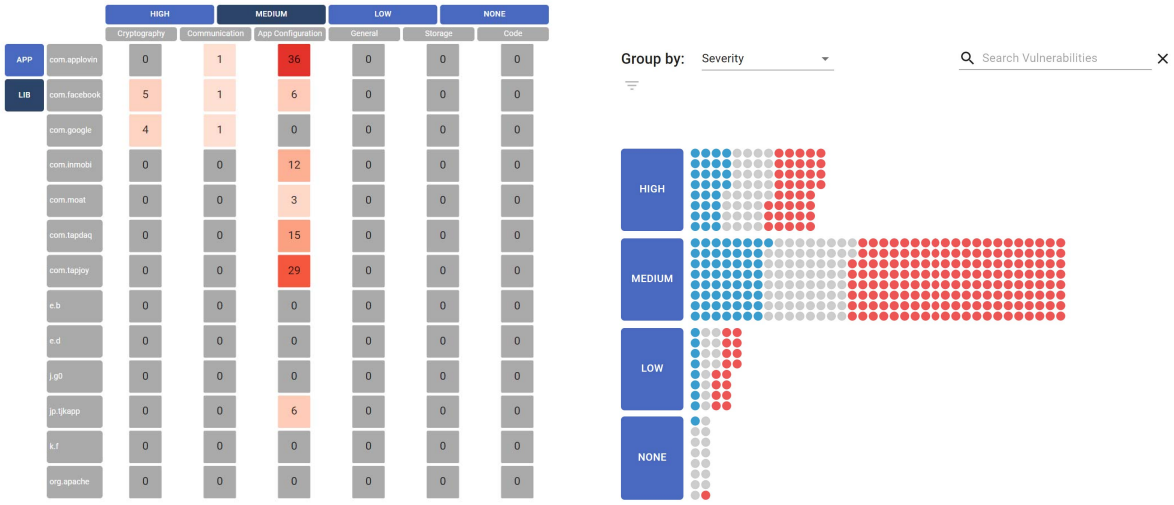
User-Centered Design of Visualizations for Software Vulnerability Reports

Steven Lamarr Reynolds* Fraunhofer IGD
Tobias Mertz† Fraunhofer IGD
Steven Arzt ‡ Fraunhofer SIT
Jörn Kohlhammer § Fraunhofer IGD, TU Darmstadt



(a) Report Overview (b) Report Comparison

Figure 1: The *Report Overview* and *Report Comparison* visualizations displaying the vulnerabilities of an application. On the left, a matrix visualization is showing a single report and the distribution of vulnerabilities across relevant attributes such as severity and origin. The red saturation marks the matrix cells with the largest amount of vulnerabilities. On the right, a unit bar chart shows the differences in vulnerabilities present in two versions of an application. The red and blue colors of some units correspond to the individual application versions while the grey color marks vulnerabilities that match between both versions. Both of these visualizations help users to assess the security status of their applications as described in use case 1 and 2 (see Sect. 5).

Abstract

Today's software systems are created by software development processes that naturally include mistakes, some of which can be exploited by attackers and are therefore called vulnerabilities. Automatic software scanners enable developers to analyze their applications to detect vulnerabilities and alert them of their presence. But often these reports are hard to understand, include false positives or overwhelm users due to the sheer number of alerts, since a report may contain hundreds to thousands of vulnerabilities. Developers must undergo a process called *vulnerability triage* to find the relevant vulnerabilities to fix. This paper presents two interactive visualizations for developers and security experts to gain an overview of the security state of their application. Users can see the distribution of vulnerabilities, find the most relevant ones, and compare differences between application versions. Our visualization design is inspired by an initial preliminary study and has been evaluated by domain experts to investigate the usability and appropriateness.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Visual analytics; Security and privacy—Software and application security; Security and privacy—Systems security—Vulnerability management—Vulnerability scanners

*e-mail: steven.lamarr.reynolds@igd.fraunhofer.de
†e-mail: tobias.mertz@igd.fraunhofer.de
‡e-mail: steven.arzt@sit.fraunhofer.de
§e-mail: joern.kohlhammer@igd.fraunhofer.de

1 Introduction

Nowadays, everyone is connected through a variety of devices, applications, and services, which can involuntarily disclose sensitive user data to third parties due to exploitable bugs or vulnerabilities. Such leaks may be possible at any time without the user's knowledge or approval. For this reason, developers must make sure that the digital infrastructure we rely on is properly secured and does not contain critical vulnerabilities. This is a difficult task, as many development teams lack the experience to prevent vulnerabilities or the budget to employ the assistance of external security experts during the software development life cycle. To assist non-experts, automatic security scanners have been proposed in the past. These scanners can automatically analyze an application, detect vulnerabilities within and alert the user to their presence.

Depending on the analysis method used by a particular scanner, it can be categorized as one of two variants: static analysis or dynamic analysis. Static code scanners parse the application to be analyzed into data structures such as abstract syntax trees or specialized intermediate languages. On the other hand, dynamic analysis tools inspect an application while it is running. This includes input and output to the user interface or storage devices. One frequently criticized issue of static analysis tools is the number of false positives [18]. A false positive occurs if the scanner result is technically incorrect, for example the scanner claims a data flow that does not

2639-4332/21/$31.00 ©2021 IEEE
DOI 10.1109/VizSec53666.2021.00013

exist. Such false positives may be reported in code that is contained in the application, but never used at runtime. Note that statically-linked applications contain entire libraries, while commonly using only a small fraction of the methods provided by each library.

Irrespective of the analysis method used, the human security analyst as the user of the analysis tool must inspect each reported finding. Even if a finding is not technically a false positive, it might still be irrelevant. The transmission may even be intended, e.g., sending the user's password to the remote backend of the application provider for verification and for accessing the user's profile. Further, not all findings are equally critical. If, for example, the number of a customer card is accessible to attackers, this is more severe for payment cards than for mere bonus cards without a payment function. In the first case, the attacker can steal the user's card balance. In the second case, they may only add more bonus points through their own purchases. This process of ruling out false positives, estimating the critically of issues, and prioritizing them for further processing is called *vulnerability triage*. Despite research efforts [15], it is still largely a manual effort that requires domain knowledge (consider the semantics of the customer card in the example) as well as a technical understanding of the application's programming language and platform.

Depending on the scope of the application, the number of detection's may vary greatly, and large numbers of vulnerabilities may overwhelm a human expert [18]. As the time required for the triage can add up quickly, development teams may opt against using automated scanners, despite possibly missing out on important security warnings. To alleviate this issue and reduce the time required for the triage, scanner output should be presented in an easily digestible format. As our example with the customer card shows, technical improvements to the scanner alone are not sufficient, as scanners lack the knowledge about the application's precise domain and environment.

