Identifying Casualty Changes in Software Patches

Adriana Sejfia
sejfia@usc.edu
University of Southern California
California, USA

Yixue Zhao
yixuezhao@cs.umass.edu
University of Massachusetts, Amherst
Massachusetts, USA

Nenad Medvidović
neno@usc.edu
University of Southern California
California, USA

ABSTRACT

Noise in software patches impacts their understanding, analysis, and use for tasks such as change prediction. Although several approaches have been developed to identify noise in patches, this issue has persisted. An analysis of a dataset of security patches for the Tomcat web server, which we further expanded with security patches from five additional systems, uncovered several kinds of previously unreported noise which we call nonessential *casualty changes*. These are changes that themselves do not alter the logic of the program but are necessitated by other changes made in the patch. In this paper, we provide a comprehensive taxonomy of casualty changes. We then develop CasCADe, an automated technique for automatically identifying casualty changes. We evaluate CasCADe with several publicly available datasets of patches and tools that focus on them. Our results show that CasCADe is highly accurate, that the kinds of noise it identifies occur relatively commonly in patches, and that removing this noise improves upon the evaluation results of a previously published change-based approach.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Software Patches, Change-based Analysis, Noise in Patches

ACM Reference Format:
Adriana Sejfia, Yixue Zhao, and Nenad Medvidović. 2021. Identifying Casualty Changes in Software Patches. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468624

1 INTRODUCTION

Software patches change a system to address security flaws, fix an algorithm's implementation, add features, boost code maintenance, etc. Beyond these immediate effects, patches are a useful source of information for software researchers and developers. They are particularly helpful in understanding the causes of past problems and providing insights for solving new ones. For instance, patches have been used to identify root-causes of bugs [33], to infer patterns of past bugs [15, 17, 18], including for specific types such as security vulnerabilities [28, 29], and to make recommendations to developers about what to change next during a coding session [34, 35].

Patches are an important source of information, but may contain *noise*. Noise refers to *changes that are nonessential or trivial in the context of the issue being addressed by a patch*. The relevant problem or feature may thus be obfuscated in noise-containing patches.

Noise in patches has been studied previously. Past work has attempted to identify low-significance changes [12]. Diffcat [14] implemented an initial approach to classify and automatically detect several types of changes that are merely noise, referred to as *nonessential changes*. Nonessential changes include renaming of variables, methods, or classes, or trivial changes such as adding a `this` keyword in programs written in Java. Changes like these may lead to wasted effort when manually analyzing patches. They have also been shown to lead to less accurate learning models in systems that use patches for further analysis [14].

Despite the availability of Diffcat for nearly a decade, noise and imprecision persist in patches [19], and researchers still identify the presence of noise as a notable concern (e.g., [15]). This has led us to hypothesize that *there are likely additional types of changes that introduce noise in patches, beyond those handled by prior work*. An added motivation was a recent empirical study [29] in which we observed frequent presence of noise in the security patches of Tomcat [22], an open-source web server implementation.

To look deeper into our hypothesis, we conducted an initial, semi-automated analysis of the Tomcat dataset from [29]. This analysis uncovered that a substantial number of patches contain previously unreported noise that we have termed nonessential *casualty changes*. Casualty changes do not themselves alter the logic of a program; however, they form part of the patch to accommodate other changes which, in turn, can refactor the program or modify its logic.

To illustrate the concept of casualty change, consider the example in Figure 1, from a Tomcat patch that addressed the CVE-2014-0230 [6] security vulnerability. Note that in this paper we make use

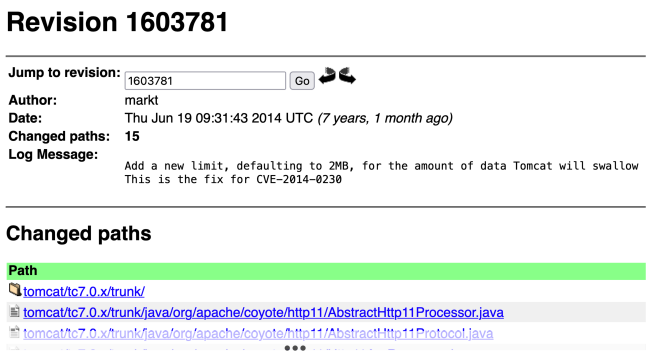

Figure 1: Partial listing of files changed in the patch that addressed vulnerability CVE-2014-0230 [6] in Tomcat 7.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468624

```
1
2 ChunkedInputFilter(int maxTSize, int maxESize) {
3           ...
4 }
```
(a)

```
1 initFilters(int maxTSize, int maxESize){
2     new ChunkedInputFilter(maxTSize, maxESize);
3 }
```
(b)

```
1 int maxSSize
2 ChunkedInputFilter(int maxTSize, int maxESize, int maxSSize) {
3         this.maxSSize = maxSSize;
4 }
```
(c)

```
1 initFilters(int maxTSize, int maxESize, int maxSSize) {
2     new ChunkedInputFilter(maxTSize, maxESize, maxSSize);
3 }
```
(d)

Figure 2: Fix of vulnerability CVE-2014-4320 [6] in Tomcat, with partial pre-patch (a,b) and post-patch (c,d) revisions shown.

of the National Vulnerability Database (NVD) [8] and its identifier for vulnerabilities (CVE). The fix from Figure 1 spans 11 Java files. However, a deeper look reveals that the functionality of only one method was changed, shown in Figure 2. The highlighted portions on the right side of Figure 2 refer to code that was added, while irrelevant code has been elided. Specifically, the change shown in Line 1 of Figure 2c introduced a global variable `maxSwallowSize`, which is set in the constructor `ChunkedInputFilter` (lines 2 and 3) and subsequently used in the class (not shown). This change "cascaded" to the remaining ten files encompassing the patch. We see an example in Figure 2d: the only updates made were to accommodate the changed signature of `ChunkedInputFilter`'s constructor. A developer or tool analyzing the changes to the program's logic made by this patch would have to go through all 11 files, only to ultimately disregard most of the changes as nonessential or trivial.

Our observation of a number of such casualty changes in Tomcat motivated us to analyze five additional systems: Zookeeper [24], Ofbiz [21], Commons Collections [25], Commons Compress [26], and Commons Email [27]. In our analysis, we inspected 1,400 patched methods, of which 22% contained only casualty changes. This directly motivated us to study these changes in a more systematic way. In particular, previous work in this area has two notable gaps that we needed to address. First, while the types of noise targeted by past work [14] at times intersected with the ones we encountered, in the majority of cases the casualty changes we uncovered were not reported previously. Second, prior work focused on grouping related changes in a patch (e.g., the changed constructor and the corresponding changed callsite in Figure 2) to help code reviewers with understanding a patch better [10] but did not classify noise.

We set to fill these gaps by approaching the problem from two directions. First, we uncovered several flavors of casualty changes, each with its own characteristics. This new knowledge needed to be *defined, organized, and put into perspective* when compared to the previously known types of noise. Second, we needed to develop *mechanisms for automatically identifying casualty changes* in patches. Our analysis indicated that the pervasiveness and, at times, subtle presentation of casualty changes renders their manual identification cumbersome, time-consuming, and error prone.

The paper makes the following contributions:

(1) the first taxonomy of casualty changes in software patches, identified and organized for a broad range of systems, with an initial focus on statically typed OO languages;
(2) CasCADe (Casualty Change Automatic Detector), an approach and accompanying tool that automatically identifies casualty changes with high accuracy;
(3) a manually curated dataset of security patches in which casualty changes are identified and labelled. This set includes 358 patches spanning six open-source systems; and
(4) an empirical analysis of datasets and tools used in two extensively referenced prior studies, to confirm the presence and analyze the impact of casualty changes.

