilities.

Most of these approaches operate on an entire software project and deliver a (frequently very large) list of potentially unsafe code. However, software grows incrementally and it is desirable to have tools to assist in reviewing these increments as well as tools to check entire projects. Most open-source projects manage their source code with version control systems (VCS) such as Git, Mercurial, CVS or Subversion. In such systems, code – including vulnerable code – is inserted into the software in the form of commits to the repository. Therefore, the natural unit upon which to check whether new code is dangerous is the commit. However, most existing tools cannot simply be executed on code snippets contained within a commit. Thus, if a code reviewer wants to check the security of a commit, the reviewer must

A modified version of the **original SZZ** but:

- Documentation files (e.g., README) are ignored.
- In addition to the blames on the deleted lines, this variant also considers the blames on **the lines around the block of new lines.**

H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). https://doi.org/10.1145/2810103.2813604

Mining VCCs: Ad hoc Approaches

Okay but

Indeed, t

C-specific

SZZ by

nored.

SZZ by

nes, this

ines

V-SZZ

Let us consider a commit that fixes a vulnerability by adding this line:

Blamed ⟵

Blamed ⟵

```
int main(int argc, char* argv[]) {
    char buff[65], *temp;
    temp = argv[1] ? argv[1] : "";
    if (argc > 0 && strlen(argv[1]) > 64)
      strcpy(buff, temp);
    printf("%s", "bye");
}
```

Rationale

Some vulnerabilities are fixed by adding missing checks, e.g., an *if* added before reading from a buffer. Hence, the **context** around the new code blocks might be responsible for the vulnerability.

H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). https://doi.org/10.1145/2810103.2813604

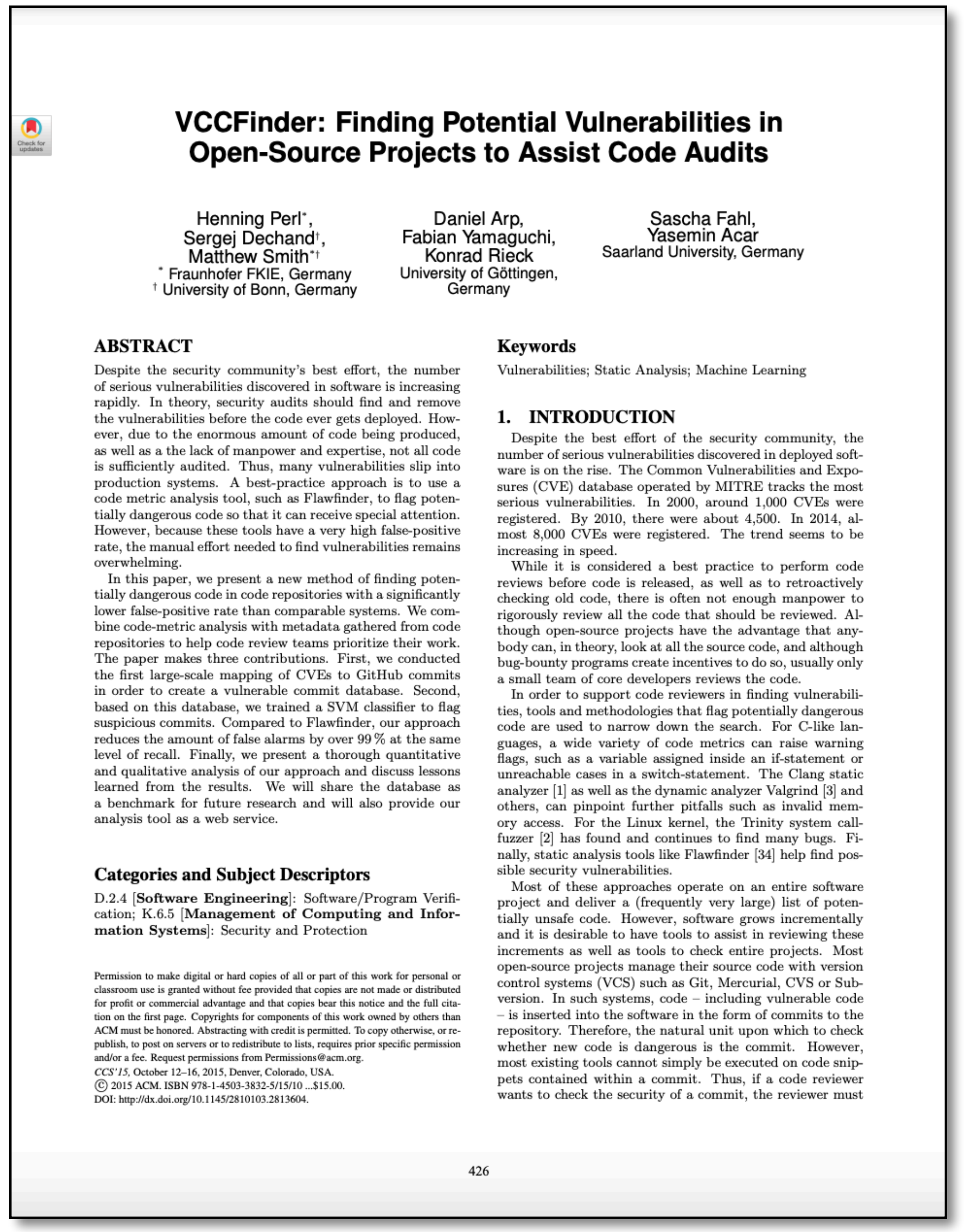
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits

Henning Perl*,
Sergej Dechand†,
Matthew Smith*†
* Fraunhofer FKIE, Germany
† University of Bonn, Germany

Daniel Arp,
Fabian Yamaguchi,
Konrad Rieck
University of Göttingen,
Germany

Sascha Fahl,
Yasemin Acar
Saarland University, Germany

ABSTRACT

Despite the security community's best effort, the number of serious vulnerabilities discovered in software is increasing rapidly. In theory, security audits should find and remove the vulnerabilities before the code ever gets deployed. However, due to the enormous amount of code being produced, as well as a the lack of manpower and expertise, not all code is sufficiently audited. Thus, many vulnerabilities slip into production systems. A best-practice approach is to use a code metric analysis tool, such as Flawfinder, to flag potentially dangerous code so that it can receive special attention. However, because these tools have a very high false-positive rate, the manual effort needed to find vulnerabilities remains overwhelming.

In this paper, we present a new method of finding potentially dangerous code in code repositories with a significantly lower false-positive rate than comparable systems. We combine code-metric analysis with metadata gathered from code repositories to help code review teams prioritize their work. The paper makes three contributions. First, we conducted the first large-scale mapping of CVEs to GitHub commits in order to create a vulnerable commit database. Second, based on this database, we trained a SVM classifier to flag suspicious commits. Compared to Flawfinder, our approach reduces the amount of false alarms by over 99 % at the same level of recall. Finally, we present a thorough quantitative and qualitative analysis of our approach and discuss lessons learned from the results. We will share the database as a benchmark for future research and will also provide our analysis tool as a web service.

Categories and Subject Descriptors

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

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813604.

Keywords

Vulnerabilities; Static Analysis; Machine Learning

1. INTRODUCTION

Despite the best effort of the security community, the number of serious vulnerabilities discovered in deployed software is on the rise. The Common Vulnerabilities and Exposures (CVE) database operated by MITRE tracks the most serious vulnerabilities. In 2000, around 1,000 CVEs were registered. By 2010, there were about 4,500. In 2014, almost 8,000 CVEs were registered. The trend seems to be increasing in speed.

While it is considered a best practice to perform code reviews before code is released, as well as to retroactively checking old code, there is often not enough manpower to rigorously review all the code that should be reviewed. Although open-source projects have the advantage that anybody can, in theory, look at all the source code, and although bug-bounty programs create incentives to do so, usually only a small team of core developers reviews the code.

In order to support code reviewers in finding vulnerabilities, tools and methodologies that flag potentially dangerous code are used to narrow down the search. For C-like languages, a wide variety of code metrics can raise warning flags, such as a variable assigned inside an if-statement or unreachable cases in a switch-statement. The Clang static analyzer [1] as well as the dynamic analyzer Valgrind [3] and others, can pinpoint further pitfalls such as invalid memory access. For the Linux kernel, the Trinity system call-fuzzer [2] has found and continues to find many bugs. Finally, static analysis tools like Flawfinder [34] help find possible security vulnerabilities.

Most of these approaches operate on an entire software project and deliver a (frequently very large) list of potentially unsafe code. However, software grows incrementally and it is desirable to have tools to assist in reviewing these increments as well as tools to check entire projects. Most open-source projects manage their source code with version control systems (VCS) such as Git, Mercurial, CVS or Subversion. In such systems, code – including vulnerable code – is inserted into the software in the form of commits to the repository. Therefore, the natural unit upon which to check whether new code is dangerous is the commit. However, most existing tools cannot simply be executed on code snippets contained within a commit. Thus, if a code reviewer wants to check the security of a commit, the reviewer must

A modified version of the **original SZZ** but:

- Documentation files (e.g., README) are ignored.
- In addition to the blames on the deleted lines, this variant also considers the blames on **the lines around the block of new lines.**
- It returns **only the most blamed commit.** In case of a tie, all the commits with the top score are returned (ex aequo).

H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). https://doi.org/10.1145/2810103.2813604

Mining VCCs: Ad hoc Approaches

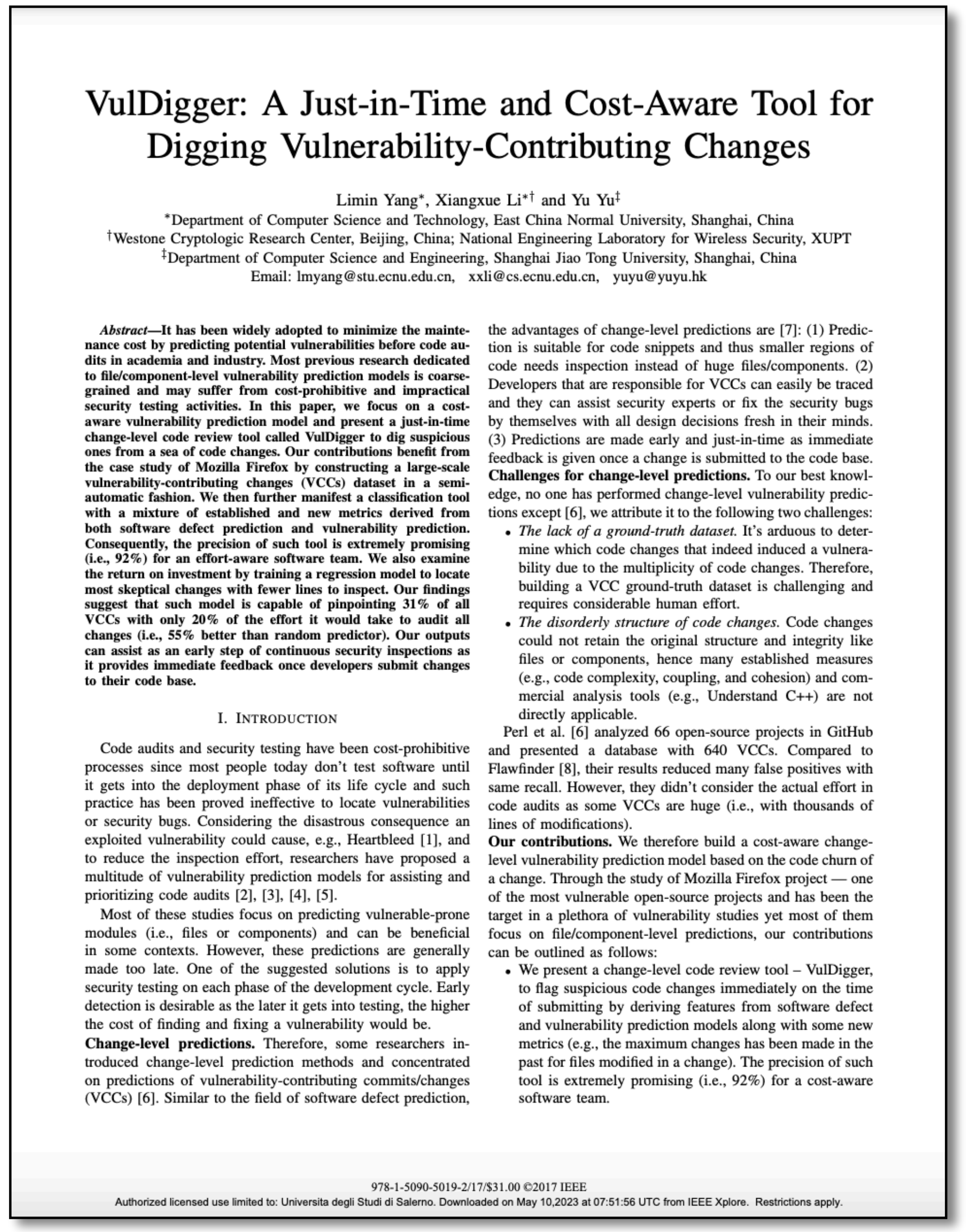
Okay but reusing the algorithms meant for bugs does not work well for VCCs.
Indeed, there are studies explaining how bugs and vulnerabilities differ.

→ **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes

Limin Yang*, Xiangxue Li*† and Yu Yu‡
*Department of Computer Science and Technology, East China Normal University, Shanghai, China
†Westone Cryptologic Research Center, Beijing, China; National Engineering Laboratory for Wireless Security, XUPT
‡Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: lmyang@stu.ecnu.edu.cn, xxli@cs.ecnu.edu.cn, yuyu@yuyu.hk

***Abstract*—It has been widely adopted to minimize the maintenance cost by predicting potential vulnerabilities before code audits in academia and industry. Most previous research dedicated to file/component-level vulnerability prediction models is coarse-grained and may suffer from cost-prohibitive and impractical security testing activities. In this paper, we focus on a cost-aware vulnerability prediction model and present a just-in-time change-level code review tool called VulDigger to dig suspicious ones from a sea of code changes. Our contributions benefit from the case study of Mozilla Firefox by constructing a large-scale vulnerability-contributing changes (VCCs) dataset in a semi-automatic fashion. We then further manifest a classification tool with a mixture of established and new metrics derived from both software defect prediction and vulnerability prediction. Consequently, the precision of such tool is extremely promising (i.e., 92%) for an effort-aware software team. We also examine the return on investment by training a regression model to locate most skeptical changes with fewer lines to inspect. Our findings suggest that such model is capable of pinpointing 31% of all VCCs with only 20% of the effort it would take to audit all changes (i.e., 55% better than random predictor). Our outputs can assist as an early step of continuous security inspections as it provides immediate feedback once developers submit changes to their code base.**

I. INTRODUCTION

Code audits and security testing have been cost-prohibitive processes since most people today don't test software until it gets into the deployment phase of its life cycle and such practice has been proved ineffective to locate vulnerabilities or security bugs. Considering the disastrous consequence an exploited vulnerability could cause, e.g., Heartbleed [1], and to reduce the inspection effort, researchers have proposed a multitude of vulnerability prediction models for assisting and prioritizing code audits [2], [3], [4], [5].

Most of these studies focus on predicting vulnerable-prone modules (i.e., files or components) and can be beneficial in some contexts. However, these predictions are generally made too late. One of the suggested solutions is to apply security testing on each phase of the development cycle. Early detection is desirable as the later it gets into testing, the higher the cost of finding and fixing a vulnerability would be.

Change-level predictions. Therefore, some researchers introduced change-level prediction methods and concentrated on predictions of vulnerability-contributing commits/changes (VCCs) [6]. Similar to the field of software defect prediction, the advantages of change-level predictions are [7]: (1) Prediction is suitable for code snippets and thus smaller regions of code needs inspection instead of huge files/components. (2) Developers that are responsible for VCCs can easily be traced and they can assist security experts or fix the security bugs by themselves with all design decisions fresh in their minds. (3) Predictions are made early and just-in-time as immediate feedback is given once a change is submitted to the code base.

Challenges for change-level predictions. To our best knowledge, no one has performed change-level vulnerability predictions except [6], we attribute it to the following two challenges:

- *The lack of a ground-truth dataset.* It's arduous to determine which code changes that indeed induced a vulnerability due to the multiplicity of code changes. Therefore, building a VCC ground-truth dataset is challenging and requires considerable human effort.
- *The disorderly structure of code changes.* Code changes could not retain the original structure and integrity like files or components, hence many established measures (e.g., code complexity, coupling, and cohesion) and commercial analysis tools (e.g., Understand C++) are not directly applicable.

Perl et al. [6] analyzed 66 open-source projects in GitHub and presented a database with 640 VCCs. Compared to Flawfinder [8], their results reduced many false positives with same recall. However, they didn't consider the actual effort in code audits as some VCCs are huge (i.e., with thousands of lines of modifications).

Our contributions. We therefore build a cost-aware change-level vulnerability prediction model based on the code churn of a change. Through the study of Mozilla Firefox project — one of the most vulnerable open-source projects and has been the target in a plethora of vulnerability studies yet most of them focus on file/component-level predictions, our contributions can be outlined as follows:

- We present a change-level code review tool – VulDigger, to flag suspicious code changes immediately on the time of submitting by deriving features from software defect and vulnerability prediction models along with some new metrics (e.g., the maximum changes has been made in the past for files modified in a change). The precision of such tool is extremely promising (i.e., 92%) for a cost-aware software team.

978-1-5090-5019-2/17/$31.00 ©2017 IEEE

A modified version of the **SZZ by Perl et al.** but:

L. Yang, X. Li and Y. Yu, "VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes," GLOBECOM 2017 - 2017 IEEE Global Communications Conference, Singapore, 2017, pp. 1-7, doi: 10.1109/GLOCOM.2017.8254428.