While research in automatic vulnerability scanning is quite active, there is a need for further exploration of the usable presentation of scanner results [19]. Furthermore, most research in this topic is not backed by user-centered design. User-centered design methods [25] put the needs of the target user first, establishing the users' tasks and subsequent design requirements before beginning development and iteratively incorporating user feedback throughout.

Within this paper we present a prototypical design of a vulnerability visualization tool for software developers and security analysts based on the output data of an automatic security scanner. Following a user-centered design approach, we first characterize the problem by defining the data and users, and then report on a preliminary study to determine the most important tasks our users need to perform with the data. From these tasks we derive design requirements for our tool. Our main contributions are:

- Characterization of the problem of effectively analyzing software vulnerability reports, including the data and users' needs.
- Definition of user tasks and derived design requirements based on a preliminary study.
- A prototypical implementation of a vulnerability visualization tool based on the data of an automatic vulnerability scanner.

2 Related Work

The publications presented in this section can be split into two main categories. In Sect. 2.1, we look at approaches that visualize vulnerabilities of software systems like in this paper. The related work contains approaches that are trying to make software systems more secure. Others try to make the development process itself better by visualizing vulnerable interactions of systems. In Sect. 2.2, we look at network security visualizations that show where a network system has certain vulnerabilities. These approaches also focus on vulnerability visualizations but with the focus of larger networks and multiple software versions where each can contain several different vulnerabilities.

2.1 Vulnerability Visualization

Recently, there have been a number of approaches that visualize vulnerabilities of software systems by using static analysis tools [3, 15]. Most of these approaches have the goal of software security and are either graph- or treemap-based. They have begun to apply visualization to the specific issues of bugs, vulnerabilities and code quality issues of software systems.

Assal et al. [3], for instance, use a treemap to visualize vulnerabilities from the FindBugs scanner to show the severity, category and location of the vulnerabilities in the codebase. They intend to use it as a collaborative code review tool and not necessarily for vulnerability triaging. Therefore, it can be hard to grasp the kind of vulnerabilities that are present. Instead it shows an overview over the source code, similar to Goodall et al. [15]. Their approach uses aggregated source code files, which are represented as blocks and sized according to the number of vulnerabilities.

Ying et al. [33] use a similar treemap with a hierarchy that shows packages and single source files. Lines of code are highlighted as well as vulnerabilities based on their severity. Developers should get an impression of both hierarchical structure of software packages as well as the severity of the source code defects. While their goal is to allow software developers to better triage the vulnerability results from multiple detection tools, it can be hard for users to assess the security state immediately. Instead, users have to map their internal representation of the source code structure to the visualization which can create a knowledge gap in users that are not familiar with this type of visualization. Additionally, there is a potential limitation of screen space by showing the actual source code structure.

These approaches are all focused on facilitating the understanding of the source code structure, a large number of vulnerabilities, or a high percentage of false positives. They either use the output of a single software scanner or the results of multiple scanner outputs combined. Access to source code can sometimes be limited for security experts analyzing external applications. These approaches did not use a prelimiary study to determine the needs and goals of the targeted user groups. Additionally, only Cesar [3] has been evaluated to determine the approach's usability. We use a preliminary study and an expert evaluation to assess the usefulness and usability of our visualizations. In summary, the goal of our approach is focused on developers understanding of how secure their software is.

2.2 Network Vulnerability Visualization

In the space of network and cybersecurity, some approaches focus on visualizations that let users analyze threats in computer networks. These have a strong focus on showing different security issues that might be distributed over multiple connected devices.

Angelini et al. [1] use a treemap bar chart that combines data from vulnerability and network scanners to allow security managers to inspect spread on networks, understand the network status, and make decisions with a large number of vulnerabilities. However, their approach only focuses on network vulnerability analysis, and cannot support a more detailed analysis of a single system. Harrison et al. [17] use a treemap to display the network structure and a histogram for an overview. Users are able to perform subsequent scans of the network and to classify vulnerabilities as fixed or non-issues. They use color to distinguish between this classification. In contrast to these treemap approaches, we decided to use aggregated matrix visualizations. They are similar to heatmaps with discrete axes [34]. Matrix visualizations are already used for bio-visualizations like Biomole by Eggermont et al. [11] or Wu et al. [35], but are not widely used in cybersecurity.

Dang et al. [10] use a node-link diagram with a bar chart overlay to visualize multiple scanner outputs of a security scan of a website with its subpages. Web security testers are able to compare performances of tools, spot similarities between webpages and view the overall vulnerability distribution. Using node-link diagrams would also be an appropriate choice for this paper. However, due to lack of emphasis on the connections between vulnerabilities, we decided to use unit bar charts that are based on the visualization grammar by Park et al. [26]. It uses a single visual element to represent a vulnerability and packs the elements according to the currently selected category. One advantage of unit visualizations is the one-to-one correspondence of elements between layouts, which we also employ by using animated transitions.

Finally, Pham et al. [27] use multiple visualizations like parallel-coordinates, a timeline and a graph to visualize the relationships of various dimensions in the Common Vulnerabilities and Exposures (CVE) catalog [29]. However, while it allows to filter topics of interest it does not use actual data of an application but rather from a vulnerability database.

These approaches are useful to analyze websites, vulnerability databases, or the security status of networks. Due to the focus on the analysis of multiple systems, most of these approaches have scalability issues, while only Vulnus [1] features an evaluation with security experts. To the best of our knowledge, there is no approach equivalent to our paper that focuses on the in-depth visualization of a single software vulnerability report.

3 Target Problem in the Domain

3.1 Background

As described in Sect. 1, there are static and dynamic vulnerability scanners. Static scanners parse the application from the human-readable source code or the binary executable, where the source code is not needed. In addition to the created intermediate representations or abstract syntax trees, scanners may compute data structures such as data flow graphs. Data flows are used to either identify leaks of privacy-sensitive data to untrusted sinks such as the Internet, or injections of untrusted data into protected resources such as databases. Vulnerability checks are usually rule-based, i.e., they apply a predefined set of rules to the intermediate representation, abstract syntax tree, or data flow graph. Static analyses reason about all possible states of the application, which allows for completeness at the cost of precision.

The monitoring of the running application in dynamic analysis tools can be achieved in parallel, e.g., by inspecting the network traffic as it happens, or through the output files generated by the application. Since the analysis matches observed events against its vulnerability patterns, it needs not approximate the application behavior. Dynamic analyses therefore usually do not suffer from false positives. On the other hand, it can only observe events, and therefore violations of the security policy, in code paths that were taken during the sample runs. In practice, code coverage is usually limited [9]. In total, dynamic analyses favor correctness of the results over completeness and may suffer from more false negatives.

