

# DaV<sup>3</sup>is: Data Flow-Based Vulnerability Verification Through Visualization

Tobias Mertz  and Steven Lamarr Reynolds  and Jörn Kohlhammer 

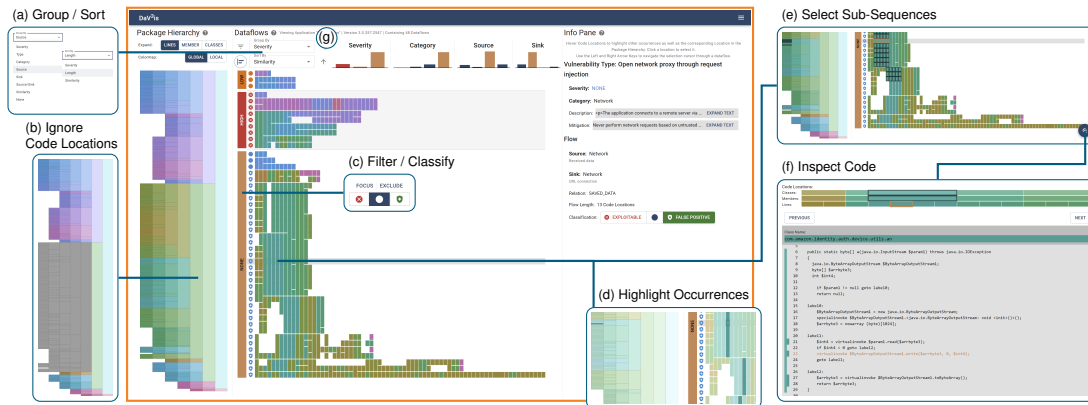


Fig. 1: An overview of the DaV<sup>3</sup>is interface in the center, showing the code structure as horizontal icicle plot and the data flows corresponding to the detected vulnerability candidates as sequence diagram. The annotations indicate the core interaction techniques. (a) shows the selection of grouping and sorting attributes, (b) shows the filtering of sub-trees within the code location hierarchy, (c) shows the application of filters or classifications via group headers, (d) shows the hover-highlighting of code location occurrences, (e) shows the selection of sub-sequences, and (f) shows the inspection of the decompiled code associated with the selected data flow. The histograms at (g) show the distribution of meta attributes across the data flows.

**Abstract**—Vulnerability verification is an important process in ensuring the security of software systems. To support users in this process, we present the design study of DaV<sup>3</sup>is, which utilizes visual event sequence analysis techniques to enable the comparison and tracing of automatically detected data flows through the software’s source code, thereby allowing users to take advantage of sequence similarities to reduce the verification workload. To that end, we characterize the domain problem based on input from domain users, describe our design rationale based on best-practices from the visual analytics literature, and evaluate individual design decisions, usability, and utility in studies with three stakeholder groups. The evaluations yielded overall positive responses, showing the suitability of our design and providing valuable insight for future research.

**Index Terms**—Visual analytics, event sequence analysis, vulnerability verification.

## 1 INTRODUCTION

Event sequences are a commonly encountered temporal data type. Each sequence consists of a series of time-stamped events, where different events are distinguishable by their event type. This abstraction is applicable to many domains, including healthcare, cybersecurity, and behavior analysis. The event type is usually a categorical attribute that can assume one of a discrete set of values. However, in some application scenarios, the event type elicits a structure, such as a hierarchy, that allows the investigation of relationships between the different event type values as part of the overall analysis. In this paper, we address the domain problem of software vulnerability verification via data flow analysis, which elicits such a hierarchical event type.

Given our increasing reliance on digital services, their security is critical. But many development teams do not have cybersecurity experts on hand, so ensuring software security falls upon regular developers without such expertise. Consequently, software scanners are being

developed, to automatically find vulnerabilities within the code. Static scanners investigate the code without execution and can, thus, facilitate high code coverage [3], but they incur false positive findings, because they approximate the run-time environment. As a result, users must perform vulnerability verification, which entails investigating the findings individually and deciding which represent exploitable vulnerabilities. Vulnerability verification is especially challenging for the results of a taint analysis [3, 10], which is a vulnerability detection technique for discovering potential data leaks or injection attacks. To verify these vulnerabilities, a developer must trace the detected data flow, a sequence of instructions, through the code base to gain an understanding of the program’s behavior along this path. Repeating this process with the entire list of vulnerabilities is very time consuming and not feasible in many cases [34, 56], but appropriate visual support can reduce the overall workload by enabling developers to take advantage of similarities between the detected data flows. Additionally, the event type of the sequential data in this application scenario is hierarchical, because it describes locations within the hierarchically structured source code.

This domain problem yields design challenges due to the nature of required insight into the data as well as the hierarchical event type, both of which have so far rarely been investigated in the event sequence literature. We tackle these challenges in our design study of DaV<sup>3</sup>is—Data flow-based Vulnerability Verification through Visualization. DaV<sup>3</sup>is is a visual interactive tool to make data flow analysis for vulnerability verification more accessible and efficient for developers. Our main

- Tobias Mertz and Jörn Kohlhammer are with Fraunhofer IGD and TU Darmstadt. E-mail: {tobias.mertz | joern.kohlhammer}@igd.fraunhofer.de.
- Steven Lamarr Reynolds is with Fraunhofer IGD. E-mail: steven.lamarr.reynolds@igd.fraunhofer.de.

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxx/TVCG.201x.xxxxxx

contributions are:

- A domain characterization of the data flow analysis process for vulnerability verification.
- The design study of DaV<sup>3</sup> is with the goal of making vulnerability verification more accessible to software developers.
- The qualitative evaluation of DaV<sup>3</sup> is with users of different experience levels, thus, offering different perspectives on the usability and usefulness of the tool.

## 2 BACKGROUND

### 2.1 Simultaneous Insight

As described above, the data flow analysis problem yields a particular design challenge, due to the nature of required insight. Monroe et al. define two distinct levels of insight to be gained from the analysis of sequential data [43]:

- **Intra-record understanding** describes the understanding of what happens within a single sequence. This includes common tasks such as the identification of reoccurring sub-sequences, or causality analysis, which require the tracing of events in the sequence.
- **Inter-record understanding** describes the understanding of relationships and trends between multiple sequences. This level is much more prominent in the literature, including such tasks as the identification of co-occurring sub-sequences, the comparison of sequences, or the analysis of event type distributions at different points in time.

For the discussion of intra-record understanding, we differentiate between first-order and higher-order sub-sequences, based on the definitions by Blaas et al. [9]. First-order sub-sequences are sub-sequences of exactly two successive events, while higher-order sub-sequences contain more than two successive events. Based on these definitions, Blaas et al. demonstrate that classical node-link diagrams can produce ambiguous visual representations, where it is no longer possible to uniquely identify a given higher-order sub-sequence if it intersects with another sequence or itself. From this limitation, they infer two desirable properties for visualizations supporting higher-order sub-sequence analysis—uniqueness, meaning that each unique sequence shall be represented by a unique visual representation, and continuity, which means that each sequence shall be discernible as one continuous entity. The combination of these two properties disambiguates the visual representation and guarantees traceability of the sequences, allowing the identification of higher-order sub-sequences. As such, the required level of intra-record understanding has a significant impact on visualization design choices. Our application scenario requires high levels of both intra-record understanding and inter-record understanding. We call this combination of both levels of understanding **simultaneous insight**. In the following, we elaborate how the required simultaneous insight manifests within the domain problem.

### 2.2 Data Flow Analysis

During taint analysis, a static scanner traces the propagation of tainted data through the codebase, starting at a data source, and detects potential paths of that data reaching a sink, such as a publicly accessible output. Tainted data refers to information that originates from untrusted or external sources, such as user input or network communication. Through this analysis, the scanner can discover vulnerabilities that are not caused by a single code location but by a coherent path through the code base. However, due to the nature of these vulnerabilities, a developer must get an in-depth intra-record understanding of the entire path to determine whether this vulnerability is a true positive.

For example, Listing 1 shows a function `foo` that calls a function `bar` twice with different parameters. Within `bar`, the second parameter (`outputSelect`) causes a branch that writes the data to an insecure sink, such as a web socket or an unprotected log file, if the parameter equals 1 or otherwise to a secure sink, for example to an encrypted file. However, the scanner does not trace values through the control flow, because it is intractable to do so generally. For this reason, it

```
1 function foo() {
2     bar(UntaintedSource.read(), 1)
3     bar(TaintedSource.read(), 2)
4 }
5 function bar(data, outputSelect) {
6     if (outputSelect == 1) {
7         InsecureSink.write(data)
8     } else {
9         SecureSink.write(data)
10    }
11 }
```

Listing 1: This example produces a false positive during taint analysis, because the scanner does not know which branch is taken in `bar`.

cannot know which of the two branches is executed within each of the two calls of `bar`. The scanner approximates the program's behavior by assuming all paths are valid. Thus, it detects a path in which the tainted data is written to the insecure sink, even though this never actually occurs in the code. A concrete example of such a finding can be found in Fig. 2. To correctly identify this data flow as a false positive, the entire path must be checked for inconsistencies.

Thus, the developer must trace the data flow through the code base to gain an understanding of the program's behavior along this path (intra-record understanding). This is a time consuming process, especially if the scanner detects many potential data flows. However, when verifying similar data flows in succession, developers can draw on knowledge gained from understanding a shared sub-sequence in the previous data flow. This reduces redundancies in the understanding process and, thus, the overall workload. But to be able to take advantage of these similarities, developers need visual support, to identify similar data flows and the sub-sequences they share (inter-record understanding). Therefore, this domain problem requires simultaneous insight.

## 3 RELATED WORK

### 3.1 Levels of Insight in Sequence Analysis

Within this section, we present an overview of the event sequence analysis literature, classified by the degree to which their application scenario requires the two levels of insight. Guo et al.'s survey of the field [30] provides insight into the present level of inter-record understanding, showing that most research focuses on the analysis of sequence collections. Examples to the contrary are *Chromograms* [71], *IDMVis* [77], and Tanahashi and Ma's storyline visualizations [57], which are designed to analyze single sequences. But they all partition the sequence into smaller chunks to apply techniques for the visualization of sequence collections. *Scribe Radar* [75], on the other hand, enables the comparison of exactly two sequences, by explicitly showing the difference between their event-frequency distributions.

In terms of intra-record understanding, the field is more diverse. Application scenarios that only require insight about single events are most rarely encountered in the literature. An example is the event frequency comparison in *Scribe Radar* [75]. The analysis of first-order sub-sequences [12, 33, 70, 73, 78] or higher-order sub-sequences [13, 38, 39, 44, 49, 50, 71] is more frequently required. Similarly, there are approaches that focus on sequence stages [24, 28, 29, 65]. Stage analysis can be interpreted as the opposite approach to higher-order sub-sequences. For higher-order sub-sequences, a sequence is analysed in terms of sub-sequences that occur within it, such as maximal sequential patterns [38]. The individual sub-sequences need to be traceable, but the composition of the entire sequence out of the sub-sequences is not necessarily relevant. Approaches focusing on stage analysis, such as *EventThread* [29], enable analysis of the high-level progression of the sequence, which is represented by the detected stages, but within each individual stage, sequences are not traceable. This is the main distinction between these types of approaches and those that require understanding of entire sequences [5, 8, 15, 40, 43, 57, 74, 77], where each sequence must be cohesive and traceable in its entirety.