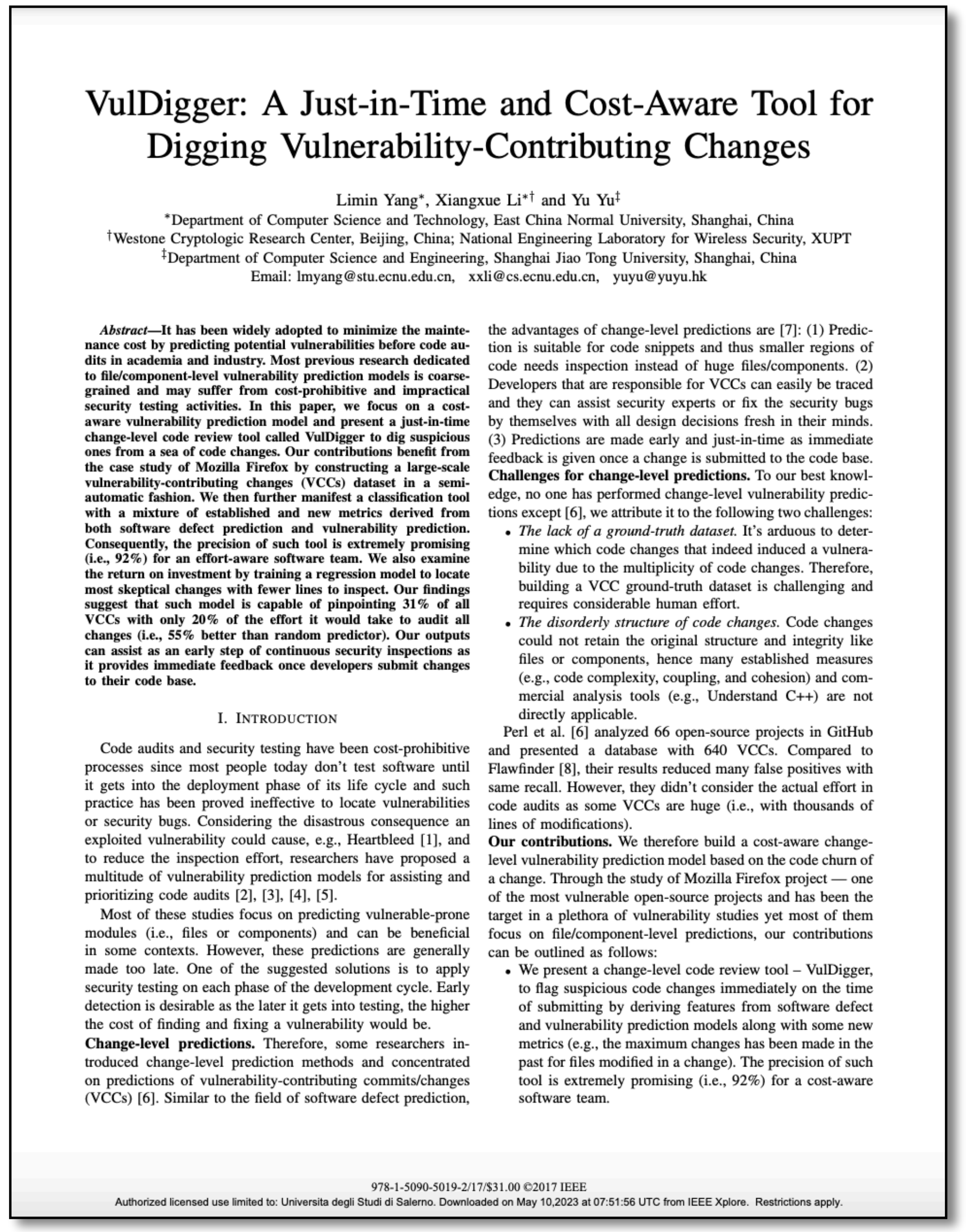
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs.
Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes

Limin Yang*, Xiangxue Li*† and Yu Yu‡
*Department of Computer Science and Technology, East China Normal University, Shanghai, China
†Westone Cryptologic Research Center, Beijing, China; National Engineering Laboratory for Wireless Security, XUPT
‡Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: lmyang@stu.ecnu.edu.cn, xxli@cs.ecnu.edu.cn, yuyu@yuyu.hk

***Abstract*—It has been widely adopted to minimize the maintenance cost by predicting potential vulnerabilities before code audits in academia and industry. Most previous research dedicated to file/component-level vulnerability prediction models is coarse-grained and may suffer from cost-prohibitive and impractical security testing activities. In this paper, we focus on a cost-aware vulnerability prediction model and present a just-in-time change-level code review tool called VulDigger to dig suspicious ones from a sea of code changes. Our contributions benefit from the case study of Mozilla Firefox by constructing a large-scale vulnerability-contributing changes (VCCs) dataset in a semi-automatic fashion. We then further manifest a classification tool with a mixture of established and new metrics derived from both software defect prediction and vulnerability prediction. Consequently, the precision of such tool is extremely promising (i.e., 92%) for an effort-aware software team. We also examine the return on investment by training a regression model to locate most skeptical changes with fewer lines to inspect. Our findings suggest that such model is capable of pinpointing 31% of all VCCs with only 20% of the effort it would take to audit all changes (i.e., 55% better than random predictor). Our outputs can assist as an early step of continuous security inspections as it provides immediate feedback once developers submit changes to their code base.**

I. INTRODUCTION

Code audits and security testing have been cost-prohibitive processes since most people today don't test software until it gets into the deployment phase of its life cycle and such practice has been proved ineffective to locate vulnerabilities or security bugs. Considering the disastrous consequence an exploited vulnerability could cause, e.g., Heartbleed [1], and to reduce the inspection effort, researchers have proposed a multitude of vulnerability prediction models for assisting and prioritizing code audits [2], [3], [4], [5].

Most of these studies focus on predicting vulnerable-prone modules (i.e., files or components) and can be beneficial in some contexts. However, these predictions are generally made too late. One of the suggested solutions is to apply security testing on each phase of the development cycle. Early detection is desirable as the later it gets into testing, the higher the cost of finding and fixing a vulnerability would be.

Change-level predictions. Therefore, some researchers introduced change-level prediction methods and concentrated on predictions of vulnerability-contributing commits/changes (VCCs) [6]. Similar to the field of software defect prediction, the advantages of change-level predictions are [7]: (1) Prediction is suitable for code snippets and thus smaller regions of code needs inspection instead of huge files/components. (2) Developers that are responsible for VCCs can easily be traced and they can assist security experts or fix the security bugs by themselves with all design decisions fresh in their minds. (3) Predictions are made early and just-in-time as immediate feedback is given once a change is submitted to the code base.

Challenges for change-level predictions. To our best knowledge, no one has performed change-level vulnerability predictions except [6], we attribute it to the following two challenges:

- *The lack of a ground-truth dataset.* It's arduous to determine which code changes that indeed induced a vulnerability due to the multiplicity of code changes. Therefore, building a VCC ground-truth dataset is challenging and requires considerable human effort.
- *The disorderly structure of code changes.* Code changes could not retain the original structure and integrity like files or components, hence many established measures (e.g., code complexity, coupling, and cohesion) and commercial analysis tools (e.g., Understand C++) are not directly applicable.

Perl et al. [6] analyzed 66 open-source projects in GitHub and presented a database with 640 VCCs. Compared to Flawfinder [8], their results reduced many false positives with same recall. However, they didn't consider the actual effort in code audits as some VCCs are huge (i.e., with thousands of lines of modifications).

Our contributions. We therefore build a cost-aware change-level vulnerability prediction model based on the code churn of a change. Through the study of Mozilla Firefox project — one of the most vulnerable open-source projects and has been the target in a plethora of vulnerability studies yet most of them focus on file/component-level predictions, our contributions can be outlined as follows:

- We present a change-level code review tool – VulDigger, to flag suspicious code changes immediately on the time of submitting by deriving features from software defect and vulnerability prediction models along with some new metrics (e.g., the maximum changes has been made in the past for files modified in a change). The precision of such tool is extremely promising (i.e., 92%) for a cost-aware software team.

978-1-5090-5019-2/17/$31.00 ©2017 IEEE

A modified version of the **SZZ by Perl et al.** but:

☛ Test and non-C/C++ files are ignored. Changes to comments, empty lines, and whitespaces are ignored as well.

L. Yang, X. Li and Y. Yu, "VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes," GLOBECOM 2017 - 2017 IEEE Global Communications Conference, Singapore, 2017, pp. 1-7, doi: 10.1109/GLOCOM.2017.8254428.

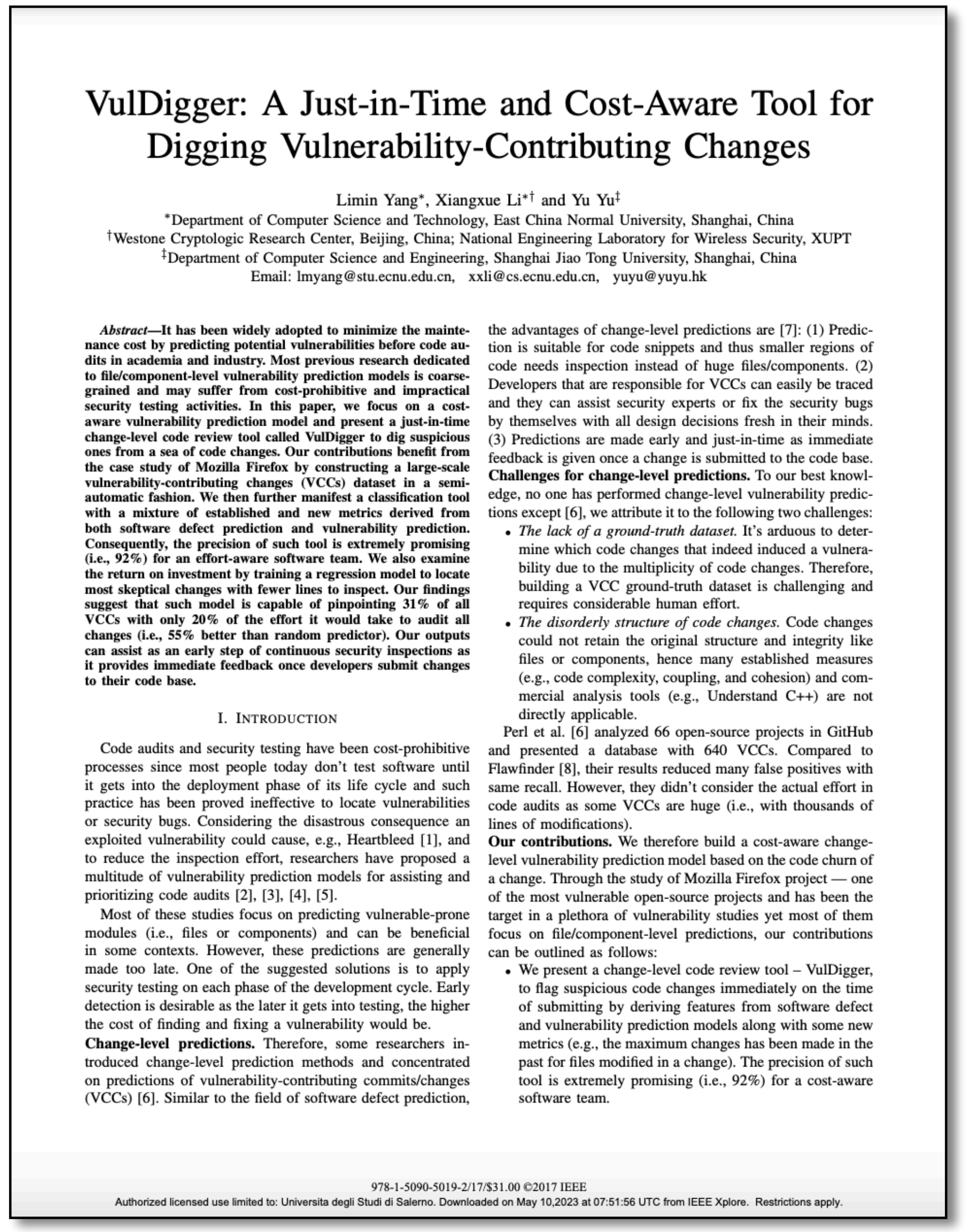
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes

Limin Yang*, Xiangxue Li*† and Yu Yu‡
*Department of Computer Science and Technology, East China Normal University, Shanghai, China
†Westone Cryptologic Research Center, Beijing, China; National Engineering Laboratory for Wireless Security, XUPT
‡Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: lmyang@stu.ecnu.edu.cn, xxli@cs.ecnu.edu.cn, yuyu@yuyu.hk

***Abstract*—It has been widely adopted to minimize the maintenance cost by predicting potential vulnerabilities before code audits in academia and industry. Most previous research dedicated to file/component-level vulnerability prediction models is coarse-grained and may suffer from cost-prohibitive and impractical security testing activities. In this paper, we focus on a cost-aware vulnerability prediction model and present a just-in-time change-level code review tool called VulDigger to dig suspicious ones from a sea of code changes. Our contributions benefit from the case study of Mozilla Firefox by constructing a large-scale vulnerability-contributing changes (VCCs) dataset in a semi-automatic fashion. We then further manifest a classification tool with a mixture of established and new metrics derived from both software defect prediction and vulnerability prediction. Consequently, the precision of such tool is extremely promising (i.e., 92%) for an effort-aware software team. We also examine the return on investment by training a regression model to locate most skeptical changes with fewer lines to inspect. Our findings suggest that such model is capable of pinpointing 31% of all VCCs with only 20% of the effort it would take to audit all changes (i.e., 55% better than random predictor). Our outputs can assist as an early step of continuous security inspections as it provides immediate feedback once developers submit changes to their code base.**

I. INTRODUCTION

Code audits and security testing have been cost-prohibitive processes since most people today don't test software until it gets into the deployment phase of its life cycle and such practice has been proved ineffective to locate vulnerabilities or security bugs. Considering the disastrous consequence an exploited vulnerability could cause, e.g., Heartbleed [1], and to reduce the inspection effort, researchers have proposed a multitude of vulnerability prediction models for assisting and prioritizing code audits [2], [3], [4], [5].

Most of these studies focus on predicting vulnerable-prone modules (i.e., files or components) and can be beneficial in some contexts. However, these predictions are generally made too late. One of the suggested solutions is to apply security testing on each phase of the development cycle. Early detection is desirable as the later it gets into testing, the higher the cost of finding and fixing a vulnerability would be.

Change-level predictions. Therefore, some researchers introduced change-level prediction methods and concentrated on predictions of vulnerability-contributing commits/changes (VCCs) [6]. Similar to the field of software defect prediction, the advantages of change-level predictions are [7]: (1) Prediction is suitable for code snippets and thus smaller regions of code needs inspection instead of huge files/components. (2) Developers that are responsible for VCCs can easily be traced and they can assist security experts or fix the security bugs by themselves with all design decisions fresh in their minds. (3) Predictions are made early and just-in-time as immediate feedback is given once a change is submitted to the code base.

Challenges for change-level predictions. To our best knowledge, no one has performed change-level vulnerability predictions except [6], we attribute it to the following two challenges:

- *The lack of a ground-truth dataset.* It's arduous to determine which code changes that indeed induced a vulnerability due to the multiplicity of code changes. Therefore, building a VCC ground-truth dataset is challenging and requires considerable human effort.
- *The disorderly structure of code changes.* Code changes could not retain the original structure and integrity like files or components, hence many established measures (e.g., code complexity, coupling, and cohesion) and commercial analysis tools (e.g., Understand C++) are not directly applicable.

Perl et al. [6] analyzed 66 open-source projects in GitHub and presented a database with 640 VCCs. Compared to Flawfinder [8], their results reduced many false positives with same recall. However, they didn't consider the actual effort in code audits as some VCCs are huge (i.e., with thousands of lines of modifications).

Our contributions. We therefore build a cost-aware change-level vulnerability prediction model based on the code churn of a change. Through the study of Mozilla Firefox project — one of the most vulnerable open-source projects and has been the target in a plethora of vulnerability studies yet most of them focus on file/component-level predictions, our contributions can be outlined as follows:

- We present a change-level code review tool – VulDigger, to flag suspicious code changes immediately on the time of submitting by deriving features from software defect and vulnerability prediction models along with some new metrics (e.g., the maximum changes has been made in the past for files modified in a change). The precision of such tool is extremely promising (i.e., 92%) for a cost-aware software team.

978-1-5090-5019-2/17/$31.00 ©2017 IEEE

A modified version of the **SZZ by Perl et al.** but:

- ☛ Test and non-C/C++ files are ignored. Changes to comments, empty lines, and whitespaces are ignored as well.
- ☛ For each new line added, the blame around this line is considered **only if it contains a C/C++ keyword or a function call**.

L. Yang, X. Li and Y. Yu, "VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes," GLOBECOM 2017 - 2017 IEEE Global Communications Conference, Singapore, 2017, pp. 1-7, doi: 10.1109/GLOCOM.2017.8254428.

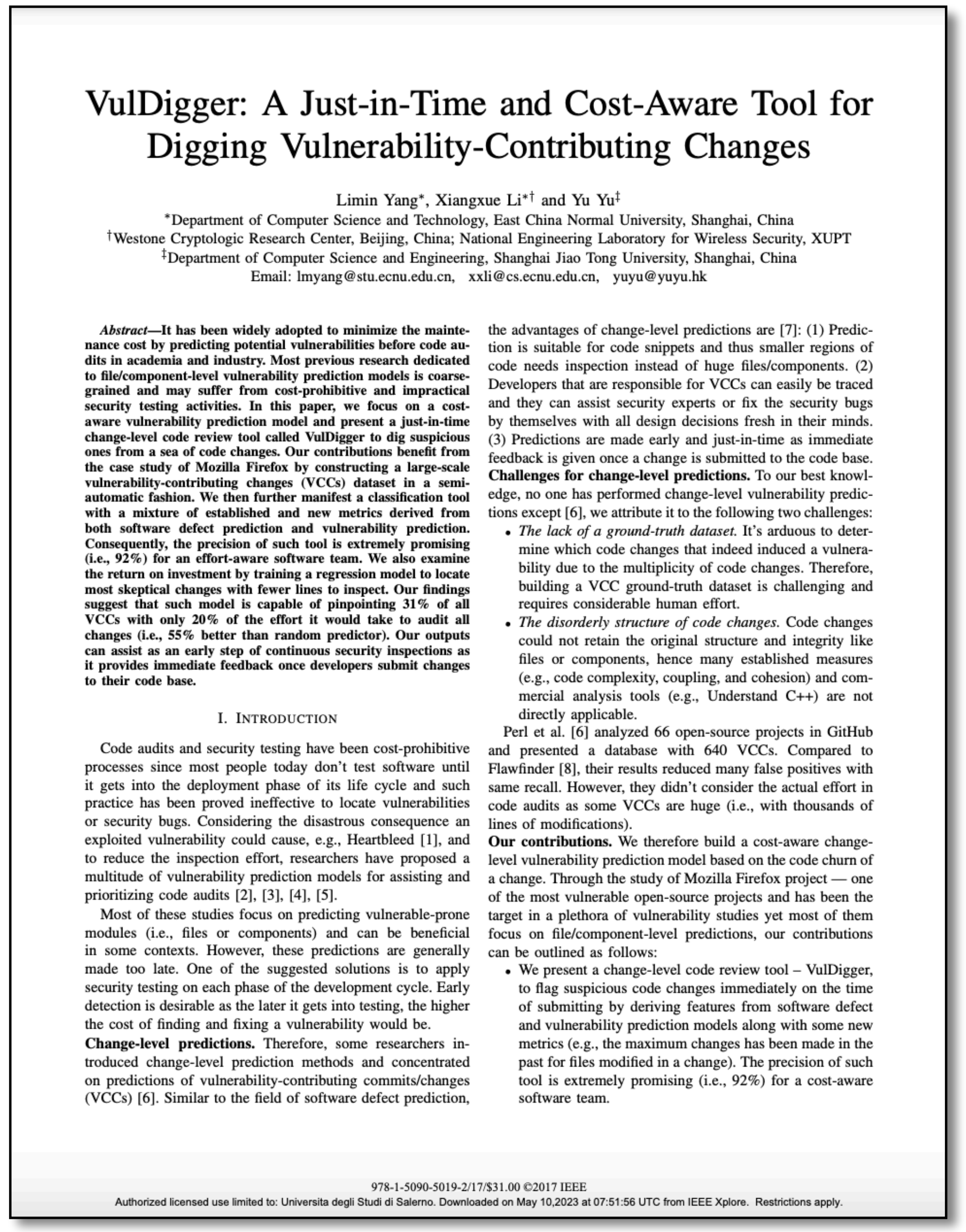
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes

Limin Yang*, Xiangxue Li*† and Yu Yu‡
*Department of Computer Science and Technology, East China Normal University, Shanghai, China
†Westone Cryptologic Research Center, Beijing, China; National Engineering Laboratory for Wireless Security, XUPT
‡Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: lmyang@stu.ecnu.edu.cn, xxli@cs.ecnu.edu.cn, yuyu@yuyu.hk

***Abstract*—It has been widely adopted to minimize the maintenance cost by predicting potential vulnerabilities before code audits in academia and industry. Most previous research dedicated to file/component-level vulnerability prediction models is coarse-grained and may suffer from cost-prohibitive and impractical security testing activities. In this paper, we focus on a cost-aware vulnerability prediction model and present a just-in-time change-level code review tool called VulDigger to dig suspicious ones from a sea of code changes. Our contributions benefit from the case study of Mozilla Firefox by constructing a large-scale vulnerability-contributing changes (VCCs) dataset in a semi-automatic fashion. We then further manifest a classification tool with a mixture of established and new metrics derived from both software defect prediction and vulnerability prediction. Consequently, the precision of such tool is extremely promising (i.e., 92%) for an effort-aware software team. We also examine the return on investment by training a regression model to locate most skeptical changes with fewer lines to inspect. Our findings suggest that such model is capable of pinpointing 31% of all VCCs with only 20% of the effort it would take to audit all changes (i.e., 55% better than random predictor). Our outputs can assist as an early step of continuous security inspections as it provides immediate feedback once developers submit changes to their code base.**

I. INTRODUCTION

Code audits and security testing have been cost-prohibitive processes since most people today don't test software until it gets into the deployment phase of its life cycle and such practice has been proved ineffective to locate vulnerabilities or security bugs. Considering the disastrous consequence an exploited vulnerability could cause, e.g., Heartbleed [1], and to reduce the inspection effort, researchers have proposed a multitude of vulnerability prediction models for assisting and prioritizing code audits [2], [3], [4], [5].