Our implementation visualizes vulnerabilities detected by the VUSC [13] code scanner, which is in turn based on Soot [21] and FlowDroid [2]. VUSC processes binary Android apps, translates them into the Jimple intermediate representation [5, 31], and builds a semantic model of the app. This model includes the app components such as activities and services, the data flows between these components, and relevant user interface controls such as password fields. VUSC evaluates its security analyses against this model. For example, it checks whether the data from a password field is transmitted to a remote server using an unencrypted protocol and reports a security violation if this is the case. VUSC is a static analyzer, i.e., it only analyzes the app's bytecode without running the app. Its results are therefore not limited by code coverage or individual runs, but may be over-approximated, i.e., individual results may be false positives.

Our visualization approach is designed to be compatible with different scanners, if they output similar data structures as VUSC. In other words, our visualization is agnostic to the analysis method used to detect the vulnerabilities. For this reason, we decided to only utilize the part of VUSC's output data described in Sect. 3.3 that is common to other vulnerability scanners as well. This makes the prototype easily adjustable to take input data from a different source, which should allow our designs to be integrated into a wider variety of workflows.

3.2 User Specification

The proposed approach has primarily been designed with developers and security experts in mind. In this context we describe two relevant user groups for this approach:

- Software Developers, who develop and maintain software technologies where possible security problems can occur.
- Security Analysts, who test and analyze software technologies.

Both of these two user groups can be seen as a homogeneous as they share many characteristics. For one, they both have knowledge of software creation process and software vulnerabilities. Security analysts typically have additional security and vulnerability knowledge that can be useful in further vulnerability analysis. We assume that both user groups are familiar with basic visualization and interaction concepts but might be overwhelmed with more complex visualization systems. Additionally, the needs of these user groups align with each other as both use software vulnerability scanners. In turn, they face similar barriers to not use them, like bad warning messages, miscommunications or poor usability [28].

3.3 Data Specification

VUSC generates its vulnerability output in a *JSON* format that can be accessed via its API. This data contains metadata for the report as well as a list of vulnerabilities that contains the following attributes:

Severity A severity score that can take the values *HIGH*, *MEDIUM*, *LOW*, or *NONE*.

Type The name of the vulnerability type, such as *"CryptoCheckAnalysis_InsecureCryptoAlgorithm"*. The type consists of the name of the VUSC module that detects this type of vulnerability and the name of the type itself, separated by an underscore. In our tool we use the entire string as an identifier.

Category VUSC discriminates between a total of 20 unique vulnerability categories, for example, the *InsecureCryptoAlgorithm* vulnerability type belongs to the category *Cryptography*.

Code Location The location of the vulnerability within the Jimple code. This includes class name, member function, line number, and the exact statement.

Description A full text description of the vulnerability.

Mitigation Hint A full text explanation of possible ways to avoid the vulnerability.

External References References to external sources regarding the vulnerability. These can be links to external vulnerability catalogues such as the *CWE* [22] or to guides that detail the proper techniques on how to prevent certain vulnerabilities.

To preserve the compatibility with other scanners, our visualizations only display the attributes above. Furthermore, VUSC can also output additional vulnerability information in the report. For example the target URL of a network communication, the individual code locations traversed in a detected dataflow, or a list of included libraries. Additionally to the data in VUSC's report, we utilize two derived attributes in our designs.

Origin This attribute discriminates between vulnerabilities within first- and third-party code. It is derived by comparing the application ID with the class name within the vulnerabilities' code location. The application ID is a unique identifier for each Android application and is contained within the report metadata. Within Android applications, all first-party class paths are prefixed with the application ID.

Delta The delta attribute is derived from the comparison of the vulnerability reports resulting from two separate scans of different application versions. When comparing two reports, all vulnerabilities within are compared pairwise. The delta attribute then discriminates between vulnerabilities occurring in only the first report, only the second report, or both reports.

3.4 User Tasks

To categorize and define the user's tasks and the system requirements we conducted a preliminary study online with 18 users. Preliminary studies can help to gather initial goals and tasks of the targeted user group [7]. Participants were asked multiple questions to determine their experience, goals and tasks with software scanners. The participants can be categorized as 7 software developers and 11 security experts with experience in security and software scanner technology. The majority of the participants have a university degree and ranked themselves high in technical skills.

We asked participants about the most important aspects in a vulnerability analysis system. We focused on four main aspects: *visual*, *interactive*, *attributes in a vulnerability report* and *attributes of a single vulnerability*.

For each aspect we defined ten items we deemed important. For example the attributes of single vulnerabilities included the title, description, category and others. Participants were asked to sort these items from least to most important. We list the three items rated highest on average for each of these four main aspects.

For the visual aspects, participants indicated *seeing the most relevant*, *overview*, and *which app version introduces new vulnerabilities* as most important. Regarding interactive aspects, *the ability to search and filter*, *viewing the position of a vulnerability in the code* and *exporting the vulnerabilites to other systems* were rated highest. For a vulnerability report, the participants think that *high severity vulnerabilities*, *medium severity vulnerabilites* and *metadata* are most important. For a single vulnerability, the *category*, *title* and *description* are most important.

The free-text responses also show that an *overview* and the *critical vulnerabilities* are most important. In addition, it was often stated that it is necessary to verify vulnerabilities and then fix the most critical vulnerabilities first. Some participants also mention various processes, such as dividing the task of fixing issues within their team, discussing them with project managers or using an issue tracking system to better manage the vulnerabilities.

To construct tasks, we used the highest average rated answers from the four aspects. Based on these, we identified the following tasks for both user groups:

T1 Assess the security status of the application

T2 Find the most relevant vulnerabilities

T3 Fix the vulnerabilities in a prioritized way

T4 Compare differences of vulnerabilities between application versions

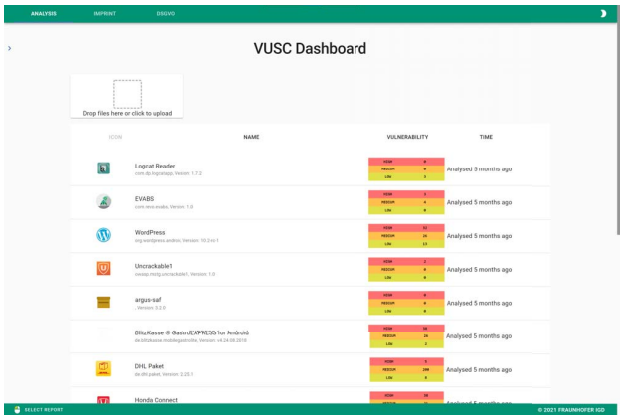


Figure 2: The dashboard page of our application.

3.5 Design Requirements

Based on the four user tasks we derived the following requirements for our design solution. These requirements correspond to the tasks in Sect. 3.4 by the referenced tasks in the parenthesis.

