Towards Better Utilizing Static Application Security Testing

Jinqiu Yang
Computer Science and Software Engineering
Concordia University, Canada
jinqiuy@encs.concordia.ca

Lin Tan
Computer Science
Purdue University, USA
lintan@purdue.edu

John Peyton, Kristofer A Duer
HCL, USA
john.peyton, kristofer.duer@hcl.com

***Abstract*—Static application security testing (SAST) detects vulnerability warnings through static program analysis. Fixing the vulnerability warnings tremendously improves software quality. However, SAST has not been fully utilized by developers due to various reasons: difficulties in handling a large number of reported warnings, a high rate of false warnings, and lack of guidance in fixing the reported warnings.**

In this paper, we collaborated with security experts from a commercial SAST product and propose a set of approaches (*Priv*) to help developers better utilize SAST techniques. First, *Priv* identifies preferred fix locations for the detected vulnerability warnings, and group them based on the common fix locations. *Priv* also leverages visualization techniques so that developers can quickly investigate the warnings in groups and prioritize their quality-assurance effort. Second, *Priv* identifies actionable vulnerability warnings by removing SAST-specific false positives. Finally, *Priv* provides customized fix suggestions for vulnerability warnings.

Our evaluation of *Priv* on six web applications highlights the accuracy and effectiveness of *Priv*. For 75.3% of the vulnerability warnings, the preferred fix locations found by *Priv* are identical to the ones annotated by security experts. The visualization based on shared preferred fix locations is useful for prioritizing quality-assurance efforts. *Priv* reduces the rate of SAST-specific false positives significantly. Finally, *Priv* is able to provide fully complete and correct fix suggestions for 75.6% of the evaluated warnings. *Priv* is well received by security experts and some features are already integrated into industrial practice.

***Index Terms*—static security application testing, static bug detection, utilization of software engineering tools, software reliability**

I. Introduction

Static analysis techniques are widely used in practice to ensure the quality of software [1], [2]. In particular, developers often rely on static application security testing (SAST) techniques to detect security vulnerabilities. SAST performs static program analysis for finding software vulnerabilities, and is different from dynamic approaches that requires penetration tests. Hence, SAST is able to detect potential vulnerabilities that remain uncovered after in-house testing. SAST has been shown to be effective in improving the security of applications [3], [4], [5].

Aside from research prototypes [3], [6], [7], there are many popular commercial SAST products, such as AppScan Source [8], CheckMark [9], and Fortify [10]. However, as shown in previous studies [11], [12], developers often encounter many challenges when using static bug detection techniques, which results in underuse of SAST techniques. We have been working with security experts from AppScan Source for the past three years. Together with the security experts, we identified several challenges that prevent developers from fully utilizing SAST techniques.

First, SAST techniques usually detect a large number of vulnerability warnings with insufficient support to help developers prioritize their quality-assurance effort (i.e., time and resources spent to fix the reported vulnerability warnings). Current SAST techniques categorize the detected vulnerabilities based on various factors (e.g., severity, confidence, vulnerability type, etc.). However, such categorization does not prioritize developers' quality-assurance effort based on the commonality among the detected warnings [11], [12]. One commonality is the *preferred fix location (pFixLoc)*, which may be commonly shared by many warnings. Applying the fix at one *pFixLoc* will eliminate multiple warnings. Second, similar to static bug detection, SAST techniques produce many false warnings: many detected warnings are not true vulnerabilities, and hence, do not require developers' effort for fixing. A high rate of false warnings makes developers lose interests in the detection results [11]. Finally, SAST techniques often do not offer fix suggestions on how to fix the detected warnings. SAST techniques may provide generic guidance, i.e., one remediation page for one type of vulnerability warnings. However, such generic guidance only provides high-level information, but lack of specific information which varies for each vulnerability warning.

Therefore, to address the above-mentioned challenges that developers may encounter when using SAST techniques, we work closely with security experts and propose a set of approaches to improve current SAST solutions. We implement the approaches as a tool, named *Priv*. *Priv* is well-received by security experts and some features are already integrated into commercial products. In this paper, we discuss how we leverage visualization and code analysis to build *Priv* upon a mature SAST product (AppScan Source). We also illustrate the usage and effectiveness of *Priv* in improving the usability of SAST techniques.

The prototype of *Priv* targets the most prominent vulnerability types [3]: cross-site scriptings (XSS), SQL injections (SQLi), path traversals (PATHtrv), command injections (COMMi), and second-order injections (SECi). A prior

study [13] shows that cross-site scriptings, SQL injections, and parameter tampering are accounted for more than one-third of web application attacks. We applied *Priv* (+ AppScan Source) on one closed source (i.e., AltoroJ, an internal testing application for evaluating AppScan Source) and five open-source web applications (WebGoat, JavaVulnerable Lab, Vulnerable Web, Bodgeit, and HeisenBerg) that are commonly used for studying security vulnerabilities and evaluating SAST techniques. The evaluation shows that *Priv* automatically finds preferred fix locations that are identical to developers' annotated fix locations for 75.3% of the evaluated vulnerability warnings. A global-view visualization based on the *pFixLocs* is useful in guiding developers to prioritize quality-assurance effort. Moreover, *Priv* can reduce the rate of SAST-specific false positives from 14.8%–88.6% to 0. Finally, *Priv* can provide customized fix suggestions for the evaluated vulnerability warnings: fully complete and correct for 75.6% of them, partially compilable/correct for 4.2% of them, and fix template only for 2.01% of them.

The main contributions of this paper are:

- We proposed a set of approaches to help developers better utilize SAST techniques.
- We provided an implementation of the proposed approaches in *Priv*, which is built upon a mature commercial SAST product–AppScan Source.
- We evaluated *Priv* on six web applications and show that *Priv* can help prioritize developers' quality-assurance effort, reduce false warnings, and provide customized fix suggetions for vulnerability warnings.
- *Priv* is currently in the process of being integrated into industrial practice, and some developed features are already offered to customers of AppScan Source.

Paper organization. The rest of the paper is organized as follows. Section II describes the background of one commercial SAST technique–AppScan Source. Section III discusses three challenges that developers may encounter when using SAST, their impact, and our solutions (*Priv*) to mitigate the challenges. Section IV describes our evaluation of *Priv*. Section V describes internal and external threats. Section VI surveys related work. Finally, section VII concludes the paper.