Section 2 summarizes prior work and our preliminary analysis of noise in open-source systems. Section 3 details our taxonomy of casualty changes. Section 4 describes CasCADe, our approach for detecting casualty changes. Section 5 presents our evaluation results, Section 6 the limitations of our work, and Section 7 our conclusions.

2 BACKGROUND

This section overviews prior work in identifying noise in software patches. We then describe our analysis of six-open source systems, conducted to better understand this phenomenon. This analysis directly motivated our taxonomy and development of CasCADe.

2.1 Related Work

A sizable body of prior work has studied ways of improving the quality of software patches. In some cases, authors have sought to weed out noise [12, 14]. In others, their goal has been to better organize groups of changes, to aid code reviewers [10]. Yet others have considered the quality of patches from a broader perspective, analyzing past patches and the numbers of issues that are addressed in each [19]. Specific types of patches and their quality have also been studied [16]. Lastly, prior research has provided suggestions on how to write better patches [30].

Our goal of detecting noise in patches is most closely aligned with Diffcat [14]. The definition of noise in Diffcat partly overlaps with our definition of casualty change. For instance, certain changes that stem from variable updates can potentially lead to casualty changes. One such case is an update to a variable's name, a scenario handled by Diffcat [14]. However, Diffcat also targets a set of more trivial changes, such as adding a `this` keyword in Java programs, replacing the simple name of a type with its fully qualified name (e.g., from `List` to `java.util.List`), or updating comments. These types of noise do not stem from other changes and are thus different from casualty changes. As such, they are outside our scope. On the flip side, most casualty changes stem from system updates other than variable or method renamings, and are not considered by Diffcat.

Fluri and Gall detail a taxonomy of low-significance changes [12]. This taxonomy does not deal with changes that cascade from others, but rather stand-alone changes and their impact in the code. It thus has a different scope and complements our taxonomy.

Our approach is also similar to the work of Barnett et al. [10]. However, their end-goal is different: They aim to group changes via program analysis so that code reviewers can consider each change in a patch and decide whether to approve it. We also use program analysis for the auxiliary purpose of exposing dependencies between different changes in a patch, but our goal is to isolate noise.

Grouping of related changes in patches and their improved presentation has also been the focus of the research work by Huang et al. [13]. They predefine five scenarios, referred to as links, which designate changes that should be grouped. Those scenarios are similar in nature to our casualty changes. However, our work presents a more extensive taxonomy and corresponding detection approach that essentially extends the five links.

Patches and their quality have garnered the interest of researchers because numerous tools and techniques rely on patches. They have been used for bug finding/understanding [28, 33] and for recommender systems [34, 35]. Additionally, patches have been employed as a learning basis for automated program repair techniques [15, 17, 18]. A number of these publications cite the quality of patches as a concern. Several have restricted to at most two-line patches in their datasets, in part due to the cost associated with the noise in patches [15]. We anticipate that our approach will help this line of work. To illustrate that, we use publicly available datasets from two of the previous studies [14, 33], and investigate the presence of casualty changes therein and their impact.

2.2 Preliminary Analysis

To obtain a better understanding of noise in patches, we conducted a preliminary analysis of several open-source systems. Our initial analysis focused on noise present in security patches (recall Section 1). We subsequently verified our findings with other types of patches, as discussed in other sections.

We extended the original Tomcat-based dataset [29] with data from five additional systems: Commons Compress [26], Commons Email [27], Commons Collections [25], Zookeeper [24],and Ofbiz [21]. As with Tomcat, we selected these systems because their vulnerability data is publicly available and use NVD's nomenclature [8] to specify the patches that fixed vulnerabilities (CVEs).

We implemented crawlers to collect the necessary information from the six subject systems: the IDs of the CVEs, the patches that fixed each CVE, and the relevant files involved in each patch. We then had to manually analyze the changes in patches, to determine whether they are noise and, if so, what kind of noise. This manual analysis was conducted by one of this paper's authors, who has extensive familiarity with the NVD spanning multiple large projects.

The six subject systems yielded a total of 358 security patches which contained a total of 1,400 changed methods. For each of the methods, we manually explored the rationale behind the change and labeled any instances of noise we found. This step uncovered noise in a number of patches that was different from that reported by prior work. These were changes that developers had to make as "casualties" of the changes actually required to patch a problem. Our analysis uncovered different types of casualty changes. We formally define these changes in Section 3. Each identified method was explored at least twice to ensure consistency in labeling. We have made the labeled dataset publicly available [9], to facilitate independent confirmation of our results and subsequent studies in this area.

Our manual analysis uncovered that 75 out of the 358 patches (21%) contained casualty changes. Moreover, 22% of the methods that were updated, added, or deleted included *only* casualty changes, i.e., the changes in these methods consist entirely of noise.

3 TAXONOMY OF CASUALTY CHANGES

As introduced above, nonessential *casualty changes* are defined as changes that happen *as a result of* and *to accommodate* other changes, but that themselves do not alter the logic of a program. Our analysis of the six open-source systems uncovered a range of casualty changes. We have organized these changes, augmenting them with their possible, although not always observed in our subject systems, variations.

The transition from the individual examples observed in the subject systems to the taxonomy occurred iteratively. We first collaboratively analyzed the collected examples. In each instance, one of the authors would propose a taxonomy category and possibly its further breakdown. This initiated a discussion and, as needed, refinement of each category, which was terminated only after all authors reached a consensus. In the final stage, we analyzed the taxonomy for completeness. In this stage, we added further categories inspired by related literature and our experience, even if we had not observed the corresponding examples in our dataset.

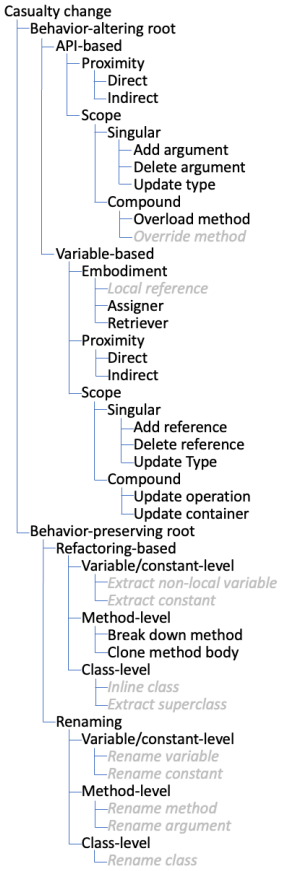

Figure 3: Taxonomy of casualty changes. Elements handled by prior work or not found in our dataset are greyed out.

This yielded the taxonomy presented in Figure 3. We believe that, while not necessarily complete, the taxonomy is reasonably comprehensive and will be useful both conceptually and in practice: *conceptually*, the types and nature of casualty changes differ across categories, and so does the rationale behind classifying them as such; *in practice*, these differences influence the ways in which casualty changes can be automatically identified and thus impact an approach such as ours. Although it is inspired by findings from a dataset of statically-typed OO systems, our taxonomy is applicable for, more broadly, systems written in procedural languages.

Next, we first explain several concepts related to casualty changes and then use such concepts to detail our taxonomy.

3.1 Concepts Underlying Casualty Changes

A *pre-patch revision* is the revision of a software system immediately before a given patch is applied. Conversely, a *post-patch revision* is the revision immediately after a patch is applied.

A *code element* is a program portion changed in a patch. Code elements can be expressions, statements, methods, classes, or packages.

We classify code elements as simple (expressions, statements) and composite (methods, classes, packages) elements. Composite code elements contain simple elements and possibly other composite elements (e.g., a class contains both statements and methods).

