ALETHEIA: Improving the Usability of Static Security Analysis

Omer Tripp Salvatore Guarnieri Marco Pistoia Aleksandr Aravkin
IBM T. J. Watson Research Center, Yorktown Heights, New York, USA
{otripp,sguarnieri,pistoia,saravkin}@us.ibm.com

ABSTRACT

The scale and complexity of modern software systems complicate manual security auditing. Automated analysis tools are gradually becoming a necessity. Specifically, static security analyses carry the promise of efficiently verifying large code bases. Yet, a critical usability barrier, hindering the adoption of static security analysis by developers, is the excess of false reports. Current tools do not offer the user any direct means of customizing or cleansing the report. The user is thus left to review hundreds, if not thousands, of potential warnings, and classify them as either actionable or spurious. This is both burdensome and error prone, leaving developers disenchanted by static security checkers.

We address this challenge by introducing a general technique to refine the output of static security checkers. The key idea is to apply statistical learning to the warnings output by the analysis based on user feedback on a small set of warnings. This leads to an interactive solution, whereby the user classifies a small fragment of the issues reported by the analysis, and the learning algorithm then classifies the remaining warnings automatically. An important aspect of our solution is that it is user centric. The user can express different classification policies, ranging from strong bias toward elimination of false warnings to strong bias toward preservation of true warnings, which our filtering system then executes.

We have implemented our approach as the ALETHEIA tool. Our evaluation of ALETHEIA on a diversified set of nearly 4,000 client-side JavaScript benchmarks, extracted from 675 popular Web sites, is highly encouraging. As an example, based only on 200 classified warnings, and with a policy biased toward preservation of true warnings, ALETHEIA is able to boost precision by a threefold factor ($\times 2.868$), while reducing recall by a negligible factor ($\times 1.006$). Other policies are enforced with a similarly high level of efficacy.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, statistical methods*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*user interfaces*

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

General Terms

Security, Verification, Human Factors, Algorithms, Measurement

Keywords

usability, false alarms, static analysis, information-flow security, classification, machine learning

1. INTRODUCTION

The scale and complexity of modern software systems often lead to missed security vulnerabilities. Static analysis has emerged as a promising solution for automated security auditing, which can scale to millions of lines of code while accounting for nontrivial program behaviors, resulting in the detection of severe and sometimes also subtle vulnerabilities [29, 7, 28, 8].

In particular, static analysis has shown great value when applied to *information-flow vulnerabilities* [22, 5]. These are the most serious and prevalent forms of vulnerability in today's landscape of Web and mobile applications,[1] which divide into two broad categories: *integrity* and *confidentiality* violations. Integrity violations include cross-site scripting (XSS), whereby unvalidated input may execute on the victim's browser; cross-application scripting (XAS), whereby tags injected into widgets can alter their appearance and behavior; and SQL injection (SQLi), whereby unintended SQL commands are executed. Confidentiality is the dual problem. It arises when sensitive data emanating from the application is released to unauthorized observers.

Static Information-flow Analysis.

A popular method of detecting information-flow vulnerabilities using static analysis, known as *taint analysis*, is by solving a "reachability" problem. Statements reading untrusted user input like the URL query string (or dually, sensitive information like the user's location) are defined as the *sources*. Statements that perform security-critical functionality like updating a database (or dually, release of information) are defined as the *sinks*. The third and final category is statements that sanitize or validate the input (or dually, declassify sensitive data), which are dubbed *downgraders*.

The analysis checks for reachability between source and sink statements via a downgrader-free path. Such paths, which we refer to as *witnesses* or *counterexamples*, are reported as potentially vulnerable. As an illustration, we provide in Figure 1 an exemplary witness reported by a commercial taint analysis for client-side JavaScript, which we have also used for our experiments.[2]

[1] `owasp.org`.

[2] We have anonymized the actual file names to avoid from disclosing the Web site's identity.