Our application scenario is at the intersection between requiring understanding of entire sequences in a sequence collection, thus it requires simultaneous insight. This combination can also be found in *CoCo* [40], *LifeFlow* [74], *Sequence Braiding* [5], and *Sequence Synopsis* [15]. However, none of these approaches can be applied directly to our domain problem. *CoCo*'s design is focused on the statistical comparison of patient cohorts. In vulnerability verification, the statistical comparison between sequences is not relevant, because the existence of a frequently occurring sub-sequence does not necessarily indicate criticality or likelihood of a true positive. *LifeFlow*'s aggregated tree visualization manages to display very large sequence collections on one screen, but the aggregation method relies on exact matches of sequence prefixes, which we rarely encountered in vulnerability data flows. To ease this constraint, Monroe et al. extend the system with filter- and aggregation-based sequence simplification options in *EventFlow* [43]. This is similar to our approach, but we utilize the hierarchical code structure for event merging. *Sequence Braiding*, on the other hand, combines the tracing of individuals from storyline visualizations with the visualization of event distributions from Sankey diagrams. However, the approach will cause many edge crossings for larger amounts of sequences while also making it difficult to trace an individual line in a set of multiple parallel lines. Thus, for data sets with many sequences, *Sequence Braiding* focuses on facilitating inter-record understanding, while its ability to facilitate intra-record understanding diminishes. *Sequence Synopsis* employs the minimum description length principle to show many sequences at once. This approach shows a shared summary of an aggregated group of sequences along with the variations of the individual sequences from the summary. To make tracing easier, the raw sequences can also be displayed. While this design can facilitate simultaneous insight, Chen et al. justify the design of the summary view with supporting users in compiling descriptive information about groups of sequences. Thus, the summary view is primarily designed for inter-record understanding, while the intra-record understanding is achieved by the display of the raw sequences. This summary task is not relevant for our domain problem due to the same reasons we elaborated on in our discussion of *CoCo*. In addition, the split of the insight levels into separate visualizations introduces a level of abstraction between which users need to translate their findings. The comparatively small size of the typical data sets in our domain problem (see Tab. 1) makes such high levels of abstraction unnecessary and allows for simpler and more intuitive designs. Our approach is most similar to *Sequen-C* [39], but we incorporate the hierarchical structure of the event type into the similarity metrics and visual encoding. In addition, we use a different approach for sequence simplification, because the code locations in a data flow are all of equal importance to the analysis.

### 3.2 Structured Event Types

In many real world applications, the event type is an abstraction of more complex data to reduce visual complexity and make the identification of common patterns easier. But this abstraction loses information in the process. While it is possible to map different data dimensions to different visual channels, this is rarely done in event sequence visualization. The more common approach is to switch a singular encoding between the individual data dimensions that are associated with the events [66]. But this approach can only show one dimension at a time. Few approaches have so far utilized the non-temporal structure within the event data to reduce this information loss.

For example, *VisTracer* [20] implements a visualization of traceroute paths to detect routing anomalies. The IP addresses traversed during a traceroute are part of a three-level hierarchical structure, consisting of the individual IP addresses at the leaf-level, the autonomous systems containing the IP addresses at the intermediate level, and countries on the upper level. The visualizations in *VisTracer* can be toggled between the hierarchy levels to facilitate analyses at multiple levels of granularity. Similarly, Liu et al. [38] encountered hierarchical event types within their analysis of web clickstreams, because the URLs associated with each view of the web page have a hierarchical structure. But they discovered that the hierarchy lacks sufficient granularity to capture the clickstreams' semantics, as different interactions can happen within the

same view. Pretorius and Van Wijk [51] visualize state transition graphs with multivariate data associated with each state. They utilize this data by performing a hierarchical clustering on the states and visualizing the cluster hierarchy itself. The transition sequences are then shown as edges between the leaves of the hierarchy. *Cadence* [25] also uses a hierarchical clustering approach to derive event types from multivariate event sequences. But in contrast to Pretorius and Van Wijk's approach, the clustering happens interactively, guided by the user.

While the visualization of Pretorius and Van Wijk displays the sequences in their hierarchical context, the other approaches flatten the hierarchical structure to a list of categories to be used within a traditional event sequence visualization. This enables the use of prevalent visual encodings, but it loses the hierarchical context in the process. On the other hand, the hierarchical visualization of Pretorius and Van Wijk prohibits sequence tracing and thus limits the intra-record understanding to insights based on first-order sub-sequences. Due to the necessity of intra-record understanding of the entire sequence and the importance of the hierarchical event type, our approach includes the hierarchical structure directly, without flattening, into the event type encoding, user interaction, and computation within an event sequence visualization.

### 3.3 Vulnerability and Data Flow Visualization

Most research on vulnerability visualization focuses on the categorical attributes of the detected vulnerabilities [16, 53]. Usually, each vulnerability is associated with a single location in the code [23, 69] or with a single device in a network [1, 32]. For the analysis of the data flow, these approaches are not sufficient, because they do not consider coherent paths. But their findings regarding the meta data can be applied to the meta data present in data flow analysis as well.

Data flow visualizations usually focus on the data flow between different devices [63], processes [18] or applications [37]. These approaches often focus on investigating first-order sub-sequences or singular entire sequences. An exception is the system presented by Fujiwara et al. [21], which can display multiple entire sequences, but is limited in its scalability to larger data sets. The examination of static taint analysis results has not yet been addressed in visualization research. Most taint analyzers are command line applications that produce results in a machine-readable text format only. For some scanners, frontend GUIs exist, allowing users to view the list of findings, but to our knowledge, *Apprecium* [60] and *VisualDroid* [58] are the only tools to facilitate the visual analysis of taint propagation. But *Apprecium*'s Node-Link diagram-based visualization produces very cluttered graphs at function-level granularity, when multiple data flows are displayed. Furthermore, the usefulness and usability of the visualization have not been evaluated with end users. In contrast, Tang et al. employ a user-centered approach to develop *VisualDroid* [58]. While *VisualDroid* is based on the prevalent taint analyzer *FlowDroid* [3], it does not actually visualize the analysis results. Instead, it shows the application's call graph to facilitate the manual identification of suspicious behaviors.

While the visual analysis of taint propagation has not yet been sufficiently addressed, it is strongly related to the field of malware analysis. In their survey of the field, Wagner et al. [67] classify malware analysis into three categories. Visualizations for individual malware analysis display the behavior of single malware to facilitate an in-depth understanding of its effects [36, 52]. Meanwhile, malware comparison tools show derived features [26] or image representations [31] of many malwares to enable comparison. Finally, malware summarization tools [76] display common patterns within a set of malware to give a coarse understanding of the entire set at once. While the individual malware analysis tools manage to facilitate intra-record understanding of the malware's behavior, and the other two categories focus on inter-record understanding. Wagner et al. conclude that the combination of the techniques, and thus the facilitation of simultaneous insight, is an important direction for future research that is left open.

## 4 DOMAIN CHARACTERIZATION

The design of DaV<sup>3</sup> builds on *VUSC* [2], a commercial code scanner that includes a static taint analyzer based on *FlowDroid* [3]. *FlowDroid* is well-suited to this effort, because it is widely used and is one of few

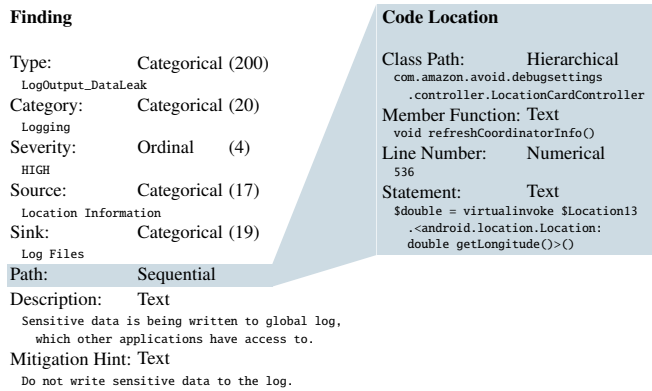


Fig. 2: The data specification of the findings resulting from VUSC's taint analysis. Values in braces indicate the number of admissible values. Each attribute is followed by a value from a real-world finding as example.

Application	# Flows	Max. Length	# Locations	Figure
ibisPaint	44	49	289	Fig. 9
WhatsApp	99	31	321	Supp. C.1
Prime Video	48	71	267	Fig. 1
Free Gold Messenger	17	47	105	Supp. C.1
Honda Connect	14	17	79	Supp. C.1

Table 1: Volume of several example scans of real-world applications. Data sets and more figures are included in the supplemental materials.

taint analyzers whose output includes full path information. Most taint analyzers only determine whether a data sink is reachable from a data source by tracking taint propagation. *FlowDroid* enhances the precision of the finding by tracing the tainted data backwards, starting from the sink, to determine exact paths through the individual statements that propagated the taint. This approach provides valuable information to users for the verification and mitigation of the findings. Despite multiple existing studies [34, 46, 56] showing that insufficient information is one of the most prominent usability issues in static analysis tools, most static taint analyzers omit this analysis step to reduce computation time.

For this reason, software developers in general are currently not familiar with the verification process of static taint analysis results. During our characterization of the domain problem, we, hence, synthesized our findings from the domain literature as well as from input of software developers, our target user group, and cybersecurity experts, who are familiar with the verification process. To that end, we interviewed stakeholders from both groups and observed their strategies during vulnerability verification. We focused on learning the details of the verification process as it is intended to be performed by the experts, as well as the additional information needs of software developers. To that end, we interviewed four software developers from our research group without cybersecurity background and no affiliation to the project as well as two cybersecurity experts from the *VUSC* development team. We then abstracted a model of the vulnerability verification workflow of the experts using a hierarchical task characterization [77]. This methodology structures the workflow in terms of high level goals that are incrementally divided into sub-tasks, down to the low-level queries and interactions to be executed with the tool.

#### 4.1 Data

Figure 2 details the schema of *VUSC*'s vulnerability findings. Each finding is described by its type, as well as the category and estimated severity of that type. To ease understanding, *VUSC* includes a textual description and mitigation hint for each vulnerability type. The severity is an ordinal scale that indicates the associated risk, were the vulnerability to be identified as a true positive. It ranges from *none*, describing code smells and similar code style violations, to *high*, indicating significant security concerns, such as potential remote code execution or

arbitrary file modifications. Additionally, each vulnerability finding resulting from the taint analysis contains a source and sink categorization, indicating what type of input the data is read from and what kind of output it is written to, as well as the path of the data flow. It includes the sequence of statements that contributed to the taint propagation, such as assignments, method calls, and return statements. Each individual code location within the data flow is described by its class path, the name of the member function, the line number within the class file as well as the statement within that line. The data flow presents users with a description of how the tainted data propagates through the system, but to understand the program behavior along this execution path, the intermediate statements that are part of the control flow but not of the data flow must be reviewed as well. Thus, it is important to analyze the data flow in the context of the entire code base. To that end, *VUSC* provides the content of the individual class files via a separate API.

One defining characteristic of *FlowDroid* is that it can scan compiled Android applications. In contrast to compiled native code, Android packages contain readable class paths and member function names, which, together with the line numbers, form a strict hierarchy that mirrors the structure of the source code. To survey data set scale, we scanned 50 of the top apps in the German Google Play Store. The findings included up to 100 data flows and up to 70 code locations per data flow. The total number of unique code locations could theoretically be as large as the product of both, but we observed only up to 400 unique code locations, indicating a large overlap between the data flows. Table 1 shows the volume of several of our scans.

#### 4.2 Users

Our target user group consists of software developers, who are experienced in neither vulnerability verification nor the use of advanced visual analytics tools. From our developer interviews, we learned that while developers have good knowledge of the application's code and its structure, they may have difficulties understanding the specific cybersecurity terminology or the semantics behind individual vulnerability types. Additionally, vulnerability verification is not software developers' primary task, but only a small part of software development, which is why they can not dedicate a large amount of time to it. For this reason, the system must be easy to understand and should not employ advanced visualization or interaction techniques that would require extensive training. Furthermore, prior research has shown that the large amount of false positives in the output and the laborious process of verification are main detractors from the widespread adoption of static analysis tools [34, 56]. This is a further indication that software developers do not want to spend much time analyzing the detected vulnerabilities and their associated data flows. In fact, interview participants stated, the ideal help would be a scanner that creates a prioritized issue list of vulnerabilities that they need to mitigate. However, as stated by the security experts, no scanner can create such a list directly, because they do not possess all the information that is required for verification and prioritization. Thus, our design must enable developers to create their issue list from the scanner's findings by applying their knowledge of the application's run-time environment to prioritize and verify the detected vulnerability candidates. To be useful in practical application, this process has to be completed as quickly as possible.

