applied sciences

MDPI

Article

CVMan: A Framework for Clone-Incurred Vulnerability Management

Jian Shi [1], Deqing Zou [1,*], Shouhuai Xu [2] and Hai Jin [3]

[1] National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China; shijianwh@hust.edu.cn
[2] College of Engineering and Applied Science, University of Colorado Colorado Springs, Colorado Springs, CO 80918, USA
[3] National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
* Correspondence: deqingzou@hust.edu.cn

Abstract: Software clones may cause vulnerability proliferation, which highlights the importance of investigating clone-incurred vulnerabilities. In this paper, we propose a framework for automatically managing clone-incurred vulnerabilities. Two innovations of the framework are the notion of the *spatial clone-relation graph*, which describes clone-based relationships between software programs, and the *temporal clone-relation graph*, which describes the evolution of clones in software over time. As a case study, we apply the framework to analyze eight versions of Ubuntu while drawing a number of insights, such as: (i) clones are prevalent with about one-sixth of the codebase being clones; (ii) intra-program clones are often attributed to polymorphisms or functional similarities between procedures, while inter-program clones are often attributed to shared code repositories and the reuse of libraries; (iii) the clone surface of Linux remains stable at around 0.6, meaning that spatial and temporal clones in Linux account for about 60% of the codebase, while the lifetime of 53% clones spans eight versions; and (iv) the clone-incurred vulnerability surface in Linux is small, while vulnerable clones and non-vulnerable clones have similar lifetimes.

Keywords: vulnerability; risk management; clone

Citation: Shi, J.; Zou, D.; Xu, S.; Jin, H. CVMan: A Framework for Clone-Incurred Vulnerability Management. *Appl. Sci.* **2023**, *13*, 4948. https://doi.org/10.3390/app13084948

Academic Editor: Luis Javier García Villalba

Received: 4 March 2023
Revised: 8 April 2023
Accepted: 10 April 2023
Published: 14 April 2023

1. Introduction

We are in an era where software is no longer constructed completely from scratch but by combining operating system features, development frameworks, and third-party code or libraries. A study based on 2400 codebases shows that 97% of these codebases contain open-source code and that 81% of these codebases have at least one known vulnerability [1]. This is largely caused by various flavors of software reuse, which is a popular practice in modern software development, perhaps because of the pressures to improve production efficiency and product quality [2].

Another study showed that 70% of the code on GitHub consists of clones [3]. This is not surprising because developers often use software repositories to share code and reuse third-party code, such as PyPI [4] for python code, RubyGems [5] for Ruby code, NPM [6] for JavaScript code, PEAR [7] and PECL [8] for PHP code, and various OS repositories (e.g., Debian [9] and Ubuntu [10]).

In the literature, the following four kinds of software clones have been defined [11,12]:

- **Type 1: exact clones**. In this case, one piece of code is entirely copy-and-pasted from another piece of code, without modifications.

- **Type 2: renamed clones**. In this case, one piece of code is syntactically identical to another piece of code, with possible differences in the variable types, identifiers, comments, and/or whitespace.
- **Type 3: restructured clones**. In this case, one piece of code may be obtained from another piece of code by somewhat modifying program structures, such as deleting, inserting, or rearranging some statements.
- **Type 4: semantic clones**. In this case, two pieces of code may differ in syntax but have the same functionality (i.e., semantic equivalence).

It is known that at least Types 1–3 clones can cause duplications of vulnerabilities [13]. Although there have been studies on detecting clone-incurred vulnerabilities (e.g., [13–16]), these studies do not offer any overall characteristics of the clone landscape. In this paper, we address this problem by investigating how to automate the management of vulnerabilities incurred by code clones.

Our contributions. First, we propose an automated clone-incurred vulnerability management framework to track the risks associated with clone-incurred vulnerabilities. This framework can be used by defenders to understand, characterize, and manage clone landscapes as well as clone-incurred vulnerability landscapes. Second, we apply the framework to conduct a case study on Ubuntu, leading to interesting insights, including: (i) clones are prevalent in most programs, with about one-sixth of the codebase being clones; (ii) intra-program clones are often attributed to polymorphisms or functional similarities between programs, while inter-program clones are often attributed to shared code repositories and reuse of libraries; (iii) most clones remain unchanged or similar across multiple consecutive software versions, while the lifetime of 53% clones spans eight versions in Linux; and (v) the clone-incurred vulnerability surface in Linux is small, while vulnerable clones and non-vulnerable clones have similar lifetime distributions.

The remainder of the paper is organized as follows. Section 2 reviews related prior work. Section 3 describes our framework. Section 4 implements a prototype system. Section 5 presents a case study on the Ubuntu repository. Section 6 discusses the limitations of the present study. Section 7 concludes the paper.

2. Related Work

We divide the related prior studies into the following categories: *characterizing clones* vs. *leveraging clones* for vulnerability detection.

2.1. Characterizing Clones

Clone granularities. Clones have been defined in four granularities: *token level* [17–19], *line level* [16], *procedure level* [20–22], and *file level* [23,24]. The present study chooses the *procedure level* granularity by leveraging existing clone-detection algorithms.

Clone characterization. First, there are studies related to what we call *spatial* clone-relation graphs. Kamiya et al. [17] analyzed clones across three operating systems (FreeBSD, NetBSD, and Linux) and found that FreeBSD and NetBSD have similar source code, but they are different from Linux. Mockus [25] analyzed projects and packages in Linux and BSD and found that more than 50% of the files were used in more than one project. Roy et al. [26] examined open-source C, Java and C# systems and found that 15% of the files in the C systems, 46% of the files in the Java systems, and 29% of the files in the C# systems were associated with exact clones.

Heinemann et al. [27] analyzed 20 open-source Java projects and found that 9 of them had a code reuse rate of 50% or higher. Lopes et al. [3] analyzed github projects and found that 70% of the code consists of clones of previously created files. Koschke and Bazrafshan [28] analyzed C/C++ projects and found that 80% of the projects had at least one Type-2 clone. Lozano et al. [29] assessed the effect of clones on maintenance and showed that clone may increase maintenance effort.

Hotta et al. [30] compared the maintainability of 15 open-source systems with vs. without clones and found that presence of clones had a negative impact on software

evolution. Lozano et al. [31] showed that code clones cause more changes to code, while a similar phenomenon was observed in [32]; in contrast, other studies [33,34] showed that cloned code is more stable. In contrast, we propose a framework for managing the clone-incurred vulnerabilities. Even for the case study of clones, we are the first to investigate the matter of clones in Ubuntu at the procedure granularity level.

Second, there are studies related to what we call *temporal* clone-relation graphs. Kim et al. [35] and Aversano et al. [36] studied clone genealogy in Java projects and found that long-lived clones changed consistently. Saha et al. [37] examined clone group evolution in 17 open-source systems and found that most clone groups changed consistently in the subsequent releases. Zibran et al. [38] studied the patterns of clone changes and removals in six software systems. Thongtanunam et al. [39] investigated six open-source Java systems from the perspective of clone lifetime and found that many clones had a short lifetime.

2.2. Leveraging Clones to Detect Bugs and Vulnerabilities

There are studies on clone-incurred bugs. For example, Sajnani et al. [40] explored the relationship between clones and bug patterns and found that the defect density in cloned code was 3.7-times less than the rest of the code. Rahman et al. [41] analyzed the relationship between cloning and defect proneness and found that the great majority of bugs were not significantly associated with clones. Li et al. [42] found that 4% of the bugs were duplicated across more than one product or file.

Islam et al. [43] studied the evolution of clones and clone-incurred bugs. These studies are primarily from a software engineering perspective. By contrast, we focus on the evolution of clones and clone-incurred vulnerabilities, which are primarily from a cybersecurity perspective. Since these studies are not based on Ubuntu, which is our focus, their results and ours are not comparable; otherwise, it would be interesting to see whether the patterns exhibited by clone-incurred bugs are similar to the patterns exhibited by clone-incurred vulnerabilities or not.

There are studies on leveraging clones to detect vulnerabilities. For example, Li et al. [13] leveraged patches to automatically select clone-detection algorithm(s) suitable for detecting vulnerabilities incurred by clones. Kim et al. [14] generated fingerprints of vulnerable procedures and used a hash approach to detect vulnerable clones. Islam and Zibran [44] studied vulnerabilities in clones in Java programs, and Islam et al. [45] studied the vulnerabilities incurred by clones in C programs by leveraging clone relations similar to what we call spatial clone graphs.

Studies [44,45] also investigated vulnerability severity in different clone types and non-clone code, while noting that their experiments were based on 97 Java programs and 34 C programs. By contrast, we analyze eight Ubuntu versions, meaning that the programs they analyzed and the programs we analyze are different. Moreover, we consider both spatial and temporal clones, which lead to the clone and vulnerability landscape (i.e., overview of the situation regarding vulnerabilities incurred by clones).