A *change* is the application of a single *change type*—move, update, delete, or insert—to a code element. We associate each change with the finest-grained code element involved in that change. For example, this means that a change of type update is never associated with methods, classes, or packages, since one can always isolate a finer-grained, simple code element that was updated within them. This decision is not arbitrary: identifying the simple code elements (1) helps one pinpoint the location of a change while (2) allowing coarser-grained changes to be represented at the level of a composite element if needed. A *change set* is a set of changes that happen to one or more simple code elements within a composite element.

These concepts allow us to elaborate on the above definition of *casualty change*: it is a logic-preserving change of a simple code element within a program, necessitated by another change made elsewhere in the program. Each casualty change is, by definition, contained inside a single method, which we term the *casualty method*.

The original change to a program that causes casualty changes is the *root change*. In cases when this change is contained within a method (as opposed to, e.g., a change to a class variable), we refer to such a method as the *root method*. Specifically, a casualty change is introduced in the code when the changed code element has a control, data, or call dependency with the code element of the root change. A casualty change can occur in three different ways: (1) to accommodate the root change directly; (2) as a result of a call to a casualty method whose API changed, which renders the calling method a casualty, too; or (3) by cascading through another casualty change within a single casualty method. In the third case, the code element of the cascaded casualty change has a forward control or data dependency with the code element of the initial casualty change. We thus refer to the cascaded change as a *casualty dependent*.

Casualty changes additionally differ based on the type of root changes from which they stem. As reflected in Figure 3, root changes can be *behavior-altering* or *behavior-preserving*. A behavior-altering root change is one that modifies the program's logic in order to patch a problem. We have identified two types of such root changes: changes in APIs and in variables. By contrast, behavior-preserving root changes are introduced to improve the structure, readability, and/or maintainability of a program. These can be code refactorings or changes that rename code elements. We further categorize each casualty change based, as appropriate, on its embodiment (dimension of *Variable-based* in Figure 3), proximity to the root change, (dimension of *API-based* and *Variable-based*), scope of analysis (dimension of *API-based* and *Variable-based*), or purpose of the underlying activity in the root change (dimensions of *Refactoring-based*).

We elaborate on three of the principal casualty change types next: API-, variable-, and refactoring-based. As discussed above, renaming changes are more straightforward and have been handled by prior work [14]. We thus do not focus on them in this paper.

3.2 API-Based Casualty Changes

Casualty changes in this category occur when at least one parameter is inserted or deleted, or one or more of the existing parameters' types are updated, in a method's signature. The involved method is the root method with respect to the change.

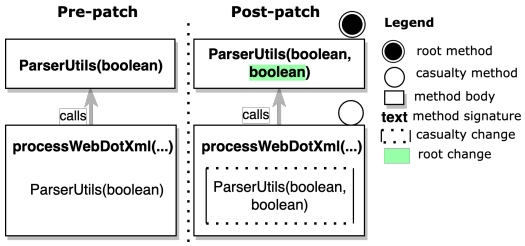

Figure 4: Partial fix of CVE-2013-4590 [5], showing an API-based casualty change.

A change to root method's API mandates changes all of its callsites, by adding, removing, or updating the arguments that correspond to the parameter(s) in the root change. We categorize such callsite changes as API-based casualty changes. Moreover, any casualty dependents—occurring when an added, removed, or updated argument is declared in the casualty method and then passed to the callsite—are also categorized as API-based casualty changes.

For example, consider a portion of the patch that fixed CVE-2013-4590 [5] in Tomcat, schematically depicted in Figure 4 with the code irrelevant to our discussion elided. The constructor of the `ParserUtils` class changed in the post-patch revision. Specifically, the API of this constructor was altered by adding a new `boolean` parameter – the root change in this case. The effect of the root change cascaded to the callsites of the constructor. One of the callsites is contained in the method `processWebDotXml` and the change to it is considered an API-based casualty change.

In general, API-based casualty changes can take several different forms (Figure 3), depending on (1) proximity to the root change and (2) scope of analysis. We discuss these two dimensions next.

1) Proximity considers the distance of the casualty method from the root method. In the context of API-based changes, casualties can follow the root change directly or indirectly. A *direct* casualty change is induced by the root change without intervening program elements, i.e., the casualty method in which they are contained calls the root method. For instance, in Figure 4 the callsite in `processWebDotXml` was changed because the `ParserUtils` constructor invoked there changed its API and is a root method. An *indirect* casualty change is one in which the casualty method is separated from the root method by more than one edge in the callgraph. This happens when the APIs of the methods along the path between the casualty and root methods are changed to reflect the root change. These intermediate methods can be thought of as root proxies.

A method is a *root proxy* when it contains an API-based casualty change that modifies said method's own API. This happens when the root proxy's API has a data dependency to the code element of another casualty change in the method, making the API change a casualty dependent. With the API change, the callsites of the root proxy have to be changed as well; those changes are indirect casualty changes with respect to the original, i.e., root change.

For example, consider the partial depiction of the Tomcat patch that addressed CVE-2013-4322 [4] in Figure 5. The root change is the API change of the `ChunkedInputFilter` class's constructor. That change induced direct casualty changes in the `initializeFilters`

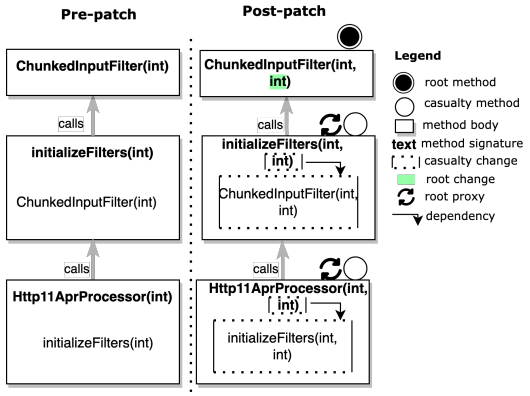

Figure 5: Partial fix of CVE-2013-4322 [4], depicting direct and indirect API-based casualty changes.

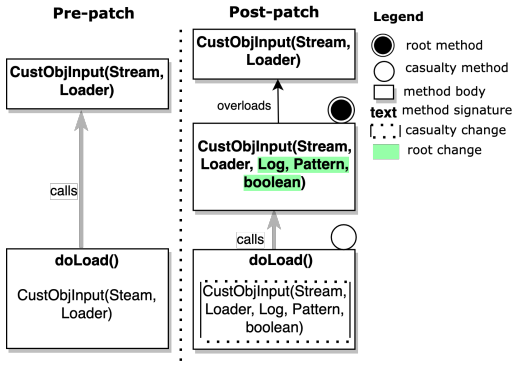

Figure 6: Partial fix of CVE-2013-4590 [5], an example of a compound API-based casualty change.

method: the change in the callsite as well as in the method's API. In this case, there is a data dependency between the parameter added in the API of `initializeFilters` and the argument in the initial casualty change (the callsite to `ChunkedInputFilter`). Hence, the API change is a casualty dependent and is caused by the root change as well. As explained above, the change to `initializeFilters`' API renders it a root proxy since its callsites now have to be changed as well. One such updated callsite is inside `Http11AprProcessor`. This change is an indirect casualty change: the casualty method housing the change (`Http11AprProcessor`) reaches the root method (`ChunkedInputFilter`) in the system's callgraph through an intermediate node (`initializeFilters`) that acts as a root proxy.

Observe that `Http11AprProcessor`'s API was also changed. That change is a casualty dependent with respect to the callsite's casualty change, due to a data dependency. `Http11AprProcessor` is therefore also a root proxy, and its own callsites will experience further indirect casualty changes. The casualty methods including those callsites will be separated from the root method by two nodes in the callgraph (`Http11AprProcessor` and `initializeFilter`). The prospect of indirect casualty changes with an arbitrary number of intermediate nodes induces complexity in an approach aiming to identify API-based casualties of a given root change.