II. Background of AppScan Source

AppScan Source [8] is one of the leading commercial SAST products. AppScan Source presents detected vulnerability warnings to developers in a list and also provides different ways to categorize vulnerability warnings (e.g., severity). When developers click on one of the presented vulnerability warnings, AppScan Source will show the details of the warning (as shown in Figure 1). Figure 1 includes three main components: the data-flow path of the vulnerability warning (left-top component), a window to show relevant code (left-bottom component), and a remediation page to show generic guidance on how to fix this vulnerability type. AppScan Source stores the details of the warnings in assessment files. We built *Priv* by analyzing such assessment files.

Typical SAST techniques target information-flow vulnerabilities. An information-flow vulnerability starts with an incoming untrusted input (i.e., the source in a data-flow path, such as *getParameter(...)* in Figure 1) and ends at code that performs security-critical functionalities (i.e., the sink in a data-flow path), such as executing SQL queries (*executeQuery(...)* in Figure 1). For example, SQL injections are caused by writing unsanitized inputs that contain malicious activities to a database. The unsanitized input may come from web API calls (e.g., through a method call *javax.servlet.http.HttpSession.getAttribute* in Java), and being passed to database execution APIs (e.g., *java.sql.Statement.executeUpdate*).

We use an example of SQL injection to explain the representation of a data-flow path (also referred to as *trace*) in AppScan Source (the top-left component in Figure 1). The trace marks source and sink in red color. The source calls *getParameter(...)* to obtain input from an HTTP request. Such input may be malicious, and thus is considered as an untrusted input. The sink executes a SQL query to the database (i.e., using *java.sql.Statement.executeQuery*). The other nodes in the trace show how the untrusted input is propagated from the source to the sink. For example, the blue node on top (namely *root caller*), i.e., method `doPost`, calls `getParameter` and passes the obtained parameter value to the sink through method `getUserInfo`. Since the data flow between the source and the sink does not validate the untrusted input and the sink does not call *PreparedStatement*, this data flow is detected as a potential SQL injection vulnerability by AppScan Source.

III. Challenges, Impact and Solutions

In this section, we discuss the challenges that prevent developers from fully utilizing SAST techniques, the resulting impact, and our proposed solutions.

C1: Handling a large number of vulnerability warnings

Description. SAST techniques may detect hundreds or even thousands of vulnerability warnings in one application. All the detected warnings are presented to developers, and developers need to investigate them one-by-one. Developers receive limited support to have an overview of all the detected warnings, especially in a way that focuses on commonality, so that developers can prioritize their quality-assurance effort. Moreover, when working on one warning (i.e., the target warning), the developer may not know the impact of fixing this warning on other warnings, especially when the target warning share commonality with other warnings.

Impact. Developers are overwhelmed by the large number of vulnerability warnings detected by SAST. Failing to identify common problems shared by multiple vulnerability warnings leads to inefficiency in developers' quality-assurance effort. Moreover, commonalities among vulnerability warnings may indicate complications in how to fix them. For instance, adding sanitizing code at one code location may introduce side-effects. Such side-effects at common code location may

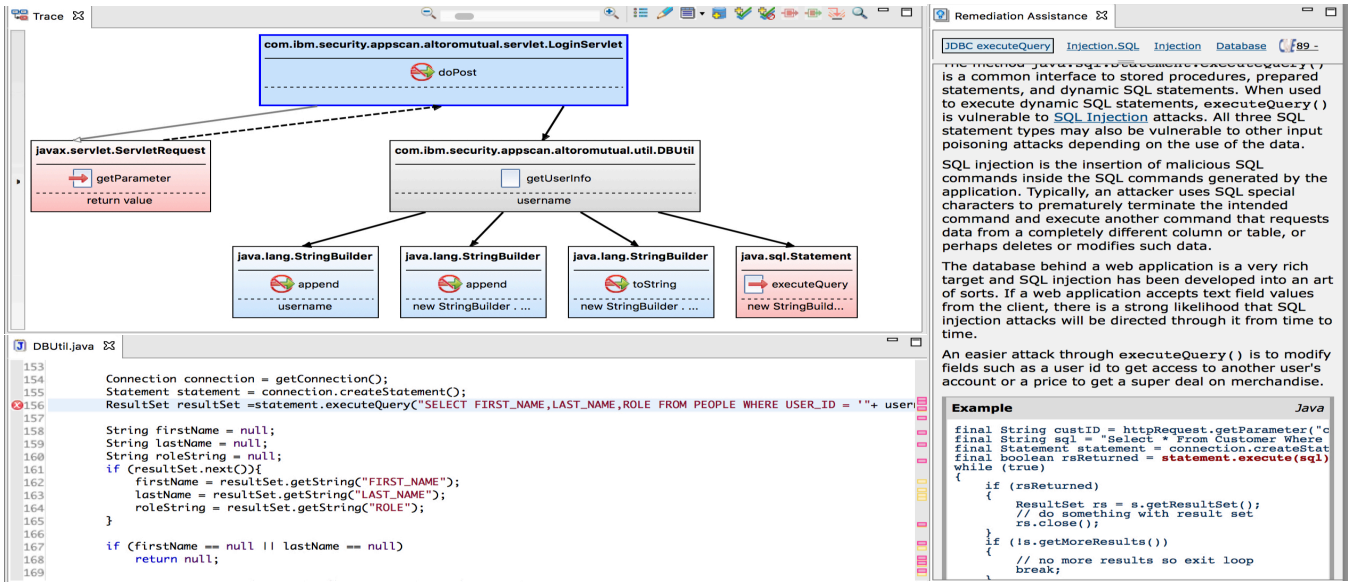


Fig. 1: For each vulnerability warning, AppScan Source shows the trace (i.e., to visualize the data-flow from source to sink), a code window to show the code snippet of one trace node (i.e., clicked by the user), and a generic remediation page (e.g., text description, examples of buggy code and fix).

improperly obstruct all the data-flow paths that cover the common code location.

Proposed Solution. To help developers efficiently resolve vulnerability warnings, we first propose an approach to identify the preferred fix locations (namely *pFixLocs*) for the warnings (e.g., adding a fix at one location can resolve the maximum number of warnings). Then we propose a global-view visualization that leverages the commonality in *pFixLocs* and presents the detected vulnerability warnings in groups, i.e., warnings in the same group share the *pFixLocs*. The visualization also highlights the complex interferences among groups. Below, we discuss the solutions in detail.