3. The Clone-Incurred Vulnerability Management (CVMan) Framework

3.1. Motivation: Practitioners' Questions (PQs)

The NIST Cybersecurity Framework [46] defines a number of functionalities for managing enterprise cybersecurity. One of these functionalities is *identify*, which aims to identify the cybersecurity risks associated with an enterprise network or IT infrastructure. This highlights the importance of knowing the spectrum of risks. In this paper, we focus on the particular risk associated with the software vulnerabilities that are incurred by code clones or *clone-incurred vulnerabilities.* This prompted us to propose the following Practitioners' Questions (PQs):

- **PQ1:** What does the intuitive notion of the *clone landscape* of an enterprise network look like? Although the term clone landscape is intuitive, we need to define it precisely.

- **PQ2:** How does the clone landscape evolve with time? Answering this question allows the requester to keep track of the dynamic clone landscape and manage the dynamic situational awareness of the clone landscape.
- **PQ3:** What are the clone-incurred *vulnerability landscapes* (e.g., which clones are associated with what vulnerabilities)? Addressing this question allows the requester to keep track of clone-incurred vulnerabilities.
- **PQ4:** How should a requester leverage the clone and/or vulnerability landscape to manage the risks of clone-incurred vulnerabilities? For example, what should the requester do when a new vulnerability becomes known (e.g., disclosed by a software vendor)?

3.2. Framework Overview

Figure 1 highlights the proposed CVMan framework for managing clone-incurred vulnerabilities. CVMan has three components: *clone management* (for addressing PQ1 and PQ2), *vulnerability management* (for addressing PQ3 and PQ4), and the *service interface* (for facilitating interaction between a requester and the former two components). In principle, the framework is equally applicable to both source code and binary code. The present study focuses on source code.

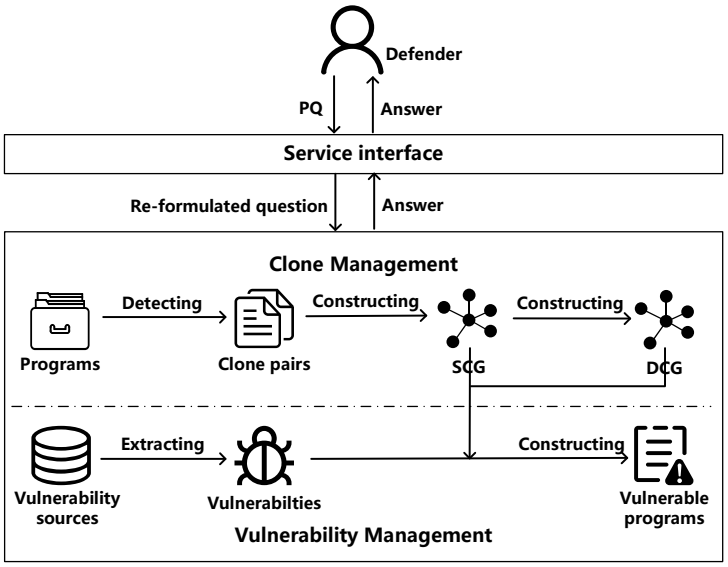

Figure 1. Overview of the CVMan framework.

Terminology. We use the term *program* to represent software that is designed for a specific application or service and is provided by a specific vendor (including open source). For example, we treat Firefox as one program and Linux as another program, while noting that Ubuntu-like software is treated as a set of programs (e.g., Ubuntu consists of multiple programs, including Firefox, Linux, and Thunderbird). Since we focus on programs with source code, a program is a set of source code *files* written in some programming language(s). This means that a *file* consists of a set of *procedures*, where each procedure is an ordered set of statements or lines of code performing certain tasks. Although procedures are often in the form of program functions, we prefer the term procedure because we will use the term *function* to indicate the mathematical functions for computing the similarity between two pieces of code.

3.3. The Clone Management Component

In order to describe and characterize the intuitive concept of a *clone landscape*, as mentioned in PQ1 and PQ2, we need to extract and manage the code clone relations

associated with a given set of software programs. We propose achieving this in two steps: *detecting clone relations* and *constructing clone landscape.* These steps are elaborated below.

3.3.1. Detecting Clone Relations

For this purpose, we first need to determine the notion of *clone granularity* because clones can be defined at multiple levels, such as: file vs. procedure vs. multiple lines of codes (LoCs) vs. single LoC. In order to unify presentations, we use the term *code fragment* to describe a piece of code at a granularity of interest, such as the four granularities mentioned above. Since the framework is equally applicable to any possible granularity, we choose to focus on the procedure as the unit of code clones because this granularity provides both syntactic and symbolic information. In order to make the CVMan framework widely applicable, we make it able to accommodate any clone-detection algorithm. For prototyping purposes, we consider one specific clone-detection algorithm, but other algorithms of the same nature can be incorporated in a plug-and-play fashion. Moreover, the clone-relation-detection sub-component, which is different from the clone-detection algorithm, extracts code fragments from programs at a granularity of interest and then analyzes the clone relations between these code fragments.

Formally, let $P_1, \ldots, P_n$ denote n programs for which we study their clone landscape, where $n \geq 1$. For example, $n = 1$ means the clone landscape between the code fragments in a single given program; $n = 2$ means both the clones within each of the two programs and the clones between the two programs. Each P_i is treated as an ordered set of *statements* (i.e., line of code). For program P_i, where $1 \leq i \leq n$, let $C_{i,1}, \ldots, C_{i,\ell_i}$ denote the ℓ_i code fragments, where each fragment is treated as an ordered set of statements; we assume that these fragments are extracted from P_i by an appropriate pre-processing algorithm. Note that these code fragments do *not* necessarily correspond to a partition of program P_i, meaning it is possible that $P_i \neq C_{i,1} \cup \ldots \cup C_{i,\ell_i}$. Moreover, it is also possible that $C_{i,j_1} \cap C_{i,j_2} \neq \emptyset$ for some $j_1 \neq j_2$ because two code fragments may have one or multiple statements in common. Let $\mathcal{C}$ denote the universe of code fragments.

Given n programs $P_1, \ldots, P_n$, there are many algorithms for detecting clone relations. The CVMan framework can incorporate any algorithm for this purpose as long as an algorithm is based on the following two concepts, which have been widely used in the literature and may guide the design of future algorithms.

One concept is to determine whether a pair of code fragments has a clone relation, namely that one is a clone of the other and vice versa. For this purpose, we need a mathematical function $f : \mathcal{C} \times \mathcal{C} \to [0, 1]$ to measure the similarity between a pair of code fragments. There are many realizations of function f. One example is to measure similarity by treating each code fragment as a set of *tokens* (e.g., identifier, operator, and procedure name) and then to leverage the similarity between their respective tokens [17]. Then, one can measure the ratio between the size of the common tokens and the size of the larger code fragment [19], namely:

$$f(C_{i_1,j_1}, C_{i_2,j_2}) = \frac{\left|C_{i_1,j_1} \cap C_{i_2,j_2}\right|}{\max(\left|C_{i_1,j_1}\right|, \left|C_{i_2,j_2}\right|)} \tag{1}$$

where $|\cdot|$ returns the size of a set.

The other concept is threshold parameter $\tau \in [0, 1]$ for determining when a pair of code fragments are, indeed, a clone. Intuitively, τ reflects the types of clones of interest. For example, when the requester only cares about Type-1 clones, the requester can set $\tau = 1$; whereas, $\tau < 1$ means the requester aims to accommodate code manipulations to clones. It is intuitive that the clone relation is not necessarily transitive.

Definition 1 (clone relation)**.** *A clone relation between a pair of code fragments* $(C_{i_1,j_1}, C_{i_1,j_1})$ *holds if* $f(C_{i_1,j_1}, C_{i_2,j_2}) \geq \tau$*, where* C_{i_1,j_1} *is extracted from program* P_{i_1} *and* C_{i_2,j_2} *is extracted*

from program P_{i_2} using an appropriate pre-processing algorithm, $1 \leq i_1, i_2 \leq n$, $1 \leq j_1 \leq \ell_{i_1}$, $1 \leq j_2 \leq \ell_{i_2}$, and $(i_1, j_1) \neq (i_2, j_2)$ but possibly $i_1 = i_2$ or $j_1 = j_2$.

3.3.2. Constructing a Clone Landscape

We propose defining a clone landscape via two kinds of clone-relation graphs, which are specified by leveraging Definition 1.