2) Scope refers to the portion of a program that needs to be analyzed to identify a root change, and can be singular or compound. The *singular* case was illustrated above: a method's API is changed via parameter addition, deletion, or update. To detect the API change, it is sufficient to check just that one root method and to employ call, data-flow, and/or control-flow analyses to identify the root's casualties. The examples in Figures 4 and 5 represent singular API-based casualty changes: each root change involved adding a parameter and each casualty method altered its callsite with a corresponding added argument. The cases in which parameters are deleted or updated in the root method analogously involve the deletion or update of corresponding arguments.

By contrast, in the *compound* case, the root change involves multiple methods. Such cases occur when an API is changed through overloading a method within a class or overriding a method in classes related through inheritance. This can happen when an overloading or overriding method is inserted, or when such methods are deleted. In the former case, a patch results in two or more methods that share names but differ in arity, parameter types, or implementations. In the latter case, at least one method in the patch has the same name as the deleted method(s), but different parameters or implementations. Finding the root change in the compound case requires identifying all overloaded methods within a class or overriden methods across classes in the inheritance hierarchy, and then analyzing their callsites for the presence of casualty changes.

To illustrate compound casualty changes, consider a partial Tomcat fix of CVE-2013-4590 [5] in Figure 6. As part of the fix, the constructor for `CustomObjIn` (method name abbreviated for space) was overloaded. The class has retained the original constructor, but now has a second constructor with three additional parameters. The method `doLoad`, which initially called `CustomObjIn`'s original constructor, calls the overloaded one in the post-patch revision.

We consider each such new constructor *a* root method of the API-based change. If a patch involves overloading or overriding a constructor with more than one method, all of the added methods would, together, comprise *the* compound root of the API-based change. Uncovering the casualty changes in such cases requires locating and distinguishing the (unchanged) callsites of the original constructors from the (casualty) callsites of the new constructors. Conversely, if an overloading or overriding method is deleted, the remaining method(s) comprise the root method. Uncovering casualty changes then requires distinguishing the cases where a callsite is deleted from those in which it is updated to a remaining method.

3.3 Variable-Based Casualty Changes

This category deals with changes to local variables of a method or member variables of a class. Casualty changes can occur both when existing variables are modified (i.e., deleted or updated) and when new variables are added. When an existing variable is modified in a patch, all code locations in which it is referenced need to be modified as well; the former change is the root change, while the latter ones are potential casualty changes. When the root change is an added variable, that change is usually accompanied by new code that implements some needed functionality referencing the variable; parts of this new code may constitute a casualty change.

Previous work [14] addressed several scenarios involving variables local to a method. With a couple of exceptions, we will thus focus our discussion on more extensive changes stemming from member variables of a class; the exceptions will zero in on local variable-based changes omitted from [14]. Our taxonomy includes

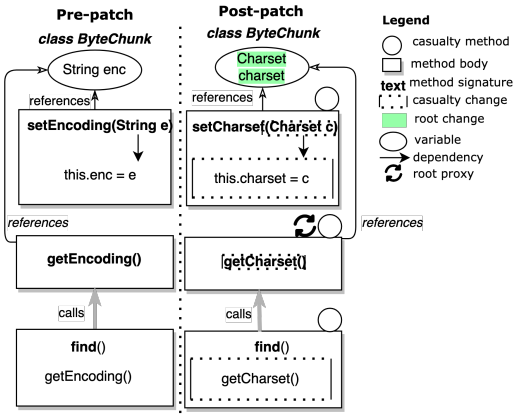

Figure 7: Partial fix of CVE-2012-0022 [3], showing direct and indirect variable-based casualty changes.

additions, deletions, and updates of variables including their types, variable assignment and retrieval methods ("setters" and "getters", respectively), as well as changes to container data structures and/or methods that manipulate such structures.

As an illustration of variable-based casualty changes, consider the patch that fixed CVE-2012-0022 [3] in Tomcat, partially depicted in the top portion of Figure 7. As part of this fix, in the class named `ByteChunk`, a member variable named `enc` of type `String` was updated to Java's type `Charset` and renamed to `charset`. The variable's setter method also had to be updated: its name was changed from `setEncoding` to `setCharset`, the lone parameter was changed in its API, and assignment to the member variable was updated in its body. These updates represent a variable-based casualty change set.

As shown in Figure 3, we categorize variable-based casualty changes along three dimensions: (1) embodiment, (2) proximity, and (3) scope. We elaborate on them next.

1) Embodiment is concerned with the manner in which variable-based casualty changes are manifested. In the case of individual methods, changes to a variable may result in casualty changes in the variable's *local references* within the method's implementation. Prior work [14] addressed a number of cases where casualties are caused by variable additions and removals; it did not deal with variable updates that may cause changes such as type casting.

Beyond changes to local variables, our taxonomy also encompasses member variables of a class. Specifically, changes to member variables may induce casualties in methods that assign or retrieve the variables' values. A method that acts only as the *retriever* of a variable's value is typically referred to as a "getter", and one that acts only as the *assigner* a "setter". For example, `getCharset` in Figure 7 is the getter and `setCharset` the setter for the `charset` variable.

It is possible for methods acting as retrievers or assigners to serve purposes other than specifically getting/setting a variable's value and to include additional, arbitrarily complex functionality. For example, a method that initializes one or more variables in a system is also an assigner; a method that logs the values of certain variables for offline analysis is also a retriever. An example of a more complex retriever is Tomcat's `find` method partially depicted in Figure 7: `find`'s non-trivial functionality includes code that retrieves and returns the values of certain variables without using them in its own internal computations; any changes to such variables will be root changes and will need to be reflected as casualties inside `find`.

2) Proximity refers to the distance of a casualty change from a variable-based root change. Casualty changes can flow directly or indirectly from a root change. In the *direct* case, a change to a variable results in changes to methods that reference the variable. Most often such methods are the variable's getters/setters, but they can be any methods that assign values to and/or retrieve values from the variable. Updates to these methods do not modify the logic of the system, but are necessitated because the variable itself has changed. Changes to the setter method from Figure 7 discussed above are examples of direct casualty changes. Also note that, by definition, casualty changes stemming from a method's own local variables are direct casualty changes.

By contrast, *indirect* variable-based casualty changes are caused by a sequence of one or more calls in the system's callgraph, ending with a method that underwent a direct variable-based casualty change to its API, making this last method a root proxy. Any methods that include calls to the root proxy must have their corresponding callsites updated. Note that indirect variable-based casualty changes apply only in cases of class member variables, and not local method variables whose scope is limited inside a method.

An example indirect variable-based casualty change can be seen in the bottom portion of Figure 7. The method `getEncoding` was changed to `getCharset` because of the change to the class variable (recall the above discussion), and is thus a direct casualty change and root proxy. In turn, the corresponding callsite inside the method `find` had to be updated and is an indirect casualty change.

3) Scope refers to the intricacy of analysis required to unearth the casualty change. Specifically, the analysis can take two forms: singular and compound. The *singular* case includes casualty changes in which the type in a reference to a variable is modified to reflect the change in the root. The analysis required to uncover such changes thus has to perform a singular check: whether the two changes match. Figure 7 shows an example of a singular variable-based casualty change, where both the type and the name of a class variable are updated, and that is reflected in the `setCharset` method.