#### 4.3 Tasks

Table 2 shows an excerpt of the abstracted verification process, containing the three main high-level goals and associated low-level user tasks, which we observed during our security expert interviews. The table also contains an abstraction of each task according to the multi-level task typology by Brehmer and Munzner [11]. The full task hierarchy can be found in the supplemental materials.

The analysis process is split into multiple phases of decision-making and sense-making. Experts start off by identifying the most relevant data flows (**G1**) via their various meta attributes. For example, flows with high severity are probably the most relevant. This is an iterative decision-making phase in which the experts first review the distribution of an individual meta data attribute (**T1**) and select the values of high relevance or discard flows with values of low relevance (**T2**). This is

Goals	Tasks
<b>G1:</b> Find the most relevant data flow.	<b>T1:</b> View meta data distribution of a relevant attribute. <i>discover — browse — summarize</i>   meta data distribution → relevant values
	<b>T2:</b> Apply meta data filters to data flows. <i>produce</i>   set of data flows → relevant sub-set
<b>G2:</b> Verify the selected data flow.	<b>T3:</b> View selected data flow's meta data <i>discover — lookup — summarize</i>   data flow → meta data values
	<b>T4:</b> Trace selected data flow through the code base. ▶ <i>discover — browse — summarize</i>   data flow → code locations
	<b>T5:</b> View code context of the selected code location. ▶ <i>discover — lookup — summarize</i>   code location → surrounding code
	<b>T6:</b> Select classification for the selected data flow. <i>produce</i>   data flow → classification
<b>G3:</b> Find the most similar data flow.	<b>T7:</b> Find similar candidate flows. ▲ <i>discover — browse — summarize</i>   set of data flows → similar sub-set
	<b>T8:</b> Estimate similarity between two data flows. ▲▶ <i>discover — browse — compare</i>   two data flows → shared sub-sequences

Table 2: The three main analysis goals and corresponding tasks extracted from our expert interview. We also include abstractions of the individual tasks according to the questions *why?* and *what?* of the multi-level task typology by Brehmer and Munzner [11]. Some of these tasks facilitate intra-record understanding (▶) or inter-record understanding (▲).

repeated iteratively over the available meta attributes, to reduce the set of data flows to those most relevant. From the resulting sub-set, a starting data flow is chosen arbitrarily to be the first verification target.

The actual verification of the selected data flow is the second phase of the analysis (**G2**). This phase contains both decision-making and sense-making components. Here, the experts refer to the meta data again. Based on the type of vulnerability, as well as the source and sink (**T3**), users may be able to immediately classify a finding as a false positive (**T6**). In this case, **G2** is a pure decision-making process, which saves much analysis time, because the time-intensive sense-making step can be skipped. But if this decision is not clear, the user must understand the path of the data flow and make a judgment based on the possibility of that path being executed (**T4**). Experts first refer to the start and end of the data flow to understand the input and output of the flow. They then investigate the individual code locations sequentially (**T5**). Based on the knowledge gathered during this investigation, experts can determine whether the vulnerability represents a true positive (**T6**).

According to the experts, the sense-making process (**T4–T5**) is the most time consuming part of the analysis. Saving time during this process by reducing redundant work can significantly reduce the overall workload. This can be achieved via visual support that allows to view similarities and differences between the data flows. If two data flows share a common sub-sequence of instructions, the semantics of that sub-sequence only need to be learned once. However, for this approach to work, the analysis tool needs to provide two pieces of information. First, users must be able to identify which data flows are actually similar, in order to review them successively, such that the information about shared sub-sequences remains in their working memory. Second, users must see which exact statements in the two data flows are the same and which differ, to be able to decide which statements require close inspection and which they can skip.

As described above, no currently available tool offers appropriate support to capitalize on the similarity between data flows for vulnerability verification. As such, this process optimization is not part of the current expert workflow. However, the security experts we interviewed confirmed that this approach is very likely to reduce the workload if appropriately supported. For this reason, we include a third decision-making phase into the process with the high-level goal of estimating the most similar data flow to proceed with (**G3**). To be able to find the most similar data flow, users first need to be able to perform coarse comparisons over the entire data set, to find similar candidates (**T7**). Then, they perform detailed pairwise comparisons between the previously verified flow and the candidates (**T8**). If no similar candidate remains, the user starts the process over with the prioritization of the remaining data flows via meta attributes (**G1**).

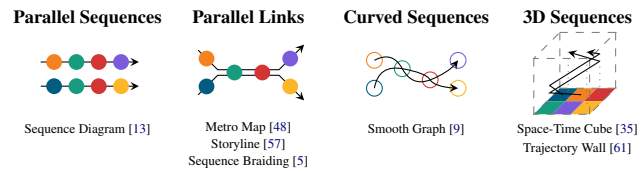


Fig. 3: The four approaches to disambiguate sequence intersections in visualizations of sequence collections. For each approach, the corresponding visualization types are listed below. The pictogram for the Parallel Sequences approach has been adapted from Guo et al. [30].

In the context of software development, the verification process is followed by a prioritization of the identified true positives to result in the final issue list of vulnerabilities to mitigate. While users' considerations during this prioritization may differ from their priorities during the verification itself, their decisions are based on the same information. Thus, this last step is largely equivalent to **G1**.

## 5 DESIGN RATIONALE

We developed the design of DaV<sup>3</sup> iteratively by incorporating feedback from the two groups that we also interviewed during our domain characterization. From the feedback sessions with the cybersecurity experts, we gained insight into the relevant characteristics of the data, the required functionality to support the workflow, as well as the understandability of our visual representations. From the software developer interviews, we gained more detailed feedback regarding our visual designs, usability concerns, as well as the information needs of users without cybersecurity expertise. Within this section, we present the considerations that led to the final design of DaV<sup>3</sup>, which can be seen in Fig. 1. It consists of a three column layout with the code structure represented as a horizontal icicle plot in the left column, the data flows as a sequence diagram from left to right in the center column, and a detail view in the right column, showing the attributes of the currently focused element. The remainder of this section presents our design considerations structured by the core design challenges.

### 5.1 Facilitating Simultaneous Insight

Due to the target user group, which does not possess expertise in visual analytics and has limited time available, we set the goal of facilitating both the comparison (**T7, T8**) and the tracing (**T4**) of the data flows with the same visualization type. This reduces the cognitive load on the users because they do not have to translate their findings between different representations. We consider this approach of simultaneous insight within a single visualization to be the main design challenge.

Visualizations for simultaneous insight incur two core requirements. First, the visualization needs to display many sequences at once, for inter-record understanding. Second, the visualization needs to ensure that each individual sequence is entirely traceable, for intra-record understanding. As demonstrated by Blaas et al. [9], the latter implies a need for uniqueness and continuity of the sequences' visual representations. This requires careful consideration about how to handle visual intersections of sequences, without introducing ambiguities for sequence tracing. It must always be possible to determine which of the sequences entering an intersection corresponds to which of the sequences leaving the intersection. During our survey of the related literature, we found that many visualization types for sequential data could not fulfill this requirement. This prominently includes traditional node-link diagrams [33] or Sankey diagrams [54], because they introduce ambiguities at intersections. But four general approaches fulfill both requirements (Fig. 3). Before discussing the advantages and disadvantages of these approaches for our application scenario, we will briefly describe each approach. To establish a common frame of reference, we use the terminology of node-link diagrams, where the events are represented by nodes and the sequences by chains of links.

The *Parallel Sequences* approach arranges sequences as parallel straight lines, so they never intersect. Here, the same node may occur multiple times, thus a separate encoding of the nodes is necessary. Due

to the absence of intersections, the links themselves are usually not displayed explicitly. The *Parallel Links* approach loosens this constraint. Shared links are drawn as parallel lines, while sequences as a whole are not parallel. Additionally, links that are incident with the same node are drawn through the node in parallel. Intersection angles between links are also frequently constrained in this approach. The *Curved Sequences* approach relies on curve-continuity instead of parallelism to distinguish sequences. Each sequence is represented by a continuous spline curve. The continuity constraint of the splines makes sure that the curvature of a sequence entering a node is consistent with the curvature leaving the node, so the two links can be associated. Finally, the *3D Sequences* approach incorporates a third display dimension, to avoid intersections either by stacking lines along this dimension, as in the trajectory wall, or by orienting the sequence progression along this dimension, as in the space-time cube. While these techniques have yet only been applied to spatio-temporal data, they could also be applied to abstract spaces, especially in the field of software visualization, where the geographical metaphor of a code city is frequently applied [72]. However, due to 3D visualizations' issues with perspective distortions, occlusion, and viewport navigation, we quickly disregarded this approach and only include it here for the sake of completeness.

To aid our decision among the three remaining approaches, we characterized the comparison problem (**G3**) in terms of the comparison elements, challenges, and strategies, according to Gleicher's abstraction [22]. Based on this categorization, the most problematic design challenges can be identified and a suitable visual design can be selected. In our application scenario, the comparison element is the set of data flows in which similarities need to be identified. The main challenge is that the set contains a large number of items that are themselves of a complex nature. The workflow within **G1** and **G3** corresponds to the *select subset* strategy. Thus, our visual design must be able to cope with many complex items while facilitating the identification of similarities and the *select subset* comparison strategy.

Gleicher also classifies design solutions within three categories: juxtaposition, superposition, and explicit encodings of similarity. Since the actual comparison items need to be visible for intra-record understanding, we can apply juxtaposition (*Parallel Sequences*) or superposition (*Parallel Links* or *Curved Sequences*). However, juxtaposition separates items, making it harder to compare those that are far apart, while superposition tends to cause clutter as the number of items increases. The cluttering issue is even further amplified by item complexity.

There are approaches to deal with both limitations. Common ways to reduce clutter in visualizations rely on aggregation, but the aggregation of links violates the uniqueness property and, thus, limits intra-record understanding. Meanwhile, the effects of separation can be counteracted with a similarity-based ordering. Such methods place similar items in close proximity, to ease their comparison, while highly differing items, whose differences are more apparent, are placed further apart. Due to the importance of intra-record understanding (**T4**, **T5**, **T8**) in our application scenario, we believe that clutter is the more severe limitation. To avoid this problem, we decided to utilize the juxtaposition-based approach with the sequence diagram. In the following sections, we describe how we consolidate the visualization of the sequential data flows with the hierarchical source code and how we address the separation issues arising from the juxtaposed design.

## 5.2 Sequences of Hierarchically Related Elements

Because software developers are very familiar with their own code base and unfamiliar with software vulnerabilities, we decided to firmly anchor the visualization in the context of the code. As described earlier, the code base has an inherent hierarchical structure, given by the class paths, the member functions, and the line numbers. Visually representing this structure allows developers to navigate the code base (**T4**, **T5**) in a reference structure that they are used to within their development environment and, thus, to more easily translate between this familiar structure and the code locations on display in the sequence diagram. We approach this anchoring with two design decisions.

First, to visually communicate the code structure in the sequence diagram, we apply a hierarchical color map to the code locations. Our

algorithm is based on *Tree Colors* [59], but is optimized to achieve a better color map quality for comparisons between the hierarchy leaves [42]. This algorithm recursively partitions the available range of hues among the branches of the hierarchy, starting with the full 360 degrees of hue at the root node. Each node is then assigned the hue value at the center of its range. Luminance and chroma are varied throughout the layers of the hierarchy to distinguish between ancestors and their descendants. To avoid implying criticality via the code location colors, we exclude red and orange hues from the color map. We chose *Tree Colors* as basis for our implementation, because it is the state-of-the-art algorithm for coloring an entire hierarchy. More recent algorithms have instead focused on generating colors for a subset of hierarchy nodes, such as individual hierarchy levels in a semantic zooming environment [68].