1) Finding preferred fix locations: For an information-flow vulnerability (as described in Section II), there may exist multiple code locations that developers can add a fix (e.g., code to validate untrusted input) to mitigate the security risk. To reduce code maintenance effort and manual effort (e.g., debugging and creating fixes), developers may prefer to add validation code at the most *effective* code location, where applying one fix can resolve a large number of vulnerability warnings. Such cost-effective fix location is referred to as *preferred* fix location (*pFixLoc*). *Priv* finds *pFixLoc* for four types of vulnerability warnings: SQL injection (SQLi), cross-site scripting (XSS), command injection (COMMi), and path traversal (PATHtrv).

Combining security experts' years of experience, we make the following strategy to find *pFixloc* for warnings. A data-flow path contains a source (i.e., where the input comes in), a sink (i.e., where the input become exploitable), a root caller (i.e., an indirect caller of both source and sink), and other nodes (i.e., either callers of the source or callers of the sink).

- For **XSS**, *Priv* finds three types of *pFixLocs*: the source, the direct caller of the source in the data-flow path, and Java classes that occur in the first half of the data-flow path (i.e., from the source to the root caller).
- For **SQLi, COMMi, and PATHtrv**, *Priv* finds three types of *pFixLocs*: the sink, the direct caller of the sink in the data-flow path, and Java classes that occur in the second half of the data-flow path (i.e., from the root caller to the sink).

We derived the rule-based strategy to find *pFixLoc* for the four above-mentioned vulnerability types. The rule-based strategy is proposed based on years of experience from the senior AppScan Source security experts. There are two factors that security experts and we consider when deriving the rule-based strategy for suggesting *pFixLoc*. The first factor is to maximize the fixing ability of *pFixLoc*. A *pFixLoc* is preferred if adding a fix at this *pFixLoc* resolves more vulnerability warnings. The second factor is to minimize the side effects that are introduced by adding validation code at *pFixLoc*. Such side effect refers to the impact of validation code on *other data-flow paths* that also go through the same *pFixLoc*. By considering the first factor only, *pFixLoc* should be as close to the source as possible (i.e., where untrusted data goes in) to maximize the fixing ability. Also, security experts from AppScan Source proposed the idea of validating Java classes (e.g., when initializing class fields), which also provides good fixing ability as it may cover all the data-flow paths where the Java object occurs. Considering the second factor, *pFixLoc* is preferred to be close to the sink, because the validation added at the sink (which is the end of the data-flow path), is less likely to introduce side-effect to other data-flow paths. Figure 2 shows an example trace for a cross-site scripting. The root (`_jspService`) gets untrusted input from the source (`getString()`) through the call chain that includes `getBankUsers` and `getBankUsernames`. Then, the root passes the unsanitized data to the sink

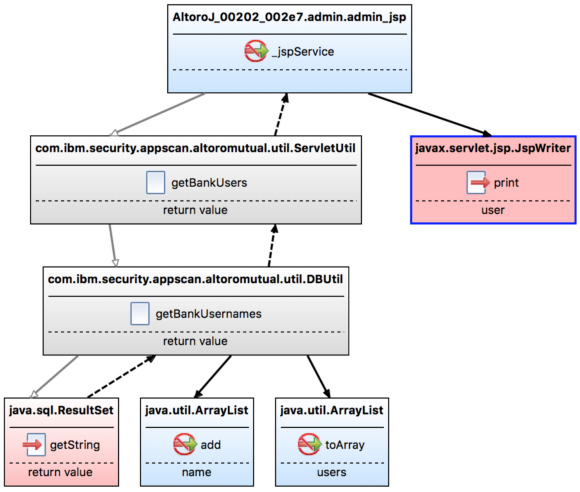


Fig. 2: An example trace of a cross-site scripting vulnerability.

(`JspWriter.print()`). The unsanitized data, if it contains a malicious script, will be executed by a victim's browser. *Priv* finds the following *pFixLoc* for this cross-site scripting: the source, the direct caller of the source, and Java objects that occur in the path from the source to the root caller. *Priv* highlights a list of potential *pFixLocs*, and does not make the final decision about where to fix. Instead, *Priv* uses the found *pFixLocs* to provide a global-view visualization to help developers decide the fix location for each vulnerability warning.

2) Grouping and visualizing vulnerability warnings based on shared pFixLocs: *Priv* applies a force-directed graph [14] to group and visualize all the detected vulnerability warnings in one application. In particular, a node represents either a vulnerability warning (*vul-node*) or a *pFixLoc* (*pFixLoc-node*). There exists a link between a vulnerability warning and all of its identified *pFixLocs*. Force-directed graph drawing algorithm requires no prior knowledge of the graph layout and tries to limit crossing links as few as possible. The force-directed graph algorithm centers vul-nodes around related pFixLoc-nodes, and pushes away the vul-nodes that are not linked to the centered pFixLoc-node. Therefore, using the force-directed graph drawing algorithm, *Priv* naturally groups vulnerability warnings based on shared *pFixLocs* and also presents a visualization.

Figure 3 shows an example of the global-view visualization that *Priv* generates for WebGoat 5.3. The visualization contains several groups: some complicated groups (e.g., the top-left group), and some simple yet large groups (e.g., the bottom-left group). This global-view visualization guides developers to work on the large groups for prioritizing quality-assurance effort, and informs developers that there are interferences between vulnerability groups, such as the complicated group on the top-left corner in Figure 3. In particular, when developers navigate vulnerability warnings in groups, they can observe that a fix added to one particular *pFixLoc* may potentially fix all the vulnerability warnings in the same group.

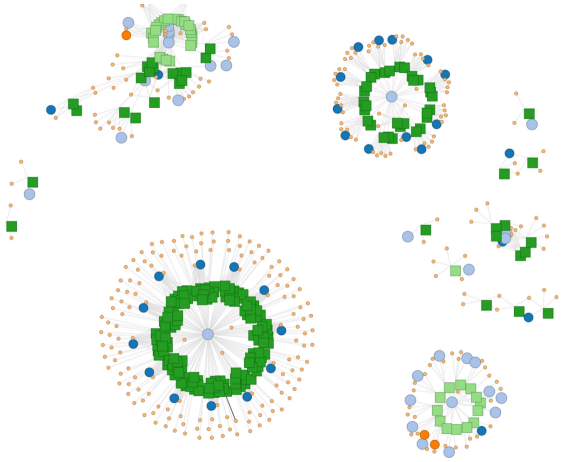


Fig. 3: *Priv* provides the global-view visualization for WebGoat 5.3. Rectangles with various colors represent different types of vulnerability warnings. Circles present trace nodes. Blue (both light and dark) circles are the *pFixLocs*. Orange circles are the trace nodes other than *pFixLocs*. This visualization is interactive: When developers click on one node, more detailed information will be shown.