The *compound* case involves changes to variables of complex types, typically containers. Such changes occur in at least two different scenarios. In one scenario, the class member or local method variable may be of a simple type, but it is used to populate a container elsewhere in the system, e.g., by applying operations that manipulate the variable and store its values. If this variable's type is updated as the root change in a patch, all corresponding operations may need to be updated to reflect this, resulting in a variable-based casualty change. In another scenario, the class member or local method variable is itself a container. Then, updates to this container's type in a patch (the root change) will need to be reflected throughout the system, including the manner in which individual container elements are accessed (the casualty change). In both scenarios, discovering whether the portions of the system that represent and manipulate the variables in question after the application of a patch are functionally equivalent to their original implementations may require sources of knowledge beyond the code alone – another reason we refer to these changes as compound.

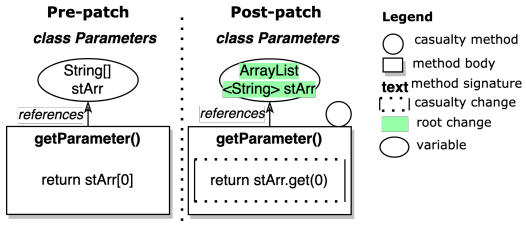

Figure 8: Partial fix of CVE-2012-0022 [3], a depiction of a compound variable-based casualty change.

For illustration, Figure 8 depicts another portion of CVE-2012-0022's fix. The root change modified the type of the container class variable `stArr` from `String[]` to `ArrayList<String>`. In turn, this induced the change in the method `getParameter`, which references the class variable. However, establishing this fact requires understanding the types of the involved variables pre- and post-patch. While these two types are similar, they are indexed in different ways: arrays are indexed directly, while array lists rely on the `get()` method. A human knowledgeable about the two types, would know from Figure 8 that, despite the change, the functionality remains the same since the first element in the container is retrieved both pre- and post-patch. The change to the `getParameter` method is thus a compound variable-based casualty change.

3.4 Refactoring-Based Casualty Changes

Refactoring-based casualty changes result from restructuring the code while preserving its behavior. Sometimes such activities cause additional restructuring changes in the code, which *are* casualty changes. We categorize refactoring-based changes at the level of (1) variables/constants, (2) methods, and (3) classes.

Note from Figure 3 that several of the dimensions of refactoring-based changes—extracting non-local variables and constants, inlining a class, and extracting a superclass—have been greyed out. For the most part, this is because we did not encounter them in our dataset. We will thus not focus on them in this section. One exception is constant extraction, which we did encounter but which has been handled by prior work [14]. Nonetheless, we will briefly describe and illustrate it below. We will then describe the two dimensions of method-level refactoring changes that have not been handled previously: breaking down and cloning method body.

1) Constant extraction is a refactoring activity through which a literal constant is assigned to a named constant and then replaced throughout the system. One such example is the root change highlighted in Figure 9, where the literal constant `1024` is assigned to

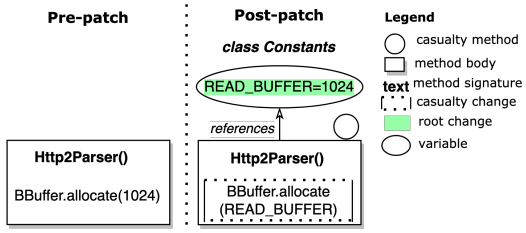

Figure 9: Partial fix of CVE-2016-6817 [7], representing a constant extraction casualty change.

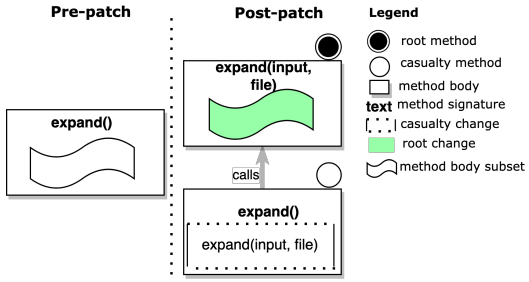

Figure 10: Partial fix of CVE-2012-0022 [3], showing a method breakdown casualty change.

`READ_BUFFER` in the post-patch revision of Tomcat that fixed CVE-2016-6817. All subsequent updates in which `1024` is replaced with `READ_BUFFER` are casualty changes. One such casualty change is the call to `BBuffer.allocate`, shown at the bottom of Figure 9.

2) Method breakdown consists of extracting one method from another by (1) introducing a new method, moving a subset of the original method's code into it, and (3) adding a call inside the original method to the new method. This added call is a casualty change since its entire purpose is to preserve the original functionality. The extracted method is the root method since it contains the intended changes. In the example depicted in Figure 10, a subset of the body of method `expand` has been extracted into a new method, with the same name but different parameters. The added call to the extracted method in the original method is the casualty change.

3) Cloning method body happens when a method is deleted and then the code from its body added to all its previous callsites. In this case, the method deletion is the root change, while the cloning of the method's body is the behavior-preserving casualty change. An example of this type of casualty change is depicted in Figure 11: the method `log` is called in the pre-patch revision by the method `handleQueryParameters`; `log` is deleted by the patch and the call to it is replaced with its actual body.

4 APPROACH

CasCADe is our approach for automatically identifying casualty changes. CasCADe can be tailored to produce different kinds of information about such changes. It can identify a casualty change at various levels of granularity, starting from a line of code. Further, it can pinpoint the root change that induced a given casualty. Finally, it can determine whether entire change sets within methods contain solely casualty changes. This can be particularly useful with sanitizing the datasets of change-based analyses that frequently treat method change sets as a single unit in their models [14, 35].

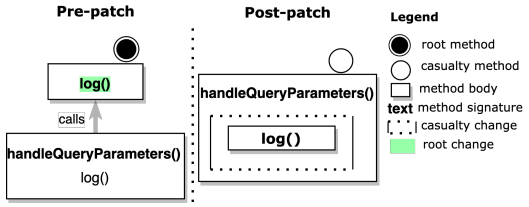

Figure 11: Partial fix of CVE-2009-2693 [2], an example of a casualty change caused by cloned method body.

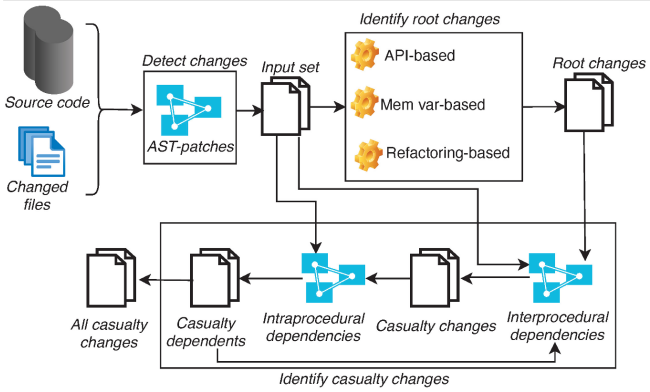

Figure 12: CasCADe's workflow.

CasCADe currently handles the casualty changes identified in our taxonomy from Figure 3, with some exceptions. As mentioned above, for practical reasons we have focused on casualty changes encountered in our datasets, and particularly the types of changes not handled by previous work. Thus, for example, we have not included simple renamings of code elements or changes to local variables. Similarly, CasCADe currently focuses on identifying changes by analyzing source code alone. This excludes the compound variable-based changes, *update operation* and *update container*, whose identification would require additional information, e.g., about data type similarity in a given programming language.

CasCADe's high-level workflow is depicted in Figure 12. We describe it next. We then detail the manner in which CasCADe determines both direct and indirect casualty changes. Finally, we overview the implementation of CasCADe's prototype.