As second representation of the code structure, we display the hierarchy explicitly along with its color map in a supplementary visualization. Here, we apply a hierarchical visualization type that displays the inner nodes, to enable user interaction with these nodes. We also believe the hierarchical nature of the color map is easier to understand if the inner nodes and their colors are visible. This decision further reinforced *Tree Colors* as a suitable color generation algorithm, because it computes colors for the inner nodes as well. For space efficiency, we narrowed our design space to implicit hierarchy visualizations, resulting in icicle plots and sunbursts as potential candidates. We finally chose a horizontal icicle plot, because it can be used in a rectangular but non-square shape, which enables a more efficient use of screen space in a three column application layout. A second advantage of a horizontal icicle plot is that it resembles the nested representation of directory structure as it is shown in most code editors. The icicle plot in our design serves as legend for the color map, to help developers associate the colors with the familiar code structure, and as interaction widget. The possible interactions with the hierarchy are described in [Sec. 5.4](#).

To make this connection explicit, we interactively link the two visualizations by highlighting all occurrences of a hovered (see [Fig. 1d](#)) or selected (see [Fig. 1e](#)) code location in either visualization. In our feedback sessions, we observed that participants intuitively associated the vertical position of the hierarchy's leaves with the vertical positions of the data flows. But these two axes are unrelated. For this reason, we decided to flip the hierarchy horizontally, resulting in the unusual orientation from right to left. This moves the leaves further away from the sequences and places the body of the hierarchy in-between, reducing the implicit connection between the two axes. Since this change, we observed no further user confusion regarding the vertical positions of code locations in the hierarchy or data flows in the sequence diagram.

## 5.3 Separation of Sequences

As described above, juxtaposed designs incur the issue of separation, impeding the comparison of items that are far apart [22]. This issue can be addressed with a similarity-based ordering, reducing the distance between similar items that require detailed comparisons. This problem can be interpreted as a matrix reordering problem, in which we try to maximize values close to the diagonal of the sequence-similarity matrix. In their survey of matrix reordering approaches, Behrisch et al. [7] describe this optimization target as a *block pattern*. During their experiments, they observed that optimal leaf ordering algorithms [4] produce well organized block patterns, with the added benefit that they also promote the visual identification and the explicit extraction of clusters by applying a threshold, both of which can support users in **T7**.

The hierarchical clustering algorithm at the basis of our approach is an agglomerative clustering of the similarity matrix, because it is the most commonly applied hierarchical clustering algorithm and easy to implement. During the clustering process, we need to compute similarity or distance on three different levels of abstraction—between code locations, between data flows, and between clusters of data flows. For all three levels, we investigated various distance metrics. Between clusters, we investigated single, average, and complete linkage. We finally selected complete linkage as default, because it resulted in the most balanced dendrograms in our preliminary experiments, but we offer the option to switch to the others as well.

For the distance between data flows, we investigated commonly

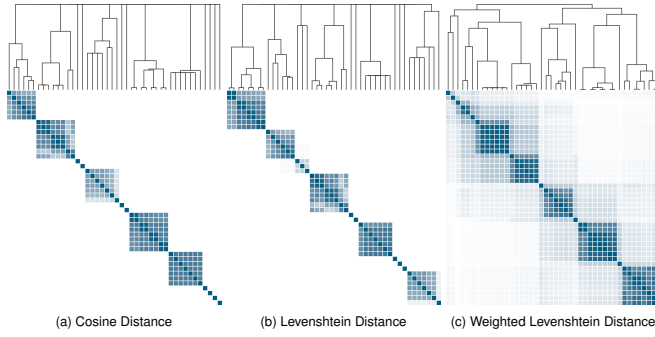


Fig. 4: Three similarity matrices and dendrograms of the ibisPaint data set resulting from different distance metrics. Similarity is given by the normalized distance between zero (blue) and one (white).

applied distance metrics from the event sequence literature. However, standard metrics, such as cosine distance (Fig. 4a) or Levenshtein distance (Fig. 4b), compare the elements within the sequences for exact equality, because the event type is usually treated as a categorical variable. In our application scenario, the code locations are not a flat categorical list, but are organized hierarchically. This hierarchical structure can be incorporated into the sequence distance, by employing a weighted Levenshtein distance.

$$D_{i,j} = \min \begin{cases} D_{i-1,j} & +c_+ \\ D_{i,j-1} & +c_- \\ D_{i-1,j-1} & +c_s(x_i, y_j) \end{cases} \quad (1)$$

Like the regular Levenshtein distance, the weighed Levenshtein distance is computed by filling the cells of a dynamic programming matrix  $D$ . Here, each cell  $D_{i,j}$  in the matrix represents the minimum cost of modifying the first  $j$  items of sequence  $y$  to correspond to the first  $i$  items of sequence  $x$ . The total distance between the two sequences is then read from the bottom right cell of the matrix. The first row and column of the matrix are initialized with  $D_{i,0} = i \cdot c_+$  and  $D_{0,j} = j \cdot c_-$  and subsequent cells are computed according to Eq. (1). This computation applies scalar costs to each of the edit operations, insertion ( $c_+$ ), deletion ( $c_-$ ), and substitution ( $c_s$ ). In the case of the regular Levenshtein distance,  $c_+$  and  $c_-$  represent a constant cost of 1 while  $c_s$  yields 0 if  $x_i = y_j$  and 1 otherwise. Meanwhile, for the weighted Levenshtein distance, the three cost parameters can be defined to suit the application scenario in question. For example, in spelling correction, the substitution cost between two letters is based on the distance between the letters on the keyboard [41]. This weighted adaptation can capture more nuanced notions of similarity than the exact equality criterion.

To apply this approach, we require a meaningful distance metric between code locations. Our chosen metric is based on good programming practices: the structure of the code base represents semantic relationships [6]. Hence, we can infer semantic relationships between code locations from their distance in the hierarchical structure of the code base. The standard formalization of the distance between two nodes in a hierarchy is the tree distance [17]. The tree distance between two hierarchy nodes  $a$  and  $b$  is given as the number of edges traversed on the path between them. We normalize this metric by the maximum possible distance between two hierarchy nodes, which is twice the total height of the hierarchy. In our application scenario,  $a$  and  $b$  represent different code locations. The resulting metric is given by:

$$d_{Tree}(a,b) = \frac{d(a) + d(b) - 2 \cdot d(a \cap b)}{2 \cdot \hat{d}} \quad (2)$$

Where  $d(x)$  denotes the length of the path between the node  $x$  and the root node,  $a \cap b$  denotes the closest common ancestor of  $a$  and  $b$ , and  $\hat{d}$  denotes the total height of the hierarchy. The computation of the tree distance introduces an additional factor in  $\mathcal{O}(\hat{d})$  to the computational

complexity, resulting in  $\mathcal{O}(n^2 l^2 \hat{d})$  for the computation of the distance matrix. Here,  $n$  is the number of sequences and  $l$  is the sequence length. However, the impact of this additional complexity is small, because  $\hat{d}$  is usually small in comparison to  $n$  and  $l$ . To further reduce this impact, we reduce  $\hat{d}$  by merging chains of nodes without branches.

The application of the tree distance as substitution cost for the weighted Levenshtein distance ( $c_s(x_i, y_j) = d_{Tree}(x_i, y_j)$  with  $c_+ = c_- = 1$ ) reveals additional structure in the distance matrices. As can be seen in Fig. 4, the values for the regular Levenshtein and cosine distances are very low within clusters of similar data flows, while the between-cluster distances are all equal to 1. This results in very clearly defined boundaries between the clusters, but we cannot extract any information about the similarities between clusters. In contrast, the similarity matrix yielded by the weighted Levenshtein distance (Fig. 4c) shows a range of different values for the between cluster distances. This allows the computation of relationships between clusters and provides more meaningful information to the hierarchical clustering and optimal leaf ordering algorithms.

To further improve the similarity perception between data flows (T7, T8), we also apply multiple sequence alignment. We decided to use global sequence alignment because it has been shown to yield better user performance for sequence comparison tasks [55] and the scope of data sets in this domain scenario is small enough to allow for its application. We use the progressive sequence alignment [19] algorithm, because of two main benefits. First, the progressive sequence alignment is computed in a bottom-up approach throughout a hierarchy of sequences, allowing us to re-use our cluster hierarchy and skip the tree-building step of the algorithm. Second, progressive sequence alignment computes pairwise alignments of sequences or already aligned groups of sequences throughout the process. For these pairwise alignments, we can apply the Needleman Wunsch algorithm [47], which is the standard algorithm for pairwise global sequence alignments. The advantage of this approach is that the Needleman Wunsch algorithm uses a very similar scoring function to the Levenshtein distance. This function simply assigns a constant mismatch score  $s_{mismatch}$  to mismatching items and match score  $s_{match}$  to matching items.

$$s(a,b) = (s_{mismatch} - s_{match}) \cdot d_{Tree}(a,b) + s_{match} \quad (3)$$

Our variant uses the tree distance to interpolate between the two scores (Eq. (3)), causing structural relationships between code locations to be reflected in the alignment. We achieve good results with  $s_{mismatch} = s_{gap} = -1$  and  $s_{match} = 3$ , where  $s_{gap}$  is the score for a gap in the alignment. The complexity of the progressive alignment is  $\mathcal{O}(nl^2 \log n)$  in the best-case and  $\mathcal{O}(n^2 l^2)$  in the worst-case, depending on the balance of the cluster hierarchy. Thus, the additional computational factor of  $\mathcal{O}(\hat{d})$  is, again, negligible in most real-world scenarios.

According to Gleicher, the three design approaches, juxtaposition, superposition, and explicit encoding, can also be combined to address some of their shortcomings [22]. In this manner, we include an element of explicit encoding into our juxtaposed design. We visually merge adjacent code locations that correspond to the same node in the hierarchy (see Fig. 5 right). This improves the differentiation between rectangles with very similar and exactly identical colors (T8). The application of sequence alignment has the added benefit that exact matches are more likely to be aligned and can be merged. The weighted variant of the alignment scoring function capitalizes on this aspect even more. With the adapted sequence alignment, not only exact matches, but also similar code locations are likely to be aligned. Thus, users can more easily determine whether the aligned code locations are only similar or exactly identical by checking whether they are merged or not.

#### 5.4 Select Subset

To encourage the *select subset* comparison strategy, we include several methods of drilling down into the data set. First, we allow users to group the data flows based on our similarity measure or the meta attributes (T1, Fig. 1a). The groups can then be filtered, to focus on the most relevant data flows (T2, Fig. 1c). Similarly, the code base can be drilled down by filtering individual sub-trees (Fig. 1b), displaying

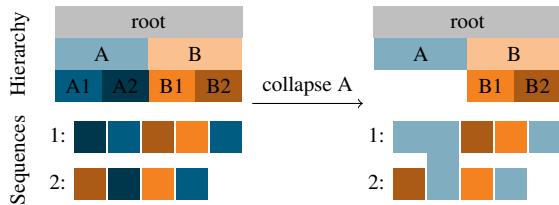


Fig. 5: Example of the code hierarchy collapsing action. The left column shows the expanded hierarchy and two sequences. In the right column, the sub-tree rooted at node A has been collapsed, thus its descendants are merged and color coded according to A in the sequence view.



Fig. 6: Example code hierarchy section with visible expand indicators. (a) shows the hierarchy collapsed to class-level. In (b), a hovered class member function indicators are highlighted, to be expanded by a click-interaction. (c) shows the result of the click.

those code locations in gray and hiding data flows consisting entirely of filtered code locations. We also provide the option to recompute the hierarchical color map for the remainder of the hierarchy. This results in more distinguishable colors for those code locations, easing comparison. However, users in our feedback sessions were confused by the change in coloring, thus this option is disabled by default.

We also allow seamless control of the hierarchy’s abstraction level to accelerate the *subset selection* process. To that end, sub-trees of the hierarchy can be collapsed by clicking on their roots. As demonstrated in Fig. 5, the code locations corresponding to nodes in the collapsed sub-tree are then merged and recolored accordingly. This reduces the number of visual elements on the screen at any time, simplifying the comparison, and allows coarse comparisons at higher levels of abstraction. After ruling out totally unrelated data flows, the granularity can be increased to compare the remaining data flows in higher detail.