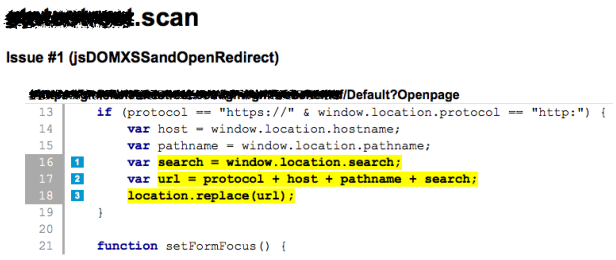

Figure 1: Security Witness Reported by a Commercial Static Security Checker for JavaScript

The reported vulnerability in this case is *open redirect*, which occurs when the user is able to influence the target URL of a redirection operation. The analysis bases this warning on information flow between a statement that obtains the URL query string (stored as `location.search`) and a statement performing redirection (`location.replace(...)`). The intermediate statement concatenates the return value from the source (pointed-to by variable `search`) with other strings, and thus the taint tag is carried across to the resulting `url` string.

Generalizations and extensions of taint analysis include features such as string sensitivity, whereby the analysis explicitly tracks string values and their structure for increased precision [26]; typestate-based tracking rules to refine the security specification [13]; as well as quantitative notions of information flow for more accurate and informative warnings [14]. In all of these cases, however, the key concept of verifying disjointness between sources and sinks modulo permitted exceptions remains the same.

Static security verification is not a silver bullet. To handle industry-scale applications, the analysis must apply aggressive approximations. Notable dimensions of precision loss include *flow insensitivity*, whereby the analysis does not track the order in which memory updates occur and instead conservatively accounts for all possible update orders; *path insensitivity*, whereby the analysis ignores path conditions, thereby traversing infeasible execution paths; and *context insensitivity*, whereby the analysis refrains from modeling the calling context of a method, thereby analyzing infeasible invocation scenarios.

These (and other) sources of inaccuracy, which we discuss in more detail and illustrate in Section 2.1, shrink the analysis' state space (also known as the *abstract state*) by an exponential factor, and hence contribute significantly to the scalability of the analysis. As an example, if path conditions are accounted for, then the analysis has to track two different runtime program states when reaching a branching condition. Path insensitivity saves the analysis from this type of state-space blowup.

Usability of Static Analysis Tools.

Approximations applied by the analysis can result in false alarms. These may be due to various reasons, including e.g. infeasible control flow, if the witness contains invalid branching decisions or call sites that are resolved incorrectly; ignoring of downgrading operations, if a proprietary downgrader is applied or downgrading occurs inline; and imprecise tracking of data flow, e.g. due to coarse modeling of string operations and/or aliasing between variables.

Indeed, while approximation enables scalability, it often also results in an excess of false alarms.These plague the reports by static analysis tools [15], thereby hindering their usability. According to interviews with professional developers, false alarms are the most significant barrier to adoption of tools based on static program analysis by developers [11]. As things now stand, developers prefer to release and deploy insecure software rather than find and fix latent vulnerabilities using off-the-shelf static security checkers. This is unfortunate.

Scope and Approach.

Our goal in this paper is to improve the usability of static security checkers by cleansing their output. We put forward two basic requirements:

- Generality: The cleansing technique should not be specific to a given tool or analysis technique. It should rather treat the analysis algorithm as opaque for wide applicability and ease of integration into legacy as well as new analysis tools.
- Customizability: Different users have different preferences when reviewing security warnings. Some prefer to aggressively suppress false alarms, even at the expense of eliminating certain true issues, whereas others may opt for completeness at the price of more false warnings.

Driven by these requirements, we have developed a method for filtering false warnings that combines lightweight user interaction with heavyweight automation. In our approach, the user classifies a small portion of the raw warnings output by the analysis. The user also specifies a tradeoff between elimination of false alarms and preservation of true findings. These two inputs are fed into a statistical learning engine, which abstracts the warnings into feature vectors that it uses to automatically build a filter. The filter, which is instantiated according to the user-specified policy, is next applied to the (vast majority of) remaining warnings, resulting in (many) less warnings than those initially reported by the security checker. Importantly, our learning-based approach is guided solely by the warnings themselves. It does not make any assumptions about, or access to, the internals of the analysis tool.

We have implemented our approach as the ALETHEIA system. To build the filter, ALETHEIA searches through a library of classification algorithms. It computes a competency score for each of these algorithms based on the user-classified warnings. The most favorable candidate is then applied to the remaining warnings.