- *Spatial Clone-Relation Graph* (SCG): Given programs $P_1, \ldots, P_n$, we can derive a SCG to describe the clone relations between a pair of programs (P_i, P_j), where $1 \leq i \leq n$ and $i \leq j \leq n$. Intuitively, a SCG describes the clone landscape from given snapshots of programs, which may run in an enterprise network. In the case $n = 1$, the resulting SCG would capture the clone relations between the code fragments that are extracted from a given program; whereas $n > 1$ means that the resulting SCG captures not only the clones within each program but also the clones between two programs.
- *Temporal Clone-Relation Graph* (TCG): Given a sequence of programs $P_1, \ldots, P_n$, which correspond to n versions of a program (e.g., n versions of Firefox), we can derive a TCG to describe the clone relations in a single version of the program and the clone relations between two versions of the program. Intuitively, a TCG additionally captures, when compared with SCG, the evolution of clones with time.

Note that the notion of TCG can be naturally extended to include programs other than the sequence of different versions of the same programs (e.g., 1 version of Linux and 10 versions of Firefox). Another extension is to consider the evolution of clone relations in the software stacks running in an enterprise network. In this case, the notion of "version" mentioned above would need to be replaced with the notion of "snapshot" (e.g., the entire set of programs that run in the enterprise at each point in time or during each time interval of interest, such as daily or weekly). Since the extension is straightforward, we choose to present the simpler notion of TCG mentioned above so that we can highlight the ideas while simplifying the presentation. Now, we proceed with formal definitions.

Definition 2 (SCG)**.** *Given a set of pairs of code fragments with a clone relation, namely $\{(C_{i_1,j_1}, C_{i_2,j_2})\}_{1 \leq i_1, i_2 \leq n, 1 \leq j_1 \leq \ell_1, 1 \leq j_2 \leq \ell_2}$ extracted from programs $P_1, \ldots, P_n$, a SCG is an undirected graph denoted by $G_S = (V_S, E_S)$, where $V_S = \bigcup_{1 \leq i_1, i_2 \leq n, 1 \leq j_1 \leq \ell_1, 1 \leq j_2 \leq \ell_2} \{C_{i_1,j_1}, C_{i_2,j_2}\}$ is the set of nodes representing code fragments and $E_S = \{(C_{i_1,j_1}, C_{i_2,j_2})\}$ is the set of edges representing the clone relations.*

As an example of SCG, consider two programs P_1, P_2 (e.g., Firefox 45.0.0 and Thunderbird 38.6.0). Suppose four code fragments are extracted from P_1, namely $C_{1,1}, C_{1,2}, C_{1,3}, C_{1,4}$. Suppose three code fragments are extracted from P_2, namely $C_{2,1}, C_{2,2}, C_{2,3}$. Then, one possible scenario is that $(C_{1,1}, C_{1,2})$ and $(C_{1,3}, C_{1,4})$ are two pairs of clones associated with program P_1, $(C_{2,1}, C_{2,2})$ is a pair of clones associated with program P_2, and $(C_{1,1}, C_{2,3})$ is a pair of clones, respectively, associated with programs P_1 and P_2.

Definition 3 (TCG)**.** *Given a set of n versions of a program, also denoted by $P_1, \ldots, P_n$ for the sake of minimizing notations, the TCG is a graph denoted by $G_T = (V_T, E_T)$, where V_T is the sets of nodes representing code fragments and E_T is the set of edges representing the code pairs. Specifically, G_T is derived as follows:*

- *For each P_i where $1 \leq i \leq n$, there is a SCG corresponding to it, denoted by $G_S^{(i)} = \left(V_S^{(i)}, E_S^{(i)}\right)$, which captures the clone relation between the code fragments of P_i.*
- *For each pair of programs (P_{i_1}, P_{i_2}) where $1 \leq i_1, i_2 \leq n$ and $i_1 < i_2$, there is a SCG corresponding to it, denoted by $G_S^{(i_1,i_2)} = \left(V_S^{(i_1,i_2)}, E_S^{(i_1,i_2)}\right)$, which captures the clone relation between the code fragments of P_{i_1} and the code fragments of P_{i_2}. Note that $V_S^{(i_1,i_2)}$ consists of two sets of nodes, namely one set that corresponds to the code fragments in P_{i_1} and another set that corresponds to the code fragments in P_{i_2}. We denote the former set by $\left[V_S^{(i_1,i_2)}\right]_{i_1}$ and the*

later set by $\left[V_S^{(i_1,i_2)}\right]_{i_2}$. *It is possible that* $V_S^{i_1} \bigcap \left[V_S^{(i_1,i_2)}\right]_{i_1} \neq \emptyset$ *and/or* $V_S^{i_2} \bigcap \left[V_S^{(i_1,i_2)}\right]_{i_2} \neq \emptyset$, *meaning that some code fragment is not only cloned in a single version (say* P_{i_1}*) but also cloned in another version (say* P_{i_2}*).*

- *Let* $V_T^{(i_1)} = V_S^{(i_1)} \bigcup \left(\bigcup_{i_1<i_2\leq n} [V_S^{(i_1,i_2)}]_{i_1}\right)$, *for* $1 \leq i_1 \leq n$, *which is the set of code fragments that belong to* P_{i_1} *and have been cloned in* P_{i_1} *or another version* P_{i_2}. *This leads to a layer-*i_1 *graph* $G_T^{(i_1)} = \left(V_T^{(i_1)}, E_S^{(i_1)}\right)$, *where the edges only describe the clone-relation between the code fragments in* P_{i_1}. *The resulting clone-relation graph is an n-layer network* $G_T = (V_T, E_T)$, *where* $V_T = V_T^{(1)} \cup \ldots \cup V_T^{(n)}$, *and* $E_T = \left(\bigcup_{1\leq i\leq n} E_S^{(i)}\right) \bigcup \left(\bigcup_{1\leq i_1<i_2\leq n} E_S^{(i_1,i_2)}\right)$, *meaning that the edges additionally capture the clone relations between code fragments in different versions of the program.*

Note that, from a Graph Theory point of view, a SCG is a standard graph or network, whereas a TCG is a multilayer network with each layer corresponding to one version of a program. We use Figure 2 to illustrate this distinction via a simple scenario of two versions of a program. Suppose we extract five code fragments from P_1, namely $C_{1,1}, C_{1,2}, C_{1,3}, C_{1,4}, C_{1,5}$. Suppose we extract four code fragments from P_2, namely $C_{2,1}, C_{2,2}, C_{2,3}, C_{2,4}$.

Then, the SCG corresponding to P_1 only contains two clone relations, namely $(C_{1,1}, C_{1,4})$ and $(C_{1,3}, C_{1,5})$ which are indicated by dashed edges; the SCG corresponding to P_2 only contains one clone relation, namely $(C_{2,1}, C_{2,4})$ which is indicated by a solid edge. The SCG corresponding to (P_1, P_2) would capture all the clone relations as shown in Figure 2 but *cannot* distinguish the dashed edges from the solid edges, whereas the TCG representation distinguishes the two kinds of clone relations. As we will see, this distinction allows us to capture the evolution of clones associated with a sequence of versions of a program.

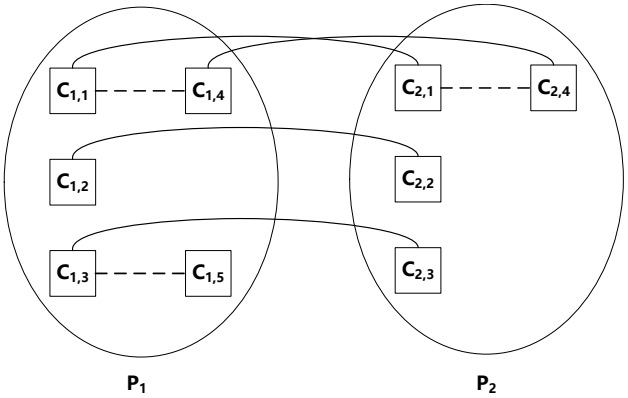

Figure 2. Illustration of TCG of two versions of a program, denoted by P_1 and P_2. The boxes represent code fragments, the dashed edges represent the clone relations between code fragments in a single version of the program (i.e., P_1 or P_2), and the solid edges represent the clone relations between code fragments in different versions of the program (i.e., between P_1 and P_2).

Given the clone relations, it is straightforward to generate SCG and TCG, by simply following their definitions. Having obtained SCG and TCG, we can use them to formalize the intuitive notion of the clone landscape as follows.

Definition 4 (clone landscape)**.** *Given a set of programs* $P_1, \ldots, P_n$, *the clone landscape is defined as:*

- *The SCG that is derived from these programs when no temporal information about the programs is provided (i.e., the programs are for different applications/purposes or provided by different vendors).*
- *The TCG that is derived from these programs when temporal information about the programs is provided (e.g., some or all of the programs are different versions of the programs).*

In order to unify representation, we use the clone landscape to indicate either SCG or TCG, which should be clear from the context.