Most of these studies focus on predicting vulnerable-prone modules (i.e., files or components) and can be beneficial in some contexts. However, these predictions are generally made too late. One of the suggested solutions is to apply security testing on each phase of the development cycle. Early detection is desirable as the later it gets into testing, the higher the cost of finding and fixing a vulnerability would be.

Change-level predictions. Therefore, some researchers introduced change-level prediction methods and concentrated on predictions of vulnerability-contributing commits/changes (VCCs) [6]. Similar to the field of software defect prediction, the advantages of change-level predictions are [7]: (1) Prediction is suitable for code snippets and thus smaller regions of code needs inspection instead of huge files/components. (2) Developers that are responsible for VCCs can easily be traced and they can assist security experts or fix the security bugs by themselves with all design decisions fresh in their minds. (3) Predictions are made early and just-in-time as immediate feedback is given once a change is submitted to the code base.

Challenges for change-level predictions. To our best knowledge, no one has performed change-level vulnerability predictions except [6], we attribute it to the following two challenges:

- *The lack of a ground-truth dataset.* It's arduous to determine which code changes that indeed induced a vulnerability due to the multiplicity of code changes. Therefore, building a VCC ground-truth dataset is challenging and requires considerable human effort.
- *The disorderly structure of code changes.* Code changes could not retain the original structure and integrity like files or components, hence many established measures (e.g., code complexity, coupling, and cohesion) and commercial analysis tools (e.g., Understand C++) are not directly applicable.

Perl et al. [6] analyzed 66 open-source projects in GitHub and presented a database with 640 VCCs. Compared to Flawfinder [8], their results reduced many false positives with same recall. However, they didn't consider the actual effort in code audits as some VCCs are huge (i.e., with thousands of lines of modifications).

Our contributions. We therefore build a cost-aware change-level vulnerability prediction model based on the code churn of a change. Through the study of Mozilla Firefox project — one of the most vulnerable open-source projects and has been the target in a plethora of vulnerability studies yet most of them focus on file/component-level predictions, our contributions can be outlined as follows:

- We present a change-level code review tool – VulDigger, to flag suspicious code changes immediately on the time of submitting by deriving features from software defect and vulnerability prediction models along with some new metrics (e.g., the maximum changes has been made in the past for files modified in a change). The precision of such tool is extremely promising (i.e., 92%) for a cost-aware software team.

978-1-5090-5019-2/17/$31.00 ©2017 IEEE

A modified version of the **SZZ by Perl et al.** but:

- Test and non-C/C++ files are ignored. Changes to comments, empty lines, and whitespaces are ignored as well.
- For each new line added, the blame around this line is considered **only if it contains a C/C++ keyword or a function call**.
- Unlike the Perl et al. variant, it considers the blames around blocks of new lines **only if they do not contain new functions**.

L. Yang, X. Li and Y. Yu, "VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes," GLOBECOM 2017 - 2017 IEEE Global Communications Conference, Singapore, 2017, pp. 1-7, doi: 10.1109/GLOCOM.2017.8254428.

Mining VCCs: Ad hoc Approaches

Okay but
Indeed, t

SZZ by

SZZ by

V-SZZ

C-specific

t:

nges to
are

C/C++
he

he
they do

Let us consider a commit that fixes a vulnerability by adding this line and a new function:

Blamed ←

Blamed ←

<u>NOT blamed</u> ←

```
int main(int argc, char* argv[]) {
    char buff[65], *temp;
    temp = argv[1] ? argv[1] : "";
    if (argc > 0 && my_len(argv[1]) > 64)
      strcpy(buff, temp);
    printf("%s", "bye");
}
int my_len(char* buff) {
  return strlen(buff);
}
```

Rationale

Functions can be added anywhere in the file. Hence, the local context does not always involve meaningful parts.

L. Yang, X. Li and Y. Yu, "VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes," GLOBECOM 2017 - 2017 IEEE Global Communications Conference, Singapore, 2017, pp. 1-7, doi: 10.1109/GLOCOM.2017.8254428.

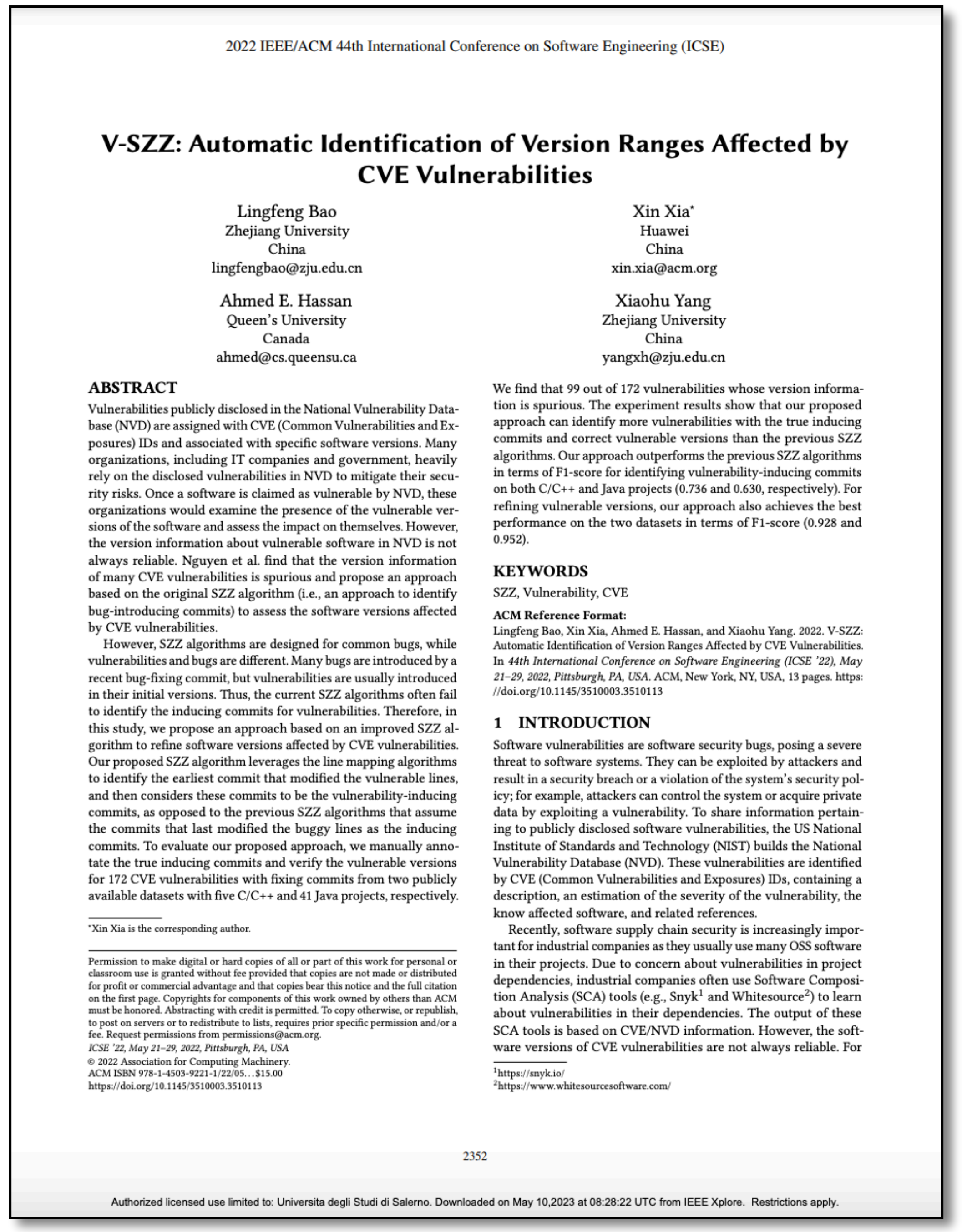
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)

V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities

Lingfeng Bao
Zhejiang University
China
lingfengbao@zju.edu.cn

Xin Xia*
Huawei
China
xin.xia@acm.org

Ahmed E. Hassan
Queen's University
Canada
ahmed@cs.queensu.ca

Xiaohu Yang
Zhejiang University
China
yangxh@zju.edu.cn

ABSTRACT

Vulnerabilities publicly disclosed in the National Vulnerability Database (NVD) are assigned with CVE (Common Vulnerabilities and Exposures) IDs and associated with specific software versions. Many organizations, including IT companies and government, heavily rely on the disclosed vulnerabilities in NVD to mitigate their security risks. Once a software is claimed as vulnerable by NVD, these organizations would examine the presence of the vulnerable versions of the software and assess the impact on themselves. However, the version information about vulnerable software in NVD is not always reliable. Nguyen et al. find that the version information of many CVE vulnerabilities is spurious and propose an approach based on the original SZZ algorithm (i.e., an approach to identify bug-introducing commits) to assess the software versions affected by CVE vulnerabilities.

However, SZZ algorithms are designed for common bugs, while vulnerabilities and bugs are different. Many bugs are introduced by a recent bug-fixing commit, but vulnerabilities are usually introduced in their initial versions. Thus, the current SZZ algorithms often fail to identify the inducing commits for vulnerabilities. Therefore, in this study, we propose an approach based on an improved SZZ algorithm to refine software versions affected by CVE vulnerabilities. Our proposed SZZ algorithm leverages the line mapping algorithms to identify the earliest commit that modified the vulnerable lines, and then considers these commits to be the vulnerability-inducing commits, as opposed to the previous SZZ algorithms that assume the commits that last modified the buggy lines as the inducing commits. To evaluate our proposed approach, we manually annotate the true inducing commits and verify the vulnerable versions for 172 CVE vulnerabilities with fixing commits from two publicly available datasets with five C/C++ and 41 Java projects, respectively. We find that 99 out of 172 vulnerabilities whose version information is spurious. The experiment results show that our proposed approach can identify more vulnerabilities with the true inducing commits and correct vulnerable versions than the previous SZZ algorithms. Our approach outperforms the previous SZZ algorithms in terms of F1-score for identifying vulnerability-inducing commits on both C/C++ and Java projects (0.736 and 0.630, respectively). For refining vulnerable versions, our approach also achieves the best performance on the two datasets in terms of F1-score (0.928 and 0.952).

KEYWORDS

SZZ, Vulnerability, CVE

ACM Reference Format:
Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510113

1 INTRODUCTION

Software vulnerabilities are software security bugs, posing a severe threat to software systems. They can be exploited by attackers and result in a security breach or a violation of the system's security policy; for example, attackers can control the system or acquire private data by exploiting a vulnerability. To share information pertaining to publicly disclosed software vulnerabilities, the US National Institute of Standards and Technology (NIST) builds the National Vulnerability Database (NVD). These vulnerabilities are identified by CVE (Common Vulnerabilities and Exposures) IDs, containing a description, an estimation of the severity of the vulnerability, the know affected software, and related references.

Recently, software supply chain security is increasingly important for industrial companies as they usually use many OSS software in their projects. Due to concern about vulnerabilities in project dependencies, industrial companies often use Software Composition Analysis (SCA) tools (e.g., Snyk[1] and Whitesource[2]) to learn about vulnerabilities in their dependencies. The output of these SCA tools is based on CVE/NVD information. However, the software versions of CVE vulnerabilities are not always reliable. For

---

*Xin Xia is the corresponding author.

[1] https://snyk.io/

[2] https://www.whitesourcesoftware.com/

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510113

A modified version of the **SZZ by Kim et al.** but:

L. Bao, X. Xia, A. E. Hassan and X. Yang, "V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 2022, pp. 2352-2364, doi: 10.1145/3510003.3510113.

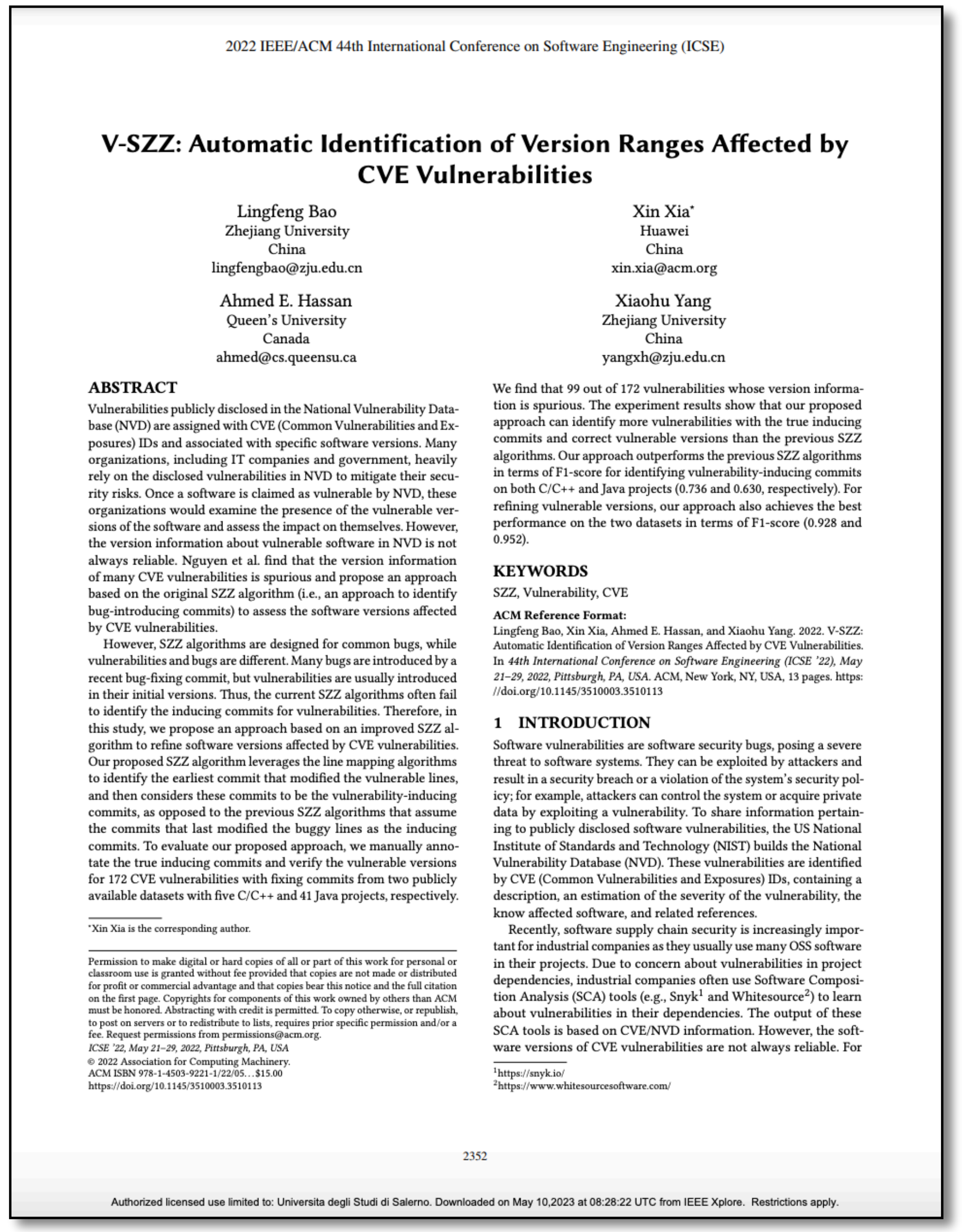
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs.
Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)

V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities

Lingfeng Bao
Zhejiang University
China
lingfengbao@zju.edu.cn

Xin Xia*
Huawei
China
xin.xia@acm.org

Ahmed E. Hassan
Queen's University
Canada
ahmed@cs.queensu.ca

Xiaohu Yang
Zhejiang University
China
yangxh@zju.edu.cn

ABSTRACT

Vulnerabilities publicly disclosed in the National Vulnerability Database (NVD) are assigned with CVE (Common Vulnerabilities and Exposures) IDs and associated with specific software versions. Many organizations, including IT companies and government, heavily rely on the disclosed vulnerabilities in NVD to mitigate their security risks. Once a software is claimed as vulnerable by NVD, these organizations would examine the presence of the vulnerable versions of the software and assess the impact on themselves. However, the version information about vulnerable software in NVD is not always reliable. Nguyen et al. find that the version information of many CVE vulnerabilities is spurious and propose an approach based on the original SZZ algorithm (i.e., an approach to identify bug-introducing commits) to assess the software versions affected by CVE vulnerabilities.

However, SZZ algorithms are designed for common bugs, while vulnerabilities and bugs are different. Many bugs are introduced by a recent bug-fixing commit, but vulnerabilities are usually introduced in their initial versions. Thus, the current SZZ algorithms often fail to identify the inducing commits for vulnerabilities. Therefore, in this study, we propose an approach based on an improved SZZ algorithm to refine software versions affected by CVE vulnerabilities. Our proposed SZZ algorithm leverages the line mapping algorithms to identify the earliest commit that modified the vulnerable lines, and then considers these commits to be the vulnerability-inducing commits, as opposed to the previous SZZ algorithms that assume the commits that last modified the buggy lines as the inducing commits. To evaluate our proposed approach, we manually annotate the true inducing commits and verify the vulnerable versions for 172 CVE vulnerabilities with fixing commits from two publicly available datasets with five C/C++ and 41 Java projects, respectively. We find that 99 out of 172 vulnerabilities whose version information is spurious. The experiment results show that our proposed approach can identify more vulnerabilities with the true inducing commits and correct vulnerable versions than the previous SZZ algorithms. Our approach outperforms the previous SZZ algorithms in terms of F1-score for identifying vulnerability-inducing commits on both C/C++ and Java projects (0.736 and 0.630, respectively). For refining vulnerable versions, our approach also achieves the best performance on the two datasets in terms of F1-score (0.928 and 0.952).

KEYWORDS

SZZ, Vulnerability, CVE

ACM Reference Format:
Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510113

1 INTRODUCTION

Software vulnerabilities are software security bugs, posing a severe threat to software systems. They can be exploited by attackers and result in a security breach or a violation of the system's security policy; for example, attackers can control the system or acquire private data by exploiting a vulnerability. To share information pertaining to publicly disclosed software vulnerabilities, the US National Institute of Standards and Technology (NIST) builds the National Vulnerability Database (NVD). These vulnerabilities are identified by CVE (Common Vulnerabilities and Exposures) IDs, containing a description, an estimation of the severity of the vulnerability, the know affected software, and related references.

Recently, software supply chain security is increasingly important for industrial companies as they usually use many OSS software in their projects. Due to concern about vulnerabilities in project dependencies, industrial companies often use Software Composition Analysis (SCA) tools (e.g., Snyk[1] and Whitesource[2]) to learn about vulnerabilities in their dependencies. The output of these SCA tools is based on CVE/NVD information. However, the software versions of CVE vulnerabilities are not always reliable. For

---

*Xin Xia is the corresponding author.