4.1 Overview of CasCADe

CasCADe takes as input two revisions of a system related by a patch—the pre- and post-patch revisions—and the set of files that have changed in that patch. It first compares the abstract syntax trees of the two revisions to identify the differences between them. We refer to the discovered differences as the *input set* to CasCADe's analysis. CasCADe then scans the input set for changes affecting a method's API, a member variable of a class, and any refactored (e.g., extracted or "inlined") portions of a method. If it finds such a change, it marks it as a potential root change and searches for any of the root's dependents among the changed code elements in the input set. For the type of changes CasCADe focuses on, the direct dependents of a root change span method boundaries (recall the examples in Section 3). Specifically, CasCADe leverages inter-procedural data flows (for variable root changes) and call dependencies (for API and refactoring root changes) for this step.

If any root-change dependents are found in the input set, CasCADe determines whether the change affecting the dependent element was induced by the root and, if so, marks the change as a casualty. Specifically, CasCADe first checks for a match in the change type (move, insert, delete, or update) between the root and the change under investigation. Such a match is the first indicator that the changes may be related. If a match is found, CasCADe checks for the evolution of the dependency between the two changes across the two revisions. If the dependency has been preserved, CasCADe concludes that the change under investigation would not have happened without the root change and thus is a potential casualty change. We detail this process in Section 4.2.

At this point, CasCADe attempts to isolate casualty dependents of the initial casualty change by further traversing intra-procedural forward dependencies to the code element of the initial casualty change; any code element in that traversal with a matching change type as the initial casualty is also marked as a casualty change. In the case of API-based changes, CasCADe marks any casualty method whose API has been changed due to a casualty dependent as a potential root proxy (recall Section 3) and repeats the above steps to find indirect casualty changes. CasCADe also repeats the above steps in cases when a variable-based change affects the API of its assigner or retriever. We discuss this process further in Section 4.3.

4.2 Identifying Casualty Changes

A change is tagged as a casualty following the above flow. However, each casualty change category entails certain distinct steps.

In *API-based changes*, the key distinction is how CasCADe determines that the dependency has been preserved between the root change and the candidate casualty change. For singular API-based changes, CasCADe simply checks whether a callsite that is the casualty-candidate code element existed in both the pre- and post-patch revisions. The analogous check for compound API-based changes is more involved. CasCADe must isolate from the input set all methods that have been inserted in the post-patch revision or deleted from the pre-patch revision. It then marks all methods in the pre-patch revision that share names with the above methods. These methods are separated by name into sets that represent candidates for compound API-based changes. In each such set, all methods inserted by a patch are overload*ing* methods, whereas the original, pre-patch method is the overload*ed* method; when a method is deleted by a patch, it is the overload*ed* method, while any remaining methods with the same name are overload*ing*.[1] CasCADe tags each overloading method as a potential root method, and checks each callsite of the original, overloaded method to determine whether it has changed. If the callsite has changed to reflect an overloading method's API, CasCADe marks the callsite as a casualty change.

In member *variable-based changes*, there are two distinct scenarios for casualty change identification. Variable additions and deletions require CasCADe only to check for matching change types between the root and candidate casualty changes. For example, the addition of a member variable would result in the addition of an assigner-retriever method pair. In the case of variable updates, CasCADe must additionally ensure that the dependencies between the root and its potential casualty are preserved. In both cases, CasCADe further needs to confirm that the code element of the candidate casualty change strictly assigns or retrieves the member variable that is the root change. CasCADe does so by building a program dependence graph (PDG) for the candidate casualty method. If the method contains an assignment to the member variable, CasCADe ensures that there are no forward control and data dependencies of the code element in the change; if such dependencies were to exist, the code element would be involved in a program logic-modifying functionality. If the member variable is referenced

[1] Method overriding (recall Figure 3) would be handled similarly, but we have not encountered it in our dataset.

but not assigned in a given code element, CasCADe ensures that (1) there is a `return` statement to which the code element has a forward data dependency and (2) that the member variable is not used or defined in between. If the above conditions hold, then CasCADe marks the change under investigation as a casualty change.

Finally, in *refactoring-based changes*, CasCADe employs a specialized notion of dependency preservation. In the case of method breakdown, it checks whether a call to the extracted method is added in the original method. In the case of cloned method body, CasCADe checks if the pre-patch revision of the method that receives the body contained a call to the now-deleted method. In essence, in the first case, CasCADe "migrates" the dependency from the PDG to the call-graph, and vice-versa in the second case.

4.3 Indirect Casualty Changes

Recall from Section 3 that casualty changes can be indirect, when caused by changes to a method that is itself a casualty. Their root changes can be API-based or variable-based.

In the case of *API-based* indirect changes, CasCADe first groups root proxies with their root methods. To do so, CasCADe selects all methods whose APIs have changed. It constructs a callgraph between these methods, adding only the edges for calls that preserve the dependencies for API-based changes as described in Section 4.2. Note that this callgraph need not be connected, e.g., if it contains multiple API-based casualty changes that involve disjoint methods.

To find each root method and its proxies, CasCADe starts the traversal at a random node. At each node, it identifies the callsites to the nodes accessed via its outgoing edges. For each callsite, CasCADe checks whether the node's corresponding API has changed and is a casualty dependent of the change in the callsite. If so, CasCADe marks the current node as a casualty method and transitions to the next node, following the appropriate outgoing edge. If the next node has no outgoing edges corresponding to the API in question, it is the root; otherwise, CasCADe marks the current node as a root proxy and the above process repeats. Once CasCADe reaches the root, it continues by randomly selecting an unvisited node until all nodes in the callgraph have been processed.

Note that a previously unvisited node may be an API-based casualty of a node that is itself already marked as a casualty; in that case, the latter is additionally tagged as a root proxy. We illustrate this with the example from Figure 5. If CasCADe starts the callgraph traversal at `initializeFilters`, it will originally mark that method as a casualty and `ChunkedInputFilter` as the root method. However, CasCADe would eventually visit `Http11Processor`. CasCADe would mark it as `initializeFilters`'s API-based casualty, and would tag `initializeFilters` as a root proxy. Finally, note that any identified root proxy may have callsites that were not part of the callgraph described above—when the APIs of the methods that enclose such callsites do not change. Those callsites are submitted to the checks described in Section 4.2.

CasCADe approaches *variable-based* indirect casualty changes differently. After identifying casualty dependents as described in Section 4.1, CasCADe checks whether any of them affect the API of their enclosing method. If the API has changed, CasCADe builds a callgraph of all identified changed methods from the input set. It traverses the graph, searching for methods that call the affected casualty and checks for changes affecting the corresponding callsite. Note that, even though they stem from a variable, indirect variable-based casualty changes ultimately affect callsites. As such, CasCADe applies the same check as it would for a singular API-based change, described in Section 4.2.

4.4 CasCADe's Implementation

We implemented an extensible and configurable CasCADe prototype in Java. CasCADe uses the Gumtree [11] tool for obtaining AST-represented patches, and Soot [32] for intra- and inter-procedural program analysis. Lastly, CasCADe uses RMiner [31] for identifying refactoring activities. CasCADe integrates these tools and provides its own capabilities with an additional 11 KSLOC.

Our prototype implements tasks for identifying each type of casualty change in separate modules, with the objective of making it extensible to include additional types. Further, the tool is configurable: a user can choose what types of tasks should be carried out at each step, e.g. identify methods that contain API-based casualties. CasCADe's implementation is available online [9].

5 EVALUATION

Our evaluation aims to establish (1) the viability and accuracy of automatically detecting casualty changes with CasCADe, (2) the prevalence of casualty changes in real patches obtained from three different datasets, and (3) the usefulness of explicitly considering casualty changes in real development scenarios supported by a previously published technique. Due to space constraints, in this section we summarize the key evaluation results. All of our datasets and complete results of our evaluation are available online [9].