In order to characterize clone landscapes and their evolution, we propose defining the following metrics to characterize the dynamic situation of clones via the landscape. Let $|P_i|$ denote the number of LoCs of program P_i. Recall that $V_T^{(i)}$ is the set of code fragments belonging to program P_i, including the code fragments that are cloned in P_i and the code fragments that are cloned in P_j where $j \neq i$. For a code fragment v, let $|v|$ denote its number of LoCs.

- **Clone surface**: This metric measures the fraction of code in a program that is cloned, either from another program or by another program. Specifically, for a program P_i, this metric describes the percentage of cloned LoCs in P_i, namely $\frac{\sum_{v \in V_T^{(i)}} |v|}{|P_i|}$.
- **Clone prevalence**: This metric measures how prevalent clones are in a set of programs. Specifically, for a given code fragment C, this metric is defined as the number of clones of C in programs $P_1, \ldots, P_n$.
- **Clone lifetime**: This metric measures the lifecycle from a cloned code fragment. Suppose a given code fragment C appears in the given set of programs $P_1, \ldots, P_2$ at time t_1 at the earliest and at time t_2 at the latest. Then, the lifetime of C with respect to $P_1, \ldots, P_n$ is defined as $t_2 - t_1$.

  Note that the *clone lifetime* metric is only applicable to TCG-based clone landscapes.

3.4. The Vulnerability Management Component

In order to describe and characterize the intuitive concept of *clone-incurred vulnerability landscape* mentioned in PQ3 and PQ4, we propose leveraging the notion of clone landscape. Intuitively, when a vulnerability pertinent to a code fragment (i.e., a vulnerable code fragment) becomes known, the requester can leverage the clone landscape to identify all of the vulnerable clones. As a result, when the patch to a vulnerability becomes available, the requester can possibly patch all of the vulnerable programs; when patches to some or all of the vulnerable programs are not available, the requester can take alternate countermeasures in monitoring and protecting those vulnerable programs (e.g., carefully examining the input to those programs to prevent exploitation of their vulnerabilities). This component has two sub-components: *extracting vulnerabilities* and *constructing clone-incurred vulnerability landscape*.

3.4.1. Extracting Vulnerabilities

Vulnerability information can be extracted from various sources, such as: vulnerability databases, vendor security advisories, bug trackers, and the commit history of software projects. The National Vulnerability Database (NVD) [47] is a public vulnerability database and maintains standardized information about reported software vulnerabilities. Vendor security advisories are issued by vendors and disclose vulnerability details and the affected software versions.

Bug trackers are software application that keeps track of reported software bugs and their patches, typically keeping detailed information about vulnerabilities (also known as security bugs) and patches. Traversing the commit history of a program can also allow one to identify and extract vulnerability information. We implemented multiple crawlers to extract vulnerability information from multiple sources, such as crawlers for MFSA, crawlers for NVD, crawlers for USN, etc. We extracted the programs, program versions affected by vulnerabilities, and vulnerabilities' patches.

3.4.2. Constructing Clone-Incurred Vulnerability Landscape

The preceding sub-component extracts each vulnerability as a piece of vulnerable code, which is treated as a vulnerable code fragment. Given a set of vulnerabilities (i.e., vulnerable code fragments) obtained by the preceding sub-component, this sub-component

constructs clone-incurred vulnerability landscape by identifying and annotating the nodes (representing code fragments) in clone landscape with the pertinent vulnerabilities. Ideally, the vulnerability information should be as detailed as possible, including what the affected programs and their versions are, what the patches are, and how they can be exploited. In practice, vulnerability information is often incomplete. For example, the requester may only be given the affected program versions.

Given a set of vulnerabilities, which are vulnerable code fragments extracted by the preceding sub-component, we need to determine whether a code fragment corresponding to a node in the clone landscape is vulnerable or not. For this purpose, there can be many algorithms. For example, one can require that two code fragments are similar enough (i.e., they can be seen as clones of each other); another method is to require that the code fragment contains all of the statements of the vulnerable code fragment; a conservative method is to treat a code fragment as vulnerable if it contains any statement of the vulnerable code fragment. This leads to:

Definition 5 (clone-incurred vulnerability landscape)**.** *Given a clone landscape derived from programs $P_1, \ldots, P_n$ and a set of vulnerabilities that can be treated as vulnerable code fragments, the* clone-incurred vulnerability landscape *is defined as the annotated clone landscape, such that each node in the clone landscape, if deemed as vulnerable according to the given vulnerabilities, is annotated with the information of the pertinent vulnerability.*

In order to characterize clone-incurred vulnerability landscapes and their evolution, we propose defining the following metrics to characterize the dynamic situation of clone-incurred vulnerabilities via the landscape. Let $V_L^{(i)}$ be the set of nodes in the clone landscape, which correspond to the code fragments belonging to program P_i. Let $V_{L'}^{(i)} \subseteq V_L^{(i)}$ be the subset of nodes in the clone landscape, which correspond to the *vulnerable* code fragments belonging to program P_i. Recall that $|P_i|$ is the number of LoCs of program P_i and that for a code fragment v, $|v|$ denotes its number of LoCs.

- **Clone-incurred vulnerability surface**: This metric measures the fraction of code in a program that is cloned and vulnerable. Specifically, for a program P_i, this metric describes the percentage of cloned LoCs in P_i where the clone is also vulnerable, namely $\frac{\sum_{v \in V_{L'}^{(i)}} |v|}{|P_i|}$.
- **Clone-incurred vulnerability prevalence**: This metric measures how prevalent clones are in a set of programs. Specifically, for a given vulnerable code fragment C, this metric is defined as the number of clones of C in the programs $P_1, \ldots, P_n$.
- **Clone-incurred vulnerability lifetime**: This metric measures the lifecycle from a cloned vulnerable code fragment. Suppose that a given vulnerable code fragment C appeared in the given set of programs $P_1, \ldots, P_2$ at time t_1 at the earliest and at time t_2 at the latest. Then, the lifetime of vulnerable clone code fragment C with respect to $P_1, \ldots, P_n$ is defined as $t_2 - t_1$.

Note that the *clone-incurred vulnerability lifetime* metric is only applicable to TCG-based clone landscapes.

3.5. The Service Interface

The service interface allows one to interact with CVMan by asking questions, such as the aforementioned PQ1–PQ4. For each question, a request is accompanied with a number of parameters. The service interface may re-formulate the question according to its interface with the clone or vulnerability management component and then determines which component or sub-component should be given the re-formulated question, collects the answer from the clone or vulnerability management component, and translates the answer to a format that may be easier for the requester to understand. Moreover, the service

interface may be extended to incorporate appropriate visualization tools for presenting the answer to the requester. Specifically, the interface can offer the following services.

In order to answer PQ1, the interface takes the input provided by the requester, which includes a set of programs. Then, the interface feeds the input to the clone management component, which detects clones, as discussed above, and formulates the corresponding clone landscape, represented by a SCG. The clone management component reports back the metrics that are measured from the SCG, namely the clone surface of each program and the clone prevalence of each cloned code fragment in the set of programs.

In order to answer PQ2, the interface takes the input provided by the requester, which includes a set of n versions of some program(s). Then, the interface feeds the input to the clone management component, which detects clones and constructs the corresponding clone landscape TCG. The clone management component would report back the metrics that are measured from the TCG, namely the clone surface of each program, the clone prevalence of each cloned code fragment in the set of programs, and the clone lifetime. These metrics can be leveraged to conduct further analyses.

In order to answer PQ3, the interface takes the input provided by the requester, which includes a set of n versions of some program(s). The interface feeds the input to the clone management component, which detects clones and constructs the corresponding clone landscape TCG. Then, CVMan extracts vulnerabilities from vulnerability sources, annotates the code fragments that have at least one line in common with a vulnerable code fragment, and, finally, outputs these vulnerable code fragments, as well as the programs that belong to the requester.

In order to answer PQ4, the requester is given a specific vulnerability (for example) by a trusted third party. The requester would want CVMan to identify all of the vulnerable programs that contain clones of the given vulnerable code. The interface takes the new vulnerability and feeds it to the vulnerability management component. Then, CVMan annotates the code fragments in the existing clone landscape that have at least one line in common with a vulnerable code and, finally, outputs these vulnerable code fragments (programs) and TCG with annotated vulnerability.