To evaluate ALETHEIA, we ran a commercial static JavaScript security checker on a set of 1,700 HTML pages, taken from a diversified set of 675 top-popular Web sites, which resulted in a total of 3,758 warnings. These were then classified by a security specialist as either true or false warnings. We report on a wide set of experiments over this dataset. The results are highly encouraging. As an example, for a policy biased toward preservation of true positives, given only 200 classified warnings, ALETHEIA is able to boost precision by a factor of 2.868 while reducing recall by a negligible factor ($\times$1.006). Conversely, if the user is more biased toward elimination of false alarms, still based on only 200 classified warnings, ALETHEIA achieves a recall-degradation factor of only 2.212 and a precision-improvement factor of 9.014. In all cases, and across all policies, ALETHEIA is able to improve precision by a factor ranging between $\times$2.868 and $\times$16.556 in return to user classification of only 200 warnings.

Contributions.

This paper makes the following principal contributions:

1. **Boosting Usability via Learning:** We propose a novel and general technique to boost the usability of static security checkers. In our approach, the users invest tolerable effort in classifying a small portion of the warnings, and in return a statistical learning engine computes a filter over the remaining

warnings. The filter is parameterized by the user's preference regarding the tradeoff between true and false alarms.

2. **Characterization of Security Warnings:** We characterize different features of static security warnings, and draw general conclusions about their correlation with the correctness of a reported warning. This insight is of independent value, and can serve for other studies as well. We also discuss and analyze the relative merits and weaknesses of different classification algorithms, which is again of general applicability.
3. **Experimental Evaluation:** We have implemented our approach as the ALETHEIA system, which is to be featured in a commercial product: IBM Security AppScan Source. Experimental evaluation of ALETHEIA on a diversified set of benchmarks has yielded highly encouraging results.

2. OVERVIEW

In this section, we motivate the need for the ALETHEIA system and provide a high-level description of its main components and properties.

2.1 Limitations of Static Analysis

As highlighted above, static program analysis has inherent limitations in precisely modeling runtime behaviors of the subject program. Added to these, the analysis often deliberately sacrifices precision (even when a precise model is possible) in favor of scalability. Following are design choices that are often made for the analysis to scale to large codes:

- Flow insensitivity: The analysis does not track the order in which memory updates occur, and instead conservatively accounts for all possible update orders. A simple example is the following:

```
x.f = read(); x.f = ""; write(x.f);
```

  Even though the value of `x.f` is benign when it flows into the `write(...)` sink, the analysis abstracts away the order of updates to `x.f` and simply records that at some point that field was assigned an untrusted value.
- Path insensitivity: Path conditions are ignored. Instead, the analysis assumes that all paths through the control-flow graph (CFG) are feasible. Here is an example:

```
x.f = ""; if (b) { x.f = read(); } if (!b) { write(x.f); }
```

  The above code is not vulnerable, as the conditions governing the source and sink statements are mutually exclusive. Yet a path-insensitive analysis would report the infeasible trace through both the source and the sink as a warning.
- Context insensitivity: The analysis abstracts away the context governing the invocation of a method, thereby merging together different execution contexts of the same method, as this example illustrates:

```
y1 = id(x); y2 = id(read()); write(y1);
```

  Here `id(...)` is simply the identity method, which echoes back its input. In the first invocation the input is trusted, while the second invocation is with an untrusted argument. Smushing together the two contexts, the analysis conservatively judges that `id(...)` may return an untrusted value, and therefore `y1` is treated as untrusted and a vulnerability is reported when `write(...)` is called.

Design choices like the above, which are pertinent for performance and scalability, each contribute to the analysis' imprecision. Worse yet, there are significant interaction effects between the different sources of imprecision. This defines the need for complementary machinery to cleanse the analysis' output. In our approach, this step is carried out interactively, with help from the user, by casting the warnings reported by the analysis into a statistical learning framework.