[1] https://snyk.io/

[2] https://www.whitesourcesoftware.com/

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510113

A modified version of the **SZZ by Kim et al.** but:

☛ The git blame is **repeated beyond format changes** until reaching the commits that created the blamed lines. This approach is supported by both AST and string similarity matching.

L. Bao, X. Xia, A. E. Hassan and X. Yang, "V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 2022, pp. 2352-2364, doi: 10.1145/3510003.3510113.

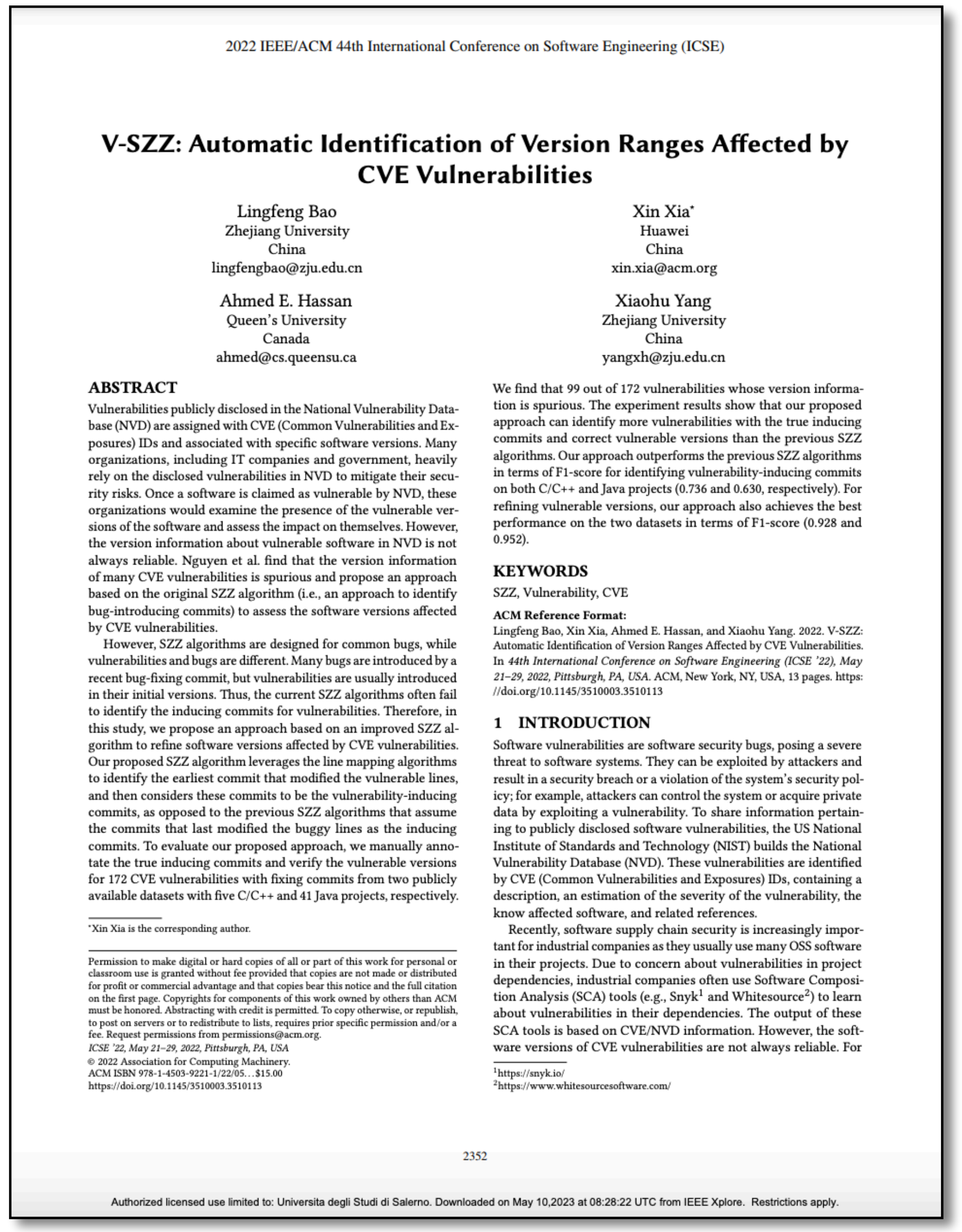
Mining VCCs: Ad hoc Approaches

Okay but reusing the algorithms meant for bugs does not work well for VCCs. Indeed, there are studies explaining how bugs and vulnerabilities differ. → **We need other VCC-specific techniques!**

SZZ by Perl et al.

SZZ by Yang et al.

V-SZZ by Bao et al.



2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)

V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities

Lingfeng Bao
Zhejiang University
China
lingfengbao@zju.edu.cn

Xin Xia*
Huawei
China
xin.xia@acm.org

Ahmed E. Hassan
Queen's University
Canada
ahmed@cs.queensu.ca

Xiaohu Yang
Zhejiang University
China
yangxh@zju.edu.cn

ABSTRACT

Vulnerabilities publicly disclosed in the National Vulnerability Database (NVD) are assigned with CVE (Common Vulnerabilities and Exposures) IDs and associated with specific software versions. Many organizations, including IT companies and government, heavily rely on the disclosed vulnerabilities in NVD to mitigate their security risks. Once a software is claimed as vulnerable by NVD, these organizations would examine the presence of the vulnerable versions of the software and assess the impact on themselves. However, the version information about vulnerable software in NVD is not always reliable. Nguyen et al. find that the version information of many CVE vulnerabilities is spurious and propose an approach based on the original SZZ algorithm (i.e., an approach to identify bug-introducing commits) to assess the software versions affected by CVE vulnerabilities.

However, SZZ algorithms are designed for common bugs, while vulnerabilities and bugs are different. Many bugs are introduced by a recent bug-fixing commit, but vulnerabilities are usually introduced in their initial versions. Thus, the current SZZ algorithms often fail to identify the inducing commits for vulnerabilities. Therefore, in this study, we propose an approach based on an improved SZZ algorithm to refine software versions affected by CVE vulnerabilities. Our proposed SZZ algorithm leverages the line mapping algorithms to identify the earliest commit that modified the vulnerable lines, and then considers these commits to be the vulnerability-inducing commits, as opposed to the previous SZZ algorithms that assume the commits that last modified the buggy lines as the inducing commits. To evaluate our proposed approach, we manually annotate the true inducing commits and verify the vulnerable versions for 172 CVE vulnerabilities with fixing commits from two publicly available datasets with five C/C++ and 41 Java projects, respectively. We find that 99 out of 172 vulnerabilities whose version information is spurious. The experiment results show that our proposed approach can identify more vulnerabilities with the true inducing commits and correct vulnerable versions than the previous SZZ algorithms. Our approach outperforms the previous SZZ algorithms in terms of F1-score for identifying vulnerability-inducing commits on both C/C++ and Java projects (0.736 and 0.630, respectively). For refining vulnerable versions, our approach also achieves the best performance on the two datasets in terms of F1-score (0.928 and 0.952).

KEYWORDS

SZZ, Vulnerability, CVE

ACM Reference Format:
Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510113

1 INTRODUCTION

Software vulnerabilities are software security bugs, posing a severe threat to software systems. They can be exploited by attackers and result in a security breach or a violation of the system's security policy; for example, attackers can control the system or acquire private data by exploiting a vulnerability. To share information pertaining to publicly disclosed software vulnerabilities, the US National Institute of Standards and Technology (NIST) builds the National Vulnerability Database (NVD). These vulnerabilities are identified by CVE (Common Vulnerabilities and Exposures) IDs, containing a description, an estimation of the severity of the vulnerability, the know affected software, and related references.

Recently, software supply chain security is increasingly important for industrial companies as they usually use many OSS software in their projects. Due to concern about vulnerabilities in project dependencies, industrial companies often use Software Composition Analysis (SCA) tools (e.g., Snyk[1] and Whitesource[2]) to learn about vulnerabilities in their dependencies. The output of these SCA tools is based on CVE/NVD information. However, the software versions of CVE vulnerabilities are not always reliable. For

---

*Xin Xia is the corresponding author.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510113

[1] https://snyk.io/
[2] https://www.whitesourcesoftware.com/

2352

A modified version of the **SZZ by Kim et al.** but:

👉 The git blame is **repeated beyond format changes** until reaching the commits that created the blamed lines. This approach is supported by both AST and string similarity matching.

Rationale

According to certain studies, many vulnerabilities are *foundational*, i.e., introduced early in the project, even before the first release.

L. Bao, X. Xia, A. E. Hassan and X. Yang, "V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 2022, pp. 2352-2364, doi: 10.1145/3510003.3510113.

Are we sure they work?

Performance Indicators

How can we be sure VCC mining algorithms work as expected? We want our algorithm to minimize:

Performance Indicators

How can we be sure VCC mining algorithms work as expected? We want our algorithm to minimize:

False Positives The algorithm returned a commit that was not a real VCC.

False Negatives The algorithm did not return one (or more) real VCC.

Performance Indicators

How can we be sure VCC mining algorithms work as expected? We want our algorithm to minimize:

False Positives The algorithm returned a commit that was not a real VCC.

False Negatives The algorithm did not return one (or more) real VCC.

From the *Information Retrieval* world, we commonly use these metrics to evaluate such approaches:

Performance Indicators

How can we be sure VCC mining algorithms work as expected? We want our algorithm to minimize:

False Positives The algorithm returned a commit that was not a real VCC.

False Negatives The algorithm did not return one (or more) real VCC.

From the *Information Retrieval* world, we commonly use these metrics to evaluate such approaches:

$$\boxed{Precision = \frac{|correct \cap identified|}{|identified|}}$$

“Among the found VCCs, how many are correct?”

Performance Indicators

How can we be sure VCC mining algorithms work as expected? We want our algorithm to minimize:

False Positives The algorithm returned a commit that was not a real VCC.

False Negatives The algorithm did not return one (or more) real VCC.

From the *Information Retrieval* world, we commonly use these metrics to evaluate such approaches:

$$Precision = \frac{|correct \cap identified|}{|identified|}$$

“Among the found VCCs, how many are correct?”

$$Recall = \frac{|correct \cap identified|}{|correct|}$$

“Among the correct VCCs, how many did I find?”

Performance Indicators

How can we be sure VCC mining algorithms work as expected? We want our algorithm to minimize:

False Positives The algorithm returned a commit that was not a real VCC.

False Negatives The algorithm did not return one (or more) real VCC.

From the *Information Retrieval* world, we commonly use these metrics to evaluate such approaches:

$$Precision = \frac{|correct \cap identified|}{|identified|}$$

“Among the found VCCs, how many are correct?”

$$Recall = \frac{|correct \cap identified|}{|correct|}$$

“Among the correct VCCs, how many did I find?”

$$F - measure = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

“Trade-off between precision and recall”

Performance Indicators

How can we be sure VCC mining algorithms work as expected? We want our algorithm to minimize:

False Positives The algorithm returned a commit that was not a real VCC.

False Negatives The algorithm did not return one (or more) real VCC.

From the *Information Retrieval* world, we commonly use these metrics to evaluate such approaches:

$$Precision = \frac{|correct \cap identified|}{|identified|}$$

$$Recall = \frac{|correct \cap identified|}{|correct|}$$

$$F - measure = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

But how do we determine this “correct” set?

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

- **Exhaustive Labeling**
- **Bisect-driven Labeling**
- **Precision Assessment**
- **Developer-informed Oracle**

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

- **Exhaustive Labeling**
- **Bisect-driven Labeling**
- **Precision Assessment**
- **Developer-informed Oracle**

For each vulnerability, we manually inspect all the commits in the project and assess whether it is a VCC. Complete but time-consuming.

Recommended when… we want to be exhaustive (!) or just want to analyze a few vulnerabilities.

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

- Exhaustive Labeling
- **Bisect-driven Labeling**
- Precision Assessment
- Developer-informed Oracle

For each vulnerability, we run git bisect until we find at least one VCC. Inspired by the Meneely et al. mining technique. Less complete but faster, reducing the workload by a logarithmic factor.

Recommended when… we don’t need a complete *correct* set, and we have time to inspect.

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

Exhaustive Labeling

Bisect-driven Labeling

Precision Assessment

Developer-informed Oracle

For each commit flagged as VCC by the algorithm, we inspect it to assess whether it is a real VCC. This will not produce the *correct* set, but only *correct* ∩ *identified*. Hence, we are not aware of the “missed” VCCs.

Recommended when… we are only interested in assessing the precision.

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

- Exhaustive Labeling
- Bisect-driven Labeling
- Precision Assessment
- **Developer-informed Oracle**

For each vulnerability, we process the fixing commit message to retrieve mentions of the culprit commit(s). Developers sometimes explicitly indicate the commit where the vulnerability was introduced. This method has a fully automated part based on NLP/text mining and an (optional) manual assessment part.

Recommended when… we don’t need a complete *correct* set and, we want developers’ experience.

G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In Proceedings of the 43rd International Conference on Software Engineering (ICSE '21). IEEE Press, 436–447. https://doi.org/10.1109/ICSE43902.2021.00049

Building the Ground Truth

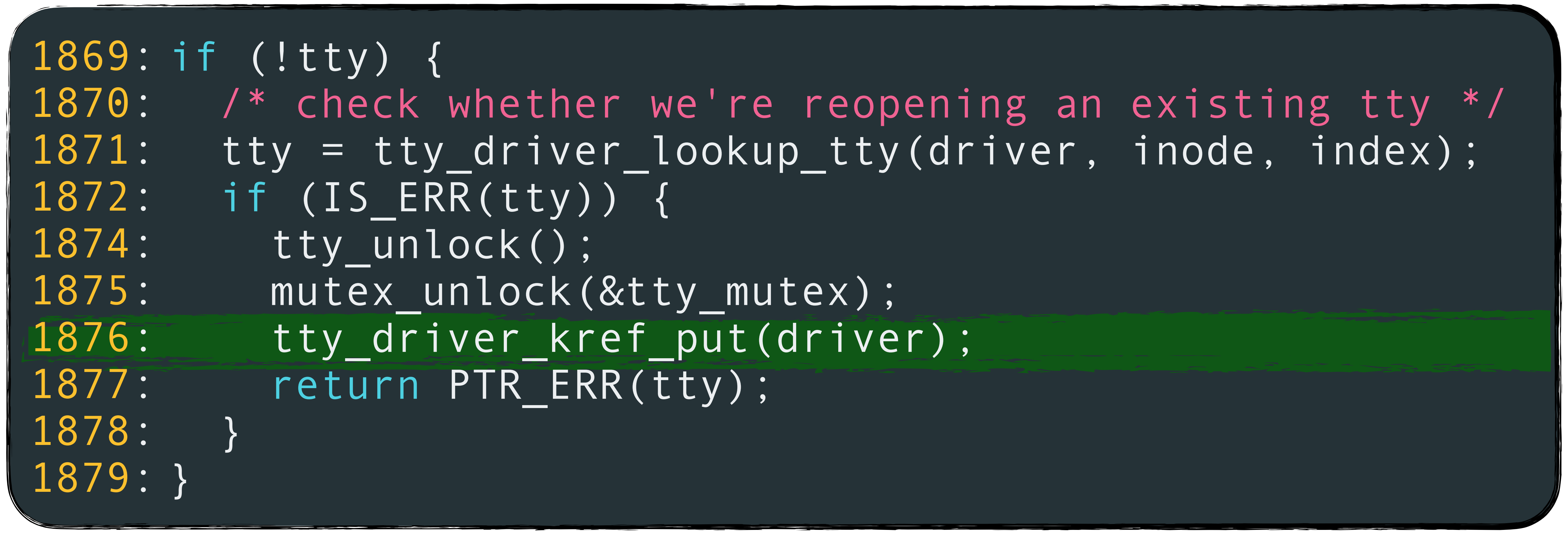
We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, <u>a dataset of true VCCs and non-VCCs</u>. We can employ some methods:

- Exhaustive Labeling
- Bisect-driven Labeling
- Precision Assessment
- **Developer-informed Oracle**

EXAMPLE

CVE-2011-5321 (NULL pointer dereference) in Linux Kernel was fixed in commit *c290f835* by just adding a single line of code.

> *TTY: drop driver reference in tty_open fail path*
> *When tty_driver_lookup_tty fails in tty_open, we forget to drop a reference to the tty driver. This was added by commit 4a2b5fd (Move tty lookup/reopen to caller). […]*



```
if (!tty) {
  /* check whether we're reopening an existing tty */
  tty = tty_driver_lookup_tty(driver, inode, index);
  if (IS_ERR(tty)) {
    tty_unlock();
    mutex_unlock(&tty_mutex);
    tty_driver_kref_put(driver);
    return PTR_ERR(tty);
  }
}
```

G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In Proceedings of the 43rd International Conference on Software Engineering (ICSE '21). IEEE Press, 436–447. https://doi.org/10.1109/ICSE43902.2021.00049

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the “standard” for evaluating the algorithms. In other words, <u>a dataset of true VCCs and non-VCCs</u>. We can employ some methods:

- Exhaustive Labeling
- Bisect-driven Labeling
- Precision Assessment
- **Developer-informed Oracle**

EXAMPLE

CVE-2011-5321 (NULL pointer dereference) in Linux Kernel was fixed in commit *c290f835* by just adding a single line of code.

> “
> *TTY: drop driver reference in tty_open fail path*
> *When tty_driver_lookup_tty fails in tty_open, we forget to drop a reference to the tty driver. This was added by commit 4a2b5fd (Move tty lookup/reopen to caller). […]*

According to the developer who fixed this vulnerability, this is a VCC (which involuntarily introduced the vulnerability while refactoring some code)**.**

G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In Proceedings of the 43rd International Conference on Software Engineering (ICSE '21). IEEE Press, 436–447. https://doi.org/10.1109/ICSE43902.2021.00049

How can I use them?

Available Tools

OpenSZZ

Command-line tool written in Java implementing the **standard SZZ**, analyzing GitHub repositories and Jira issues.

SZZUnleashed

Collection of Python and Java scripts implementing the **SZZ by Williams and Spacco** (not seen).

PyDriller

Python library for repository mining, including an implementation of **SZZ by Kim et al.**

Archeogit

Command-line tool written in Python implementing the **SZZ by Perl et al.**

V-SZZ

Collection of Python scripts replicating **V-SZZ by Bao et al.**

PySZZ

Collection of Python scripts implementing **several SZZ variants** with a uniform interface.

Isn't there something ready to use?

Available Datasets

Curated

Available Datasets

Curated

Vulnerability History Project

Database of curated histories of 2,677 vulnerabilities of eight open-source projects. Built by **class assignments in a Master's degree course held at RIT.**

Available Datasets

Curated

Vulnerability History Project	Database of curated histories of 2,677 vulnerabilities of eight open-source projects. Built by **class assignments in a Master's degree course held at RIT.**
Java VCC Dataset	Dataset of 100 VCCs of 71 known vulnerabilities affecting popular Java projects. Built by **manually analyzing the history aided by blames on fixing commits**.

Available Datasets

Curated

Vulnerability History Project	Database of curated histories of 2,677 vulnerabilities of eight open-source projects. Built by **class assignments in a Master's degree course held at RIT.**
Java VCC Dataset	Dataset of 100 VCCs of 71 known vulnerabilities affecting popular Java projects. Built by **manually analyzing the history aided by blames on fixing commits**.