4. Implementing CVMan Prototype System

Based on the CVMan framework described above, we implemented a prototype system. The prototype system cannot achieve end-to-end automation yet because there are some tasks that are currently done manually. This means that further development is needed in order to achieve end-to-end automation in clone-incurred vulnerability management. Nevertheless, the current implementation of the prototype system is sufficient to demonstrate the idea of CVMan and its usefulness.

4.1. The Clone Management Component

In the prototype system, this component is completely automated. As described in the framework, this component has two sub-components. First, the clone-relation-detection sub-component detects clone relations at the procedure granularity. To extract procedures from the programs, we employed Universal Ctags [48], an accurate and fast open-source regular expression-based parser. SourcererCC is leveraged to perform clone detection. Second, the clone landscape construction sub-component takes the clone relations as input to construct the clone landscape representation in SCG and TCG. In order to answer PQ1 and PQ2, procedures are designed to compute, for example, the clone surface metric, the clone prevalence metric, and the clone lifetime metric.

4.2. The Vulnerability Management Component

As described in the framework, this component has two sub-components. First, the vulnerability extraction sub-component is not completely automated, because patches exist on different websites. This sub-component includes multiple crawlers, which extract vulnerability information from multiple sources, such as the National Vulnerability

Database (NVD) [47], Mozilla Foundation Advisories (MFSA) [49], Ubuntu Security Notices (USN) [50], and Ubuntu CVE Tracker [51].

For each vulnerability, the crawler extracts the following attributes: the vulnerable programs, the vulnerable versions of the programs, and the patches. However, there are vulnerability sources that do not provide a structural representation of vulnerability information. This means that, in order to achieve complete coverage of vulnerability sources, this component in the current prototype needs to be extended to accommodate such unstructured vulnerability information, which is often provided by the vendors' security advisories. This could be achieved by leveraging natural-language-processing techniques.

Second, the clone landscape annotation sub-component is completely automated. This sub-component is implemented as a standard patch parser and an annotating procedure. The patch parser parses patches in UNIX standard format for extracting vulnerable lines of code (i.e., the fragment that includes several lines of code that changed). With the vulnerable lines of code, the annotating procedure annotates fragments in the clone landscape graph with the pertinent vulnerabilities, which is achieved by searching for fragments that have at least one line in common with a vulnerable line of code.

4.3. The Service Interface

In the prototype system, the service interface is a control procedure that parses the input parameters from the requester, calls the corresponding sub-component, and then feeds the results back to the requester. Figure 3 highlights this system in a flow chart.

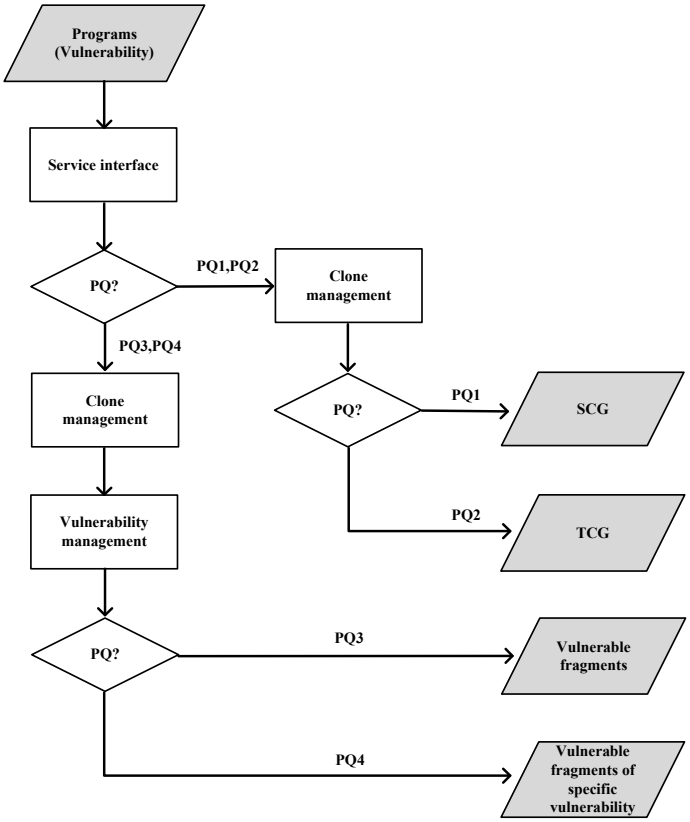

Figure 3. The flow chart of the CVMan prototype system.

5. Case Study: Applying CVMan to Analyze Clones in Ubuntu

In order to demonstrate the usefulness of CVMan, we conducted a case study by applying it to $n = 8$ versions of Ubuntu. We chose Ubuntu because it is representative of modern software in terms of its volume (about one hundred million LoCs), complexity (con-

taining various types of software components and complex reuse between the components), significance (widely used), and open-source nature (source code availability).

5.1. Experiment with the Ubuntu System

Our experiments were conducted on a server equipped with two Intel Xeon E5-2630V3 CPUs (16 cores in total) running at 2.40 GHz, 16 GB memory, and with 4 TB hard drives. The server ran the CVMan prototype system.

The Programs That Are Analyzed in the Experiments: Ubuntu Repository

Ubuntu is a Linux distribution based on Debian and composed mostly of free and open-source software. Ubuntu is stored in four repositories: *main*, *restricted*, *universe*, and *multiverse*. The main repository contains free and open-source software (with the source code) supported by the Ubuntu team. The restricted repository contains proprietary drivers for various devices. The universe repository contains free and open-source software maintained by the community with source code). It is worth mentioning that the Ubuntu team does not guarantee to provide prompt security updates for Ubuntu in the universe repository but will provide them when security updates are made available by the community.

The multiverse repository contains software that is not free. We studied the default software in the main and universe repositories of Ubuntu release for the desktop AMD64 architecture because it is perhaps the most widely used. Specifically, we selected the $n = 8$ Ubuntu releases between 2018 and 2021. We downloaded the Ubuntu source images and manifest files from the Ubuntu release website at http://old-releases.ubuntu.com/releases (accessed on 1 May 2022). The source images contain the source code, and the manifest file is the default package name and version included in an Ubuntu release.

According to the package information in the manifest file, we used a crawler to obtain the program names corresponding to the packages and to extract the source code of programs from the source images. We used command `dpkg-source -x packagename` to extract the source code of programs. Then, we cleaned the source code of the programs: (i) discarding the programs that did not include any files with the suffixes .c, .C, .cpp, .Cpp, .h, .hpp, or .c++; (ii) discarding the files that did not have the suffixes .c, .C, .cpp, .Cpp, .h, .hpp, and .c++; and (iii) removing whitespaces and comments. Table 1 presents the basic information about the $n = 8$ versions of Ubuntu, including the number of C/C++ programs and the number of lines of C/C++ code in each version. In total, the dataset consists of 5760 programs, including more than 800 million LoCs.

Table 1. Basic information about the $n = 8$ Ubuntu versions.

Version	# C/C++ Programs	# Lines of C/C++ Code
18.04	704	100,985,002
18.10	708	98,274,202
19.04	714	103,083,366
19.10	714	101,017,066
20.04	721	109,634,409
20.10	727	109,208,168
21.04	735	113,759,907

We detected clone relations in eight Ubuntu versions. If we take Ubuntu 18.04 as an example, there are 704 C/C++ programs, and the SCG has 337,837 nodes (i.e., fragments) and 4,761,650 edges. Since the experiment deals with Ubuntu, the experiment automatically collected vulnerabilities from: (i) the Ubuntu Security Notices (USN) [50], which tracks vulnerabilities in all Ubuntu releases; (ii) the Ubuntu CVE Tracker [51], which lists the notices when a security issue is fixed in an official Ubuntu program. and (iii) the National Vulnerability Database (NVD), which contains vulnerability reports.

5.2. Answering PQ1

Taking Ubuntu 18.04 as an example, in order to answer PQ1, the interface feeds the programs in Ubuntu 18.04 to the clone management component, which detects clones and constructsthe clone landscape, represented by a SCG. The clone management component would report back the metrics that are measured from the SCG, including the clone surface of each program and the clone prevalence of each cloned fragment in Ubuntu 18.04. The SCG of Ubuntu 18.04 has 337,837 nodes (i.e., fragments) and 4,761,650 edges.

5.2.1. Clone Surface

In Ubuntu 18.04, the number of clone LoCs is 18,592,560. The clone surface of Ubuntu 18.04 is 0.18. Among the 704 programs, 627 programs contain clones, with a 0.07 median clone surface (i.e., one half of the programs have a clone surface that is higher than 0.07).