2.2 System Architecture

We describe the high-level architecture of ALETHEIA with reference to Figure 2. The input to ALETHEIA is the raw warnings $\{w_1, \ldots, w_n\}$ output by the static security checker. Next, the user is asked to classify a subset of the warnings. This subset is selected at random to avoid biases. The result is a classified subset $\{(w_{i_1}, b_{i_1}), \ldots, (w_{i_k}, b_{i_k})\}$ of the warnings, where the labels b_{i_j} are boolean values (indicating whether the warning is true of false). Naturally, the accuracy of the filter computed by ALETHEIA is proportionate to the number of warnings reviewed by the user. However, as we demonstrate experimentally in Section 5, even a relatively small sample of 100 warnings suffices for ALETHEIA to construct a highly precise filter.

To cast security warnings into a statistical setting, we need to derive simple-structured features from the warnings, which are complex objects that cannot tractably be learned directly. This part of the flow is visualized in Figure 2 as the "feature mapping" box. A given warning is abstracted as a set of attributes, including *e.g.*, the respective line numbers of the source and sink statements, the time required by the analysis to compute the witness, the number of flow steps along the witness, etc.

The feature vectors, combined with the user-provided true/false tags and policy, provide the necessary data to learn and evaluate filters. Given training data of the following form:

$$\begin{array}{rcl} [\mathit{length} = 14, \mathit{time} = 2.5, \mathit{srcline} = 10, \ldots] & \mapsto & \mathit{false} \\ [\mathit{length} = 6, \mathit{time} = 1.1, \mathit{srcline} = 38, \ldots] & \mapsto & \mathit{true} \\ [\mathit{length} = 18, \mathit{time} = 3.6, \mathit{srcline} = 26, \ldots] & \mapsto & \mathit{false} \\ \ldots & & \end{array}$$

the ALETHEIA system partitions the data into training and testing sets of equal cardinality. ALETHEIA then generates a set $\mathcal{F}$ of candidate filters by training different classification algorithms on the training set. ALETHEIA also converts the policy into a scoring function SCORE. Next, each of the candidate filters is applied to the testing set, and the resulting classifications are reduced to a score via the SCORE function based on the rate of true positives, false positives and false negatives. Finally, the filter that achieves the highest score is applied to the remaining warnings. The user is presented with the findings surviving the filter.

2.3 Features and Learning Algorithms

Naturally, the efficacy of ALETHEIA is dependent on the available features and learning algorithms. We describe both in detail in Sections 3 and 4, respectively. Here we give a brief informal description of both.

Features.

As explained earlier, features are an abstraction of witnesses reported by the analysis tool. Most of the features that we have defined reflect basic characteristics of the witness, such as (i) its length (*i.e.*, the number of statements along the path), (ii) the syntactic location of the source and sink statements, or (iii) the context manipulated by the JavaScript code (plain DOM elements, Flash, JavaScript, etc). We have also defined general features that are not derivable directly from the witness itself, but rather reflect metadata associated with the witness, such as the time required by the analysis to detect the respective violation.