Mined

Available Datasets

Curated

Vulnerability History Project	Database of curated histories of 2,677 vulnerabilities of eight open-source projects. Built by **class assignments in a Master's degree course held at RIT.**
Java VCC Dataset	Dataset of 100 VCCs of 71 known vulnerabilities affecting popular Java projects. Built by **manually analyzing the history aided by blames on fixing commits**.

Mined

Secret Life Dataset	Dataset of 12,256 VCCs of 3,663 vulnerabilities affecting 1,096 open-source projects. Built by **running an SZZ variant by Iannone et al.** (not seen).

Available Datasets

Curated

Vulnerability History Project	Database of curated histories of 2,677 vulnerabilities of eight open-source projects. Built by **class assignments in a Master's degree course held at RIT.**
Java VCC Dataset	Dataset of 100 VCCs of 71 known vulnerabilities affecting popular Java projects. Built by **manually analyzing the history aided by blames on fixing commits**.

Mined

Secret Life Dataset	Dataset of 12,256 VCCs of 3,663 vulnerabilities affecting 1,096 open-source projects. Built by **running an SZZ variant by Iannone et al.** (not seen).
FrontEndART Dataset	Dataset of ~700 VCCs of 564 vulnerabilities affecting 198 Java projects. Built by **running an SZZ variant by Aladics et al.**

Available Datasets

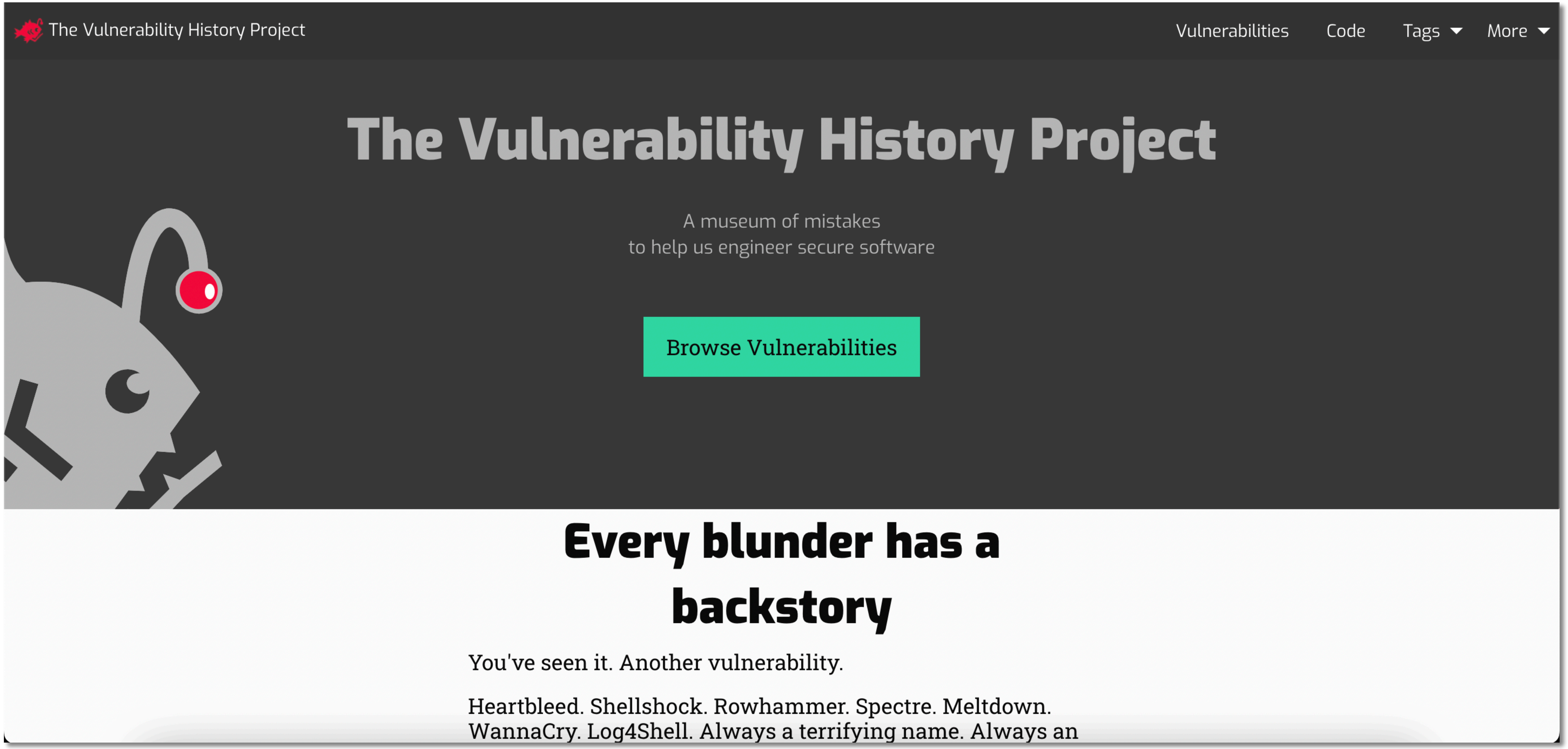
Curated

Vulnerability History Project	Database of curated histories of 2,677 vulnerabilities of eight open-source projects. Built by **class assignments in a Master's degree course held at RIT.**
Java VCC Dataset	Dataset of 100 VCCs of 71 known vulnerabilities affecting popular Java projects. Built by **manually analyzing the history aided by blames on fixing commits.**

Secret Life Dataset	Dataset of 12,256 VCCs of 3,663 vulnerabilities affecting 1,096 open-source projects. Built by **running an SZZ variant by Iannone et al.** (not seen).
FrontEndART Dataset	Dataset of ~700 VCCs of 564 vulnerabilities affecting 198 Java projects. Built by **running an SZZ variant by Aladics et al.**

Available Datasets

Vulnerability History Project



The Vulnerability History Project

A museum of mistakes
to help us engineer secure software

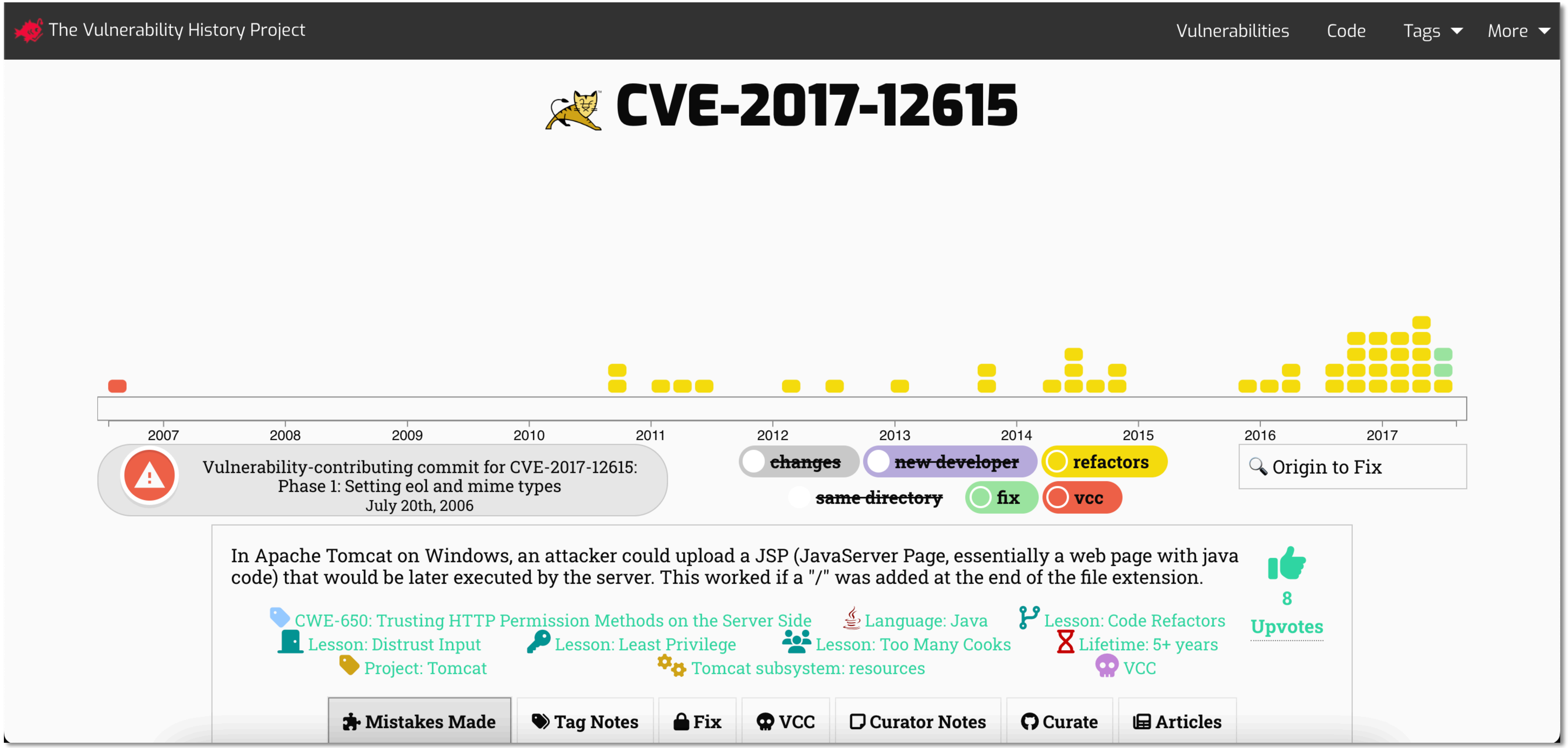
Every blunder has a backstory

You've seen it. Another vulnerability.

Heartbleed. Shellshock. Rowhammer. Spectre. Meltdown. WannaCry. Log4Shell. Always a terrifying name. Always an

Available Datasets

Vulnerability History Project



CVE-2017-12615

Vulnerability-contributing commit for CVE-2017-12615:
Phase 1: Setting eol and mime types
July 20th, 2006

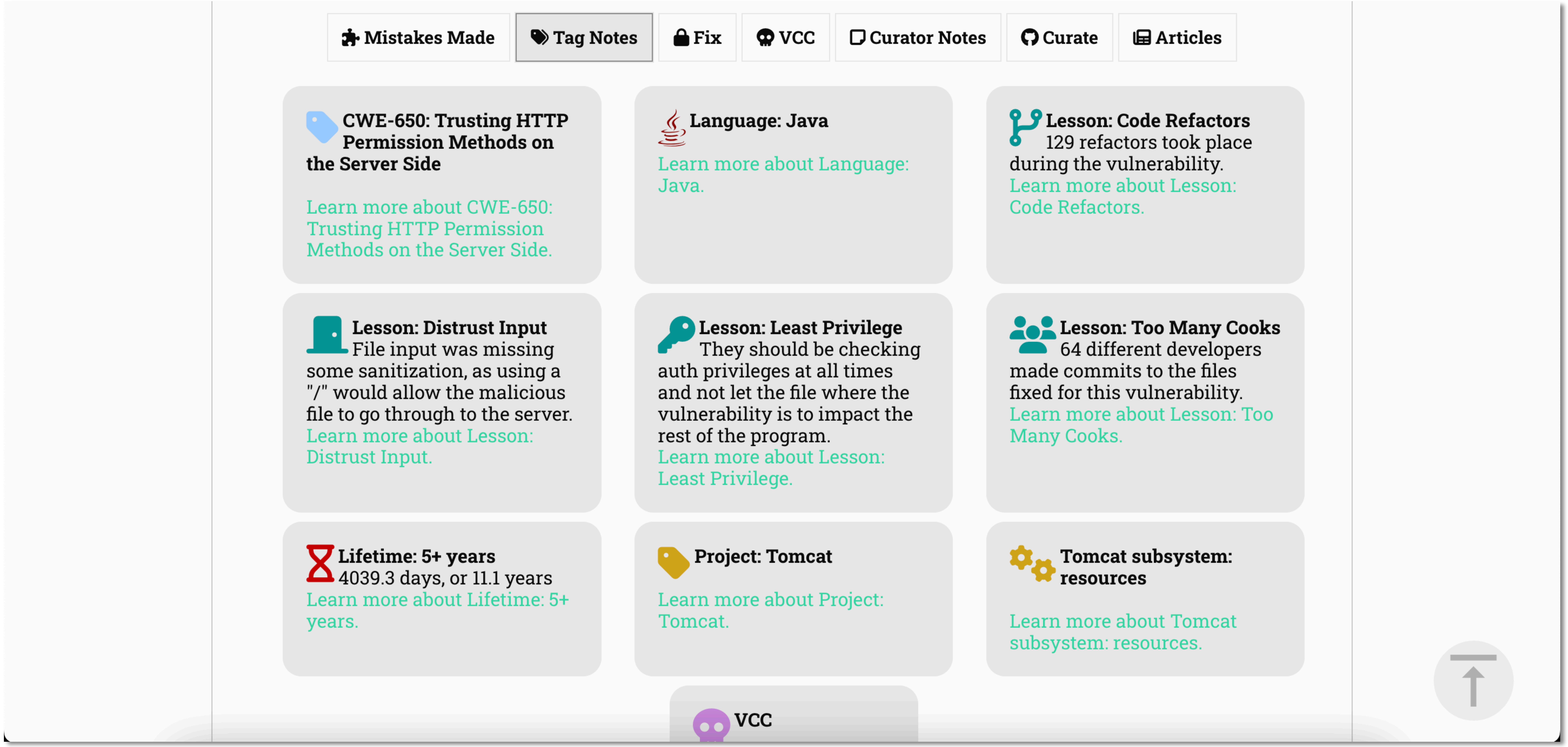
In Apache Tomcat on Windows, an attacker could upload a JSP (JavaServer Page, essentially a web page with java code) that would be later executed by the server. This worked if a "/" was added at the end of the file extension.

8 Upvotes

CWE-650: Trusting HTTP Permission Methods on the Server Side
Language: Java
Lesson: Code Refactors
Lesson: Distrust Input
Lesson: Least Privilege
Lesson: Too Many Cooks
Lifetime: 5+ years
Project: Tomcat
Tomcat subsystem: resources
VCC

Available Datasets

Vulnerability History Project



CWE-650: Trusting HTTP Permission Methods on the Server Side

Learn more about CWE-650: Trusting HTTP Permission Methods on the Server Side.

Language: Java

Learn more about Language: Java.

Lesson: Code Refactors 129 refactors took place during the vulnerability. Learn more about Lesson: Code Refactors.

Lesson: Distrust Input File input was missing some sanitization, as using a "/" would allow the malicious file to go through to the server. Learn more about Lesson: Distrust Input.

Lesson: Least Privilege They should be checking auth privileges at all times and not let the file where the vulnerability is to impact the rest of the program. Learn more about Lesson: Least Privilege.

Lesson: Too Many Cooks 64 different developers made commits to the files fixed for this vulnerability. Learn more about Lesson: Too Many Cooks.

Lifetime: 5+ years 4039.3 days, or 11.1 years Learn more about Lifetime: 5+ years.

Project: Tomcat

Learn more about Project: Tomcat.

Tomcat subsystem: resources

Learn more about Tomcat subsystem: resources.

VCC

Built by running a SZZ variant by Aladics et al.

Available Datasets

Vulnerability History Project

VHP can be mined in several ways

RESTFul API

Retrieving data with simple HTTP requests.

Raw Data

The list of vulnerabilities is available in a repository of its organization in GitHub.

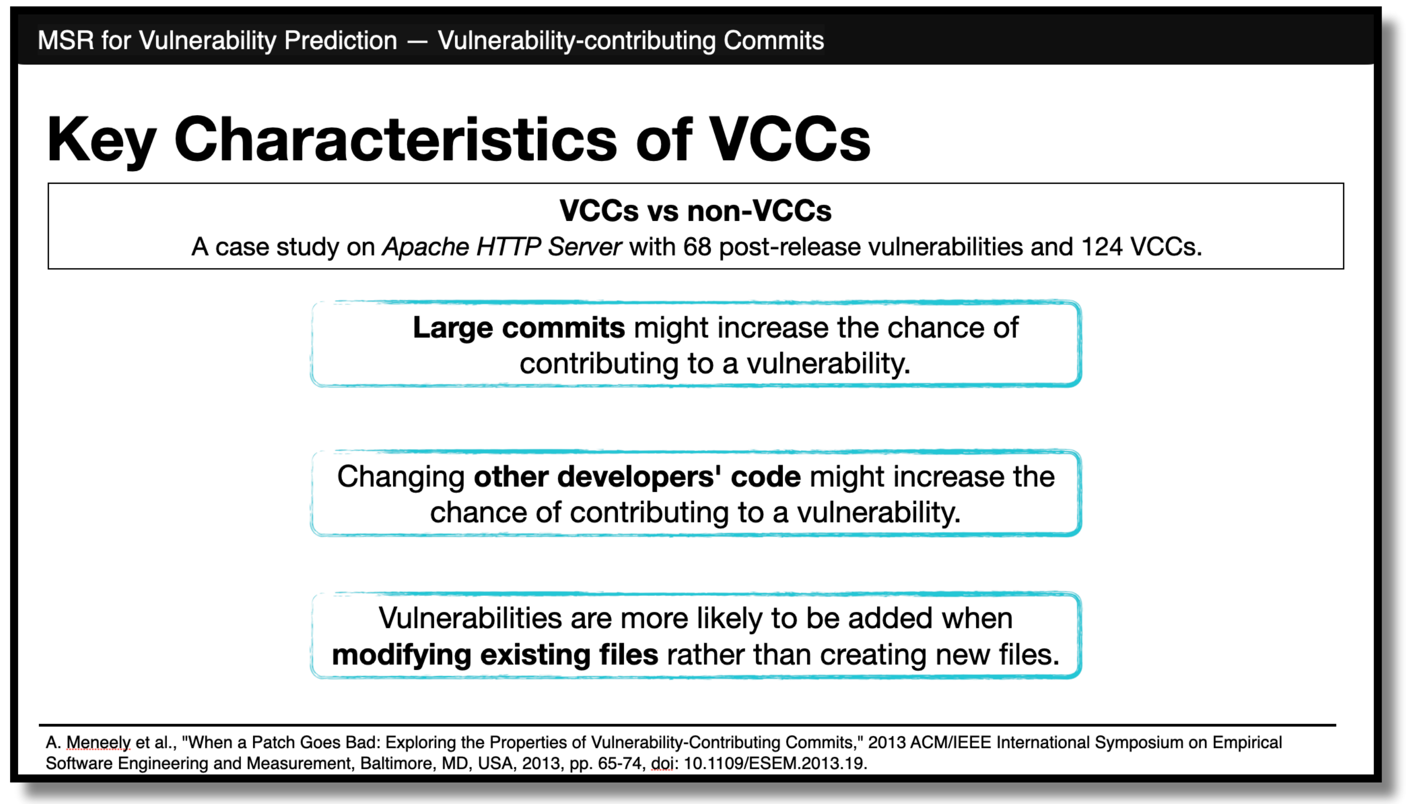
Ad Hoc Tool

The organization in GitHub offers a dedicated command-line tool.

I’m
hungry!

Wrap up

Wrap up



MSR for Vulnerability Prediction — Vulnerability-contributing Commits

Key Characteristics of VCCs

VCCs vs non-VCCs
A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

Large commits might increase the chance of contributing to a vulnerability.