5.1 CasCADe's Accuracy

We evaluate the accuracy of CasCADe based on its ability to identify the presence of casualty changes in patches. To this end, we rely on two commonly used metrics, precision and recall. Specifically, we aim to isolate the methods within which casualty changes are contained. The security dataset described in Section 2.2 comprises our ground truth. The dataset totals 1,709 changed methods. Table 1 presents the breakdown of those methods that were marked by CasCADe as *containing* casualty changes ("CC-C") and those whose change sets comprised *only* casualties ("CC-O"). Note that one method may contain casualty changes of multiple categories.

We evaluate how accurately CasCADe identifies both CC-C and CC-O methods. Our results are presented in Table 2. CasCADe

Table 1: Casualty change-containing (CC-C) and casualty change-only (CC-O) methods identified by CasCADe.

	Total	API	Var	Ref
CC-C	386	106	248	39
CC-O	313	85	199	29

Table 2: Accuracy (Precision and Recall) of CasCADe.

	CC-C			CC-O		
	API	**Var**	**Ref**	**API**	**Var**	**Ref**
Precision	0.97	0.77	0.98	1.0	0.88	1.0
Recall	0.9	0.93	1.0	1.0	0.86	1.0

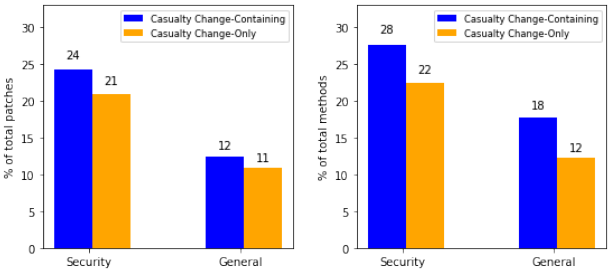

Figure 13: Casualty-containing patches (left) and changed methods (right).

yields high, and in several instances perfect, precision and recall. The accuracy tends to be slightly lower for CC-C methods, since they contain both necessary and casualty changes, and the latter require finer-grained detection. Variable-based changes have the lowest accuracy across the board. This is due to CASCADE's difficulty in identifying more intricate assigner- and retriever-method scenarios due to the limitations of its employed static analyses.

5.2 Prevalence of Casualty Changes

Sections 2.2 and 5.1 discussed the prevalence of casualty changes in a dataset of security patches from six open-source systems that we assembled. We expanded this with two previously published datasets that contain a range of other kinds of software patches. The first dataset [33] contains 231 ZXing [36] patches, 504 AspectJ [1] patches, and 3,163 Tomcat [22] patches.[2] The second dataset [14] contains 3,832 Ant [20] patches and 2,597 Xerces [23] patches. In total, the two latter datasets combined include 10,327 patches resulting in changes to 31,679 methods.

To understand the prevalence of casualty changes across these datasets, we use CASCADE to compute (1) the fraction of patches that have at least one CC-C (resp. CC-O) method, and (2) the fraction of CC-C (resp. CC-O) methods within a patch. The first measure indicates the "breadth" of casualty changes across patches, while the second measure indicates the "depth" of casualty changes in a patch.

The results from our original ("security") and new ("general") datasets are shown in Figure 13. Casualty changes exhibit both breadth across patches and depth within individual patches. The patches from our security dataset and the individual methods within them both have higher proportions of casualties than their counterparts from the general dataset. On the one hand, these results indicate that casualty changes may be more prevalent in security patches. This can be explained by the fact security patches often have to add or modify checks in the code, e.g., to fend off XSS or input validation attacks. In turn, these checks are implemented via member variables that need to be accessed across classes, hence increasing the reliance on and frequency of changes to assigner and retriever methods. Furthermore, we often encountered checks that regulate permissions to access certain data, and those kinds of checks tend to be passed through method parameters.

On the other hand, the prevalence of casualty changes in the general dataset is still significant: of the 10,327 patches, 1,240 contain at least one casualty change and 1,136 contain at least one method that comprises only casualty changes. Furthermore, of the 31,679 modified methods in this dataset, 5,702 contain at least one casualty change and 3,801 contain *only* casualty changes. This strongly suggests that CASCADE will have significant utility and will benefit existing tools that focus on entire patches (e.g., to make recommendations to developers [14] or to pinpoint the source of a bug [33]).

[2]This is a different set of patches than Tomcat's patches in our security dataset.

5.3 Impact of Casualty Changes

Noise has been shown to harm the accuracy of change-based analyses [14]. We hypothesize that casualty changes, as a specific kind of noise, have a similar negative impact. To measure that impact on an existing tool's accuracy we selected AssocChecker, the method-pair association model used for Diffcat [14, 35]. This model recommends to developers methods they should modify next based on a system's change history, and its accuracy was measured using real developer data. We also considered using Locus [33], another related approach whose evaluation dataset we already applied in our analysis, in the same manner. However, Locus's ground-truth data is collected based on a third-party algorithm and it contains casualty changes, which would bias our results and make them difficult to interpret. For this reason, we restrict our comparison to Diffcat.

Given a changed method, Diffcat's AssocChecker model recommends and ranks up to n methods that should be changed next; $n = 10$ in the model's evaluation. AssocChecker's accuracy is based on the quality of the recommendations. First, AssocChecker collects all past patches, with their corresponding sets of changed methods. For each patch, AssocChecker is trained with the sets of changed methods in all patches previous to it, and infers which methods are often changed together. A recommendation is deemed helpful if it involves a method actually changed by the developers.

As mentioned above, we obtained two of the software systems used to evaluate AssocChecker by Kawrykow and Robillard [14]. We chose these two since the source code and patch information was available. In total, we collected 3,832 patches in Ant [20] and 2,597 patches in Xerces [23]; this information was unavailable for the remaining systems [14]. We ran CASCADE on these patches to identify methods that solely contain casualty changes. We used the same number of recommendations, $n = 10$, per method [14]. We then created two separate inputs for the model: one with all the changed methods and one without those methods that contained only casualty changes; since AssocChecker analyzes its data at the method level, we had to include in the latter input all methods that contained combinations of regular and casualty changes. We were thus able to obtain a conservative measure of the impact of casualty changes, via the difference in recommended methods between the two runs.

These results are presented in Table 3. The table shows the originally used AssocChecker metrics [14]: *Tot Rec* refers to the total number of recommendations the model made; *Feedback* refers to the number of methods that received recommendations; *Prec* refers to precision (no recall results are reported in AssocChecker); *Top 3* refers to the rate of changed methods for which there was

Table 3: Results when running AssocChecker with casualty changes (All) and after they have been removed (CC free).

	Tot Rec	Feedback	Prec	Top 3	OE
All	3509	3061	0.22	0.44	0.25
CC free	2515	2331	0.26	0.53	0.17

at least one helpful recommendation among the top 3; and *Only Error (OE)* refers to the rate of methods for which there were no helpful recommendations as reported by developers.

After removing casualty changes, the relative increase in precision was 18% (0.04 on the absolute scale). The relative increase in the rate of helpful recommendations in *Top 3* was 20% (0.09). The rate of changed methods for which there are no helpful recommendations fell by 32% (0.08). Interestingly, the rate of total helpful recommendations (defined by AssocChecker as *Prec* × *Tot Rec*) decreased by 15%, from 771 to 653. However, this decrease is surpassed by the more significant dip in false positives: the substantial increase in *Prec* and *Top 3*, and the decrease in *OE*. In other words, by removing casualty changes, developers will receive fewer but better recommendations by AssocChecker.