Table 2 lists the top ten programs with the largest clone surface in Ubuntu 18.04. Among these, the program with the largest clone surface value is Fftw3, a C subroutine library utilized for computing the discrete Fourier transform. The clone surface value for Fftw3 is 0.83, which can be attributed to the presence of several similar procedures designed to handle arbitrary input sizes as well as real and complex data. Similarly, SuiteSparse, a library utilized for sparse matrix computations, has a clone surface value of 0.66. Clones in Fftw3 and SuiteSparse are primarily found within the program, which we term as intra-program clones. These clones arise due to the presence of functionally similar procedures within a program, such as polymorphism mechanisms in the C++ language or similar algorithms. Consequently, these functionally similar programs contain the same syntax structures and variables.

Another type of clone occurs between programs, which we term as inter-program clones. Clones in Libffi, Libart-lgpl, Bzip2, Libogg, Ldb, Libvpx, and Hunspell programs are mainly inter-program clones. The shared code repository is the cause of several inter-program clones. For example, the presence of several inter-program clones in Mozilla Application Suite, which encompasses Firefox, Thunderbird, and SpiderMonkey (Mozjs), can be attributed to the use of the same Mozilla source code repository. Additionally, the reuse of libraries is another contributing factor. For instance, Libffi 3.2.1 is available as a separate program in Ubuntu 18.04 and is included in Python 2.7, GCC-7, and GCC-8, whereas another version of Libffi 3.1 is included in Python 3.6, Thunderbird, Firefox, and Mozjs52.

Table 2. Top 10 programs in Ubuntu 18.04 ranked by the clone surface.

Program	LoC	Clone LoC	Clone Surface
Fftw3	264,365	219,272	0.83
M2300w	2260	1685	0.75
Libffi	25,498	17,070	0.67
Libart-lgpl	10,754	7245	0.67
Bzip2	5837	3824	0.66
SuiteSparse	539,437	358,110	0.66
Libogg	2637	1588	0.60
Ldb	67,599	39,641	0.59
Libvpx	296,977	166,979	0.56
Hunspell	18,520	10,362	0.56

Figure 4a presents the cumulative distribution function (CDF) of the clone surfaces for programs in Ubuntu 18.04. It shows that 35% of programs have a clone surface larger than 0.10, and some programs even have a clone surface greater than 0.50. Such high levels of code cloning can adversely affect the maintainability of the codebase. Figure 4 also illustrates the CDF of inter-program clone surface, revealing that 36% of programs have no inter-program clones, while 20% of the programs have an inter-program clone surface

larger than 0.10. When addressing bugs in these programs, developers should check for the presence of clones in other programs.

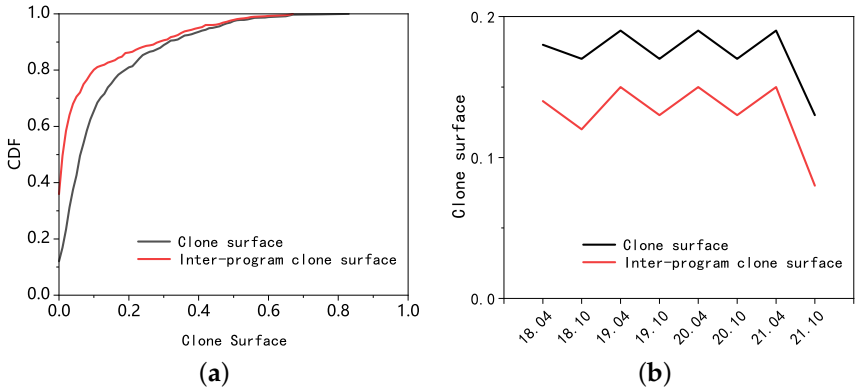

Figure 4. Clone surface. (**a**) CDF of the clone surface for programs in Ubuntu 18.04. (**b**) The clone surfaces for eight Ubuntu versions.

Furthermore, the clone surface for each version of Ubuntu was acquired by utilizing eight Ubuntu versions as input. As depicted in Figure 4a, the clone surfaces for each Ubuntu version are shown, and the changes in the clone surface and inter-program clone surface remain consistent. Notably, the clone surface in Ubuntu version .04 is larger than the clone surface in Ubuntu version .10, mainly due to the existence of two or more versions of GCC in Ubuntu version .04. This suggests that clones in Ubuntu are relatively stable, with the exception of Ubuntu 21.10, where the absence of Firefox source code in Ubuntu 21.10 results in a lower clone surface and inter-program clone surface.

5.2.2. Clone Prevalence

The clone prevalence describes the number of clones in one fragment. We obtain the code prevalence from the SCG of Ubuntu 18.04. In Ubuntu 18.04, the clone with the maximum clone prevalence is the *at_impl* procedure, located in the file *boost/fusion/container/vector/detail/cpp03/preprocessed/vector40.hpp* in the boost program, with a clone prevalence of 1558. This clone's high clone prevalence is attributed to the presence of numerous procedures with similar functionality in the same file and directory. The code for these procedures is automatically generated and is essentially similar in structure and function. On the other hand, the minimum clone prevalence is 2, indicating that the procedure has only one clone, such as the *snd_pcm_plugin_build_copy* procedure in alsa-driver and Linux.

Insight 1. *Clones are prevalent, with about one-sixth of the code being cloned. Intra-program clones are often attributed to polymorphisms or functionally-similar procedures, while inter-program clones are often attributed to shared code repositories and the reuse of libraries.*

5.3. Answering PQ2

Due to the large number of programs in each Ubuntu version, the construction of a TCG becomes time-consuming. To illustrate the process of constructing a TCG, we employ the Linux program as an example. In order to answer PQ2, the interface passes eight Linux versions to the clone management component, which detects clones and constructs the corresponding clone landscape TCG. The clone management component reports the metrics measured from TCG, including the clone surface of each program, the clone prevalence of each cloned code fragment, and the clone lifetime. In TCG, there are 870,914 nodes (i.e., fragments) and 5,073,824 edges. For the sake of clarity in presenting the program version sequence, we opted to utilize the corresponding Ubuntu version to denote the eight versions of Linux, rather than specifying their individual version numbers.

5.3.1. Clone Surface

Figure 5 shows the clone surface for Linux in each Ubuntu version. The largest clone surface is 0.63 in Ubuntu 18.10. Figure 5 reveals that the clone surface in TCG remains stable at around 0.6. This means that most of the code in subsequent versions is still similar to the previous version. For example, the procedure *memset* in the file *arch/microblaze/lib/memset.c* is similar in each Linux version.

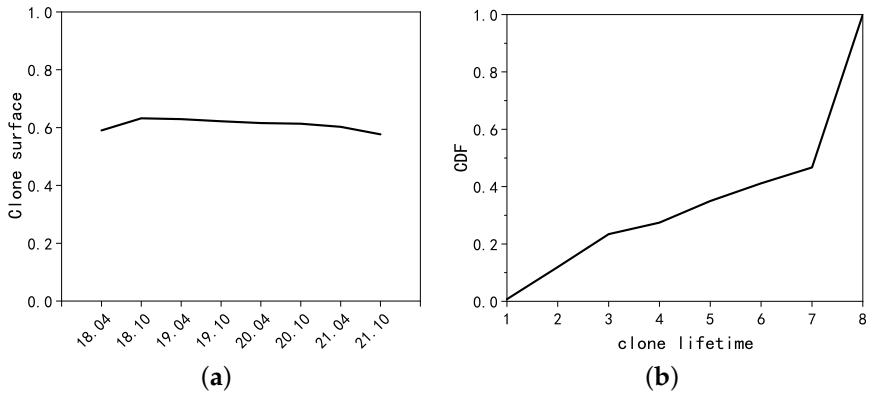

Figure 5. The clone surface and clone lifetime in the Linux clone landscape. (**a**) The clone surface for Linux. (**b**) The CDF for the clone lifetime in Linux.

5.3.2. Clone Prevalence

Recall that the clone prevalence describes the number of clones in eight Linux versions. We obtained the clone prevalence from the TCG of Linux in Ubuntu 18.04-21.10. However, the results need to be filtered because some opcodes and data are present in C/C++ code, and these code fragments bring a large number of clone relationships. After filtering, the maximum clone prevalence was observed to be 117 for the *ia_css_process_sdis2_horiproj* procedure in the *ia_css_isp_params.c* file—the reason being that this file is an automatically generated driver file in which many procedures are similar. The majority of procedures (97%) had a clone prevalence of less than 8, while 67% of procedures exhibited a clone prevalence of 8 due to the fact that most of the procedures are similar between the eight versions.

5.3.3. Clone Lifetime

Recall that the clone lifetime describes the duration of the existence of clones in Linux. To facilitate comparison of clone lifetime lengths, it is customary to employ the number of versions of a clone that exist rather than the actual duration of its existence. Figure 5b presents the cumulative distribution function (CDF) of the clone lifetime in Linux in Ubuntu 18.04-21.10. A total of 53% of the clone lifetime lengths are for eight versions, indicating that the corresponding procedures are similar across the eight consecutive versions.