R1 Show an overview of all vulnerabilities (**T1**, **T2**)

R2 Show details of a single vulnerability on demand (**T1**, **T3**)

R3 Visually discriminate between vulnerabilities by their most relevant attributes (**T1**, **T4**)

R4 Let the user search and filter vulnerabilities (**T2**)

R5 Visually discriminate between vulnerabilities of two software versions (**T4**)

4 Visualization and Design Rationale

Our prototype is implemented as a front-end web application in *TypeScript* [24], using *React* [12] and *D3.js* [6]. Additionally, we adhere to the guidelines of *Material Design* [23] and utilize its components to present the user a familiar interface with recognizable interactive elements. The application consists of three web pages.

The starting page is a dashboard providing a list of all scanned applications as well as the functionality to queue new scanning operations. The user can select one scan report from the list to inspect, which opens the *Report Overview* page.

The *Report Overview* shows an overview of all vulnerabilities within a single report grouped by their attributes. It allows the user to filter and see details on demand. Within the *Report Overview*, the user can select a second report to compare with the currently viewed report. This opens the *Report Comparison* page.

The *Report Comparison* page shows all vulnerabilities within the compared reports and classifies each vulnerability depending on which report it occurs in. Additionally, the *Report Comparison* allows the user to filter and group vulnerabilities based on their attributes as well as view details on demand.

4.1 Dashboard

The dashboard page, as shown in Fig. 2, provides two core functionalities, the ability for the user to queue application scans, and a list of all past scans to inspect. To queue new scans the user can drop a compiled Android application into a dropzone at the top of the page or click the zone to select a file through the file browser.

The list of reports is displayed below the dropzone. Each list element contains the header information of the corresponding report. This header information consists of the application icon, its name, its Java package name, its version, a quick vulnerability overview,

Figure 3: Initial matrix visualization showing an overview of how vulnerabilities are distributed across severity and their origin.

Figure 4: Matrix visualization after the *medium* severity is selected.

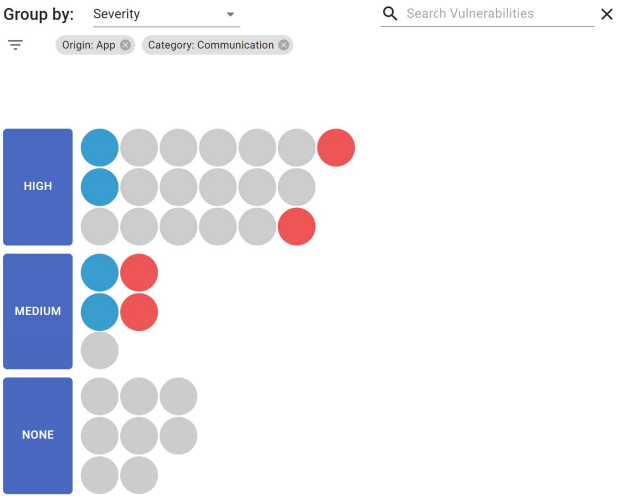


Figure 5: Unit visualization showing the comparison between two reports. Vulnerabilities are filtered by the *App* origin and the *Communication* category and grouped by severity.

and the time passed since it was scanned. The quick vulnerability overview displays the number of vulnerabilities within the report for each severity type. The three vulnerability amounts are rendered against a colored background that is red for high-severity, orange for medium-severity, and yellow for low-severity vulnerabilities. Additionally, to the colored background, textual labels for the severity's are placed next to the numbers.

The header of the report list allows the user to sort the list based on application name, weighted vulnerability count, or scan date. The weighted vulnerability count weighs vulnerabilities proportionally to their severity. Furthermore the page provides the functionality to search the list and to filter the report list by different scan date ranges.

4.2 Report Overview

The *Report Overview* page displays the report header for the selected vulnerability report. This header is displayed in the same way as the entry within the report list on the dashboard. The space below the report header is split vertically through the center. The left side of the page contains the visualization that can be seen in Fig. 3, while the right side of the page shows a list of all vulnerabilities within the report.

We chose a matrix visualization to support the ability of users to get an overview of all vulnerabilities and their distributions. To bridge the knowledge gap, this visualization resembles a simple matrix, where the elements are categorized by rows and columns. Users can simultaneously explore the associations between attributes and the distribution of vulnerabilities as can be seen in Fig. 3. The columns correspond to the different severities *HIGH*, *MEDIUM*, *LOW* and *NONE*. The rows are the origins — whether the vulnerability is located in first- or third-party code. These attributes were determined as relevant for the user groups in the preliminary study (see Sect. 3.4). The grid should allow for better orientation and comparability between elements. To direct the user's attention to the most critical cells, the relative amount of vulnerabilities is encoded with the red color saturation.

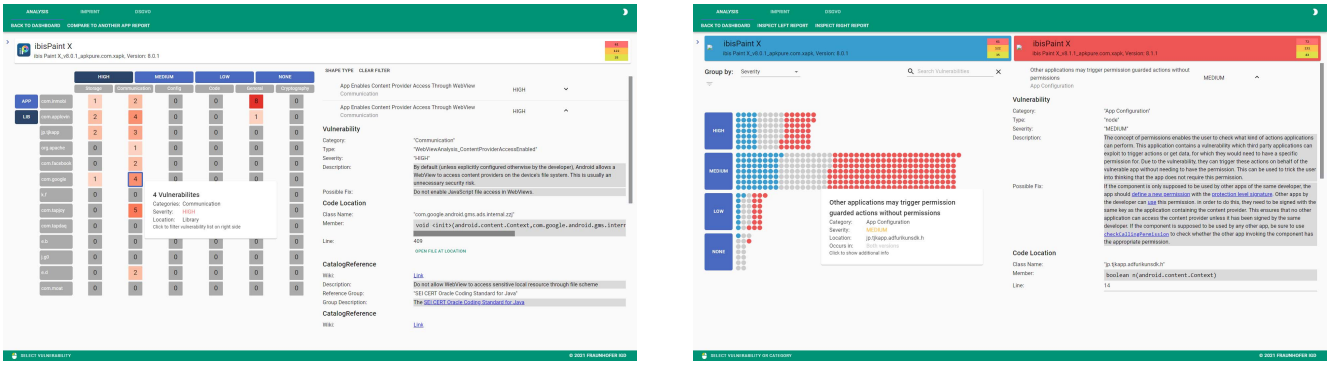
This representation should allow users to perceive distribution of vulnerabilities in the application and used libraries. To allow users to view an overview of all vulnerabilities (**R1**), we show all vulnerabilities across the grid (see Fig. 6a). Additionally, we show a list of vulnerabilities next to the visualization (**R2**). Users can select a grid element which filters the list of vulnerabilities next to it and additionally users can filter the vulnerabilities by a search field, which lets users search and filter vulnerabilities (**R4**). By selecting a column or row header element users can expand either into more detailed attributes to further investigate the vulnerability distribution (**R3**). Selecting a severity splits the columns into the individual vulnerability categories as can be seen in Fig. 4, while selecting an origin splits the rows into the individual package names within that origin. The left image in Fig. 1a shows the visualization with the medium severity column and Lib origin row selected.