We note that AssocChecher and CasCADe ultimately provide complementary analyses. Since AssocChecker uses historical data, it may include in its recommendation a method #2 *every time* another method #1 is changed because they were changed together as a root-casualty pair *at certain times* in the past. However, this approach cannot distinguish scenarios in which changes to method #1 have no relevance to method #2. For example, AssocChecker would not be able to distinguish between a change to a loop internal to method #1, which would not impact method #2, and a change to method #1's API, which would impact method #2. We hypothesize that augmenting AssocChecker's analysis with the information provided by CasCADe would address such differences. Confirming this hypothesis is part of our planned work.

6 LIMITATIONS AND THREATS TO VALIDITY

As indicated previously, for practical reasons we have made certain assumptions and specific implementation choices in our work. In turn, these assumptions and choices have induced several limitations. We discuss the limitations in terms of threats they pose to the validity our results and the corresponding employed mitigations.

1) Internal Validity – A primary threat to our work's internal validity stems from the reliance on manual steps in two parts of our approach: obtaining the ground truth and deriving the taxonomy. Manual work is known to be error prone. To minimize errors and inconsistencies, we performed several iterations of analyzing the dataset when obtaining the ground truth. Moreover, we collaboratively derived and refined the taxonomy, terminating our discussions when all authors agreed with a given categorization.

The implementation of our approach, CasCADe, relies on several third-party tools. Despite their broad use, these tools have limitations that our technique inherits. When feasible, we implemented additional processing of these tools' outputs (e.g., improving the alignment of ASTs in Gumtree) to mitigate their limitations.

2) External Validity – The threats to our work's external validity are related to the taxonomy of casualty changes. First, our dataset, used to derive the taxonomy, currently contains only programs written in Java, an OO language. This leaves unproven the taxonomy's applicability to other paradigms. However, as discussed above, we relied on the authors' experience, which spans a large number of programming languages, to refine the categories of the taxonomy. While we do not have such example-systems in our current dataset, our experience, as well as the taxonomy's constituent elements (e.g., statements, methods), strongly suggest that the taxonomy will also be directly applicable to procedural languages.

We additionally acknowledge that the taxonomy is comprehensive but not complete. Despite this limitation, the taxonomy has served as a fruitful foundation for our work and has yielded high accuracy in applying CasCADe to a representative set of open-source systems. We aim for the taxonomy to be extended further via our own on-going work and its adoption by the research community.

3) Construct Validity – The principal construct validity threats are related to the definition of casualty changes. The foundations of our work rest on correctly identifying changes that, both, cascade from other changes and preserve the logic of a program. As already discussed above, determining the preservation of logic may be non-trivial. In most cases, our ground truth was determined via multiple iterations through the dataset. In both the ground truth and our automated approach, we classified a change as casualty if and only if that change (1) had dependencies with a root or another casualty change and (2) did not have any dependencies with other (non-casualty and non-root) changes.

This approach does not account for certain cases, such as the previously discussed container data types, where the preservation of behavior is "masked" by semantically equivalent but syntactically different code. Such cases may require one-off solutions. Another strategy we are exploring is leveraging implementation-level invariants (e.g., in the form of assertions) to establish behavioral equivalence of a change introduced in a patch with the original functionality.

7 CONCLUSION

Each of the four principal contributions of our work—(1) the taxonomy of casualty changes, (2) CasCADe, (3) the curated dataset of security patches, and (4) the casualty change-based analysis of prior studies' results—can be further extended and enriched. For example, our taxonomy may include additional casualty changes as they are identified, but also other similar phenomena studied by existing work: noise, non-essential changes, low-significance changes, and so on. Along similar lines, the existing datasets can be expanded to include additional systems and to identify different types of noise, possibly simultaneously.

CasCADe itself will also evolve, both by including capabilities it has not needed to date (e.g., because the datasets we have access to do not contain certain kinds of noise) and by integrating with existing tools. One particularly interesting aspect of such an integration would leverage CasCADe's ability to identify noise at different abstraction levels – a capability largely missing from previous approaches. In turn, this opens another research avenue: gaining an understanding of the types of feedback and levels of detail that developers prefer about patches. The exclusive focus of existing approaches on methods may have been a missed opportunity.

ACKNOWLEDGMENTS

This work is supported by a Google PhD Fellowship, the U.S. National Science Foundation under grants 1717963, 1823354, and 2030859 (the Computing Research Association for the CIFellows Project), and the U.S. Office of Naval Research under grant N00014-17-1-2896.

REFERENCES

[1] [n.d.]. AspectJ.
[2] [n.d.]. CVE-2009-2693. https://nvd.nist.gov/vuln/detail/CVE-2009-2693
[3] [n.d.]. CVE-2012-0022. https://nvd.nist.gov/vuln/detail/CVE-2012-0022
[4] [n.d.]. CVE-2013-4322. https://nvd.nist.gov/vuln/detail/CVE-2013-4322
[5] [n.d.]. CVE-2013-4590. https://nvd.nist.gov/vuln/detail/CVE-2013-4590
[6] [n.d.]. CVE-2014-0230. https://nvd.nist.gov/vuln/detail/CVE-2014-0230
[7] [n.d.]. CVE-2016-6817. https://nvd.nist.gov/vuln/detail/CVE-2016-6817
[8] [n.d.]. National Vulnerability Database. https://nvd.nist.gov/
[9] [n.d.]. Website With CasCADe and the data. https://asejfia.github.io/cascade.github.io/
[10] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 134–144.
[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982
[12] Beat Fluri and Harald C Gall. 2006. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 35–45.
[13] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. 2018. ClDiff: Generating Concise Linked Code Differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 679âĂŞ690. https://doi.org/10.1145/3238147.3238219
[14] David Kawrykow and Martin P. Robillard. 2011. Non-Essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 351âĂŞ360. https://doi.org/10.1145/1985793.1985842
[15] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 213–224.
[16] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 715–726.
[17] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.
[18] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.
[19] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E Johnson, and Danny Dig. 2012. Is it dangerous to use version control histories to study source code evolution?. In *European Conference on Object-Oriented Programming*. Springer, 79–103.
[20] Apache Ant Project. [n.d.]. Apache Ant. https://ant.apache.org/
[21] Apache Ofbiz Project. [n.d.]. Apache Ofbiz. https://ofbiz.apache.org/
[22] Apache Tomcat Project. [n.d.]. Apache Tomcat. https://tomcat.apache.org/
[23] Apache Xerces Project. [n.d.]. Apache Xerces. https://xerces.apache.org/
[24] Apache Zookeeper Project. [n.d.]. Apache Zookeeper. https://zookeeper.apache.org/
[25] Commons Collections Project. [n.d.]. Commons Collections. https://commons.apache.org/proper/commons-collections/
[26] Commons Compress Project. [n.d.]. Commons Compress. https://commons.apache.org/proper/commons-compress/
[27] Commons Email Project. [n.d.]. Commons Email. https://commons.apache.org/proper/commons-email/
[28] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 757–762.
[29] Adriana Sejfia and Nenad Medvidović. 2020. Strategies for Pattern-Based Detection of Architecturally-Relevant Software Vulnerabilities. In *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 92–102.
[30] Yida Tao, Donggyun Han, and Sunghun Kim. [n.d.]. Writing acceptable patches: An empirical study of open source project patches. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 271–280.
[31] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 483–494. https://doi.org/10.1145/3180155.3180206
[32] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework *(CASCON '99)*. IBM Press, 13.
[33] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 262âĂŞ273. https://doi.org/10.1145/2970276.2970359
[34] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. 2004. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering* 30, 9 (2004), 574–586.
[35] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.
[36] ZXing. [n.d.]. ZXing. https://zxing.org/