However, during our feedback sessions, users were unable to determine, which leaves of the hierarchy could be expanded further and which were already at the deepest level. To avoid such confusion, we include indicators for the leaves’ hidden children (see Fig. 6). We designed this representation to invoke the metaphor of a fog covering the lower hierarchy levels and the indicators residing on the boundary between the visible and invisible. Thus, the indicators are displayed as gray nodes with an opacity gradient, fading into the background. To provide a preview of the expand action, we display expand indicators with their corresponding node colors if their parent node is hovered.

We also show histograms of the most important attributes (T1), to make this information always available at a glance (see Fig. 1g). In line with Gleicher’s recommendations for the *select subset* strategy [22], we stack the histograms of the visible and filtered data flows, to put the currently visible data into the context of the overall data set, helping users orient themselves while applying new filters. The histograms also serve as a shortcut to change the grouping attribute or apply filters.

### 5.5 Intra-Record Understanding

The main view of DaV<sup>3</sup> allows users to trace each data flow through the program’s source code and view each of the individual statements along the path (T4). Through the inclusion of multiple granularity levels, we also facilitate structural understanding of the flow. We visually merge consecutive equivalent code locations (see Fig. 5), enabling developers to see the number of code locations in the same member function, class, package, or library, depending on the currently selected level of granularity. Via the interactive highlighting, developers can



Fig. 7: The three-level sequence diagram of a data flow. The selected code location is highlighted with an orange border, while its member function and class have a black border. The black borders later in the sequence indicate, that the data flow will return to this member function.

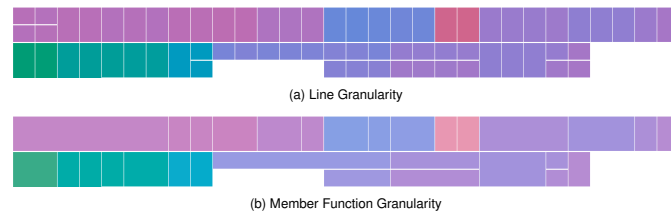


Fig. 8: Two granularity levels displaying a set of two pairs of similar flows.

determine, whether a data flow returns to a particular code location at a later stage, such as after the execution of a utility method.

However, as described in Sec. 4.1, the data flow is only a subset of the control flow, thus to understand the behavior of the program, the context of the control flow must be reviewed as well (T5). To make this context visible, we include an inspection view for individual data flows (see Fig. 1f). The inspection view replaces the center column of the interface with a view of the decompiled code of the current code location’s class. The line that corresponds to the current code location is highlighted in the code. Above the code view, we show the data flow as sequence diagram, analogous to the main view. This allows users to associate their findings between the two views. However, in the inspection view, we show line-, member function-, and class-level granularity at once, to make structural insight available at a glance (see Fig. 7). This results in a visualization similar to a *flame graph* [27], which are widely applied in software profiling and which many developers are, therefore, already familiar with. Users can navigate through the data flow via the arrow keys, visual navigation buttons, or by clicking on a code location in the sequence diagram. With this inspection view, we replicate the state-of-the-art features of GUI-based data flow analysis tools, while also allowing users to seamlessly associate their findings with the main view, and providing additional structural insight. Thus, the inspection view provides the necessary context to understand the progression of the data flow and its surrounding control flow through the code.

## 6 USE CASE

In this section, we describe how DaV<sup>3</sup> supports the workflow described in Sec. 4.3. To that end, we consider a fictional software developer, Alice, that is tasked with the verification of detected vulnerability findings in the Prime Video Android application. Upon loading the report, Alice sees the view displayed in Fig. 1. To identify the most relevant findings as a starting point (G1), Alice first views the distribution of vulnerability severity via the group headers (T1, Fig. 1a) and applies the filter to focus on high-severity findings (T2, Fig. 1c). After reviewing the other meta attributes via grouping (Fig. 1a) and histograms (Fig. 1g), Alice decides that this sub-set can not be reduced further, so she arbitrarily decides to proceed from top to bottom.

During this process, Alice soon arrives at two very similar data flows (top pair in Fig. 8a). She notices that they only differ in the first two code locations, because all others are merged across both flows. From the meta data (T3), Alice finds, the vulnerability lies in the app writing location information to a global device log. If this data flow represents a real behavior, a malicious third-party application could access this log file and read the location information without being granted access to the device’s location API. Looking at the data flows’ source description, Alice further realizes that the two data flows correspond to the propagation of latitude and longitude respectively. Through inspection of the first two code locations (the ones that differ), Alice confirms that they invoke the location API, retrieving the longitude in one flow and the

latitude in the other. Now, Alice sequentially investigates the remainder of one of the data flows in the inspection view (T4–T5, Fig. 1f). Between the last two code locations, Alice spots an inconsistency in the control flow. The logging function reaches the detected sink (the last code location) if its first parameter is a 3, but its call in the second to last code location passes a value of 4. This is a real-world occurrence of our initial example in Listing 1. This data flow can never be executed, hence Alice classifies this finding as a false positive (T6, Fig. 1c). Furthermore, Alice can immediately apply the same classification to the other finding (G3), because she has already seen that the inconsistency occurs in the shared sub-sequence (T8) between the two data flows.

Now, Alice advances to the next data flow and notices that it also has a similar sibling (bottom pair in Fig. 8a), which equals in the first eight code locations. From the meta data, Alice learns that, here, user account information may be written to the global log, but the scanner does not know precisely what information that is, because it does not know the semantics of the account information object and its contents. Inspecting both ends of the flows, Alice sees that the application indeed requests user account data from the phone's account management API at the start, and passes the taint to the logging API at the end. Investigating one flow sequentially, Alice finds that in the first eight code locations, the application extracts the user ID from the account data and checks for error codes. The ninth code location (the first that differs between the flows) handles the retrieval result by calling the `onSuccess` method. Alice hypothesizes that the other data flow may be a different branch, handling a different result. She confirms this by checking the ninth code location of the other data flow and finds that it contains a call to the `onError` method. To find out whether this is the only discrepancy, Alice changes the visualization's granularity to the member function level (see Fig. 8b). In this view, she sees that the result handling methods span the next eight code locations in the successful and three code locations in the error case. Afterwards, both data flows enter string formatting methods with different signatures, which prepare the message that is then passed to the logging API. From the string formatting methods' headers, Alice gathers that they differ, because the application logs two values in the error case and only one value on success. To find the logged values, Alice inspects the `onSuccess` method. Here, the application retrieves the session ID from the user's cookies and passes it on to the string formatting method. In the `onError` method, on the other hand, the application extracts the error code and error message and passes both of them on to the string formatting method. Overall, Alice determines that none of session ID, error code, and error message are significantly sensitive information, so she classifies both findings as false positives accordingly.

Alice continues this process for the remaining high-severity findings, before proceeding to findings of lower severity. After investigating all data flows, she hides the false positives and prioritizes the confirmed vulnerabilities with the same approach she used during the verification process, resulting in the issue list for her team to mitigate.

## 7 EVALUATION

Guided by the recommendations of Munzner's nested model [45], we performed validation steps at multiple nesting layers. On the encoding/interaction level, we verified two core design decisions with visualization experts and investigated the usability of DaV<sup>3</sup>is in a user study with target users. On the abstraction level, we elicited assessments of utility from both target users (software developers) as well as cybersecurity experts, focusing on qualitative methods to emulate the realistic application context more closely [14]. We selected evaluation methodologies appropriate to the nesting layer and the temporal constraints of our study participants, resulting in three distinct evaluation perspectives, which we will report on in the following. Additional details and evaluation materials can be found in the supplemental materials.

### 7.1 Visual Analytics Perspective

Two of our central design decisions from Sec. 5 were based on assumptions that, to our knowledge, have not yet been subject of user studies. These assumptions reside on the abstract encoding level of the nested model, and should be validated without domain specificity, yielding

generalizable findings. As independent validation, we chose to gather feedback in expert reviews [62] in a hallway setting. To that end, we performed separate experiments with eight participants each, which we recruited from the visual analytics research community—four from our research group and four at an international scientific conference on information visualization—at which point we reached saturation.

In the first experiment we investigated the impact of the weighted similarity measure. To get participants familiarized with the problem, we started the experiment by tasking participants to determine the grouping threshold that best represented their subjective perception of similarity in the DaV<sup>3</sup>is interface. We repeated this task with both the regular and the weighted Levenshtein distance and then asked which variant they preferred. During our interviews, seven out of eight participants preferred the weighted Levenshtein distance, because it resulted in a preferable grouping and allowed for a more granular customization, with one stating, *"I would now like to merge these two [clusters], but I can't. I can only do it, if I merge everything into one cluster."*, during their attempt to find the optimum with the regular Levenshtein distance. The eighth participant valued both metrics equally in this respect but preferred the regular Levenshtein distance overall, because it resulted in a more symmetrical cluster hierarchy.

In the second experiment, we investigated the significance of the coloring of inner hierarchy nodes. Here, the tasks were performed on paper and included the identification of related nodes in a colored icicle plot as well as the association of items between a sequence diagram and the icicle plot. We repeated the tasks with two variants of the icicle plot, starting with one in which the inner nodes were left blank, followed by the fully colored version. Finally, we showed participants both sheets and asked, which they preferred in terms of aesthetics, support in the tasks, and representation of the hierarchical structure.

All eight participants for this experiment unanimously agreed that colored inner nodes improved aesthetics and hierarchical representation. In the blank-node condition, five participants asked, whether we meant relatedness by hierarchical structure or by color, indicating that they did not perceive the hierarchical nature of the color map. For task-support, the participants were split. Five participants stated that the inner node colors help with lookup tasks, because they provide a coarse representation of the nodes in the corresponding sub-tree (*"I feel that I needed to jump around a lot less with the eyes to see which colors belong to [an inner node]."*, *"The colors reinforce the relations. You don't have to pay as much attention to the [node borders], and a [colored] area is more obvious than a line."*). The other three participants reported no perceived difference between the variants, because they relied purely on the leaf nodes anyways. In addition, two of them voiced the concern that relying on the inner node colors may introduce a level of uncertainty into the tasks (*"It's not that I couldn't relate them [the inner node colors], I didn't trust them."*, *"Since the tasks are focused on the colors of the leaves, the inner nodes are not relevant and potentially rather distracting because they are not neutral."*) Overall, this experiment supports our assumption that the hierarchical nature of the color map is easier to understand if the inner nodes are colored. While the majority of experts were of the opinion that the inner node colors also improve task performance, the opposite view point was also argued. This points towards the need of further empirical study of the effects of inner node colors.

### 7.2 Software Development Perspective

We merged our investigation of usability and utility for software developers into a single study, to better suit the time constraints of our participants. The study was set up as a combination of a think-aloud walkthrough of four pre-defined tasks and a post-test questionnaire. We started with an introduction into the basic principles of taint analysis and vulnerability verification, as well as a demonstration of DaV<sup>3</sup>is. Then we asked participants to perform a series of training tasks to learn the visual interface and interaction methods, after which we gave them the opportunity to use the tool freely. When participants were satisfied, we started with the main evaluation tasks. We designed the tasks to emulate the typical phases of the analysis workflow (see Sec. 4.3):

- **ET1:** Select one data flow that you would verify first. (G1)

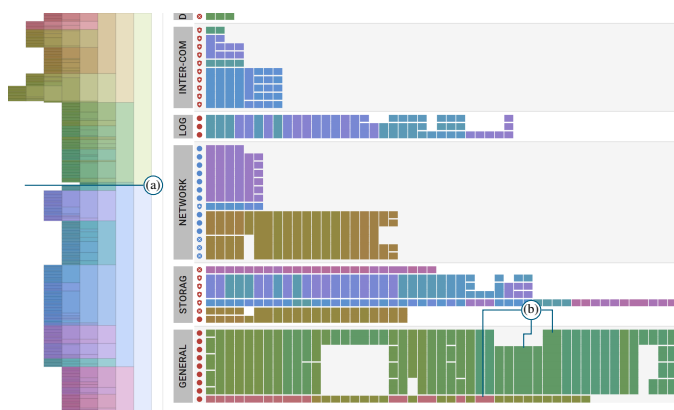


Fig. 9: The ibisPaint data set as displayed by DaV<sup>3</sup>is. Notably, no data flows pass between the two parts of the hierarchy, separated at (a), and some code locations occur twice in direct succession, such as (b).