4.3 Report Comparison

This page shows a comparison of the vulnerabilities between two versions of an Android application. The top of the page shows the report headers of both reports side by side as can be seen in Fig. 6b. These headers contain the same information and styling as the display within the report list on the dashboard and on the *Report Overview*. However, on this page the headers have colored backgrounds. The left report has a blue background while the right report has a red background. These colors are used throughout the entire page to distinguish between vulnerabilities occurring in either of the two reports. Vulnerabilities that occur in both reports are displayed in grey. This color mapping has been explicitly chosen to cause vulnerabilities that have been introduced in a new application version to be perceived as especially critical. This is achieved with a simple heuristic for the report ordering. The report with the higher version number is always chosen as the right report and therefore its vulnerabilities are marked in red.

The remainder of the page is split vertically into two halves. The right side of the page shows some general statistics of the reports, while the left half shows the visualization of the vulnerabilities. The matched vulnerabilities from both reports are displayed as a unit bar chart. According to the *ATOM* grammar for unit visualizations [26], the units are first grouped by their severity, then sorted by their delta attribute and positioned in a packed layout within their severity bins. Each element is represented by a filled circle colored according to its delta attribute.

The initial display of all vulnerabilities at the same time shows an overview of all vulnerabilities (**R1**). The merged display of both reports and the coloring corresponding to the delta attribute are



(a) Report Overview

(b) Report Comparison

Figure 6: The two visualizations of our approach. (a) *Report Overview* gives an overview over a single report. (b) *Report Comparison* lets users compare differences of vulnerabilities between two application versions.

designed to discriminate between vulnerabilities of two software versions (**R5**). Atop the visualization, a dropdown menu allows the user to select the grouping attribute. In addition to the default grouping by severity, also category, origin, or delta can be selected (**R3**).

Next to each unit bin, a button with the group name is displayed. A click on this button applies a filter to the vulnerabilities, disregarding all vulnerabilities outside of this bin. Furthermore, the user can enter a search string into a search field above the visualization to filter the displayed vulnerability data. This allows for a combination of attribute-based filtering as well as a textual search (**R4**). The visualization can be seen with two active filters in Fig. 5.

Hovering over a vulnerability shows a tooltip with the discriminating vulnerability attributes, clicking a vulnerability replaces the report statistics on the right side of the page with detailed information about the vulnerability. With the initial view, the hovering tooltip, and the explicit detail view, we achieve three levels of granularity for the vulnerability data. User can control the display of each of these granularity levels (**R2**).

The visualization as a unit bar chart was chosen for several reasons. Since the matching of vulnerabilities in binary code is still a topic of active research [16], we cannot guarantee the correctness of all matches and the user may need to verify them manually. To be able to verify the vulnerability matches, the user must be able to inspect, interact with, and track vulnerabilities individually. Additionally, according to Park et al. [26] unit visualizations are particularly effective at presenting relative percentages, because the user is easily able to estimate absolute as well as relative amounts from the visualization at the same time. We use this concept to our advantage to quickly communicate two pieces of information about each bin to the user: the total number of vulnerabilities and the relative number of vulnerabilities for each delta value.

The main downside of unit visualizations is their scalability. However, due to our extended filtering and search functionality, the user can always narrow down their exploration space to a sufficiently small set that is easily displayable on a regular desktop display.

5 Use Cases

We demonstrate the utility of our tool with two use cases that encompass the different functionalities of our prototype.

5.1 First Time User

This use case considers Alice, a hobbyist developer who has never used a security analysis tool before and tries to assess the security state of her own application. As the application is already published, Alice wants to quickly begin with fixing vulnerabilities, starting with the most severe ones.

Alice starts her analysis by opening the dashboard and loading her application for analysis. After the scan is complete, the dashboard already shows the number of high, medium, and low severity vulnerabilities within the report. To get more details, Alice selects the report and is routed to the *Report Overview*. On this page, she sees the matrix visualization as described in Sect. 4.2. The columns of the matrix discriminate between the different severity values, the rows between the two origins, and the individual cells represent all vulnerabilities corresponding to the respective row and column. The color scale of the matrix cells immediately gives an impression of the relative frequency of vulnerabilities of the different severity types within her own code as well as third-party libraries. The numbers in the matrix cells also tell her the absolute amount of vulnerabilities for each cell. This information allows her to quickly judge the number of vulnerabilities and their distribution (**T1**). As she is mostly interested in high severity vulnerabilities, she selects the corresponding column, to filter out all vulnerabilities of lower severity's. The columns of the matrix now display the vulnerability categories. Alice wants to start fixing vulnerabilities quickly, so she decides to fix her own code first before she tries to determine how to treat vulnerabilities within third-party libraries. For this reason, she selects the *APP* row of the matrix. All third-party vulnerabilities are now filtered out and the rows change to display individual classes (**T2**).

The fine-grained matrix gives a complete overview of vulnerabilities within each class and each category. Alice selects the crossing of category and library with the most vulnerabilities. On the right side of the page, the complete list of these vulnerabilities is now displayed, and Alice can compare the individual entries. Within each entry, she can view the type of vulnerability, the description, advice for a possible fix, the location within the code, as well as external references. Alice can also display the intermediate representation of the corresponding code file. This information helps Alice determine the vulnerability's threat potential.

5.2 Regular User

This use case considers Bob, a developer working as part of a larger team that is using automated security scans as part of their development cycle. Bob has just committed changes to a new version of the app and wants to see his progress in fixing the existing security vulnerabilities but also check if he created new vulnerabilities by accident.

Bob starts his analysis the same way Alice does in the first use

case. On the *Report Overview* page, Bob selects the *Compare with another app report* option. Within the popup window, all previous versions of the app are listed. To only include his most recent changes, Bob selects the previous version of the app to compare to and is routed to the *Report Comparison* page.

Within the *Report Comparison*, Bob can see both selected reports and identify the corresponding vulnerabilities by their colors (**T4**). Since the blue elements correspond to vulnerabilities that only exist in the report of the older application version and grey elements to vulnerabilities within both reports, Bob can easily estimate his progress by comparing the number of blue elements to the total number of blue and grey elements. The bar chart-like arrangement of the visual elements enables a rather quick relative estimation of these numbers. To judge whether new vulnerabilities have been created through his changes, Bob simply has to look for red elements. If there are new vulnerabilities, Bob can easily determine if the security state of his application has improved or worsened by comparing the number of vulnerabilities displayed in red with the blue ones (**T1**).

Additionally, all elements are grouped by severity by default. Since high-severity vulnerabilities are the most important vulnerabilities to fix, this grouping allows Bob not only to check his total progress, but also the progress within the individual severity classes.