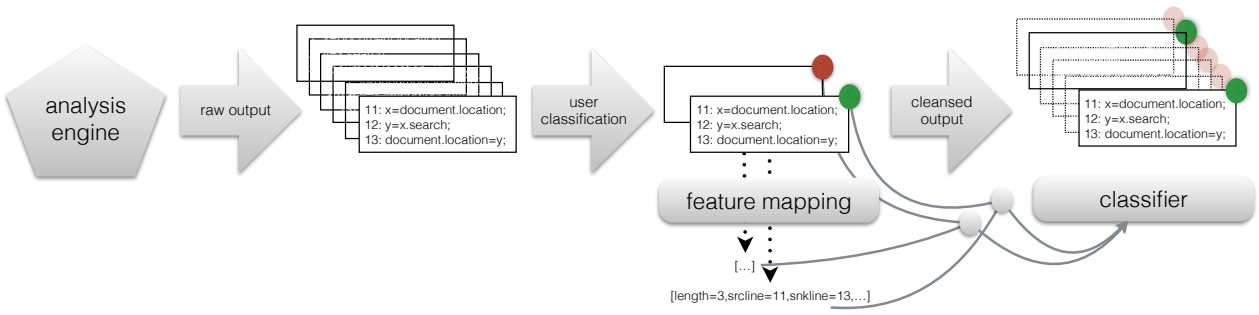

Figure 2: Visual description of the workflow of the ALETHEIA system

We note, importantly, that simply defining a large set of arbitrary features may lead the learning algorithm to discover false correlations. For this reason, we have made a careful selection of features. Behind each of the features is a justification for why it may correlate with true/false alarms. As an example, many findings involve imported code, such as the `swfobject` library for embedding Flash content in HTML files.[3] These often invoke sources and/or sinks (*e.g.*, reading or writing of the document's URL). The developer may deem flows emanating from, passing through or arriving at such libraries as spurious. This may happen, *e.g.*, if the library applies inline sanitization that the analysis misses. Thus, the source/sink location may become an important feature, which correlates well with user classifications.

Learning Algorithms.

There is a wide spectrum of classification techniques [30, 9]. These range between tree-based classification (*e.g.*, decision trees [21]); rule-based algorithms [17]; functional methods, which compute a geometrical boundary between the instances [20]; Bayesian techniques, which compute the probability of events (actual vs spurious vulnerabilities in our case) as a function of feature attributes [18]; and even (unsupervised) clustering techniques like K-means that form clusters and tag new instances according to their assigned cluster [4].

An important property of ALETHEIA is that it lets the user specify the filtering policy. As we explain in Section 4.5, and validate experimentally in Section 5.3, different policies are best served by different classification algorithms. As an example, if the user is strongly biased toward elimination of false alarms, then a rule-based classifier may identify a feature f and a threshold value v, such that (almost) any alarm for which $[\![f]\!] \geq v$ is false. This would satisfy the user's preference perfectly, but at the same the classifier may also eliminate true vulnerabilities. Striking a non-trivial balance between precision and recall, on the other hand, may leads toward a more sophisticated classifier ((*e.g.*, a decision tree or a kernel SVM). In light of this observation, we have linked into ALETHEIA 8 popular classifiers that together all the algorithmic categories above (functional, rule based, tree based, etc).

3. LEARNING FEATURES

As we highlighted earlier, in Section 2, the choice of which features to map a security witness to has important bearing on the overall quality of the filtering algorithm. The main concern with introducing arbitrary features is that the algorithm could be led to discover false correlations between such features and witness correctness. We dedicate this section to explaining the rationale behind the features we have selected. We divide the discussion according to feature categories.

[3] `https://code.google.com/p/swfobject/`

3.1 Lexical Features

A natural category of features are those recording syntactic properties of the witness. Specifically, we have defined seven such features, as follows:

- Source identifier (srcid): The name of the field or function evaluated in the source statement. An example of a source identifier is `document.location`.
- Sink identifier (srcid): The name of the field or function evaluated in the sink statement (*e.g.*, `window.open(...)`).
- Source line number (srcline): The line number of the source statement.
- Sink line number (srcline): The line number of the sink statement.
- Source URL (srcurl): The URL of the JavaScript function containing the source statement.
- Sink URL (sinkurl): The URL of the JavaScript function containing the sink statement.
- External objects (extobjs): Flags indicating whether the witness is performing `mailto` or `embed` functionality (*e.g.*, Flash).

Syntactic features are effective in uncovering patterns due to third-party libraries or usage of frameworks. They also assist in localizing noisy sources and sinks.

Indeed, our reason for recording source and sink line numbers as well as the URLs of the files enclosing the source and sink statements, which we exemplify in Section 2.3, is to capture instances where the flow either emanates from, or arrives at, third-party libraries. Another meaningful characterization of the flow is the context(s) it manipulates via `mailto` and `embed` statements. Certain attack vectors fail in such special contexts, which could be a source of false alarms.

Finally, for source and sink identifiers, the motivation is to detect "noisy" operations (*e.g.*, `document.location.url`, which can cause an open-redirect attack if assigned an untrusted value, but this happens very rarely [27]).