Insight 2. *In multiple versions of the program, most of the clones are unchanged or similar. In Linux, 53% of the clone lifetime lengths are for eight versions.*

5.4. Answering PQ3

In order to answer PQ3, the interface takes eight Linux versions as input. The interface feeds the input to the clone management component, which detects clones and constructs the corresponding clone landscape TCG. Then, CVMan annotates the procedures that have at least one line of code in common with a vulnerable code fragment and, finally, outputs these vulnerable procedures. Across eight Linux versions (Ubuntu 18.04-21.10), we matched 299 CVEs, and these vulnerabilities were distributed across 310 files. For example, 473 nodes (i.e., procedures) of Linux in Ubuntu 18.04 contained vulnerable code.

5.4.1. Clone-Incurred Vulnerability Surface

Figure 6a shows the clone-incurred vulnerability surface for Linux in each Ubuntu version. The clone-incurred vulnerability surface was about 0.0013, which decreased after Ubuntu 20.10, which is likely due to the fact that there are still vulnerabilities that remain undetected. The results indicate that there are fewer clone-induced vulnerabilities in Linux.

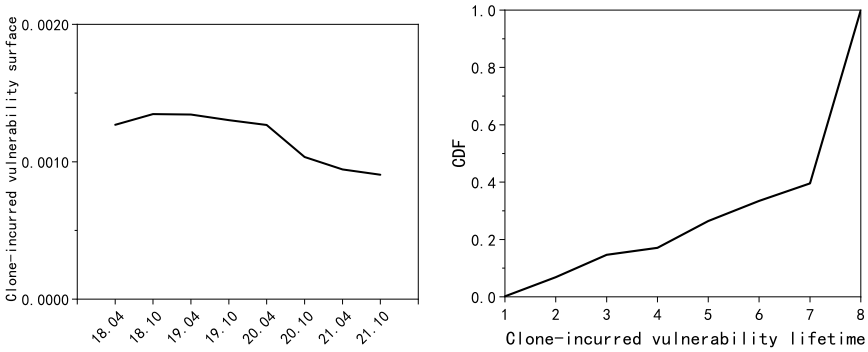

Figure 6. The clone-incurred vulnerability surface and clone-incurred vulnerability lifetime in Linux. (**a**) The clone-incurred vulnerability surface for Linux. (**b**) The CDF for the clone-incurred vulnerability lifetime in Linux.

5.4.2. Clone-Incurred Vulnerability Prevalence

The clone-incurred vulnerability prevalence describes the number of clones of a vulnerable procedure in eight Linux versions. The code prevalence can be obtained from the vulnerability landscape of Linux. There are a total of 526 vulnerable clones, where 58% of the vulnerable clones have a prevalence of 8 and 98% of the vulnerable clones have a prevalence of less than 8. This means that most of the vulnerable clones exist in eight consecutive Linux versions.

5.4.3. Clone-Incurred Vulnerability Lifetime

Figure 6 plots the lifetime of 526 vulnerable clones. A total of 60% of the vulnerable clones had a lifetime of 8. These clones contained at least one line of vulnerable code extracted from the vulnerability patch. Very few vulnerabilities affect only one Linux version; therefore, when a vulnerability is discovered, it is necessary to verify that the clone is affected by this vulnerability in the consecutive versions. Compared with Figure 5b, we find that the CDF of the clone-incurred vulnerability lifetime is approximate to the overall clone's lifetime. The number of vulnerable clones is much less than the number of non-vulnerable clones, which indicates that vulnerable clones and non-vulnerable clones have similar lifetimes.

Insight 3. *Most vulnerable clones exist in multiple consecutive versions. Vulnerable clones and non-vulnerable clones have similar lifetime distributions.*

5.5. Answering PQ4

When a new vulnerability is disclosed, the interface forwards it to the vulnerability management component. Then, the vulnerability management component extracts the vulnerable lines of code of the patch and annotates the clone landscape of the existing programs. Finally, CVMan outputs the vulnerable clones and the TCG with the annotation. Defenders can check the annotated code fragments and their clones and can verify whether they are affected by the vulnerability. Then, defenders can take appropriate defensive measures, such as fixing the vulnerability or performing input validation.

For example, the vulnerability CVE-2020-8648 was reported in the NVD. This is a use-after-free vulnerability in Linux, which is located in procedure *n_tty_receive_buf_common* in

the file *drivers/tty/n_tty.c.* CVMan annotated the code snippet in the existing clone landscape and found that Linux in Ubuntu 18.04-20.04 was affected by this vulnerability.

Table 3 presents examples of clone-incurred vulnerabilities. For example, CVE-2019-10638 affects procedure *__ipv6_select_ident*, and CVMan identified three vulnerable procedures in three Linux versions based on Linux TCG. As another example, corresponding to vulnerability CVE-2019-15214, CVMan identified 10 clones in multiple Linux versions based on Linux TCG and found clones in Alsa-driver, which is based on Ubuntu SCG.

Table 3. Examples of clone-incurred vulnerabilities.

CVE	Program	Vulnerable Procedure	Clone-Incurred Vulnerability Prevalence	Clone-Incurred Vulnerability Lifetime
CVE-2019-15214	Linux, Alsa-driver	*snd_card_disconnect*	10	5
CVE-2019-10638	Linux	*__ipv6_select_ident*	3	3
CVE-2019-11487	Linux	*buffer_pipe_buf_get*	3	3
CVE-2020-8648	Linux	*paste_selection*	5	5
CVE-2020-9383	Linux	*set_fdc*	5	5
CVE-2020-7053	Linux	*context_idr_cleanup*	3	3
CVE-2018-15471	Linux	*xenvif_set_hash_mapping*	2	2
CVE-2018-14625	Linux	*__vhost_vsock_get*	2	2

6. Discussion

Limitations of the CVMan Framework. The framework has two limitations. First, we focused on managing clone-incurred vulnerabilities in programs with source code. This is applicable to the setting where the defender of an enterprise network uses open-source software or proprietary software developed by the enterprise. This is also applicable to vendors of proprietary software, such as Microsoft, who can leverage CVMan to manage clone-incurred vulnerabilities in the lifecycle of their software.

Nevertheless, it would be interesting to extend the investigation to the following setting: The defender only has the binary code (and not the source code) of proprietary software (e.g., Windows) but may still be interested in managing the clone-incurred vulnerabilities in the binary code without having access to the source code. This is relevant because: (i) there are binary-code-based clone detectors [52–54], and (ii) some vulnerabilities are only disclosed with binary code information (i.e., no information about the vulnerable source code is given), for which a concrete example is CVE-2020-4345.

A defender can use binary code-based clone detectors to extract clone pairs and construct an SCG and TCG to describe the clone landscape. Then, the defender can use an appropriate source-code-to-binary-code mapping method (when source patches are known) or can compare the binary file before applying a patch and the binary file obtained after applying a patch to locate and annotate vulnerable code fragments.

Second, the vulnerability management component could be improved. This is because, when constructing a clone landscape, vulnerabilities are not considered. As a consequence, a code fragment that has one line of code that is similar to one line of code in a piece of vulnerable code is treated as vulnerable. However, whether the code fragment is indeed vulnerable needs to be further investigated. An alternate solution is to treat the code fragment that contains the lines of code removed by the patch as the vulnerable code fragment. However, this solution is conservative and could miss some vulnerable code fragments because vulnerable code fragments in different versions do not necessarily contain exactly the same lines of code removed by the patch.

Limitations of the Case Study. We only applied TCG to Linux in Ubuntu. Despite that Ubuntu is complex software (i.e., millions of lines of code for each version) and that we considered eight versions of it, the case study may not be sufficiently representative. As a consequence, the insights drawn from the case study may not hold for other programs.

7. Conclusions

We presented an automated clone-incurred vulnerability management framework. This framework can be used to understand, characterize, and manage both the clone landscape and the clone-incurred vulnerability landscape, so as to guide defenders to take corresponding actions when new vulnerabilities are discovered. We conducted a case study on Ubuntu and drew a number of insights.

Author Contributions: Conceptualization, J.S., D.Z. and S.X.; methodology, J.S., D.Z. and S.X.; software, J.S.; data curation, J.S.; writing—original draft preparation, J.S., D.Z. and S.X.; writing—review and editing, J.S., D.Z., S.X. and H.J. All authors have read and agreed to the published version of the manuscript.