- **ET2:** Verify the selected data flow. (G2)
- **ET3:** Select the next data flow that you would verify. (G3)
- **ET4:** Under the assumption that all data flows in this set are true positives, select the vulnerability that you would mitigate first.

After finishing the think-aloud portion of the study, we asked participants to fill out the post-test questionnaire. We acquired five software developers from our institution that were outside of our research group. Each session lasted around 45 minutes.

Overall, participants responded positively to the visual approach, stating that “A visual overview of the findings is a cool approach—especially when it’s interactive and lets you jump straight into the code”. In the questionnaire, four out of five participants highlighted the grouping and filtering functionality as especially helpful in the process of pruning the set of findings. Similarly, four participants applauded the direct linking to the corresponding sections of the code. While all participants considered the available meta data categorization during their work on **ET1** and **ET4**, four participants also included attributes of the sequences themselves into their considerations—the length (three participants) and whether a data flow passes through a particular package (two participants). Here, our choice to visualize the data flows as sequence diagram enables users to read the length with ease, and our hierarchical color map supports the localization of the code locations in the code structure. Between the two tasks, the considered meta data attributes differed, with participants ascribing greater importance to specific vulnerability types and sources in the mitigation scenario, while focusing mainly on the severity for verification. But the overall strategy to find the most relevant findings remained the same, confirming that DaV<sup>3</sup>is can support both scenarios equally. During **ET3**, four participants considered the similarity between data flows in order to take advantage of their work on the previous finding. (“If the last one was critical, I would first look at the similar ones. If it was not exploitable, I would put the similar ones last and decide the same way as for the first task (**ET1**)”). This confirms that developers benefit from visual sequence comparison during vulnerability verification and that the application of similarity sorting, sequence alignment and the visual merging of code locations supports them in this process.

In terms of usability, participants encountered one major hurdle during our tutorial tasks. When asked to identify a specific region of the code base, four participants started their search by browsing the code locations in the sequence diagram. After realizing that this task is more easily solved with the icicle plot, they showed no further confusion in this regard. This indicates that the visual representations do pose an initial barrier to entry for domain users without visualization expertise. To improve adoption, in-tool onboarding tutorials should explain the visualizations and their respective purpose. In summary, participants rated the understandability of DaV<sup>3</sup>is at an average of 3.4 out of four, indicating that the interface is easily usable after a relatively

short learning phase (“The features are intuitive, but there are many of them. After using the tool on and off for a week or so, I think you probably have understood everything.”).

### 7.3 Cybersecurity Perspective

To assess the utility of DaV<sup>3</sup>is from the perspective of expert users, we first performed a semi-structured interview with three members of the *VUSC* development team. The interview lasted around one hour in which we first demonstrated the functionality of DaV<sup>3</sup>is, using the *ibisPaint* data set as example (Fig. 9), and then discussed the visual representations and their strengths. The experts assessed the design as suitable for the verification process and did not have trouble understanding the visual representation. Notably, they found unexpected patterns in the data set that they had never observed with the regular *VUSC* user interface. One expert observed that the data set contained only data flows that occurred entirely within either application code or library code and none crossed in-between (no data flow contains locations from both sides of the line at Fig. 9a). Furthermore, the data set contains occurrences of the same line of code being executed twice in direct succession (rectangles of double width, such as Fig. 9b). This pattern caused an in-depth discussion between the experts about whether this constituted a real possible pattern or a bug in the scanner itself. This indicates that the hierarchical color map and the visual merging of code locations enable domain experts to identify overarching patterns in the data set, which they could not see with their own tools.

In addition, one of the experts agreed to participate in an extended follow-up interview. During this two hour session, we asked the expert to perform the same four tasks as the software developers (**ET1–ET4**), using the *Prime Video* data set (see Fig. 1) as example. As comparison baseline, we first asked them to use the *VUSC* web interface, which provides a list view of the detected findings with the detailed meta information on demand as well as the option to view the code corresponding to the associated data flow by stepping through the sequence. While the expert worked on the task, we observed their decisions and asked to elaborate on their reasoning if it was unclear to us. Afterwards, we asked them to repeat the same tasks with DaV<sup>3</sup>is.

Overall, the expert reiterated that “understanding every finding in detail is not always feasible” due to time constraints, which is why the findings must be ranked according to their criticality and their likelihood of being exploitable. The expert’s workflow largely mirrored that of the software developers, but their domain expertise provided deeper insight from the meta data. During the ranking process (**ET1**), the expert stated that shorter flows are easier to verify and are less likely to contain inconsistencies (such as the one in Sec. 2.2) while developers may prioritize findings in library code, because it is easier to update a library than to fix their own code. With the *VUSC* interface, flow length is not shown in the report overview and only the first and last code locations are displayed. DaV<sup>3</sup>is makes this information readily available in the overview, allowing its consideration during prioritization.

During **ET2**, the *VUSC* interface caused some confusion, when the expert analyzed a second data flow. Initially, the expert was surprised, because the code itself looked very similar to the statements in the previous data flow (“At first glance, this one looks more or less identical [to the last one]. I don’t even know why these are two separate findings”). Shortly afterwards, they realized “Oh, never mind, this is a different method”. This confusion would have been avoided with DaV<sup>3</sup>is, because the statements would have been displayed as not merged and, thus, located in different parts of the code. In addition, when reviewing a data flow with DaV<sup>3</sup>is, the expert elaborated that “If I know the code, then it could be that I know that it jumps into a library at some point. Then I can skip the application code, because I know that one, and only need to look at the library code”. To that end, they viewed the three-level sequence diagram (Fig. 7) as particularly useful, because it shows at a glance how many steps of the data flow occur in the same method or class and whether the current method is returned to at a later point during the data flow. This allows users to determine which parts of the data flow they can skip and what to expect during navigation.

For **ET3**, the expert stated that grouping findings based on the meta data alone is not sufficient, because those data flows do not necessarily

have anything in common. Referring back to the two data flows that they got confused with, they stated “*Usually [the data flows] are entirely disjunct. In our case, they didn't have anything in common*”. After we asked about the potential of shared sub-sequences specifically, the expert stated that “*If they share certain sub-paths and they are identical from a certain point onwards, I could verify them up to that point. That would help. If those shared sub-paths occur somewhere else, I only need to verify them once*”. When viewing the same data set in DaV<sup>3</sup>is, the expert was able to see the shared sub-sequences and realized that counter to their expectations, this occurs frequently. Based on this realization, they stated that “*This is helpful, because I don't need to look at [the sub-sequence] twice.*” and even reevaluated their original strategy (“*If you can take advantage of this, then I would rather look at those [data flows] with many shared locations instead of only the shortest ones*”). This confirms the potential of DaV<sup>3</sup>is to reduce verification workload by taking advantage of data flow similarities.

Since the strength of DaV<sup>3</sup>is lies in its ability to reduce the number of code locations that need to be closely investigated, we can estimate this reduction quantitatively. To that end, we count the sum of sequence-lengths in each of the data sets from Tab. 1. Without visual support, all of these code locations need to be investigated. Next, we assume that shared code locations within adjacent data flows in DaV<sup>3</sup>is only need to be investigated once, so we subtract the shared locations with the previous data flow from each flow's length, resulting in an average reduction of 52%. To estimate the saved time, we measured the expert's average time taken per code location during their work on ET2. Extrapolated to the entire Prime Video dataset, verification would take 6.56 hours without visual support and 3.54 hours with DaV<sup>3</sup>is.

## 7.4 Cumulative Findings

Taking a step back and compiling the evidence gathered throughout this paper, we have demonstrated the following: We have correctly identified domain problems that can benefit from visual support (domain expert interviews and quantitative estimate of improvement). By supporting the abstracted tasks visually, we support users in the identified domain problem (utility testing with developers and cybersecurity experts). As apparent from the literature, the chosen visual encodings and interaction techniques are suitable for the abstract tasks at hand and they are intuitive to use (usability study). Furthermore, our algorithms for color map generation and distance computation are suitable to produce the desired effects, which enables users to use the existing visual encodings more precisely (expert review of distance metrics) and more intuitively (expert review of color map), while the algorithms themselves are applicable to the real world setting (complexity analysis).

## 8 DISCUSSION

The summative evaluations showed that DaV<sup>3</sup>is provides appropriate visual support for the prioritization and verification of software vulnerability data flows. The visualizations are usable and understandable with a short introduction. In addition, DaV<sup>3</sup>is allows software developers to quickly generate several findings that are difficult to realize with existing state-of-the-art solutions. This includes the identification of similar data flows, of distinct regions of the code base that data flows pass through, as well as the structural intra-record understanding, which serves as a preview when navigating through an individual flow.

Yet the design of DaV<sup>3</sup>is was created under a series of assumptions, which we will discuss in the following. First, we will consider the main view. Both the expert's and the developers' workflows suggest that G1 is mostly reliant on meta data. Hence it would be a reasonable approach to focus the visualization on that meta data and only display the remaining set of data flows for G2–G3. However, both user groups were able to gain additional insight from the overview of all data flows. While the cybersecurity experts noticed unexpected patterns in the overall data set, which made them question the correct functionality of their own scanner, some software developers used the overview to find data flows passing through especially sensitive regions of the code base and included this information as additional criterion within their considerations for G1. This insight could not be discovered as easily if the entire set of data flows was not on display.

Similarly, we set the goal of supporting both levels of insight with the same visualization type, to reduce the cognitive load on users. In the end, however, we still implemented a secondary view (the inspection view) to improve intra-record understanding. We apply the same visualization type in both views, to reduce the mental effort in translating findings between the two views. Yet the separation of both insight levels into separate views raises the question whether the individual insight levels could be supported better with different visualization types and whether this improvement in the individual insight levels would outweigh the mental translation cost between the two visualization types. The scope of this paper does not allow us to definitively answer this question, yet we can document our initial observations:

Software developers in the evaluation initially experienced difficulties in associating the two existing visualizations (icicle plot and sequence diagram) with the corresponding data characteristics and in determining the preferred visualization for each task. This indicates that, initially, the need to relate findings between the two visualizations causes a barrier to entry, that needs to be overcome before the tool can be applied successfully. Introducing another visual representation of the data flows would likely increase this barrier further.

Furthermore, participants applauded the direct connection between the visual representation and the underlying code. The inclusion of a separate visual representation would likely have introduced an additional layer of abstraction, reducing the directness of this connection.

In summary of our technical contributions, including the hierarchical relationships in the similarity computation allows more granular customization of the data flow grouping. Their inclusion into the color map and aggregation interaction enables similarity perception on multiple granularities. In addition, the explicit display of multiple granularity levels in the three-level sequence diagram was explicitly applauded by domain experts, because it provides structural insight into the data flow and serves as a preview before navigation. While the hierarchical code structure is not available for general scans of compiled applications, we believe our techniques are also applicable to source code scans or other domains that contain sequences of hierarchically related elements, such as computer network trace routes [20], web click streams [38], organizational processes with hierarchical relationships [64], or hierarchically clustered multivariate event sequences [25, 51].

However, scalability must be kept in mind. The inclusion of the tree distance into the sequence distance and alignment computation did not introduce significant performance cost in our application scenario. When applying these techniques to other domains with larger data sets, this may change. But it is more likely that the number of sequences or the length of sequences result in the performance deterioration, rather than the depth of the hierarchy. In this case, the approach itself—computing pairwise Levenshtein distances and applying progressive global sequence alignment—needs to be reevaluated.