Developers may select the group with the most warnings for prioritizing quality-assurance effort. Finally, developers become aware of potential side-effects of placing validation code as the interferences among groups are presented to them. For example, if a whitelist-based validation code is added to one fix location, and this fix location may be in the data flow of other vulnerability warnings, developers should be alerted about the potential side-effect introduced by this fix.

C2: Having a large number of false positives in the SAST detection result

Description. Similar to static bug detection [11], SAST also reports a large number of false positive warnings, which are not real vulnerabilities and do not require developers' attention. There are two types of false positives in SAST techniques: false positives introduced by limitations of static analysis (e.g., infeasible paths), and SAST-specific false positives due to incomplete information flows related to stored injections. A solution that identifies SAST-specific false positives can help developers focus on the actionable warnings.

Impact. High false positive rate decreases developers' interests in the detection result and hence makes SAST less useful. Time and resources are wasted in investigating false warnings while true vulnerabilities may remain unfixed even after extensive quality-assurance activities.

Proposed Solution. There exists prior effort to detect false positives in static bug detection [15], [16], [17]. In this work, we focus on reducing the number of SAST-specific false positives. We first describe SAST-specific false positives in detail, and then describe our solution which is based on establishing a complete data flow from two incomplete data flows.

SAST-specific false positives are related to stored injections

(i.e., second-order injections). Security experts from AppScan Source point out two primary types of stored injections that result in many false positives: database-related (DB-related) injections and attribute-related (attr-related) injections. DB-related injections are about storing untrusted input in the database. However, such injections only become real threats when the stored input is actually exploited (e.g., being executed). A DB-related injection is actionable if the stored data is later used in database queries (i.e., causing SQL injections) or in websites (i.e., causing cross-site scriptings). Similarly, actionable attr-related injections are about storing untrusted input in JSP (JavaServer Pages) attributes, which later become exploitable.

Priv identifies actionable DB- and attr-related vulnerabilities by connecting two related data flow paths, each of which is incomplete and shows only half of a complete data flow. The first half data flow is from an entry point of untrusted data to where the data is stored (*entry path*); and the second half is from where the data is stored to where the untrusted data is exploited (*exit path*). *Priv* first finds entry paths where the sink writes to a database or JSP attribute, and exit paths where the source reads from a database or JSP attribute. Then, *Priv* connects one entry path and one exit path if they read from or write to the same database table or the same JSP attribute. *Priv* automatically extracts the name of JSP attribute by using regular expressions since the name of an attribute is often string literals. *Priv* asks developers to specify database table name. Automatically extracting database table name remains as future work, such as performing semantic analysis on database-access code (e.g., SQL queries).

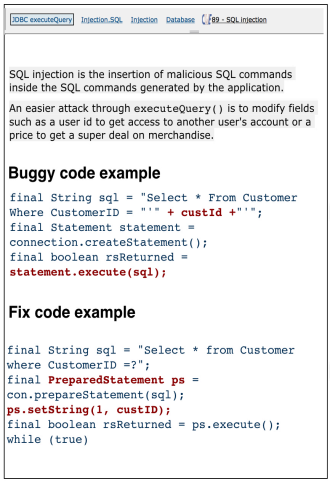
C3: Lack of customized fix suggestions

Description. Developers receive insufficient support about how to fix the detected vulnerabilities. SAST techniques may provide remediation pages to help developers understand each type of vulnerability and provide examples to show how to fix the detected vulnerability warnings. However, such example-based remediation pages are generic (e.g., one remediation page for each type of vulnerability), and can be difficult to comprehend and adapt due to their significant difference with the detected vulnerability warning.

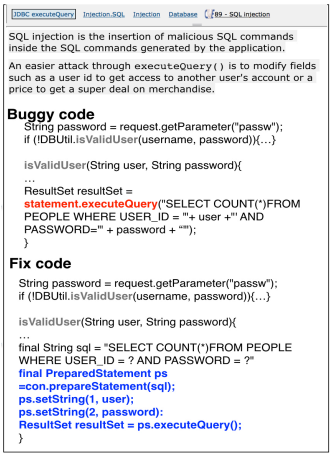
Impact. Insufficient support in fixing the detected vulnerability warnings causes a significant increase in the time that developers spend in quality-assurance activities [12]. The manual process of comprehending fix examples and applying them to new contexts is error-prone, and may lead to inaccurate fixes.

Proposed Solution. *Priv* leverages manually-derived fix templates, applies the derived fix templates in new contexts, and generates customized fix suggestions for the vulnerability warnings detected by SAST. The fix templates are manually derived from existing example-based remediation pages, i.e., generic fix suggestions (Fig. 4a). In particular, *Priv* intercepts the remediation page in AppScan Source, and replaces the generic fix information with a customized fix suggestion for each vulnerability warning.

Figure 4a shows the generic remediation page that AppScan Source provides for SQL injections. *Priv* customizes the generic remediation page by replacing the buggy code and the fix with customized ones, as shown in Figure 4b. The example fix code prevents a SQL injection by using `PreparedStatement`. A fix template is manually derived based on the example fix code. The fix code in Figure 4b is a customized fix suggestion based on the derived fix template, and is able to prevent a specific SQL injection which contains more complex semantics than the SQL injection example in the generic remediation page (Figure 4a).



(a) The remediation page of AppScan Source for SQL injections.



(b) *Priv* presents customized buggy code and fix suggestios instead of generic ones.

Fig. 4: *Priv* provides a customized fix suggestion instead of a generic example-based remediation page.

Priv re-constructs the buggy code and generates the fix suggestion by analyzing the information in the assessment file (i.e., the file where AppScan Source stores the detection result). This demonstrates that *Priv* can be integrated smoothly with AppScan Source and does not require changes to the current implementation of AppScan Source. In particular, *Priv* generates fix suggestions for four vulnerability types: cross-site scriptings, SQL injections, path traversals and command injections. Fix templates are manually concluded from the remediation pages of the four vulnerability types. It remains as future work to automate the process of manually crafting fix patterns from documentation [18].

IV. Evaluation

We evaluate *Priv* on six web applications: five are open source projects which are commonly used for studying vulnerabilities (WebGoat, Bodgeit, VulWeb, JavaVulLab, and Heisenberg), and one (AltoroJ) is for internal use by the development team of AppScan Source. Table I shows the details of the six evaluated web applications. In particular, we apply AppScan Source + *Priv* to find vulnerabilities in the six studied web applications and report changes and improvement that *Priv* introduces from the perspective of developers (the users of AppScan Source).