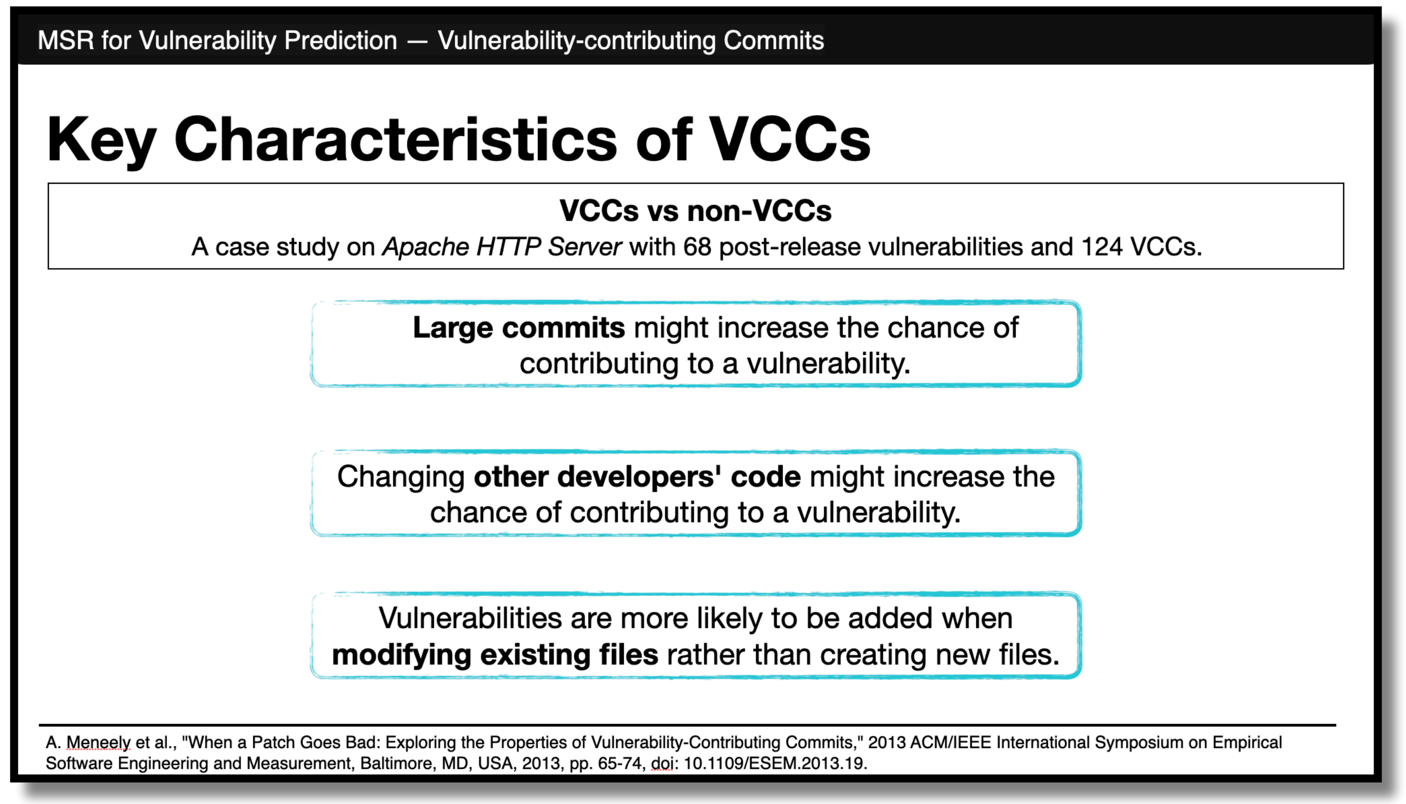
Changing **other developers' code** might increase the chance of contributing to a vulnerability.

Vulnerabilities are more likely to be added when **modifying existing files** rather than creating new files.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Definition &
Characteristics of VCCs

Wrap up



MSR for Vulnerability Prediction — Vulnerability-contributing Commits

Key Characteristics of VCCs

VCCs vs non-VCCs

A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

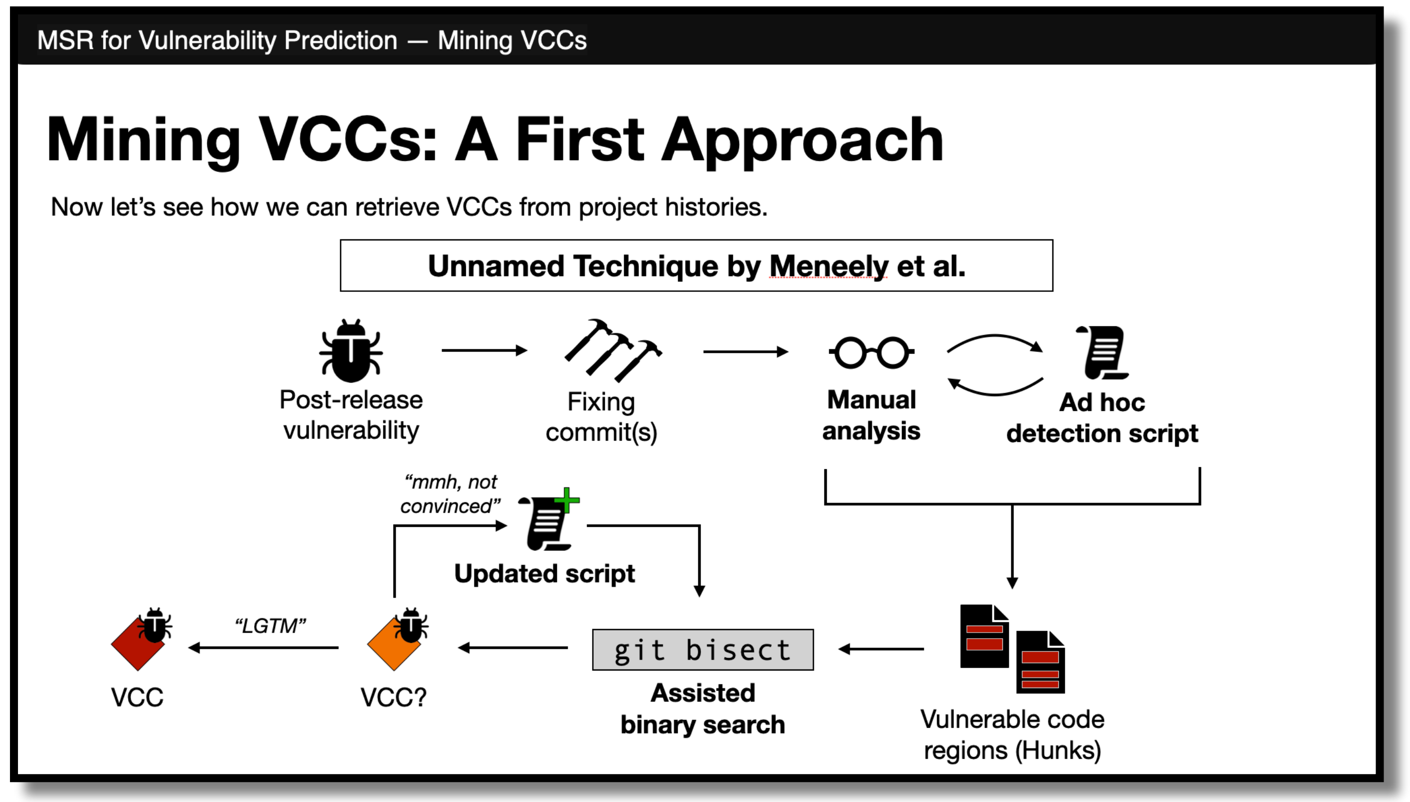
Large commits might increase the chance of contributing to a vulnerability.

Changing **other developers' code** might increase the chance of contributing to a vulnerability.

Vulnerabilities are more likely to be added when **modifying existing files** rather than creating new files.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Definition & Characteristics of VCCs



MSR for Vulnerability Prediction — Mining VCCs

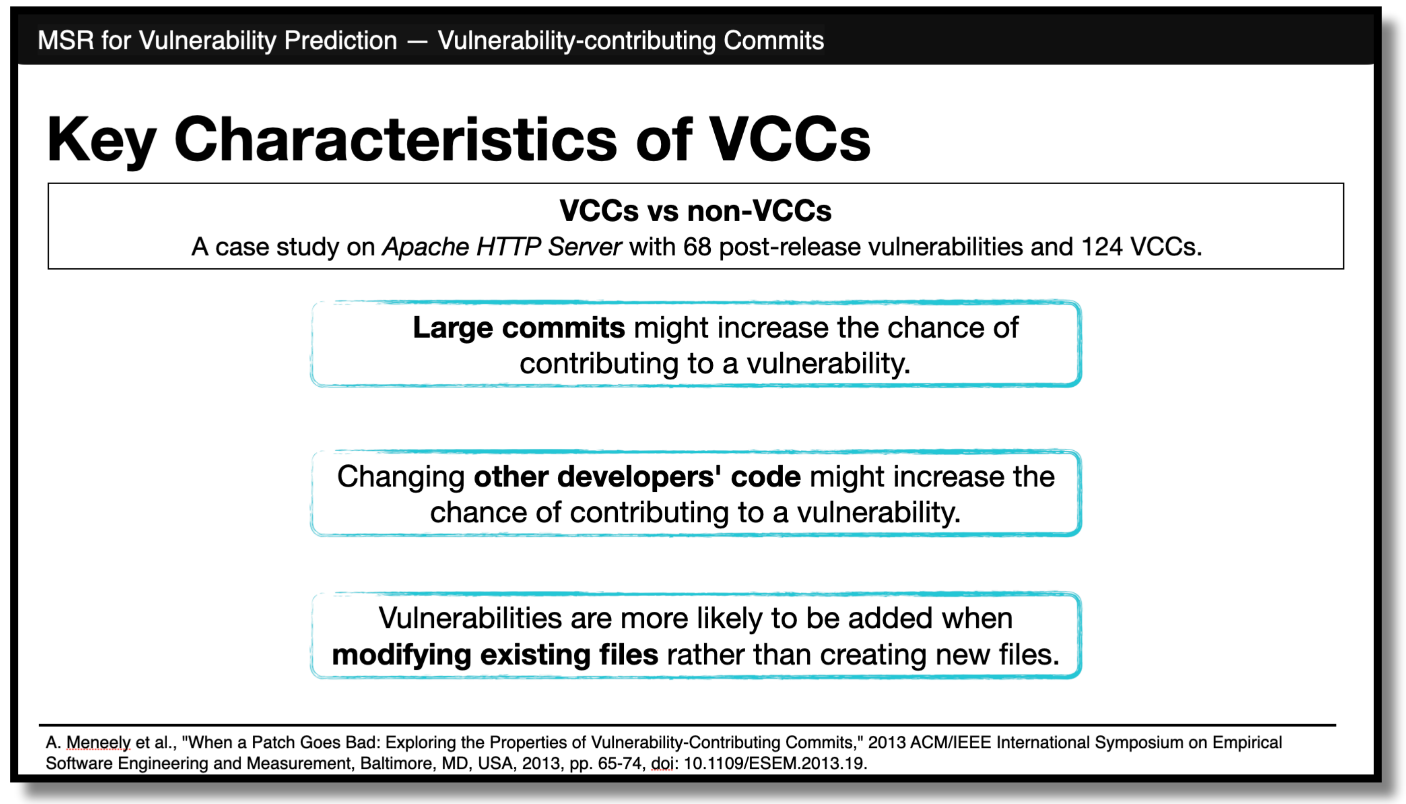
Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

Meneely et al. technique (git bisect)

Wrap up



MSR for Vulnerability Prediction — Vulnerability-contributing Commits

Key Characteristics of VCCs

VCCs vs non-VCCs
A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

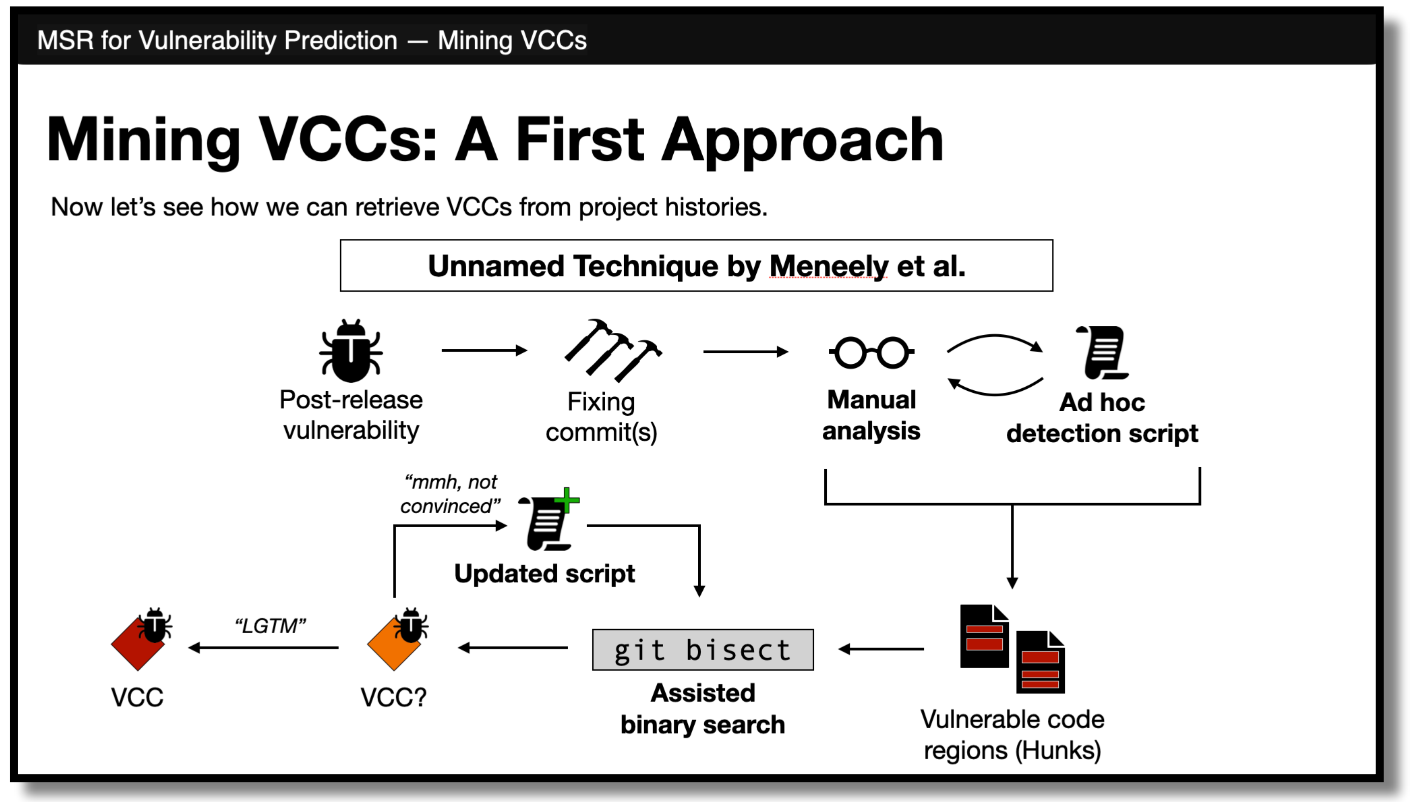
Large commits might increase the chance of contributing to a vulnerability.

Changing **other developers' code** might increase the chance of contributing to a vulnerability.

Vulnerabilities are more likely to be added when **modifying existing files** rather than creating new files.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Definition & Characteristics of VCCs



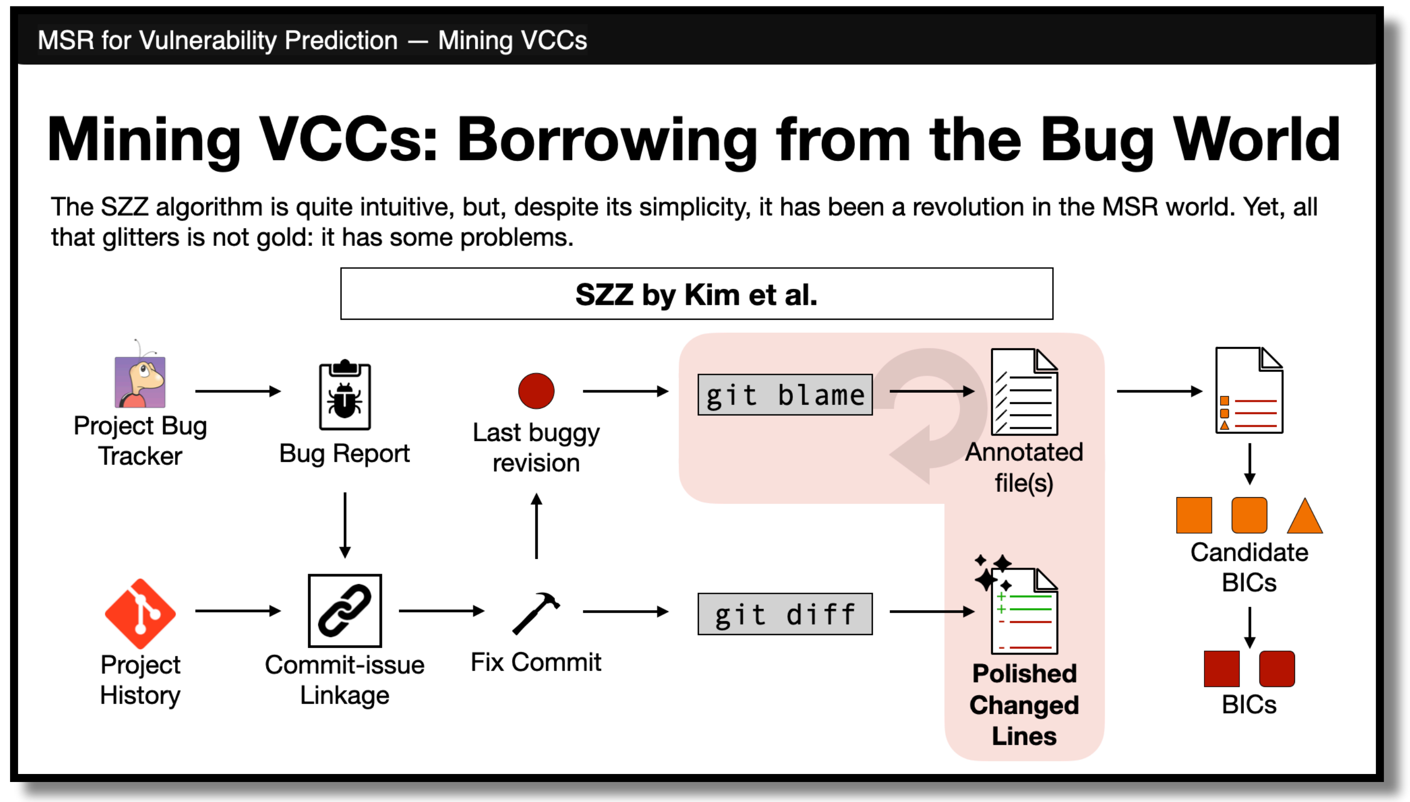
MSR for Vulnerability Prediction — Mining VCCs

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

Meneely et al. technique (git bisect)



MSR for Vulnerability Prediction — Mining VCCs

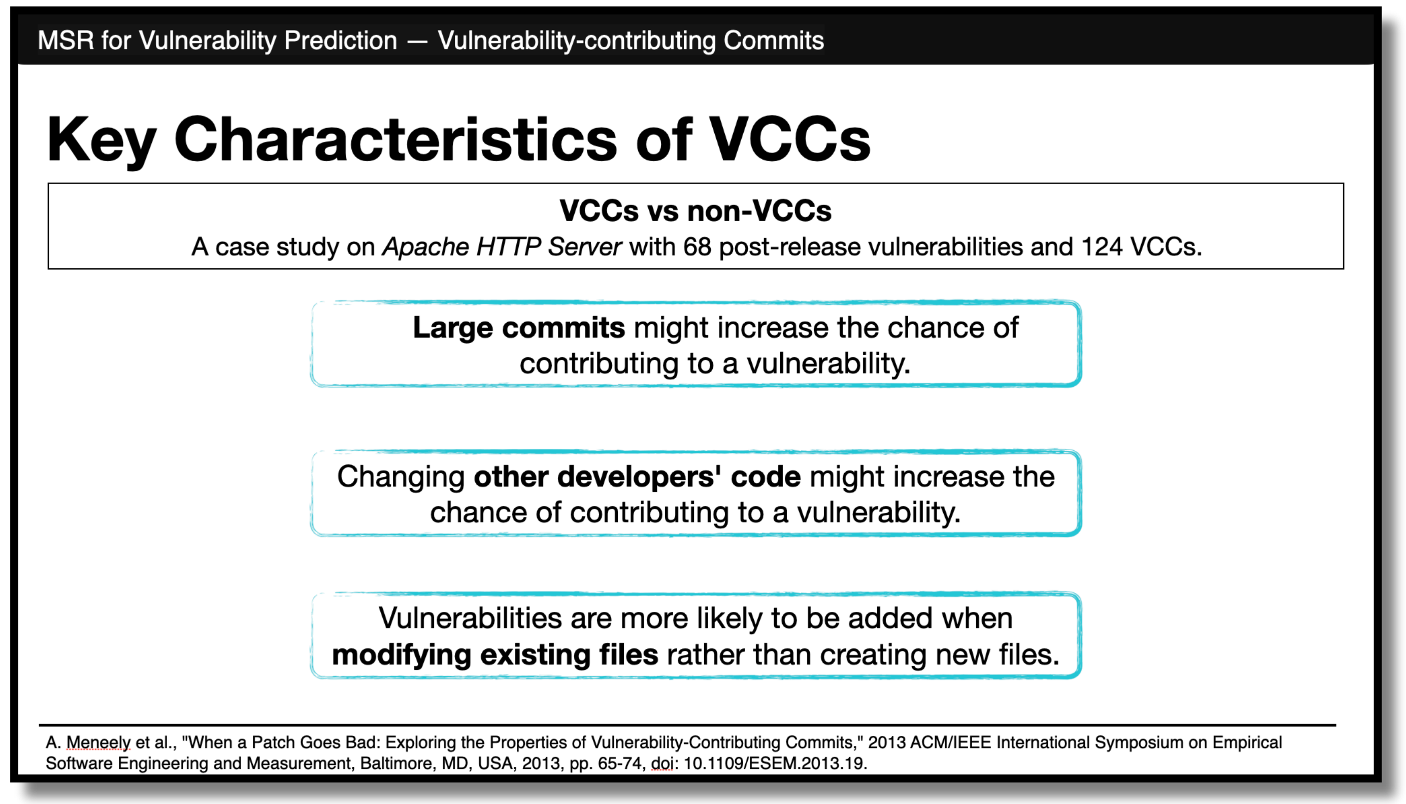
Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

SZZ by Kim et al.