Funding: The research of the authors affiliated with Huazhong University of Science and Technology was funded by the National Natural Science Foundation of China under Grant No. 62172168. The work of S. Xu was supported in part by NSF Grants #2122631 and #2115134 and Colorado State Bill 18-086.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are available by request from the corresponding author.

Acknowledgments: The authors thank the editors and anonymous reviewers for their helpful suggestions on how to improve the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sonatype. *2022 Open Source Security and Risk Analysis Report*; Technical Report; Sonatype: Fulton, MD, USA, 2022.
2. Shi, Z.; Tang, J.; Yu, H.; Xing, Y.; Liu, Z.; Bai, W.; Li, T. A quantitative benefit evaluation of code search platform for enterprises. *Sci. China Inf. Sci.* **2020**, *63*, 1–3. [CrossRef]
3. Lopes, C.V.; Maj, P.; Martins, P.; Saini, V.; Yang, D.; Zitny, J.; Sajnani, H.; Vitek, J. DéjàVu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang.* **2017**, *1*, 1–28. [CrossRef]
4. Python Software Foundation. PyPI · The Python Package Index. Available online: https://pypi.org/ (accessed on 27 February 2023).
5. Ruby Community. RubyGems.org. Available online: https://rubygems.org/ (accessed on 27 February 2023).
6. npm, Inc. npm. Available online: https://www.npmjs.com/ (accessed on 27 February 2023).
7. The PHP Group. PEAR—PHP Extension and Application Repository. Available online: https://pear.php.net/ (accessed on 27 February 2023).
8. The PHP Group. PECL: The PHP Extension Community Library. Available online: https://pecl.php.net (accessed on 27 February 2023).
9. Debian. Debian—Packages. Available online: https://www.debian.org/distrib/packages (accessed on 27 February 2023).
10. Canonical Ltd. Ubuntu—Ubuntu Packages Search. Available online: https://packages.ubuntu.com (accessed on 27 February 2023).
11. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [CrossRef]
12. Roy, C.K.; Cordy, J.R.; Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.* **2009**, *74*, 470–495. [CrossRef]
13. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Qi, H.; Hu, J. Vulpecker: An automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications, Angeles, CA, USA, 5–8 December 2016; pp. 201–213.
14. Kim, S.; Woo, S.; Lee, H.; Oh, H. Vuddy: A scalable approach for vulnerable code clone discovery. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 595–614.
15. Liu, Z.; Wei, Q.; Cao, Y. Vfdetect: A vulnerable code clone detection system based on vulnerability fingerprint. In Proceedings of the 2017 IEEE third Information Technology and Mechatronics Engineering Conference (ITOEC), Chongqing, China, 3–5 October 2017; pp. 548–553.
16. Jang, J.; Agrawal, A.; Brumley, D. ReDeBug: Finding unpatched code clones in entire os distributions. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 48–62.
17. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
18. Li, Z.; Lu, S.; Myagmar, S.; Zhou, Y. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.* **2006**, *32*, 176–192. [CrossRef]

19. Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.K.; Lopes, C.V. Sourcerercc: Scaling code clone detection to big-code. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 1157–1168.
20. Yamaguchi, F.; Lottmann, M.; Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; pp. 359–368.
21. Wu, Y.; Zou, D.; Dou, S.; Yang, S.; Yang, W.; Cheng, F.; Liang, H.; Jin, H. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis. In Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Virtual, 21–25 December 2020; pp. 821–833.
22. Zou, Y.; Ban, B.; Xue, Y.; Xu, Y. CCGraph: A PDG-based code clone detector with approximate graph matching. In Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Virtual, 21–25 December 2020; pp. 931–942.
23. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 96–105.
24. Sasaki, Y.; Yamamoto, T.; Hayase, Y.; Inoue, K. Finding file clones in FreeBSD ports collection. In Proceedings of the 2010 seventh IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; pp. 102–105.
25. Mockus, A. Large-scale code reuse in open source software. In Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007), Minneapolis, MN, USA, 20–26 May 2007; p. 7.
26. Roy, C.K.; Cordy, J.R. Near-miss function clones in open source software: An empirical study. *J. Softw. Maint. Evol. Res. Pract.* **2010**, *22*, 165–189. [CrossRef]
27. Heinemann, L.; Deissenboeck, F.; Gleirscher, M.; Hummel, B.; Irlbeck, M. On the extent and nature of software reuse in open source java projects. In *Proceedings of the International Conference on Software Reuse*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 207–222.
28. Koschke, R.; Bazrafshan, S. Software-clone rates in open-source programs written in C or C++. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 3, pp. 1–7.
29. Lozano, A.; Wermelinger, M. Assessing the effect of clones on changeability. In Proceedings of the 2008 IEEE International Conference on Software Maintenance, Beijing, China, 28 September 2008–4 October 2008; pp. 227–236.
30. Hotta, K.; Sano, Y.; Higo, Y.; Kusumoto, S. Is duplicate code more frequently modified than non-duplicate code in software evolution? An empirical study on open source software. In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), Antwerp, Belgium, 20–21 September 2010; pp. 73–82.
31. Lozano, A.; Wermelinger, M.; Nuseibeh, B. Evaluating the harmfulness of cloning: A change based experiment. In Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007), Minneapolis, MN, USA, 20–26 May 2007; p. 18.
32. Mondal, M.; Roy, C.K.; Schneider, K.A. An empirical study on clone stability. *ACM SIGAPP Appl. Comput. Rev.* **2012**, *12*, 20–36. [CrossRef]
33. Krinke, J. Is cloned code more stable than non-cloned code? In Proceedings of the 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, Beijing, China, 28–29 September 2008; pp. 57–66.
34. Göde, N.; Koschke, R. Frequency and risks of changes to clones. In Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA, 21–28 May 2011; pp. 311–320.
35. Kim, M.; Sazawal, V.; Notkin, D.; Murphy, G. An empirical study of code clone genealogies. In Proceedings of the tenth European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, 5–9 September 2005; pp. 187–196.
36. Aversano, L.; Cerulo, L.; Di Penta, M. How clones are maintained: An empirical study. In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07), Amsterdam, Netherlands, 21–23 March 2007; pp. 81–90.
37. Saha, R.K.; Asaduzzaman, M.; Zibran, M.F.; Roy, C.K.; Schneider, K.A. Evaluating code clone genealogies at release level: An empirical study. In Proceedings of the 2010 Tenth IEEE Working Conference on Source Code Analysis and Manipulation, Timisoara, Romania, 12–13 September 2010; pp. 87–96.
38. Zibran, M.F.; Saha, R.K.; Roy, C.K.; Schneider, K.A. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1123–1130.
39. Thongtanunam, P.; Shang, W.; Hassan, A.E. Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empir. Softw. Eng.* **2019**, *24*, 937–972. [CrossRef]
40. Sajnani, H.; Saini, V.; Lopes, C.V. A comparative study of bug patterns in java cloned and non-cloned code. In Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, Canada, 28–29 September 2014; pp. 21–30.
41. Rahman, F.; Bird, C.; Devanbu, P. Clones: What is that smell? *Empir. Softw. Eng.* **2012**, *17*, 503–530. [CrossRef]
42. Li, J.; Ernst, M.D. CBCD: Cloned buggy code detector. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 310–320.

43. Islam, J.F.; Mondal, M.; Roy, C.K. Bug replication in code clones: An empirical study. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 1, pp. 68–78.
44. Islam, M.R.; Zibran, M.F. A comparative study on vulnerabilities in categories of clones and non-cloned code. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 3, pp. 8–14.
45. Islam, M.R.; Zibran, M.F.; Nagpal, A. Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, Canada, 9–10 November 2017; pp. 20–29.
46. Cybersecurity, Critical Infrastructure. Framework for Improving Critical Infrastructure Cybersecurity. 2018. Available online: https://nvlpubs.Nist.Gov/nistpubs/CSWP/NIST.CSWP (accessed on 27 February 2023).
47. National Institute of Standards and Technology. NVD—Vulnerabilities. Available online: https://nvd.nist.gov/vuln (accessed on 27 February 2023).
48. Ctags. Universal Ctags. Available online: https://github.com/universal-ctags/ctags (accessed on 27 February 2023).
49. Mozilla. Mozilla Foundation Security Advisories. Available online: https://www.mozilla.org/en-US/security/advisories/ (accessed on 27 February 2023).
50. Canonical. Ubuntu Security Notices. Available online: https://ubuntu.com/security/notices (accessed on 27 February 2023).
51. Ubuntu. Ubuntu CVE Tracker. Available online: https://people.canonical.com/~ubuntu-security/cve/ (accessed on 27 February 2023).
52. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
53. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 896–899.
54. Duan, Y.; Li, X.; Wang, J.; Yin, H. Deepbindiff: Learning program-wide code representations for binary diffing. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2020.