Because Bob cooperates with multiple other developers on the same application, Bob may not need to see and investigate all vulnerabilities. Let's say Bob's area of responsibility is the implementation of network communication. For this reason he wants to focus on investigating the vulnerabilities within the corresponding category first. Bob can group all vulnerabilities by their category instead of their severity, and select the *communication* category as a filter. This hides all other vulnerabilities. Similarly, Bob may want to exclude vulnerabilities within third-party libraries. To achieve this, he can change grouping to the origin attribute and select the *APP* group. Bob can also use the text search if he is looking for a more specific set of vulnerabilities. This allows Bob to narrow down the set of vulnerabilities to the ones he is most interested in (**T2**).

However, since the matching of vulnerabilities between two reports is not trivial, some matches may not be recognized as such by the comparison algorithm. This means that Bob may need to compare individual blue elements with red elements to refine his progress estimation. To facilitate this triage process, Bob can change the vulnerability grouping and apply additional filters. Since most attributes of a vulnerability should remain the same between application versions, applying additional filters helps reduce the amount of vulnerabilities to investigate to the actually plausible matches. By hovering over a vulnerability and viewing the tooltips, Bob can see if two contain the same information. If Bob suspects a match, he can click on the vulnerabilities to view their full details. Using this information as well as the ability to open the intermediate representation of the code location within a pop-up window, Bob can determine whether the two vulnerabilities are actually the same.

After verifying which of the red visual elements actually correspond to new vulnerabilities, Bob can include these vulnerabilities in his task list and continue making the application more secure (**T3**).

6 Evaluation

We developed our visualizations with an iterative design process including feedback from several visualization experts from our team in each iteration. Some alternative designs were developed and later discarded in favor of our current design during this process. Additionally, we performed a qualitative evaluation of our current design with experts from the security domain. Within this section, we briefly present alternative designs, present the gathered feedback and highlight the advantages and disadvantages in comparison with our current designs.

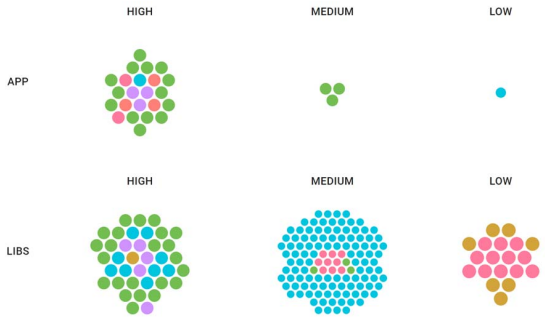


Figure 7: First iteration of the report overview visualization. Similarly to the matrix visualization, vulnerabilities are grouped horizontally by their severity and vertically by their origin.

6.1 First Iteration

The *Report Overview* was initially designed as a unit visualization, as can be seen in Fig. 7. Similarly to the matrix visualization, this unit visualization is also split into several bins of equal sizes. Vulnerabilities are grouped horizontally by their severity and vertically by their origin and placed into the corresponding bin according to a packed circular layout. One of the primary reasons why this visualization type was chosen was the ability to interact with individual vulnerabilities directly and track them through transitions. However, the main requirement for this visualization is the presentation of an overview over the application's security state. The details for individual vulnerabilities are only required on demand, after the user has already determined which smaller subset of vulnerabilities they want to focus on.

Due to the limited amount of space for each grouping, we decided to scale the vulnerabilities to fit within the available space for each group. When using the same size for all units, large variances in group size could cause some groups to appear small. If the absolute number of vulnerabilities within a group was large, the individual units were also too small to be properly usable. Since vulnerabilities with *high* severity are usually less frequent than those of lower severity's, the group of *high* severity vulnerabilities was often one of the groups that appeared small. This was a very much unintended side effect of our scale calculation. As our preliminary survey showed, very severe vulnerabilities should be the ones most prominently visible. The unit visualization with equally sized vulnerabilities was not able to achieve this.

Using different unit sizes for the individual groups is also not the ideal solution. Sizes can be computed such that vulnerabilities always fit their bin. This way all groups appear to be filled, but the different unit sizes imply different levels of importance. This is an implication that may not necessarily be true. Groups of low importance may also contain few vulnerabilities, which would cause their units to be displayed with a greater size.

The *Report Comparison* page was initially designed with circular groupings, as can be seen in Fig. 8. The groups are further distinguished by a colored border, whose color mapping is displayed in a legend next to the visualization. The attribute serving as basis for the grouping can be selected from a drop-down menu. The vulnerabilities are represented by individual visual units and displayed as filled circles, colored according to their delta attribute. However the identification of the individual groups is unnecessarily difficult, as the groups have to be manually identified by looking up the border color in the legend. Additionally, circular layouts are very space inefficient, which can be problematic if the comparison contains many vulnerabilities.

For both visualizations, we used circular packed layouts. These

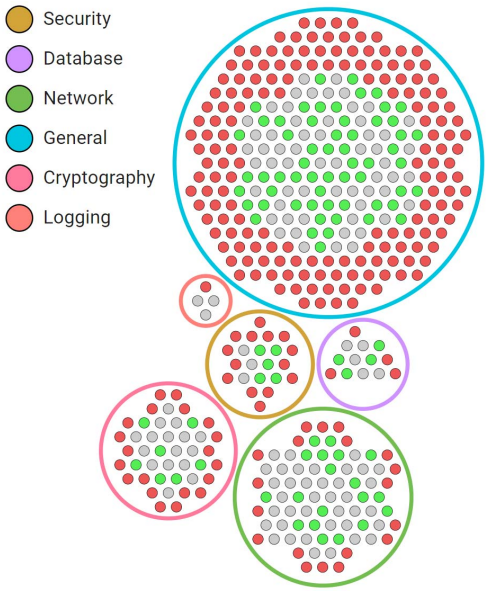


Figure 8: First iteration of the report comparison visualization. Vulnerabilities are grouped by their category and color-coded according to their delta. The category groups are surrounded by a colored circular border and can be identified in the accompanying legend.

layouts caused the overall visualizations to be very unstable during change transitions. This made the tracking of individual elements nearly impossible, which voided one of the main advantages of the unit visualization.

Furthermore, we used color to discriminate between vulnerability categories. Since VUSC outputs vulnerabilities in 20 categories, these colors were naturally difficult to distinguish. For this reason, we categorized the vulnerabilities with two hierarchy levels. The lowest level consisted of the 20 original categories output by VUSC, while the higher level contained 6 coarser categories that were created by grouping those lower level categories that were semantically similar. Then we applied a hierarchical coloring to the category tree, such that categories within the same coarse category group are colored similarly. Users can switch between the display of the original 20 categories or the reduced set.

However, we still found the colorization to be confusing. Multiple different elements on the page are colored with different color scales, as there is a color scale for severity, one for categories, and one for the delta attribute. Additionally, these scales may overlap in colors. So different interactive elements on the screen may have the same color, but that color indicates different information. During the second design iteration of both pages, the layout changed such that the categories are already sufficiently distinguishable by their positions and textual labels. Therefore, we decided that category colors were no longer necessary.