In addition, the readability suffers from an increase in the data set size. To guarantee a minimum size of visual elements, that facilitates interaction, we grow the visualization beyond the viewport and display scroll bars for large data sets. The similarity-based sorting, enables the relevant comparisons even if not all sequences are on screen at the same time. In our domain scenario, where the data set is reduced through successive application of filters via the sequence meta data, this does not cause visual problems for data sets up to 100 sequences. But in other domains, with larger data sets, this may differ. In these cases, the aggregation of sequences as in *Sequence Synopsis* [15] may be unavoidable. Furthermore, with an increase in the hierarchy size, the colors become indistinguishable at deeper hierarchy levels. To compensate for this, dynamically adjusting color maps can be applied, recomputing colors on the fly and improving discriminability. But the algorithm must make sure to keep colors as consistent as possible to avoid confusing users. Such algorithms already exist for semantic zooming environments [68], but as of today, there are no dynamic algorithms that color the inner nodes as well.

## 9 CONCLUSION

We have presented the design study of DaV<sup>3</sup>is, a visual analytics tool for the analysis of software vulnerabilities detected by the VUSC taint

analyzer. This domain problem contains several design challenges in that it requires simultaneous insight of the underlying sequences and that these sequences consist of hierarchically related elements. Another challenge is the lack of experience of the target user group, software developers, with both cybersecurity and visual analytics. We designed and evaluated DaV<sup>3</sup>is with input from three groups, cybersecurity experts without expertise in visual analytics, visual analytics experts without expertise in cybersecurity, and developers with expertise in neither field, yielding positive responses from all groups. We believe our technical contributions to be transferable to other domains as well.

## SUPPLEMENTAL MATERIALS

All supplemental materials are available on OSF at <https://osf.io/n5xzc/>, released under a CC-BY 4.0 license. In particular, they include (1) the example data sets listed in Tab. 1 in JSON format, (2) the full task hierarchy, (3) additional screenshots of DaV<sup>3</sup>is, including a picture-walkthrough of the use case from Sec. 6, (4) additional evaluation details as well as used materials, and (5) a video demonstrating the core functionality of DaV<sup>3</sup>is.

## ACKNOWLEDGMENTS

This research work was supported by the National Research Center for Applied Cybersecurity ATHENE.

## REFERENCES

- [1] M. Angelini, G. Blasilli, T. Catarci, S. Lenti, and G. Santucci. Vulnerus: Visual Vulnerability Analysis for Network Security. *IEEE TVCG*, 25(1):183–192, Jan. 2019. doi: 10.1109/TVCG.2018.2865028 3
- [2] S. Arzt. VUSC - the Code Scanner. [www.sit.fraunhofer.de/en/vusc/](http://www.sit.fraunhofer.de/en/vusc/). 3
- [3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Oceau, 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, June 2014. doi: 10.1145/2666356.2594299 1, 3
- [4] Z. Bar-Joseph, D. K. Gifford, and T. S. Jaakkola. Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics*, 17(suppl\_1):22–29, June 2001. doi: 10.1093/bioinformatics/17.suppl\_1.S22 6
- [5] S. D. Bartolomeo, Y. Zhang, F. Sheng, and C. Dunne. Sequence Braiding: Visual Overviews of Temporal Event Sequences and Attributes. *IEEE TVCG*, 27(2):1353–1363, Feb. 2021. doi: 10.1109/TVCG.2020.3030442 2, 3, 5
- [6] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. An empirical study on the developers' perception of software coupling. In *ICSE*, pp. 692–701, May 2013. doi: 10.1109/ICSE.2013.6606615 7
- [7] M. Behrisch, B. Bach, N. Henry Riche, T. Schreck, and J.-D. Fekete. Matrix Reordering Methods for Table and Network Visualization. *Computer Graphics Forum*, 35(3):693–716, 2016. doi: 10.1111/cgf.12935 6
- [8] J. Bernard, C.-M. Barth, E. Cuba, A. Meier, Y. Peiris, and B. Shneiderman. IVESA – Visual Analysis of Time-Stamped Event Sequences. *IEEE TVCG*, 31(4):2235–2256, Apr. 2025. doi: 10.1109/TVCG.2024.3382760 2
- [9] J. Blaas, C. Botha, E. Grundy, M. Jones, R. Laramée, and F. Post. Smooth Graphs for Visual Exploration of Higher-Order State Transitions. *IEEE TVCG*, 15(6):969–976, Nov. 2009. doi: 10.1109/TVCG.2009.181 2, 5
- [10] D. Boxler and K. R. Walcott. Static Taint Analysis Tools to Detect Information Flows. In *SERP*. Athens, Greece, 2018. 1
- [11] M. Brehmer and T. Munzner. A Multi-Level Typology of Abstract Visualization Tasks. *IEEE TVCG*, 19(12):2376–2385, Dec. 2013. doi: 10.1109/TVCG.2013.124 4, 5
- [12] N. Cao, Y.-R. Lin, X. Sun, D. Lazer, S. Liu, and H. Qu. Whisper: Tracing the Spatiotemporal Process of Information Diffusion in Real Time. *IEEE TVCG*, 18(12):2649–2658, Dec. 2012. doi: 10.1109/TVCG.2012.291 2
- [13] B. C. Cappers and J. J. van Wijk. Exploring Multivariate Event Sequences Using Rules, Aggregations, and Selections. *IEEE TVCG*, 24(1):532–541, Jan. 2018. doi: 10.1109/TVCG.2017.2745278 2, 5
- [14] S. Carpendale. Evaluating Information Visualizations. In *Information Visualization: Human-Centered Issues and Perspectives*, pp. 19–45. Springer Berlin Heidelberg, Berlin, 2008. 9
- [15] Y. Chen, P. Xu, and L. Ren. Sequence Synopsis: Optimize Visual Summary of Temporal Event Data. *IEEE TVCG*, 24(1):45–55, Jan. 2018. doi: 10.1109/TVCG.2017.2745083 2, 3, 11
- [16] T. T. Dang and T. K. Dang. An Extensible Framework for Web Application Vulnerabilities Visualization and Analysis. In *FDSE*, pp. 86–96. Springer International Publishing, Cham, 2014. doi: 10.1007/978-3-319-12778-1\_7 3
- [17] R. Diestel. *Graph Theory*, p. 8. Graduate Texts in Mathematics. Springer, Berlin, Heidelberg, 5th ed., 2017. doi: 10.1007/978-3-662-53622-3 7
- [18] W. Fang, B. P. Miller, and J. A. Kupsch. Automated tracing and visualization of software security structure and properties. In *VizSec*, pp. 9–16. ACM, New York, NY, USA, Oct. 2012. doi: 10.1145/2379690.2379692 3
- [19] D.-F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, Aug. 1987. doi: 10.1007/BF02603120 7
- [20] F. Fischer, J. Fuchs, P.-A. Vervier, F. Mansmann, and O. Thonnard. Vis-Tracer: A visual analytics tool to investigate routing anomalies in traceroutes. In *VizSec*, pp. 80–87. ACM, New York, NY, USA, Oct. 2012. doi: 10.1145/2379690.2379701 3, 11
- [21] T. Fujiwara, P. Malakar, K. Reda, V. Vishwanath, M. E. Papka, and K.-L. Ma. A Visual Analytics System for Optimizing Communications in Massively Parallel Applications. In *VAST*, pp. 59–70. IEEE, Phoenix, AZ, USA, Oct. 2017. doi: 10.1109/VAST.2017.8585646 3
- [22] M. Gleicher. Considerations for Visualizing Comparison. *IEEE TVCG*, 24(1):413–423, Jan. 2018. doi: 10.1109/TVCG.2017.2744199 6, 7, 8
- [23] J. R. Goodall, H. Radwan, and L. Halseth. Visual analysis of code security. In *VizSec*, pp. 46–51. ACM, New York, NY, USA, Sept. 2010. doi: 10.1145/1850795.1850800 3
- [24] D. Gotz and H. Stavropoulos. DecisionFlow: Visual Analytics for High-Dimensional Temporal Event Sequence Data. *IEEE TVCG*, 20(12):1783–1792, Dec. 2014. doi: 10.1109/TVCG.2014.2346682 2
- [25] D. Gotz, J. Zhang, W. Wang, J. Shrestha, and D. Borland. Visual Analysis of High-Dimensional Event Sequence Data via Dynamic Hierarchical Aggregation. *IEEE TVCG*, 26(1):440–450, Jan. 2020. doi: 10.1109/TVCG.2019.2934661 3, 11
- [26] R. Gove, J. Saxe, S. Gold, A. Long, and G. Bergamo. SEEM: A scalable visualization for comparing multiple large sets of attributes for malware analysis. In *VizSec*, pp. 72–79. ACM, New York, NY, USA, Nov. 2014. doi: 10.1145/2671491.2671496 3
- [27] B. Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, May 2016. doi: 10.1145/2909476 8
- [28] S. Guo, Z. Jin, D. Gotz, F. Du, H. Zha, and N. Cao. Visual Progression Analysis of Event Sequence Data. *IEEE TVCG*, 25(1):417–426, Jan. 2019. doi: 10.1109/TVCG.2018.2864885 2
- [29] S. Guo, K. Xu, R. Zhao, D. Gotz, H. Zha, and N. Cao. EventThread: Visual Summarization and Stage Analysis of Event Sequence Data. *IEEE TVCG*, 24(1):56–65, Jan. 2018. doi: 10.1109/TVCG.2017.2745320 2
- [30] Y. Guo, S. Guo, Z. Jin, S. Kaul, D. Gotz, and N. Cao. Survey on Visual Analysis of Event Sequence Data. *IEEE TVCG*, 28(12):5091–5112, Dec. 2022. doi: 10.1109/TVCG.2021.3100413 2, 5
- [31] K. S. Han, J. H. Lim, B. Kang, and E. G. Im. Malware analysis using visualized images and entropy graphs. *International Journal of Information Security*, 14(1):1–14, Feb. 2015. doi: 10.1007/s10207-014-0242-0 3
- [32] L. Harrison, R. Spahn, M. Iannacone, E. Downing, and J. R. Goodall. NV: Nessus vulnerability visualization for the web. In *VizSec*, pp. 25–32. ACM, New York, NY, USA, Oct. 2012. doi: 10.1145/2379690.2379694 3
- [33] Z. Jin, S. Guo, N. Chen, D. Weiskopf, D. Gotz, and N. Cao. Visual Causality Analysis of Event Sequence Data. *IEEE TVCG*, 27(2):1343–1352, Feb. 2021. doi: 10.1109/TVCG.2020.3030465 2, 5
- [34] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, pp. 672–681, May 2013. doi: 10.1109/ICSE.2013.6606613 1, 4
- [35] T. Kapler and W. Wright. GeoTime information visualization. In *IEEE VIS*, pp. 25–32. IEEE, Austin, TX, USA, Oct. 2004. doi: 10.1109/INVIS.2004.27 5
- [36] J.-F. Lalande, M. Simon, and V. Viet Triem Tong. GroDDViewer: Dynamic Dual View of Android Malware. In *Graphical Models for Security*, pp. 127–139. Springer International Publishing, Cham, 2020. doi: 10.1007/978-3-030-62230-5\_7 3
- [37] Y. K. Lee, P. Yoodee, A. Shahbazian, D. Nam, and N. Medvidovic. SEALANT: A detection and visualization tool for inter-app security vulnerabilities in android. In *ASE*, pp. 883–888. IEEE, Urbana, IL, USA, Oct. 2017. doi: 10.1109/ASE.2017.8115699 3
- [38] Z. Liu, Y. Wang, M. Dontcheva, M. Hoffman, S. Walker, and A. Wilson. Patterns and Sequences: Interactive Exploration of Clickstreams to Understand Common Visitor Paths. *IEEE TVCG*, 23(1):321–330, Jan. 2017. doi: 10.1109/TVCG.2016.2598797 2, 3, 11