TABLE I: The summary of the six web applications that are used in the evaluation. 'Tot. warnings' is the number of warnings reported by AppScan Source.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
KLOC	3.7	321	2.6	3	1.5	4
tot. warnings	281	2,215	159	102	2,424	151
XSS	43	239	31	14	2,083	9
SQLi	55	78	5	12	94	16
COMMi	0	7	0	2	0	0
PATHtrv	3	60	0	3	18	4

We aim to answer the following three research questions:

RQ1: *What is the accuracy and effectiveness of using global-view visualization of vulnerabilities?*

RQ2: *How many actionable database- or attribute-related vulnerabilities that Priv finds?*

RQ3: *What is the quality of the fix suggestions provided by Priv?*

RQ1: What is the accuracy and effectiveness of using global-view visualization of vulnerabilities?

Motivation. The basis of *Priv*'s global-view visualization is whether or not a reasonable preferred fix location can be found by *Priv*. To assess the accuracy of the suggested *pFixLoc*, we compare the *pFixLocs* found by *Priv* with the ones annotated by developers. Moreover, one may wonder how effective the global-view visualization is in guiding developers to prioritize quality-assurance effort by working on larger groups of vulnerabilities first.

Approach. *Priv* works for four types of vulnerabilities (i.e., XSS, SQLi, COMMi, and PATHtrv). We applied *Priv* to visualize these four types of vulnerabilities detected in the six web application (Table I). A comparison-based evaluation is conducted to assess whether *Priv* finds reasonable *pFixLocs*. Also, we conduct a case study on AltoroJ to illustrate how the global-view visualization will impact developers' current workflow of fixing warnings detected by AppScan Source.

Our collaborators (i.e., security experts from AppScan Source) provided us their annotated fix locations for vulnerability warnings. Since the developers did not annotate the fix location of every single vulnerability (i.e., the annotation was for other purposes, not particularly for this evaluation), we only performed the comparison for a subset of all the XSS and SQLi warnings: In total, 190 XSS and SQLi vulnerabilities are included in this comparison. We compared the fix locations found by *Priv* with the ones annotated by security experts to evaluate whether *Priv* finds accurate preferred fix locations. Note that this comparison shall not be positioned as whether *Priv* suggests better or worse fix locations than that of developers, but should instead be a comparison which highlights the similarities and differences between the two. Also, we provide statistics to show to what extent, the visualization provided by *Priv* can help developers prioritize quality-assurance effort.

We defined and utilized the following metrics to answer this RQ. *Similarity* is used to measure the percentage of warnings that *Priv* can find fix locations that are identical to the ones annotated by developers. Since *Priv* may suggest more than one fix location for one warning, we consider that *Priv* suggests identical fix location if any of the suggested *pFixLocs* matches with the one annotated by the developers (note that *Priv* may suggest multiple fix locations and only one is identical with the developers' selection, which is measured by *cost*). Also, we used two metrics to measure the effectiveness of *pFixLocs* found by *Priv*: *reduction* and *cost*. *Reduction* is used to measure to what percentage can *Priv* narrow down the scope of fix location. For example, one vulnerability warning may have *N* code locations in the data flow (i.e., validation code can be added to one of the *N* code locations). If *Priv* suggests *only* one *pFixLoc*, then the reduction of *Priv* for this vulnerability warning is *(N-1)/N*. *Cost* is the number of unique *pFixLocs* suggested by *Priv*, which measures the cost brought by *Priv*'s found *pFixLocs* since *Priv* could suggest more than one fix locations for each vulnerability warning.

TABLE II: Comparison between the fix locations suggested by *Priv* and the ones annotated by developers.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
XSS						
w/ annotations	13	67	22	2	17	7
diff	0	29	11	1	0	0
identical	13	38	11	1	17	7
similarity	13/13	38/67	11/22	1/2	17/17	7/7
	100%	56.7%	50%	50%	100%	100%
SQLi						
w/ annotations	7	12	5	2	19	16
diff	0	1	0	2	3	0
identical	7	11	5	0	16	16
similarity	7/7	11/12	5/5	0	16/19	16/16
	100%	91.7%	100%	0	84.2%	100%

To illustrate the effectiveness of global-view visualization, we conduct a case study using AltoroJ. Global-view visualization guides developers to prioritize quality-assurance effort by first working on groups that have more vulnerabilities. Intuitively, investigating the largest group will resolve many vulnerabilities at once since the vulnerabilities share common fix locations. We performed a simulated study on AltoroJ with the assumption that developers will prefer to first work on the largest group of vulnerability warnings. We sorted the clusters based on the number of vulnerability warnings. Then, we estimated the effort of fixing the current largest group using the total number of suggested fix locations, and also presented the outcome of such fixing effort (i.e., number of warnings resolved).

Results. Table II shows the comparison result between the fix locations suggested by *Priv* and the ones annotated by developers. Because the annotation was not particularly performed for this evaluation (i.e., for other development goals), the warnings in the comparison (Table II) are a subset of all the XSS and SQLi warnings. Table II lists the number of warnings with developers' annotated fix locations, and the comparison result with *Priv*'s suggested fix location (i.e., 'diff', and 'identical'). 'Diff' means that the fix locations suggested by *Priv* are different from that of developers. 'Identical' means *Priv* suggests the same fix location as developers. *In summary, Priv achieves a 50–100% similarity when comparing the suggested fix locations with the ones annotated by developers.*

TABLE III: The table shows the statistics of preferred fix locations suggested by *Priv*. The table is divided into two parts: AppScan Source and AppScan Source + *Priv*. 'Tot. warnings' is the total number of warnings (the four studied vulnerability types) reported by AppScan Source. 'Tot. trace nodes' is the total number of trace nodes in the studied warnings. 'Tot. pFixLocs' is the total number of pFixLocs are found by *Priv*. 'Reduction' shows the percentage of 'tot. pFixLocs' in 'tot. trace nodes', i.e., reducing the number of code locations investigated. 'Cost' is the average number of code locations investigated to fix a warning.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
AppScan Source						
tot. warnings	101	384	36	31	2,195	29
tot. trace nodes	576	1,819	140	234	10,926	168
AppScan Source + ***Priv***						
tot. pFixLocs	158	495	36	87	2,297	32
reduction	72.6%	72.8%	74.3%	63.9%	79%	81%
cost	1.66	1.29	1	3.07	1.04	1.10