6.2 Qualitative Evaluation

The current prototype version was evaluated again with domain experts to help evaluate the current visualization techniques [30]. We interviewed 4 participants which is enough to detect 80% of usability problems [32]. At first we explained the visualizations and interactions, and showed possible use cases (see Sect. 5). Then the experts were able to freely try the visualization themselves with vulnerabilities from a real application. During this trial period, we encouraged the experts to voice their thoughts and any questions that come up. Afterwards, we had an informal discussion about possible improvements and if our design requirements were achieved. Finally, we asked them to fill out a system usability scale (SUS) questionnaire, which is composed of ten questions on a Likert-scale [8].

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total
P1	10	10	10	7.5	10	10	10	7.5	7.5	7.5	90
P2	7.5	10	10	10	7.5	10	10	10	10	10	95
P3	0	10	10	10	10	7.5	7.5	10	10	10	85
P4	7.5	10	10	10	7.5	10	10	10	10	10	95
Avg	6.3	10	10	9.4	8.8	9.4	9.4	9.4	9.4	9.4	91.5

Table 1: Results of System Usability Scale [8] with domain experts. The tool achieved a score of 90 out of 100. Especially if the system is unnecessarily complex (Q2) and easy to use (Q3) were rated best possible by all participants, while interest to use this system frequently (Q1) was considerably lower.

We recorded all of the user's questions and comments, as well as any noticeable patterns of behavior, such as confusion or difficulty finding certain functionalities. We categorized repeated ideas, feedback and comments from the transcript of the think-aloud phase and the discussion based on the questions for Evaluating User Experience by Lam et al. [20]. The evaluation was conducted online with video calls and screen sharing. By remotely letting users take control of the screen we could view the interactions of participants during the trial.

6.2.1 Results

Four external domain experts from the cybersecurity domain took part in our evaluation. Three participants were security researchers while one was a penetration tester. Our transcript of the think-aloud phase and the following discussions contained 112 entries. We discarded three of our observations because we believe they were due to the virtual setting of the study. The remaining notes were categorized into the 6 categories *useful feature*, *missing feature*, *feature improvement*, *limitation*, *understandability*, and *potential*. The first five of these categories were derived from the individual questions for evaluating user experience, proposed by Lam et al. [20]. We included the *potential* category in addition to the derived categories to encompass the domain experts' judgement whether further development of this prototype can yield meaningful improvements to developers' workflows.

The *useful feature* category is the largest category with 23 notes about the *Report Overview* and 19 notes about the *Report Comparison*. The category contains all user remarks regarding the usefulness of a certain feature of our tool. Overall, the comments in this category were rather positive. The most frequent comment on both visualizations was that they do provide a good overview of the distribution of vulnerabilities within the app and effective filtering functionality. Some comments also reinforced the results of our preliminary study, saying that high severity vulnerabilities within the app are most important, which is why the attributes' severity and origin are the most important.

Conversely, the *missing feature* category contains all mentions of additional features that the domain experts would like to see in our tool. This category contains 11 notes that are mostly applicable to both visualizations. Most of these comments were wishes for additional information to be included in the visualization and the vulnerability details. Furthermore, some of the domain experts asked for additional vulnerability management features, such as flagging or ignoring certain vulnerabilities as well as the ability to correct the vulnerability matches within the *Report Comparison*.

Next, we grouped all statements that suggested ways to improve the already existing functionality as *feature improvement*. This

category contains 9 statements concerning the *Report Overview* and 11 possible improvements of the *Report Comparison*. The most common criticism within this category is on the use of color in both visualizations. Some experts also wished the removal of filters in the *Report Overview* was more intuitive. Another suggestion is to allow the *Report Comparison* to filter by individual class names like it is possible in the *Report Overview*.

Only three limitations of the current prototype were observed by the experts. The *limitation* category contains wishes for a better vulnerability matching in the *Report Comparison* and the display of source code instead of the Jimple intermediate representation. However, the latter is a limitation on the data output by VUSC and independent of our visualization prototype. Additionally, one expert remarked that the *Report Comparison* may not work well on small displays, such as mobile phones.

As stated within the description of our method, we logged all moments of noticeable confusion of the experts. These moments as well as comments expressing that an expert had trouble understanding something were grouped in the *understandability* category. This category contains 9 notes about the *Report Overview*, 15 notes about the *Report Comparison*, as well as 2 notes pertaining to both visualizations. Mirroring the results of category *feature improvement*, the domain experts had the most trouble understanding the visual encoding within both visualizations. They criticize the use of the red color scale in the *Report Overview*. We initially designed the visualization with a red saturation to indicate the element with the highest amount of vulnerabilities. As their mental model associates this color with severity, it failed to convey the meaning. This conflict can be amended by using a different hue. In the *Report Comparison*, the users were required to look at the header of the page to link the colors to both reports. Since the report headers are not located right next to the corresponding data points and the grey color is not explained elsewhere, this mapping took the experts a couple of seconds to process. Although, they noted that the color choice makes sense after having figured out its meaning. The experts also had some trouble removing filters and vulnerability selection in both visualizations. Finally, none of the users tried to change the vulnerability grouping within the *Report Comparison* after already applying a filter. Subsequently, they also did not find out that it is possible to apply multiple filters at the same time.

The last category contains all the domain experts' conclusions regarding the *potential* usefulness of the prototype. This category contains 8 notes. Overall, the potential was envisioned rather positively. Multiple experts stated that the prototype is useful and accelerates the process of extracting information from the vulnerability report. The three security researchers saw a lot of potential in the *Report Comparison* especially, while the penetration tester preferred the *Report Overview*.

The SUS questionnaire resulted in the ratings presented in Table 1 with an average rating of 91.5. This implies a very high user satisfaction.

7 Discussion

We have used a preliminary study to understand the specific needs and goals of the user groups. This process helped us define the goals and tasks based on these actual needs. We used this feedback to prioritize visual and interactive elements, such as severity for vulnerabilities. In addition, free-text responses from participants confirmed these aspects. We further iterated on our initial design that was focused on unit visualization by internal feedback. We believe that by using a preliminary study in our design process we were able to create an effective visualization.

During the expert evaluation, we have gained several insights into the understanding of vulnerabilities by developers and security experts. The current design iteration and prototype implementation received a lot of positive feedback. Participants gave consideration on how they liked the design and whether they imagined that other developers would enjoy using it too. Additionally, they wanted more information on the matching algorithm for the *Report Comparison*. This is due to the data being used by the static scanner, which is not optimized to deobfuscate compiled applications. The matching algorithm for the *Report Comparison* could have further improvements, but this space is also part of ongoing research [16]. Another part of the design that participants criticized was the lack of options for further analysis. Participants wanted more options to filter and search both visualizations. While they expressed this need they did not use multiple features at the same time in the *Report Comparison*. However, this could be due to missing usability for this interaction.