- [39] J. Magallanes, T. Stone, P. D. Morris, S. Mason, S. Wood, and M.-C. Villa-Uriol. Sequen-C: A Multilevel Overview of Temporal Event Sequences. *IEEE TVCG*, 28(1):901–911, Jan. 2022. doi: [10.1109/TVCG.2021.3114868](https://doi.org/10.1109/TVCG.2021.3114868) 2, 3
- [40] S. Malik, F. Du, M. Monroe, E. Onukwugha, C. Plaisant, and B. Shneiderman. Cohort Comparison of Event Sequences with Balanced Integration of Visual Analytics and Statistics. In *IUI*, pp. 38–49. ACM, New York, NY, USA, Mar. 2015. doi: [10.1145/2678025.2701407](https://doi.org/10.1145/2678025.2701407) 2, 3
- [41] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*, p. 58. Cambridge University Press, July 2008. 7
- [42] T. Mertz and J. Kohlhammer. Towards a Quality Approach to Hierarchical Color Maps. In *IEEE VIS*, pp. 221–225. IEEE, St. Pete Beach, FL, USA, Oct. 2024. doi: [10.1109/VIS55277.2024.00052](https://doi.org/10.1109/VIS55277.2024.00052) 6
- [43] M. Monroe, R. Lan, H. Lee, C. Plaisant, and B. Shneiderman. Temporal Event Sequence Simplification. *IEEE TVCG*, 19(12):2227–2236, Dec. 2013. doi: [10.1109/TVCG.2013.200](https://doi.org/10.1109/TVCG.2013.200) 2, 3
- [44] L. Montana, J. Magallanes, M. Juarez, S. Mason, A. Narracott, L. van Gemeren, S. Wood, and M.-C. Villa-Uriol. EventBox: A Novel Visual Encoding for Interactive Analysis of Temporal and Multivariate Attributes in Event Sequences. *IEEE TVCG*, 32(1):112–122, Jan. 2026. doi: [10.1109/TVCG.2025.3633904](https://doi.org/10.1109/TVCG.2025.3633904) 2
- [45] T. Munzner. A Nested Model for Visualization Design and Validation. *IEEE TVCG*, 15(6):921–928, Nov. 2009. doi: [10.1109/TVCG.2009.111](https://doi.org/10.1109/TVCG.2009.111) 9
- [46] M. Nachtigall, M. Schlichtig, and E. Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *ISSTA*, pp. 532–543. ACM, New York, NY, USA, July 2022. doi: [10.1145/3533767.3534374](https://doi.org/10.1145/3533767.3534374) 4
- [47] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, Mar. 1970. doi: [10.1016/0022-2836\(70\)90057-4](https://doi.org/10.1016/0022-2836(70)90057-4) 7
- [48] M. Nollenburg. A Survey on Automated Metro Map Layout Methods. In *Schematic Mapping Workshop*, 2014. 5
- [49] L. Peng, Z. Lin, N. Andrienko, G. Andrienko, and S. Chen. Contextualized visual analytics for multivariate events. *Visual Informatics*, 9(2):100234, June 2025. doi: [10.1016/j.visinf.2025.100234](https://doi.org/10.1016/j.visinf.2025.100234) 2
- [50] A. Perer, F. Wang, and J. Hu. Mining and exploring care pathways from electronic medical records with visual analytics. *Journal of Biomedical Informatics*, 56:369–378, Aug. 2015. doi: [10.1016/j.jbi.2015.06.020](https://doi.org/10.1016/j.jbi.2015.06.020) 2
- [51] A. J. Pretorius and J. J. Van Wijk. Visual Analysis of Multivariate State Transition Graphs. *IEEE TVCG*, 12(5):685–692, Sept. 2006. doi: [10.1109/TVCG.2006.192](https://doi.org/10.1109/TVCG.2006.192) 3, 11
- [52] D. A. Quist and L. M. Liebrock. Visualizing compiled executables for malware analysis. In *VizSec*, pp. 27–32. IEEE, Atlantic City, NJ, USA, Oct. 2009. doi: [10.1109/VIZSEC.2009.5375539](https://doi.org/10.1109/VIZSEC.2009.5375539) 3
- [53] S. L. Reynolds, T. Mertz, S. Arzt, and J. Kohlhammer. User-Centered Design of Visualizations for Software Vulnerability Reports. In *VizSec*, pp. 68–78, Oct. 2021. doi: [10.1109/VizSec53666.2021.00013](https://doi.org/10.1109/VizSec53666.2021.00013) 3
- [54] R. C. Roberts, D. Rees, R. S. Laramée, P. Brookes, and G. A. Smith. RiverState: A Visual Metaphor Representing Millions of Time-Oriented State Transitions. In *CGVC*. The Eurographics Association, Swansea, Wales, UK, 2018. doi: [10.2312/cgvc.20181210](https://doi.org/10.2312/cgvc.20181210) 5
- [55] R. A. Ruddle, J. Bernard, H. Lücke-Tieke, T. May, and J. Kohlhammer. The Effect of Alignment on People's Ability to Judge Event Sequence Similarity. *IEEE TVCG*, 28(9):3070–3081, Sept. 2022. doi: [10.1109/TVCG.2021.3050497](https://doi.org/10.1109/TVCG.2021.3050497) 7
- [56] J. Smith, L. N. Q. Do, and E. Murphy-Hill. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *SOUPS*, pp. 221–238, 2020. 1, 4
- [57] Y. Tanahashi and K.-L. Ma. Design Considerations for Optimizing Storyline Visualizations. *IEEE TVCG*, 18(12):2679–2688, Dec. 2012. doi: [10.1109/TVCG.2012.212](https://doi.org/10.1109/TVCG.2012.212) 2, 5
- [58] L. Tang, T. Wu, X. Chen, S. Wen, L. Li, X. Xia, M. Grobler, and Y. Xiang. How Does Visualisation Help App Practitioners Analyse Android Apps? *IEEE Transactions on Dependable and Secure Computing*, 20(3):2238–2255, May 2023. doi: [10.1109/TDSC.2022.3178181](https://doi.org/10.1109/TDSC.2022.3178181) 3
- [59] M. Tennekes and E. de Jonge. Tree Colors: Color Schemes for Tree-Structured Data. *IEEE TVCG*, 20(12):2072–2081, Dec. 2014. doi: [10.1109/TVCG.2014.2346277](https://doi.org/10.1109/TVCG.2014.2346277) 6
- [60] D. Titze and J. Schütte. Apparecium: Revealing Data Flows in Android Applications. In *AINA*, pp. 579–586, Mar. 2015. doi: [10.1109/AINA.2015.239](https://doi.org/10.1109/AINA.2015.239) 3
- [61] C. Tominski, H. Schumann, G. Andrienko, and N. Andrienko. Stacking-Based Visualization of Trajectory Attribute Data. *IEEE TVCG*, 18(12):2565–2574, Dec. 2012. doi: [10.1109/TVCG.2012.265](https://doi.org/10.1109/TVCG.2012.265) 5
- [62] M. Tory and T. Moller. Evaluating visualizations: Do expert reviews work? *IEEE CG&A*, 25(5):8–11, Sept. 2005. doi: [10.1109/MCG.2005.102](https://doi.org/10.1109/MCG.2005.102) 9
- [63] A. Ulmer, D. Sessler, and J. Kohlhammer. ProBGP: Progressive Visual Analytics of Live BGP Updates. *CGF*, 40(3):37–48, 2021. doi: [10.1111/cgf.14287](https://doi.org/10.1111/cgf.14287) 3
- [64] S. van den Elzen, M. Jans, N. Martin, F. Pieters, C. Tominski, M.-C. Villa-Uriol, and S. J. van Zelst. Towards multi-faceted Visual Process Analytics. *Information Systems*, p. 102560, May 2025. doi: [10.1016/j.is.2025.102560](https://doi.org/10.1016/j.is.2025.102560) 11
- [65] S. van der Linden, B. Cappers, A. Vilanova, and S. van den Elzen. Long sequences with a lot of events (LoLo): A visual analytics approach for analyzing long event sequences. *Information Visualization*, p. 14738716251372584, Oct. 2025. doi: [10.1177/14738716251372584](https://doi.org/10.1177/14738716251372584) 2
- [66] S. van der Linden, B. M. Wulterkens, M. M. van Gilst, S. Overeem, C. van Pul, A. Vilanova, and S. van den Elzen. FlexEvent: Going beyond Case-Centric Exploration and Analysis of Multivariate Event Sequences. *CGF*, 42(3):161–172, 2023. doi: [10.1111/cgf.14820](https://doi.org/10.1111/cgf.14820) 3
- [67] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, D. A. Keim, and W. Aigner. A Survey of Visualization Systems for Malware Analysis. In *EuroVis*. The Eurographics Association, 2015. doi: [10.2312/eurovisstar.20151114](https://doi.org/10.2312/eurovisstar.20151114) 3
- [68] N. Waldin, M. Waldner, M. Le Muzic, E. Gröller, D. S. Goodsell, L. Autin, A. J. Olson, and I. Viola. Cuttlefish: Color Mapping for Dynamic Multi-Scale Visualizations. *CGF*, 38(6):150–164, 2019. doi: [10.1111/cgf.13611](https://doi.org/10.1111/cgf.13611) 6, 11
- [69] 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. *Advanced Materials Research*, 433–440:5453–5458, 2012. doi: [10.4028/www.scientific.net/AMR.433-440.5453](https://doi.org/10.4028/www.scientific.net/AMR.433-440.5453) 3
- [70] T. D. Wang, C. Plaisant, A. J. Quinn, R. Stanchak, S. Murphy, and B. Shneiderman. Aligning temporal data by sentinel events: Discovering patterns in electronic health records. In *CHI*, pp. 457–466. ACM, New York, NY, USA, Apr. 2008. doi: [10.1145/1357054.1357129](https://doi.org/10.1145/1357054.1357129) 2
- [71] M. Wattenberg, F. B. Viégas, and K. Hollenbach. Visualizing Activity on Wikipedia with Chromograms. In *Human-Computer Interaction – INTERACT 2007*, pp. 272–287. Springer, Berlin, Heidelberg, 2007. doi: [10.1007/978-3-540-74800-7\\_23](https://doi.org/10.1007/978-3-540-74800-7_23) 2
- [72] R. Wetzel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *ICSE*, pp. 551–560. ACM, New York, NY, USA, May 2011. doi: [10.1145/1985793.1985868](https://doi.org/10.1145/1985793.1985868) 6
- [73] K. Wongsuphasawat and D. Gotz. Exploring Flow, Factors, and Outcomes of Temporal Event Sequences with the Outflow Visualization. *IEEE TVCG*, 18(12):2659–2668, Dec. 2012. doi: [10.1109/TVCG.2012.225](https://doi.org/10.1109/TVCG.2012.225) 2
- [74] K. Wongsuphasawat, J. A. Guerra Gómez, C. Plaisant, T. D. Wang, M. Taieb-Maimon, and B. Shneiderman. LifeFlow: Visualizing an overview of event sequences. In *CHI*, pp. 1747–1756. ACM, New York, NY, USA, May 2011. doi: [10.1145/1978942.1979196](https://doi.org/10.1145/1978942.1979196) 2, 3
- [75] K. Wongsuphasawat and J. Lin. Using visualizations to monitor changes and harvest insights from a global-scale logging infrastructure at Twitter. In *VAST*, pp. 113–122, Oct. 2014. doi: [10.1109/VAST.2014.7042487](https://doi.org/10.1109/VAST.2014.7042487) 2
- [76] I. Yoo. Visualizing windows executable viruses using self-organizing maps. In *VizSec/DMSEC*, pp. 82–89. ACM, New York, NY, USA, Oct. 2004. doi: [10.1145/1029208.1029222](https://doi.org/10.1145/1029208.1029222) 3
- [77] Y. Zhang, K. Chanana, and C. Dunne. IDMMVis: Temporal Event Sequence Visualization for Type 1 Diabetes Treatment Decision Support. *IEEE TVCG*, 25(1):512–522, Jan. 2019. doi: [10.1109/TVCG.2018.2865076](https://doi.org/10.1109/TVCG.2018.2865076) 2, 4
- [78] J. Zhao, Z. Liu, M. Dontcheva, A. Hertzmann, and A. Wilson. MatrixWave: Visual Comparison of Event Sequence Data. In *CHI*, pp. 259–268. ACM, New York, NY, USA, Apr. 2015. doi: [10.1145/2702123.2702419](https://doi.org/10.1145/2702123.2702419) 2