TABLE IV: The result from a case study on AltoroJ to show how effective the global-view visualization is in guiding developers prioritize quality-assurance effort. 'Order' shows the order that developers follow based on size of groups to incrementally fix all vulnerability warnings from AltoroJ. 'Cumulative effort' is the cumulative number of fix locations that require manual investigation for fixing one cluster. 'Outcome' quantifies the result of an effort, i.e., number of warnings resolved.

order	1	2	3	4	5	6	7	8
cumulative effort	8	12	14	15	16	17	18	19
outcome	41	48	52	54	56	58	60	61
outcome/effort	5.13	4	3.71	3.6	3.5	3.41	3.33	3.21

Table III shows the ability of *Priv* in reducing the scope of manual investigation of fix locations. For example, for AltoroJ, there are 95 warnings of XSS, SQLi, COMMi, and PATHtrv. For these 95 vulnerability warnings, there are a total of 570 possible fix locations (i.e., aggregating all the nodes in data-flow paths). *Priv* narrows to 152 fix locations. Thus, the *reduction* of *Priv* on AltoroJ is 73.3%, i.e., (570-152)/570. Table III also shows the *cost* (i.e., how many fix locations developers will investigate per vulnerability), which shows that on average 1.12 fix locations per vulnerability warning are found by *Priv*.

Table IV shows how *Priv* can prioritize quality-assurance efforts for AltoroJ (one of the evaluated projects). The order of each column complies with the order that developers follow based on the assumption (i.e., starting with the largest group). The first row shows the aggregated number of fix locations that developers would need to investigate. Correspondingly, the second row lists the aggregated number of vulnerability warnings would be resolved after the group at this order is investigated. The last row shows the average number of vulnerability warnings that would be resolved per fix location after each selection of group. For example, when developers choose to work on the first group (the column when 'order' is 1), 41 vulnerability warnings may be resolved after investigating eight fix locations. If there is no prioritization, for each fix location, developers would resolve 3.21 warnings on average (as shown in the last column in Table IV). With the prioritization provided by *Priv*, the average warnings per fix location are 5.13 to start with, and gradually decreases to 3.21. This shows that *Priv* can indeed improve work efficiency by prioritizing quality-assurance efforts.

TABLE V: The results of applying *Priv* to identify actionable warnings and thus removing SAST-specific false positives. *Priv* is able to identify all the actionable warnings and remove all the SAST-specific false positives.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
Database						
# of total warnings	20	298	35	9	128	16
# of actionable warnings	14	233	4	0	109	0
perc. of SAST-specific false positives	30%	21.8%	88.6%	N/A	14.8%	N/A
# of warnings reported by ***Priv***	14	233	4	0	109	0
# of **actionable** warnings by ***Priv***	14	233	4	0	109	0
Attribute						
# of total warnings	17	13,324	11	0	0	29
# of actionable warnings	4	2,170	6	0	0	25
perc. of SAST-specific false positives	76.5%	83.7%	45.5%	N/A	N/A	13.8%
# of warnings reported by ***Priv***	4	2,170	6	0	0	25
# of **actionable** warnings by ***Priv***	4	2,170	6	0	0	25

Priv finds identical preferred fix locations when compared with developers' annotations for 75.3% of the studied vulnerability warnings (i.e., ranging from 50% to 100% per web application). The global-view visualization that Priv generates for AltoroJ can prioritize quality-assurance effort, i.e., guiding developer to work on the largest group.

RQ2: How many actionable database- or attribute-related vulnerabilities that Priv finds?

Motivation. *Priv* reduces the rate of SAST-specific false positives by identifying actionable database- and attribute-related vulnerabilities. Reducing the number of false positives can maintain developers' interest in the detection result [12]. This RQ answers how many actionable warnings are identified by *Priv*, and, correspondingly, how many SAST-specific false positives are excluded.

Approach. We applied *Priv* on the six web application (see Table I) to identify actionable DB- and attr-related vulnerabilities. Since *Priv* requires manual effort to specify the names of the database tables when analyzing DB-related vulnerabilities, we manually provided such information in this evaluation. For the attribute-related warnings, *Priv* first extracts the values of attributes if the parameters of relevant APIs (i.e., `getAttribute` or `setAttribute`) are string literals. If not, *Priv* then requires manual input to provide such information. We manually provide the names of attributes if the first automated method fails to extract such information.

Results. Table V shows the results of applying *Priv* to identify actionable database- and attr-related warnings in the six

evaluated web applications. Table V shows the total number of DB-related or attribute-related warnings, the number of actionable ones among all the warnings, the percentage of SAST-specific false positives that are not actionable, and the number of **actionable** warnings identified by *Priv*. The authors and security experts manually examined the actionable warnings reported by *Priv* and found that **all** of the *Priv*'s reported warnings are indeed actionable. In short, *Priv* is able to identify all the actionable warnings and thus removing all the SAST-specific false positives (i.e. reducing SAST-specific false positive rate to 0%).

> Priv *detects a total of 2,565 actionable DB-related or attr-related vulnerabilities in the six evaluated web applications after excluding SAST-specific false warnings.* Priv *reduces the rate of SAST-specific false positives from 13.8%–88.6% to 0.*

RQ3: What is the quality of the fix suggestions provided by Priv?

Motivation. *Priv* provides customized remediation pages that include customized fix suggestions to fix the detected vulnerabilities and elevate the burden from developers. Due to the difficulties in adapting generic fix templates to each detected vulnerability warning, the customized fix suggestions provided by *Priv* can be classified based on the quality. The best-quality fixes provided by *Priv* can be directly applied to the source code (i.e., with complete essential and correct semantics). Meanwhile, the lower-quality fixes provided by *Priv* may be incomplete and still require developers' manual effort. This RQ studies for how many vulnerability warnings, *Priv* can provide customized fix suggestions. Moreover, we evaluate the quality of the customized fix suggestions regarding whether the fix can be directly applied to the source code, and if not, how much extra effort is required (e.g., minor modifications).