SZZ algorithm and variants (git blame)

Wrap up



MSR for Vulnerability Prediction — Vulnerability-contributing Commits

Key Characteristics of VCCs

VCCs vs non-VCCs
A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

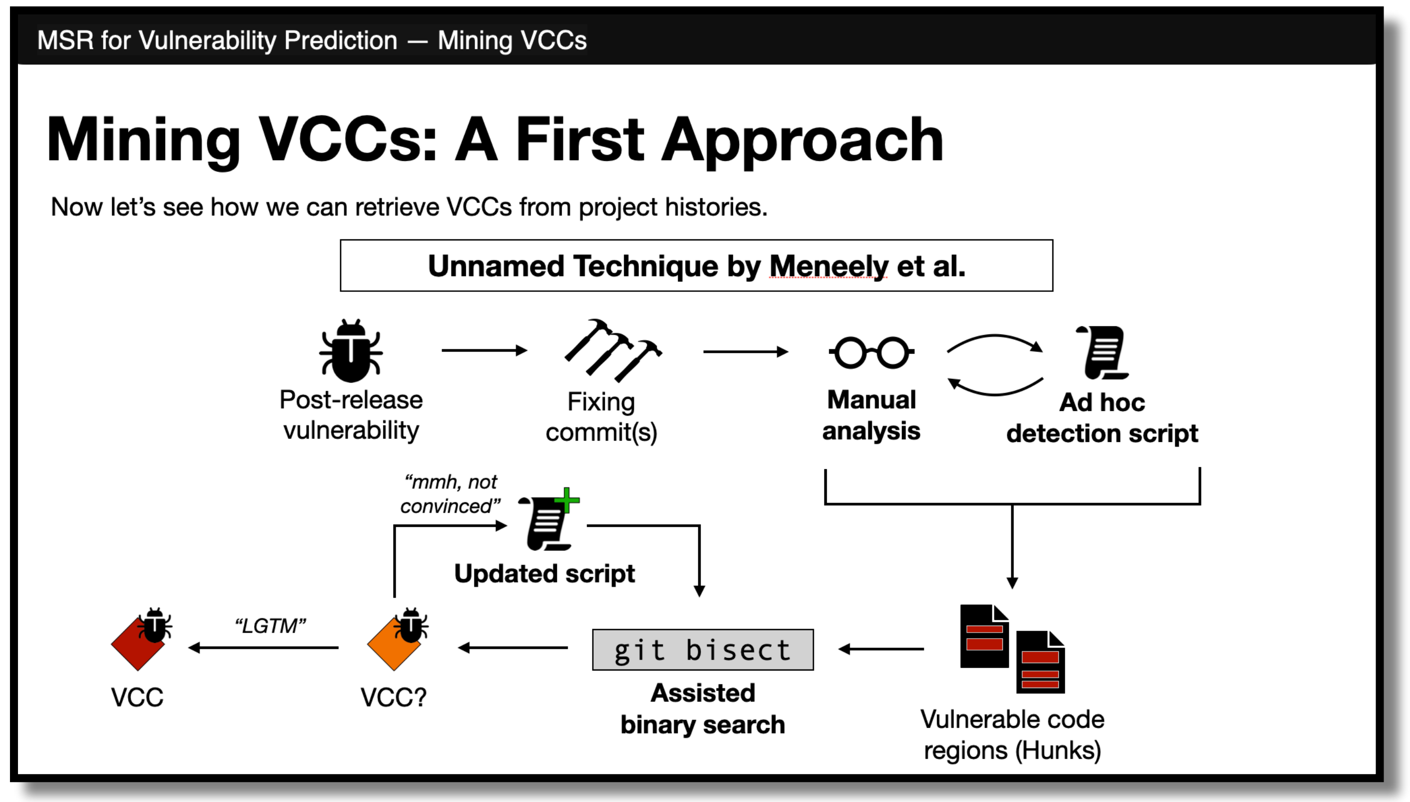
Large commits might increase the chance of contributing to a vulnerability.

Changing **other developers' code** might increase the chance of contributing to a vulnerability.

Vulnerabilities are more likely to be added when **modifying existing files** rather than creating new files.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Definition & Characteristics of VCCs



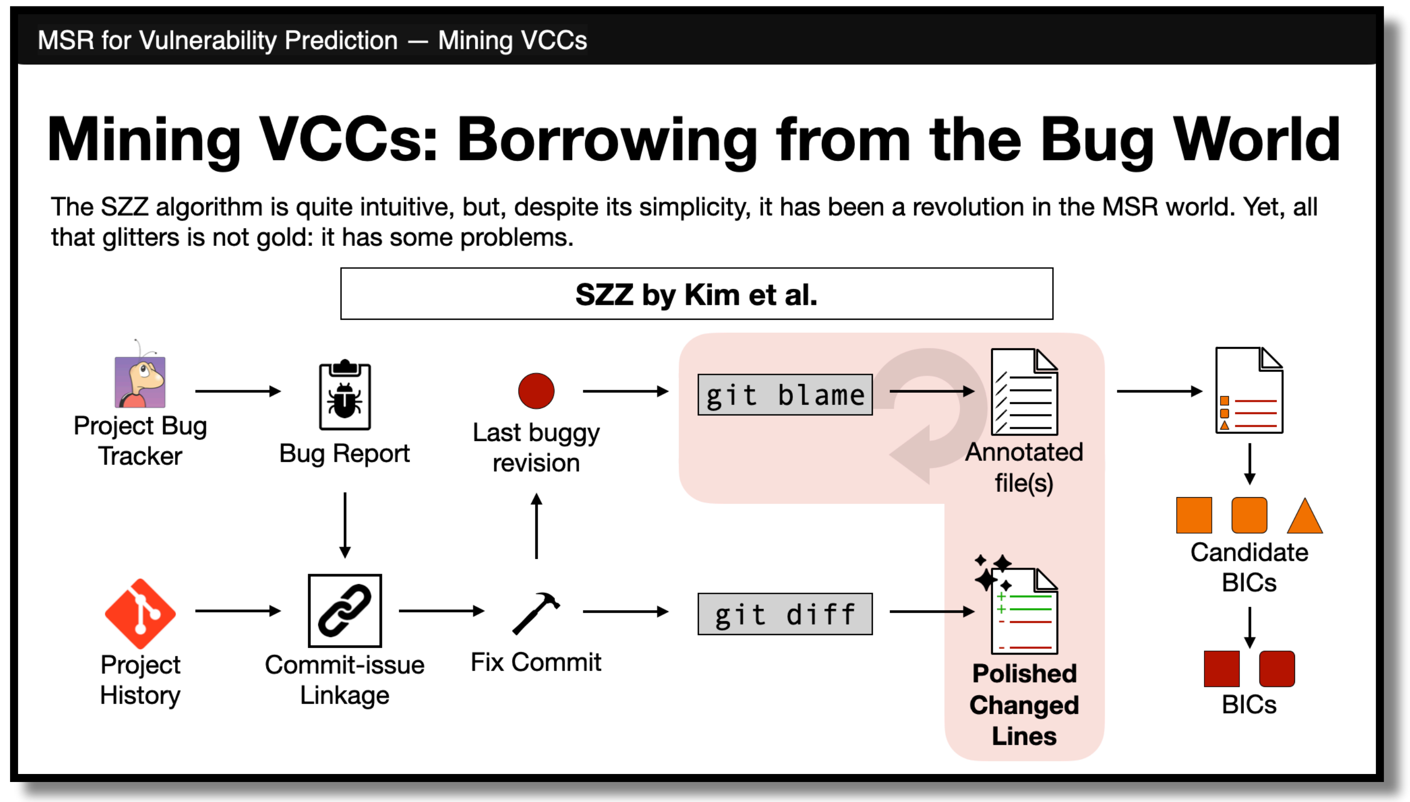
MSR for Vulnerability Prediction — Mining VCCs

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

Meneely et al. technique (git bisect)



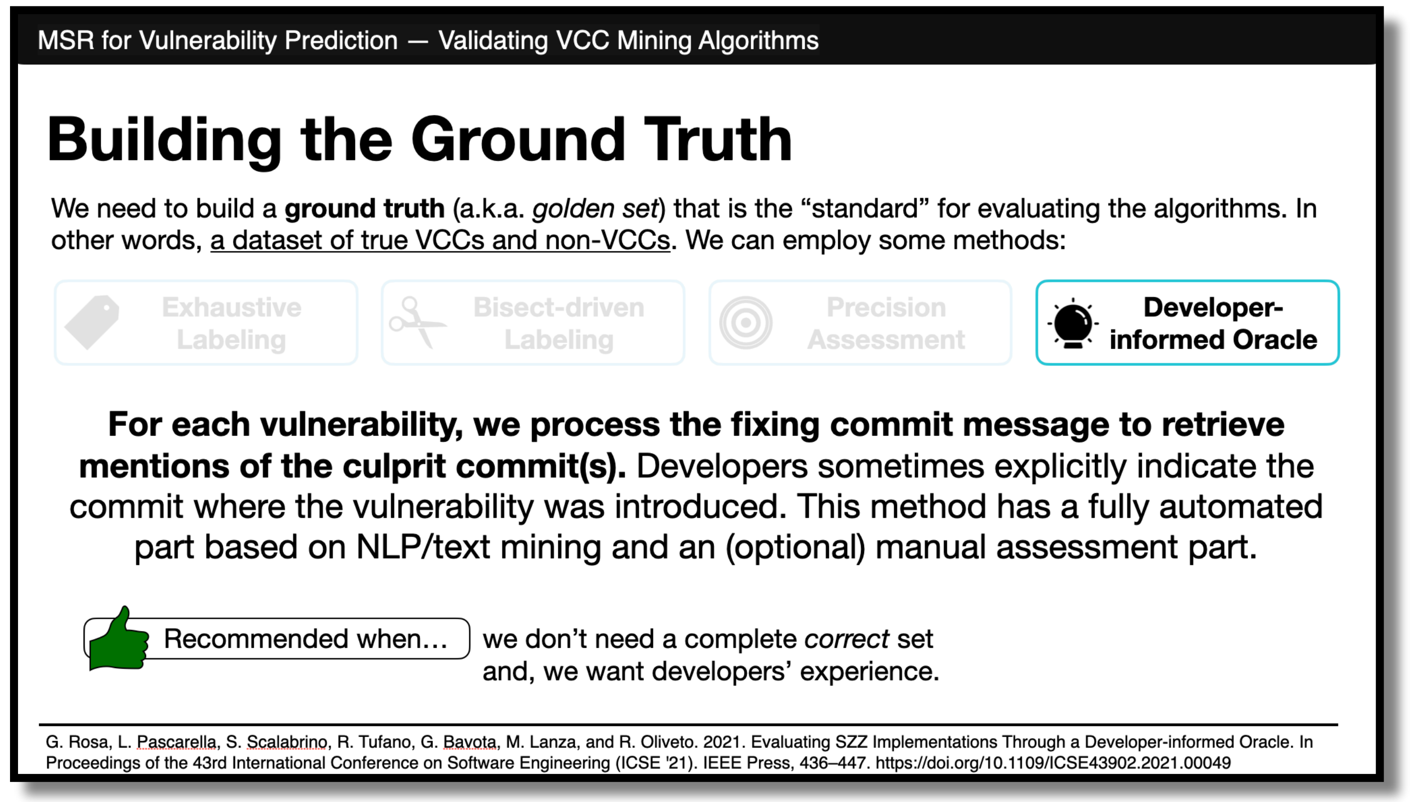
MSR for Vulnerability Prediction — Mining VCCs

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

SZZ by Kim et al.

SZZ algorithm and variants (git blame)



MSR for Vulnerability Prediction — Validating VCC Mining Algorithms

Building the Ground Truth

We need to build a **ground truth** (a.k.a. *golden set*) that is the "standard" for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

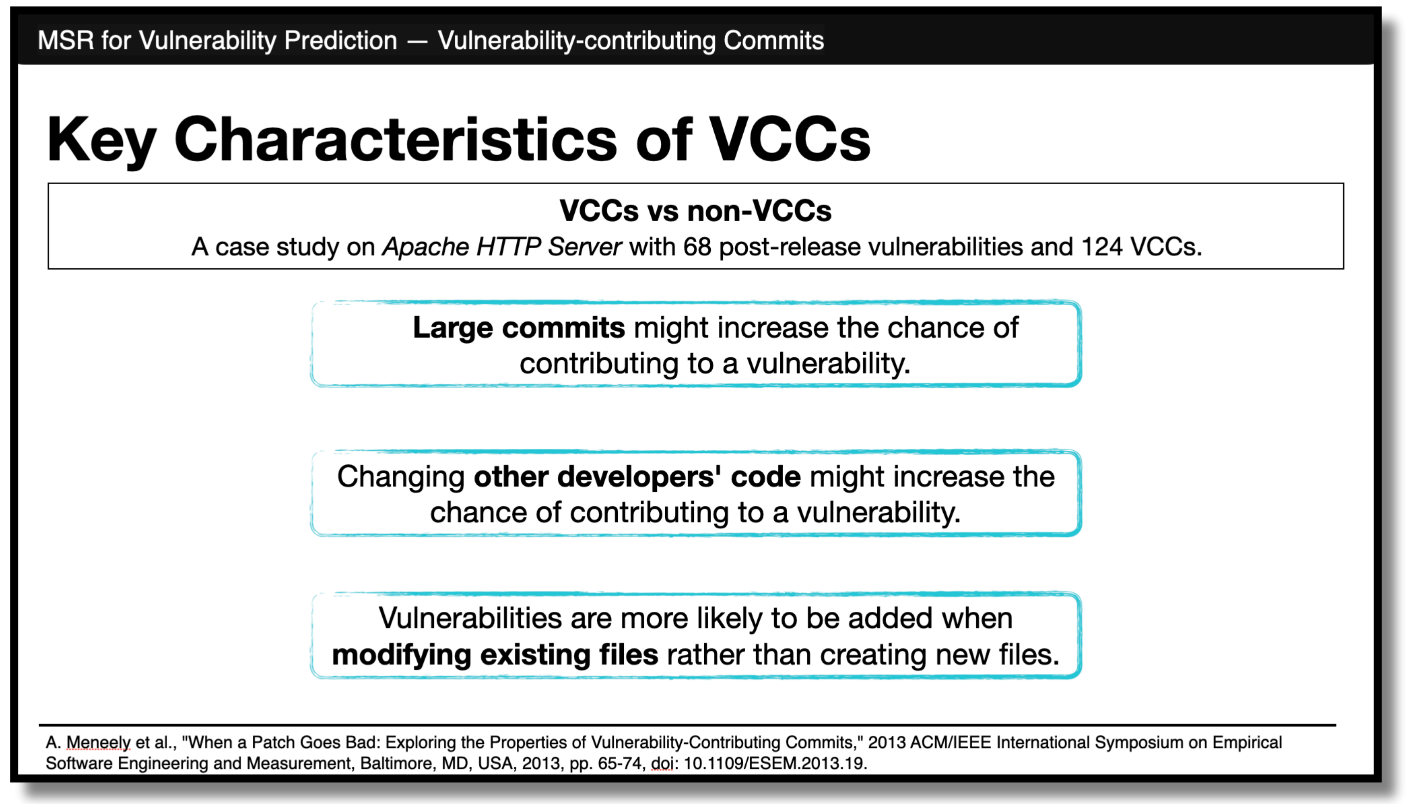
For each vulnerability, we process the fixing commit message to retrieve mentions of the culprit commit(s). Developers sometimes explicitly indicate the commit where the vulnerability was introduced. This method has a fully automated part based on NLP/text mining and an (optional) manual assessment part.

Recommended when… we don't need a complete *correct* set and, we want developers' experience.

G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In Proceedings of the 43rd International Conference on Software Engineering (ICSE '21). IEEE Press, 436–447. https://doi.org/10.1109/ICSE43902.2021.00049

Performance Metrics & Ground Truth

Wrap up



MSR for Vulnerability Prediction — Vulnerability-contributing Commits

Key Characteristics of VCCs

VCCs vs non-VCCs
A case study on *Apache HTTP Server* with 68 post-release vulnerabilities and 124 VCCs.