We also realized that the domain experts have different needs due to their backgrounds. One participant in our expert evaluation had a background in penetration testing, while the others had their background in security research. While the participant with expierence in penetration testing liked the *Report Overview* visualization the others preferred the *Report Comparison*. We think that this is due to different goals while analyzing a vulnerability report. Security experts often have an ongoing process of analyzing an application and therefore are more interested in changes over time and between versions. This is also reflected in the use cases in Sect. 5.

The quantitative results of the SUS suggests that our prototype provides *Best Imaginable* usability [4]. It should be noted that these results might be biased due to the Moderator Acceptance Bias and Social Desirability Bias [14] in the evaluation. The individual scores are shown in Table 1. We noticed that our prototype has best possible ratings in *unnecessarily complex* (Q2) and *easy to use* (Q3). In contrast, the worst rating is in *using the tool again* (Q1) due to one particularly low score. Even if the design requirements did not directly specify the need for usability, we think that the feedback confirms that the system is easy to understand and use. Overall, our system should help users to better understand and in turn increase the security status of their application.

8 Conclusion

In this paper, we have presented a new interactive visualization system for developers to analyze vulnerability reports. We summarized the related work in vulnerability visualization and network vulnerability visualization. We characterized the problem domain, including data and users' needs by performing a preliminary study. We derived four specific tasks for users of vulnerability scanners and described our additional design requirements and infrastructure in detail. We further introduced two visualizations and demonstrated the usefulness with two use cases. An evaluation with domain experts gave additional feedback that we discussed and will use in our future work.

For the future development of our tool, we will continue the iterative design process by addressing the feedback gathered from our qualitative evaluation. Furthermore, we are already working on extending our solution to incorporate more of the security analysts' needs as well as those of potential end users. To that end, we also want to implement the integration of data from additional open-source scanners.

In spirit of the iterative design process, we also want to perform a more thorough quantitative evaluation to gain additional feedback and verify the design decisions that have already been made. Finally, we are considering the improvement of our vulnerability matching algorithm. This could yield a more accurate delta estimation, which would improve the comparison performance of our tool.

Acknowledgments

This research work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] M. Angelini, G. Blasilli, T. Catarci, S. Lenti, and G. Santucci. Vulnus: Visual vulnerability analysis for network security. *IEEE transactions on visualization and computer graphics*, 25(1):183–192, 2018.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[3] H. Assal, S. Chiasson, and R. Biddle. Cesar: Visual representation of source code vulnerabilities. In *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–8. IEEE, 2016.

[4] A. Bangor, P. Kortum, and J. Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *J. Usability Studies*, 4(3):114–123, May 2009.

[5] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pp. 27–38, 2012.

[6] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[7] M. Brehmer, S. Carpendale, B. Lee, and M. Tory. Pre-design empiricism for information visualization: Scenarios, methods, and challenges. In *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization*, pp. 147–151, 2014.

[8] J. Brooke. Sus: a "quick and dirty'usability. *Usability evaluation in industry*, 189, 1996.

[9] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 429–440, 2015. doi: 10.1109/ASE.2015.89

[10] T. T. Dang and T. K. Dang. An extensible framework for web application vulnerabilities visualization and analysis. In *International Conference on Future Data and Security Engineering*, pp. 86–96. Springer, 2014.

[11] M. Eggermont, S. Knudsen, R. Pusch, and S. Carpendale. Biomole: Visualizing functional co-occurrence. In *IEEE VIS*, pp. 20–25, 2019.

[12] Facebook Inc. React.js. `https://reactjs.org/`. Accessed: 2021.

[13] Fraunhofer SIT. VUSC - Der Codescanner. `https://www.sit.fraunhofer.de/vusc/`. Accessed: 2021.

[14] A. Furnham. Response bias, social desirability and dissimulation. *Personality and individual differences*, 7(3):385–400, 1986.

[15] J. R. Goodall, H. Radwan, and L. Halseth. Visual analysis of code security. In *Proceedings of the seventh international symposium on visualization for cyber security*, pp. 46–51, 2010.

[16] I. U. Haq and J. Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), Apr. 2021. doi: 10.1145/3446371

[17] L. Harrison, R. Spahn, M. Iannacone, E. Downing, and J. R. Goodall. Nv: Nessus vulnerability visualization for the web. In *Proceedings of the ninth international symposium on visualization for cyber security*, pp. 25–32, 2012.

[18] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, 2013. doi: 10.1109/ICSE.2013.6606613

[19] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681. IEEE, 2013.

[20] H. Lam, E. Bertini, P. Isenberg, C. Plaisant, and S. Carpendale. Empirical studies in information visualization: Seven scenarios. *IEEE transactions on visualization and computer graphics*, 18(9):1520–1536, 2011.

[21] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011.

[22] R. Martin. Common weakness enumeration (cwe v1. 8). *National Cyber Security Division, US Dept. Of Homeland Security*, 2010.

[23] Material-UI. React components that implement Google's Material Design. `https://material-ui.com/`. Accessed: 2021.

[24] Microsoft Corporation. TypeScript. `https://www.typescriptlang.org/`. Accessed: 2021.

[25] D. A. Norman and S. W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., USA, 1986.

[26] D. Park, S. M. Drucker, R. Fernandez, and N. Elmqvist. Atom: A grammar for unit visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 24(12):3032–3043, 2018. doi: 10.1109/TVCG.2017.2785807

[27] V. Pham and T. Dang. Cvexplorer: Multidimensional visualization for common vulnerabilities and exposures. In *2018 IEEE International Conference on Big Data (Big Data)*, pp. 1296–1301. IEEE, 2018.

[28] J. Smith. Supporting effective strategies for resolving vulnerabilities reported by static analysis tools. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 267–268. IEEE, 2018.

[29] The MITRE Corporation. Common Vulnerabilities and Exposures. `https://cve.mitre.org/`. Accessed: 2021.

[30] M. Tory and T. Moller. Evaluating visualizations: do expert reviews work? *IEEE computer graphics and applications*, 25(5):8–11, 2005.

[31] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[32] R. A. Virzi. Refining the test phase of usability evaluation: How many subjects is enough? *Human factors*, 34(4):457–468, 1992.

[33] Y. Wan, C. Q. Tan, Z. G. Wang, G. Q. Wang, and X. J. Hong. An effective visual system for static analysis of source code. In *Advanced Materials Research*, vol. 433, pp. 5453–5458. Trans Tech Publ, 2012.

[34] L. Wilkinson and M. Friendly. The history of the cluster heat map. *The American Statistician*, 63(2):179–184, 2009.

[35] H.-M. Wu, S. Tzeng, and C.-h. Chen. Matrix visualization. In *Handbook of data visualization*, pp. 681–708. Springer, 2008.