Approach. The authors and the security experts manually examined the quality of all the customized fix suggestions provided by *Priv*, except for XSS in JavaVulnLab (AppScan Source finds a total of 2,083 XSS vulnerabilities in JavaVulnLab, so we took a sample of 100 for manual examination). To qualitatively evaluate the quality of the fixes provided by *Priv*, we classify each fix into one of the three categories: *fully complete and correct*, *partially compilable/correct* (i.e., minor modifications required), *fix template only* (i.e., developers need to add extra code, such as validation methods). *Fully complete and correct* means that one fix that can be *directly* applied to the code without modifications, which means the patched code is compilable and indeed fixes the vulnerability. *Partially compilable/correct* means that the suggested fix *cannot* be directly applied to the code as the patched code would be either not compilable or not entirely correct. The fixes of this category may require minor modifications to become *fully complete and correct*. Figure 5 shows an example of *partially compilable/correct* fixes that *Priv* generates for one SQL injection. The fix is not entirely correct: the patched code replaces `dateString` with '?' in line 13 (the unpatched code is in line 8) because *Priv* fails to identify that `dateString` is used in a condition. If `dataString` is concatenated to a SQL query, then a correct fix should replace it with a **?**.

```
1  if (startDate != null && startDate.length()>0 && ... ){
2    dateString = "DATE␣BETWEEN␣"+startDate+"00:00:00␣AND␣"\
3                 + endDate +"␣23:59:59";
4  } else if (startDate != null && startDate.length()>0){
5    dateString = "DATE␣>␣" + startDate +"␣00:00:00"; }
6  - String query = "SELECT * FROM TRANSACTIONS WHERE (" \
7  -     + acctIds.toString() + ") " \
8  -     + ((dateString==null)?"": "AND (" \
9  -     + dateString + ") ") \
10 -     + "ORDER BY DATE DESC" ;
11 + String query = "SELECT * FROM TRANSACTIONS WHERE (" \
12 +     + "?" + ") " \
13 +     + (("?"==null)?"": "AND (" \
14 +     + "?"+ ") ") \
15 +     + "ORDER BY DATE DESC" ;
16 + query.setString(1, acctIds.toString());
17 + query.setString(2, dateString);
18 + query.setString(3, dateString);
19 + query.setString(4, startDate);
20   resultSet = statement.executeQuery(query);
```

Fig. 5: A customized fix suggestion provided by *Priv* for a SQL injection vulnerability. The fix suggestion is classified as 'partially compilable/correct'. This customized fix is partially correct because *dateString* in line 8 is incorrectly replaced by ? in line 13 since *Priv* is not able to infer that *dateString* is used in a condition.

Fix template only means that a fix is the least complete, which only contains the skeleton of a fully complete and correct fix, but lacks concrete implementation (e.g., validation code to prevent malicious input). For example, to prevent a path traversal in `File file = new File(path)` since *path* can be malicious and untrusted input, *Priv* suggests to validate *path*, e.g., creating a whitelist-based validation so that only valid paths are allowed. *Priv* provides a customized fix template `if(validate(path))` and developers will need to implement the method *validate(...)*. Even for this lowest-quality category of *Priv*'s fix suggestions, *Priv* still provides a better customized fix suggestion than a generic fix template, since the generic fix template will not show that *path* is the variable that requires validation.

Results. Table VI shows the results on the quality of the fixes provided by *Priv*. Each fix is manually examined by the authors and security experts, and classified into one of the three above-mentioned categories.

For the majority of the cross-site scriptings (100% in three of the evaluated web applications), *Priv* provides complete and correct fixes. For 27 cross-site scriptings (21 from Altoroj, one from Bodgeit, and 5 from JavaVulLab), *Priv* does not provide fully complete fixes due to the limited information stored in the AppScan Source assessment file. Sometimes the AppScan Source assessment file does not store the original information about the tainted parameter in the sink; instead, an intermediate representation is stored. Below is an example of such cases, which may execute a malicious input:

```
<%= (request.getAttribute("message_feedback")!=null)?"...%>
```

In the assessment file, *request.getAttribute(...) != null* is stored as `Temp@11@0`. Thus, *Priv* is not able to infer that

TABLE VI: The results of manual examination on the quality of the fixes generated by *Priv*. N/A means that AppScan Source detects none of this vulnerability type.

	AltoroJ	WebGoat	Bodgeit	VulWeb	JavaVulLab	Heisenberg
Cross-site Scripting						
fully complete and correct	22	239	30	14	95[1]	9
partially compilable/correct	21	0	1	0	5	0
fix template only	0	0	0	0	0	0
SQL Injection						
fully complete and correct	42	78	2	0	51	0
partially compilable/correct	6	0	0	0	0	0
fix template only	7	0	3	12	43	9
Path Traversal						
fully complete and correct	0	0	N/A	0	0	0
partially compilable/correct	0	0	N/A	0	0	0
fix template only	3	60	N/A	3	18	4
Command Injection						
fully complete and correct	N/A	0	N/A	0	N/A	N/A
partially compilable/correct	N/A	0	N/A	0	N/A	N/A
fix template only	N/A	7	N/A	2	N/A	N/A

request.getAttribute() occurs in a condition, and therefore, incorrectly places *request.getAttribute()* in a validation method.

Priv generates fully complete fixes for 70.9% of the evaluated SQL injections. The main reason that *Priv* does not generate fully complete fixes is that limited information is stored in assessment files (similar to the cross-site scripting cases). For the example shown in Figure 5, AppScan Source uses intermediate variables to represent *dateString==null*. Therefore, *Priv* cannot analyze the condition accurately and produces a partially correct fix which replaces *dateString* with a '**?**'. For all the path traversals and command injections, *Priv* provides fixes in the category 'fix template only' since the validation code would require developers' domain knowledge.

Priv *generates fully complete and correct fixes for 75.6% of the vulnerability warnings, partially compilable/correct fixes for 4.2%, and fix templates for 2.1%. Even the least complete category of 'fix template only' is more customized than the generic fixes in the current remediation pages.*

V. Threats to Validity

Internal Threats. The preferred fix locations (i.e., the basis of building global-view visualization) found by *Priv* are compared with security experts' annotation. We believe such annotation is trustworthy due to the security experts' years of experience. However, there could be more than one way to fix vulnerabilities and sometimes the process is subjective. Future research should include an extensive user study to obtain such annotations of fix locations from professional developers. Also, for each vulnerability type, we only conduct the comparison on a subset of all the warnings; however, we believe developers would follow consistent rules when annotating for the remaining warnings of the same type. In the future, we plan to extend the comparison evaluation. *Priv* is not designed to identify non-SAST-specific false positives, i.e., caused by the limitations of static analysis. Therefore, some vulnerability warnings that are detected by SAST, and analyzed by *Priv* can still be false warnings. However, we believe that this should not negatively impact *Priv*'s improvement. Moreover, *Priv* can be combined with techniques that focus on reducing false positives caused by limitations of static analysis. Last, the current implementation of *Priv* relies on manual annotations to obtain database table and attribute names in reducing SAST-specific false warnings. It remains as future work to conduct a user study to assess the quality of the manual annotations and even to provide an automated solution.