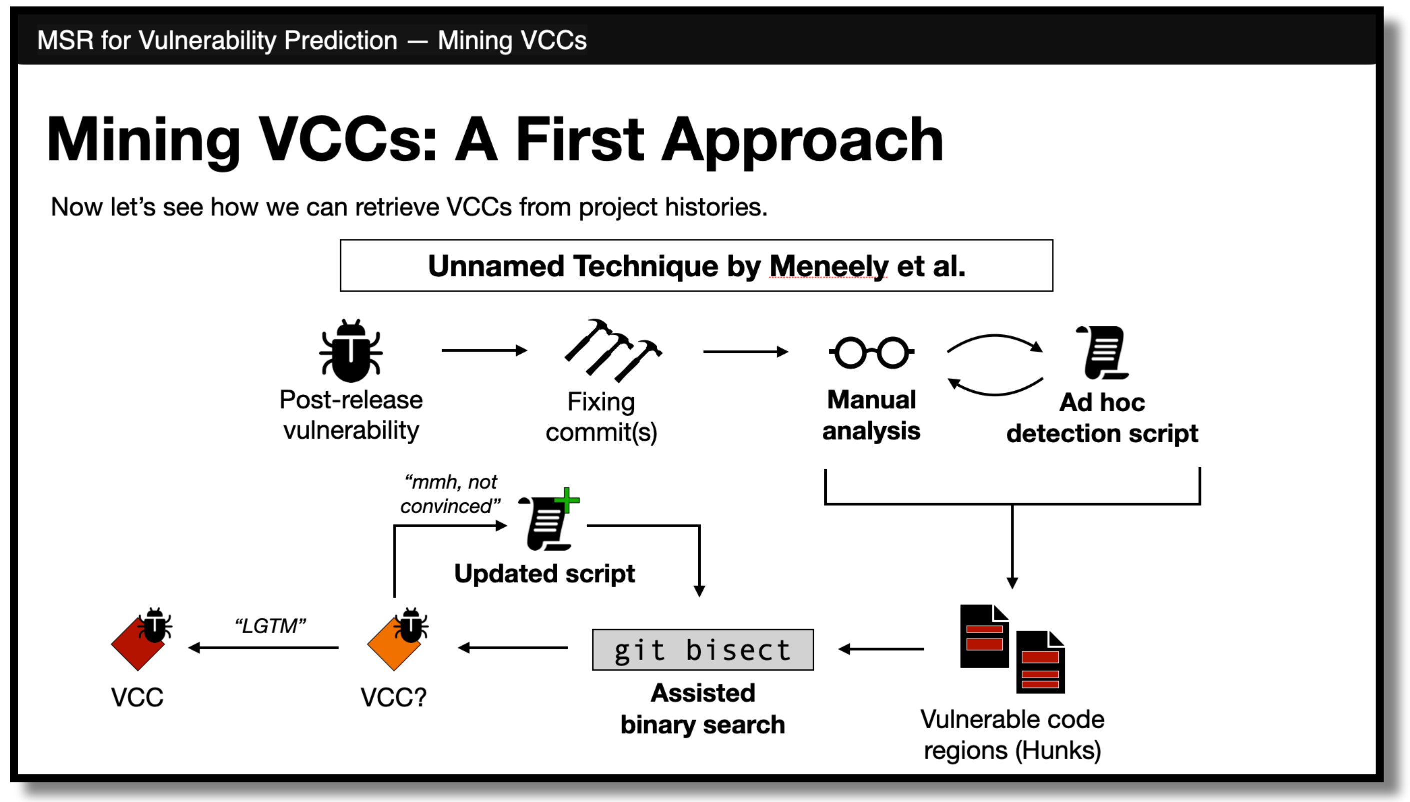
Large commits might increase the chance of contributing to a vulnerability.

Changing **other developers' code** might increase the chance of contributing to a vulnerability.

Vulnerabilities are more likely to be added when **modifying existing files** rather than creating new files.

A. Meneely et al., "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 65-74, doi: 10.1109/ESEM.2013.19.

Definition & Characteristics of VCCs



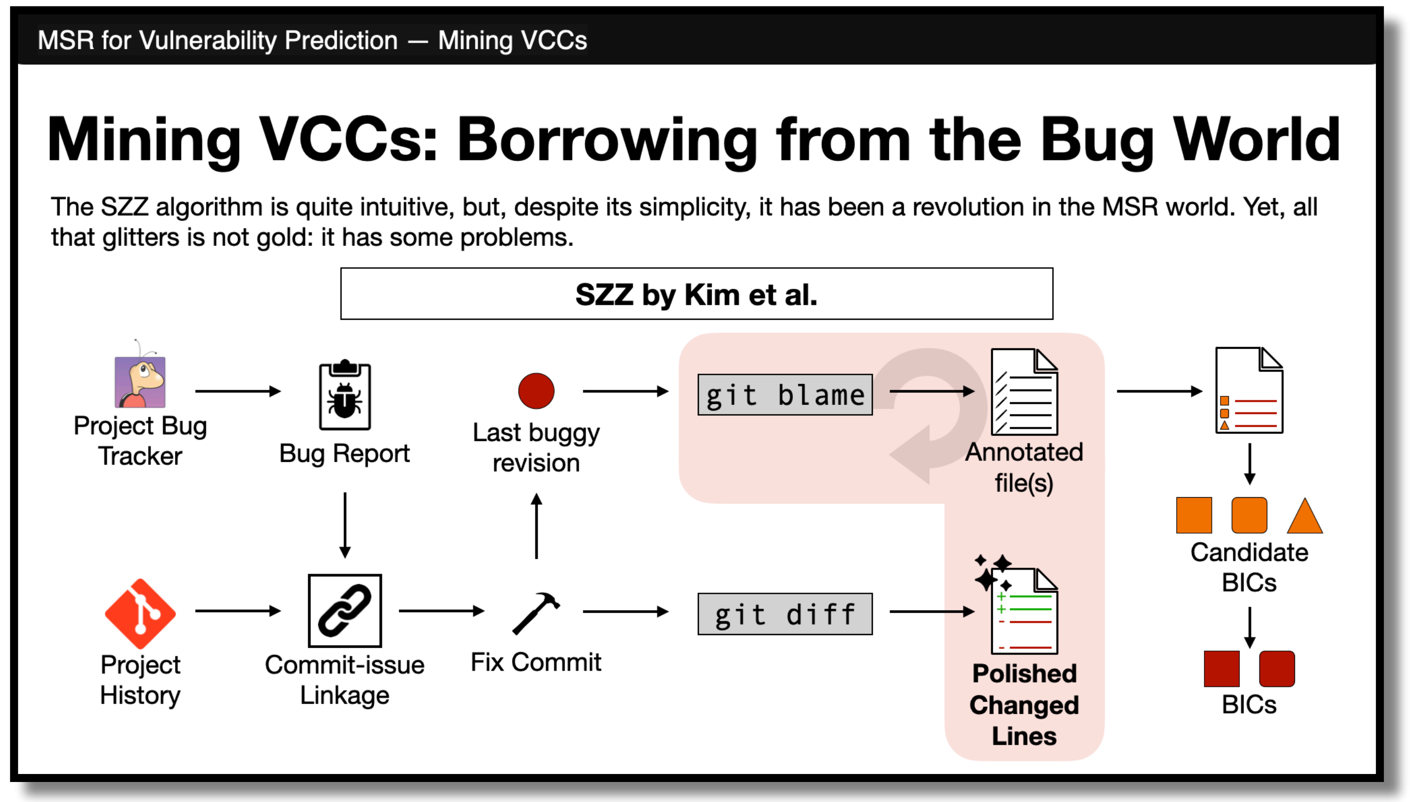
MSR for Vulnerability Prediction — Mining VCCs

Mining VCCs: A First Approach

Now let's see how we can retrieve VCCs from project histories.

Unnamed Technique by Meneely et al.

Meneely et al. technique (git bisect)



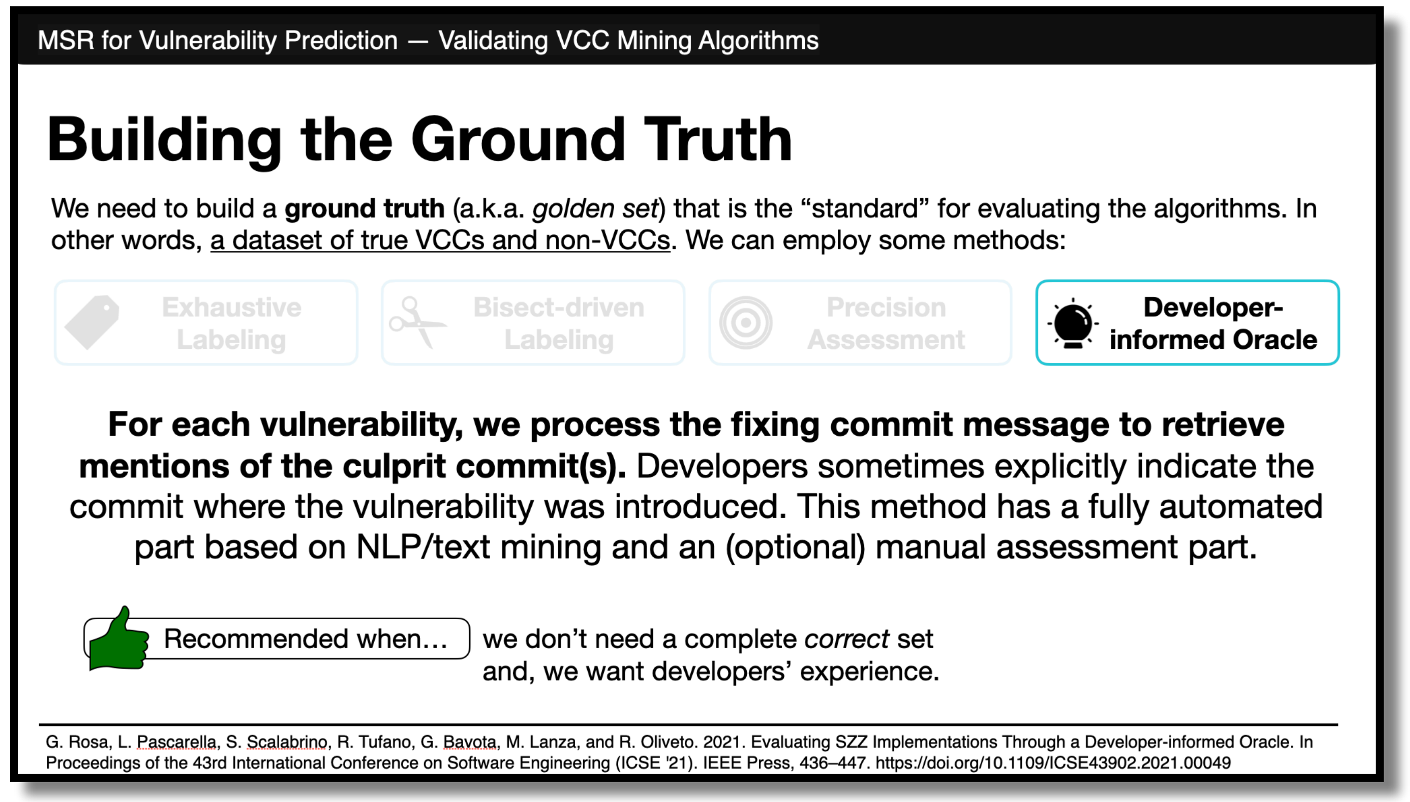
MSR for Vulnerability Prediction — Mining VCCs

Mining VCCs: Borrowing from the Bug World

The SZZ algorithm is quite intuitive, but, despite its simplicity, it has been a revolution in the MSR world. Yet, all that glitters is not gold: it has some problems.

SZZ by Kim et al.

SZZ algorithm and variants (git blame)



MSR for Vulnerability Prediction — Validating VCC Mining Algorithms

Building the Ground Truth

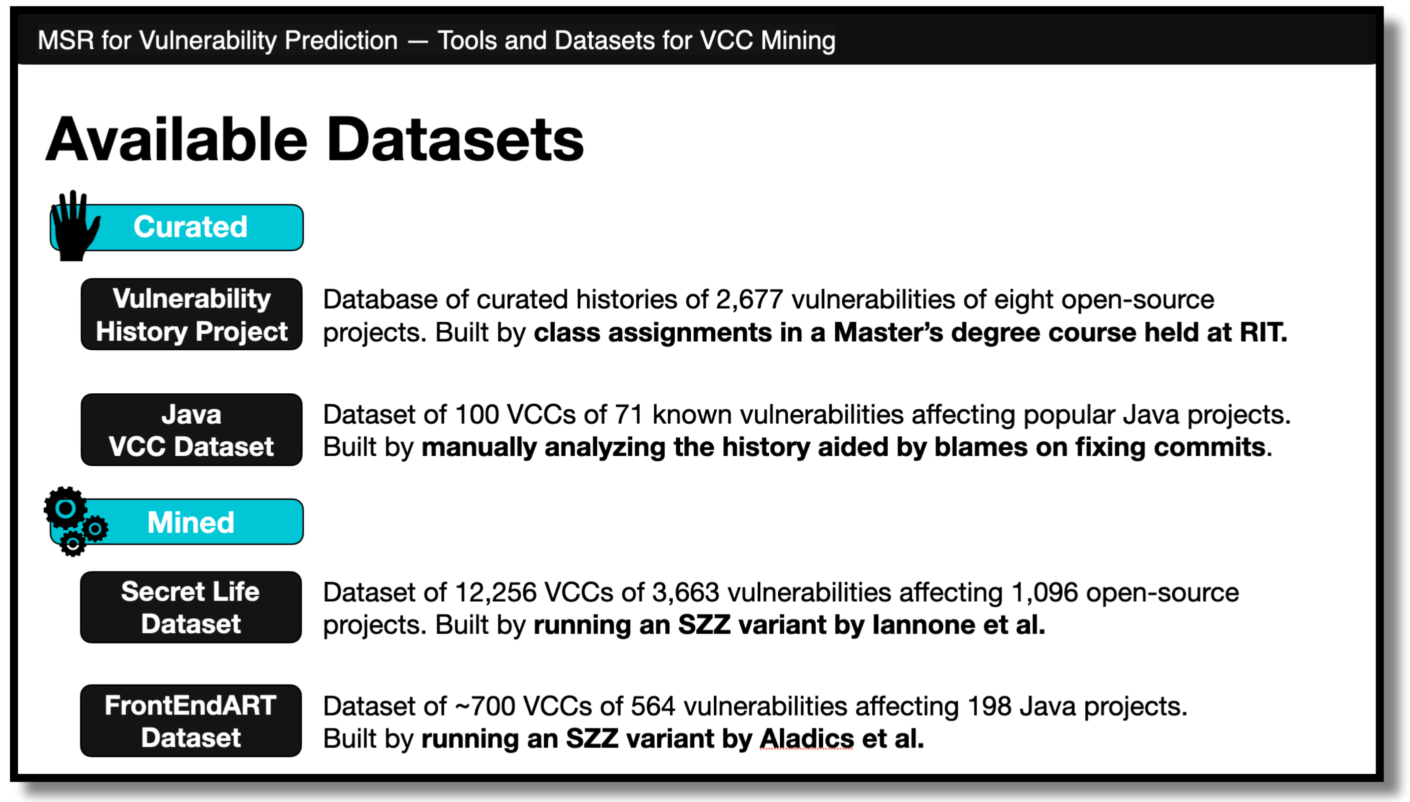
We need to build a **ground truth** (a.k.a. *golden set*) that is the "standard" for evaluating the algorithms. In other words, a dataset of true VCCs and non-VCCs. We can employ some methods:

For each vulnerability, we process the fixing commit message to retrieve mentions of the culprit commit(s). Developers sometimes explicitly indicate the commit where the vulnerability was introduced. This method has a fully automated part based on NLP/text mining and an (optional) manual assessment part.

Recommended when… we don't need a complete *correct* set and, we want developers' experience.

G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In Proceedings of the 43rd International Conference on Software Engineering (ICSE '21). IEEE Press, 436–447. https://doi.org/10.1109/ICSE43902.2021.00049

Performance Metrics & Ground Truth



MSR for Vulnerability Prediction — Tools and Datasets for VCC Mining

Available Datasets

Curated

Vulnerability History Project	Database of curated histories of 2,677 vulnerabilities of eight open-source projects. Built by **class assignments in a Master's degree course held at RIT.**
Java VCC Dataset	Dataset of 100 VCCs of 71 known vulnerabilities affecting popular Java projects. Built by **manually analyzing the history aided by blames on fixing commits.**

Mined

Secret Life Dataset	Dataset of 12,256 VCCs of 3,663 vulnerabilities affecting 1,096 open-source projects. Built by **running an SZZ variant by Iannone et al.**
FrontEndART Dataset	Dataset of ~700 VCCs of 564 vulnerabilities affecting 198 Java projects. Built by **running an SZZ variant by Aladics et al.**

Available Tools & Datasets

(Some) Open Challenges

Non-code-related Vulnerabilities

Not all vulnerabilities are caused by coding mistakes. Some of them are caused by improper configurations or, even worse, design issues.

(Some) Open Challenges

Non-code-related Vulnerabilities

Not all vulnerabilities are caused by coding mistakes. Some of them are caused by improper configurations or, even worse, design issues.

Tangled Changes

Not all fixing commits are focused on fixing the vulnerability: other collateral activities may be done.

(Some) Open Challenges

Non-code-related Vulnerabilities

Not all vulnerabilities are caused by coding mistakes. Some of them are caused by improper configurations or, even worse, design issues.

Tangled Changes

Not all fixing commits are focused on fixing the vulnerability: other collateral activities may be done.

Irrelevant Changes

Not all lines changed are directly related to the vulnerability, e.g., addition/removal of import statements, parameters reordering, etc.

(Some) Open Challenges

Non-code-related Vulnerabilities

Not all vulnerabilities are caused by coding mistakes. Some of them are caused by improper configurations or, even worse, design issues.

Tangled Changes

Not all fixing commits are focused on fixing the vulnerability: other collateral activities may be done.

Irrelevant Changes

Not all lines changed are directly related to the vulnerability, e.g., addition/removal of import statements, parameters reordering, etc.

Migrated Repositories

Many “old” projects were migrated from another VCS (e.g., svn to git), so their history might be incomplete (e.g., the initial commit is enormous).

References

Articles (1/2)

[Meneely et al.] **When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits**: https://ieeexplore.ieee.org/document/6681339

[Śliwerski et al.] **When do changes induce fixes?**: https://dl.acm.org/doi/10.1145/1082983.1083147

[Kim et al.] **Automatic Identification of Bug-Introducing Changes**: https://ieeexplore.ieee.org/document/4019564

[da Costa et al.] **A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes**: https://ieeexplore.ieee.org/document/7588121

[Rodríguez-Pérez et al.] **How bugs are born: a model to identify how bugs are introduced in software components**: https://link.springer.com/article/10.1007/s10664-019-09781-y

[Rosa et al.] **Evaluating SZZ Implementations Through a Developer-informed Oracle**: https://dl.acm.org/doi/10.1109/ICSE43902.2021.00049

[Camilo et al.] **Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project**: https://ieeexplore.ieee.org/document/7180086

References

Articles (2/2)

[Canfora et al.] **Investigating the vulnerability fixing process in OSS projects: Peculiarities and challenges**: https://www.sciencedirect.com/science/article/abs/pii/S0167404820303400

[Canfora et al.] **Patchworking: Exploring the code changes induced by vulnerability fixing activities**: https://www.sciencedirect.com/science/article/abs/pii/S0950584921001932

[Perl et al.] **VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits**: https://dl.acm.org/doi/10.1145/2810103.2813604

[Yang et al.] **VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes**: https://ieeexplore.ieee.org/document/8254428

[Iannone et al.] **The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study**: https://ieeexplore.ieee.org/document/9672730

[Bao et al.] **V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities**: https://ieeexplore.ieee.org/document/9794006

sesa lab
SOFTWARE ENGINEERING
SALERNO

SoftSEC
Institute of
Software
Security

TUHH
Hamburg
University of
Technology

MSR for Vulnerability Prediction

Mining Vulnerability-Contributing Commits

Emanuele Iannone

SeSa Lab @ University of Salerno, Italy

`eiannone@unisa.it`

Master Course “Cybersecurity Data Science”

Winter Semester 22/23