3.2 Quantitative Features

A second category of features are those that record quantitative measures of the witness. These are not to be confused with numerical yet nonquantitative features like line numbers. The latter are meaningful only inasmuch as equality checking is concerned, whereas quantitative measures can meaningfully be subjected to less-/greater-than comparisons. We define the following five quantitative features:

- Total results on (results): The overall number of findings reported on the file containing the sink statement.
- Number of steps (steps): The number of flow milestones comprising the witness path.
- Time (time): The total time spent by the analysis on the scope containing the witness.
- Number of path conditions (conditions): The number of branching statements (either loops or conditions) along the witness path.
- Number of functions (functions): The number of functions enclosing statements along the witness path.

Intuitively, all five of these features satisfy that the greater their value is, the less likely it is for the witness to be correct. First, for overall number of results, it is unlikely (albeit not impossible) for a single file to contain a large number of vulnerabilities. A more likely hypothesis is that the functions in the file are complicated, leading to their conservative and thus imprecise analysis. The time feature captures a similar pathology. Often imprecision leads the analysis to explore dead code or infeasible execution paths, which in turn lead to further imprecision, and so on. Thus, if the analysis has spent a long time on a given scope, then it is likely a symptom indicating that the model it has created is overly conservative. Finally, for methods, steps and path conditions, the higher these counts are, the more the analysis is exposed to errors due to approximations (*e.g.*, infeasible branching, incorrect resolution of call sites, etc).

3.3 Security-specific Features

The third and final category relates to features that are intrinsic to the security domain. We have identified two such features:

- Rule name (rname): The name of the violated security rule.
- Severity (severity): The severity of the violation as determined by the analysis tool.

There are various other security properties that are reported by industrial security checkers, but these change across tools, and so for generality we avoided from including them.

Similarly to the source and sink identifiers, the involved security rule may prove "noisy." This could be either because (i) the rule is perceived as less relevant by the user or (ii) the sources and sinks defined by the rules are noisy or (iii) there are defense measures (like framework-level sanitizers) that suppress the given type of vulnerabilities, which the analysis is not aware of. The severity of a finding also hints toward its correctness. In specific, low-severity witnesses are less likely to be accepted by the user as actionable.

3.4 Discussion

Our division of the features into three categories clarifies where and how our approach generalizes to other domains:

- Syntactic features are of wide applicability. Our specific choice of features, which mostly refer to sources and sinks, assumes a data-flow analysis. Defining other syntactic features, both general and specific to other forms of analysis, is straightforward.
- For quantitative features, some of the features we selected are strictly general. Measuring quantities like time and number of results is useful across virtually all forms of analysis, including even AST-level bug-finding tools like FindBugs.[4] Other quantitative properties, like the number of flow steps, assume a data-flow analysis whose results are in the form of traces through the code.
- Last, the security-specific features naturally carry values that are inherent to our domain. However, the features themselves — rule name and severity — are general, and likely remain useful in other domains.

To summarize, then, many of the features that we have defined are of general usefulness. Most of the features remain relevant under other forms of data-flow analysis (e.g., data-race detection or type-state verification), and some, like time and number of results, even apply to a wider range of analyses.

As for generalization to other languages, beyond the boundaries of JavaScript, a pleasing property of our approach is that it largely treats the analysis as a black box, instead focusing on observable outputs (warnings, running time, etc). This also renders any dependence on the underlying programming language minimal. While there are certain specific properties that relate to language libraries, extobjs being an example, ALETHEIA can be adapted to support the same style of security analysis in other languages with modest effort.

4. LEARNING ALGORITHMS

Our approach to static security analysis reduces the problem to binary classification, either in the online or offline setting. For a set of reports, we have feature data (given in Section 3) along with user-generated labels. Binary classification is a classic problem in machine learning, and a variety of methods from the field can be applied (for a survey, see [30, 3, 9]). In this section, we give a brief overview of methods we use to evaluate the approach in Section 5, and discuss their suitability for static security analysis. We discuss four categories of methods: functional, clustering, tree/rule based, and Bayesian. Methods in different classes can be combined, and some state of the art methods borrow ideas from several categories. For example, the NBTree method builds a decision tree with Naive Bayes' classifiers at the leaves. We provide a general overview, rather than delving too deeply into the details of the methods we compared. We close with a comparative discussion of the methods, identifying potential advantages of some categories, and presenting strategies for comparison between methods useful from the user's perspective.