External Threats. *Priv* is built on one commercial product–AppScan Source. Although we believe *Priv* can also benefit other SAST techniques, in this paper, we did not evaluate how effective that would be, which remains as future work. *Priv* is built on AppScan Source, and thus is applied on web applications in Java. Future work may extend *Priv* to improve SAST for other programming languages, and frameworks, e.g., mobile applications. *Priv* targets four types of vulnerabilities (XSS, SQLi, COMMi, PATHtrv). Other types of information-flow vulnerabilities may also benefit from *Priv*, but in this paper, we do not implement *Priv* for information-flow vulnerabilities other than the four above-mentioned types. However, the four types are the most common vulnerability types [13].

VI. Related Work

Detecting Vulnerabilities Using Static Analysis. Static analysis is shown to be effective in detecting vulnerabilities, especially information-flow-related vulnerabilities [5], [3], [4], such as cross-site scriptings and SQL injections. Also, static analysis is widely used to improve mobile security by detecting data leaks to protect sensitive and confidential information [19]. However, a previous study by Johnson et al. [11] shows that developers do not fully utilize static analysis techniques because of two reasons: high false positive rates and the presentation of the detected results.

Helping Developers Better Utilize Static Analysis Results. To make static bug detection more useful, researchers have been working on decreasing the high false positive rate. Junker et al. [20] convert static analysis to a model checking problem and then utilize SMT solver to check path feasibilities. Fehnker et al. [21] propose a technique to reduce false positives by leveraging refined security rules. Muske et al. [22] propose a partitioning approach to reduce false positives. The results of the static analysis techniques are divided into equivalence classes. If the leader of a class is determined as false positive, then the entire equivalence class is false positives. Chen et al. [12] propose techniques to prioritize the detected problems based on their potential impact.

In addition, many statistic-based approaches are proposed to identify false positives of static bug detection. Shen et al. [23] improve the FindBugs [1] tool by using an error-ranking strategy to increase true positive rate. Jung et al. [24] combine statistical analysis with domain knowledge to reduce false positives. Krememnek et al. [25] rank static analysis warnings by combining correlation between warnings and feedbacks from developers. Hallem et al. [26] propose a flexible extension language allowing users to specify system-specific rules for detecting bugs. Tripp et al. [15] extract features and interactively reduce false positives by using classification. Similarly,

Hanam et al. [16] use classification based on features from code patterns and leverage past unactionable alerts (i.e., false positives) as a training set. Ruthruff et al. [17] use code metrics and false warning patterns to predict actionable warnings. Le et al. [27] compute path dependencies of warnings to group relevant warnings together to reduce redundancy.

Different from prior work, *Priv* focuses on prioritizing developers' quality-assurance efforts in resolving the detected vulnerability warnings by visualizing the warnings in groups, and providing automatically-generated fix suggestions. In addition, *Priv* significantly reduces false positive rate (i.e., reduces up to 88.6%) of database-related and attribute-related vulnerability warnings, which are different types of false positives that are not addressed in previous work.

VII. Conclusions

In this paper, we document our collaborative work with security experts from AppScan Source to improve static application security testing (SAST) techniques. We propose several approaches to improve the utilization of SAST techniques, and we implemented the approaches as a tool call *Priv*. *Priv* is currently in the process of being integrated into industrial practice, and some features of *Priv* are already being offered in commercial SAST products. *Priv* helps prioritize developers' quality-assurance efforts. *Priv* also identifies actionable warnings by locating relevant warnings for database- and attribute-related warnings. Finally, *Priv* improves the current remediation pages in a commercial SAST product with customized remediation that includes the customized fix suggestion for each detected vulnerability warning. Our evaluation shows that *Priv* is effective in prioritizing quality-assurance efforts. *Priv* is able to identify actional warnings by excluding up to 88.6% SAST-specific false positives, and provides customized fix suggestions (many are fully complete and correct). In short, *Priv* can help developers better utilize SAST techniques, and thus improving software quality.

References

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sept 2008.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA, 2007, pp. 805–806.

[3] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium*, ser. SSYM, 2005, pp. 18–18.

[4] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP, 2006, pp. 258–263.

[5] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2006.

[6] W. G. J. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2005, pp. 174–183.

[7] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS, 2012, pp. 229–240.

[8] IBM, "Appscan source," 2017, https://www.ibm.com/us-en/marketplace/ibm-appscan-source.

[9] "Checkmark static code analysis," 2017, https://www.checkmarx.com/technology/static-code-analysis-sca.

[10] "Fortify static code analyzer," 2017, https://software.microfocus.com/it-it/software/sca.

[11] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE, 2013, pp. 672–681.

[12] T. H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting problems in the database access code of large scale systems - an industrial experience report," in *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, ser. ICSE-SEIP, 2016, pp. 71–80.

[13] M. Surf and A. Shulman, "How safe is it out there? zeroing in on the vulnerabilities of application security," in *Imperva Application Defense Center Paper*, 2004.

[14] R. Tamassia, *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*, 2007.

[15] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "Aletheia: Improving the usability of static security analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2014, pp. 762–774.

[16] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR, 2014, pp. 152–161.

[17] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE, 2008, pp. 341–350.

[18] J. Yang, E. Wittern, A. T. T. Ying, J. Dolby, and L. Tan, "Towards extracting web api specifications from documentation," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 454–464. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196411

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2014, pp. 259–269.

[20] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "Smt-based false positive elimination in static program analysis," in *Proceedings of the 14th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ser. ICFEM, 2012, pp. 316–331.

[21] A. Fehnker, R. Huuck, S. Seefried, and M. Tapp, "Fade to grey: Tuning static program analysis," *Electronic Notes in Theorectical Computer Science*, vol. 266, pp. 17–32, Oct. 2010.

[22] T. B. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *Proceedings of the 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM, 2013, pp. 106–115.

[23] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 299–308.

[24] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis," in *Proceedings of the 12th International Conference on Static Analysis*, ser. SAS, 2005, pp. 203–217.

[25] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT/FSE, 2004, pp. 83–93.

[26] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI, pp. 69–82.

[27] W. Le and M. L. Soffa, "Path-based fault correlations," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE, 2010, pp. 307–316.