4.1 Functional Methods

Functional methods include logistic regression (see, *e.g.*, [3]), linear support vector machines (see, *e.g.* [20, 24]), and generalizations, such as neural nets and kernel SVMs (see, *e.g.*, [9]). For convenience, we refer to these models as *functional* classification. These methods classify by learning a boundary either in feature space, or in a derived space related to features space by particular mappings. Once trained, the model can be used for prediction by noting where in the decision space an incoming feature would fall.

For example, SVMs search for a hyperplane in feature space that best separates labeled data (according to a maximum margin condition). Given a set of features x_i with labels $y_i \in \{-1, 1\}$, SVMs solve for a hyperplane w that solves a strictly convex optimization problem:

$$min_{w,\gamma} \frac{1}{2}\|w\|^2 + \lambda \sum_i max(0, 1 - y_i(x_i^T w - \gamma)) \qquad (1)$$

By inspection, the method weights a regularization term, $\|w\|^2$, against a robust classification measure, which is 0 for example i when the predicted label $x_i^T w - \gamma$ has the correct sign, and grows linearly if the sign is incorrect. The problem is strictly convex, so always guaranteed to have a unique solution, which is easily found using iterative methods.

One weakness of SVMs, and other other linear methods (*e.g.*, logistic regression) is the richness of the model space — there are

[4] findbugs.sourceforge.net/

limits to how well a linear classifier can perform. To address this issue, kernel SVMs neural net models train multiple layers of derived features from the data, while kernel SVMs make use of a kernel function Φ that maps the features to a different space, yet allows simple evaluation of inner products $\Phi(x_i)^T\Phi(x_j)$. Kernels and their applications have a rich literature, see, *e.g.*, [1, 23].

4.2 Instance-based Classification

Instance based learning requires a distance function to measure the distance of an incoming measurement to existing instances. Given such a function, one can simply find the nearest labeled instance to an unknown datapoint, and use that label to predict the class of the input.

For example, the Kstar algorithm uses a generalized distance function that models the distance between two instances through a series of possible transformations. Considering the entire set of possible transformations gives a probabilistic measure that takes into account influence from several labeled points [4].

4.3 Tree- and Rule-based Methods

Tree- and rule-based methods are divide and conquer algorithms that try to efficiently partition data instances according to labels.

For example, decision trees partition data into branches according to attribute values and labels. In order to build the tree, a feature attribute must be chosen at every branch point. These choices are made in a way that maximizes the so-called 'information gain', which is a data dependent measure that can be quickly computed. Decision trees work top-down, finding an attribute to split on at each point, and continuing recursively into the branches [21].

Rule-based methods attempt to find covering rules that describe each class, excluding others. For discrete attributes, rules can include membership in subsets of domain values (*e.g.*, use of frameworks is binary and can be used in a split or a rule). For continuous or ordinal variables, rules can include thresholds (*e.g.*, time taken to complete analysis larger than a set value). For example, the 1R classifier generates a one-level decision tree, with each rule in the set testing only one particular attribute [17].

4.4 Bayesian Methods

Bayesian methods include Naive Bayes, and Bayesian Networks, and directly model the probability of class events as a function of feature attributes. Naive Bayes assumes independence of attributes, and uses Bayes' rule to compute the probability of class given feature as:

$$P(C = c \mid X = x) = \frac{P(X = x \mid C = c)P(C = c)}{P(X = x)},$$

where the probabilities on the right hand side are learned from the data [18]. It is important to note that Naive Bayes also assumes quantitative features have a Gaussian distribution, but there are generalizations that relax this assumption. More importantly, independence of attributes can be relaxed as well, and more sophisticated methods such as Bayesian Networks are able to learn covariance structure from the data, *e.g.*, by maximum likelihood estimation [10].

4.5 Discussion

In the context of static security analysis, geometric methods such as support vector machines and logistic regression models (which fit a hyperplane through feature space) and instance-based learning that rely on Euclidean distance between features have disadvantages, since they rely on the usefulness of the numerical *quantities* present in feature space (or related space). This assumption may not hold for applications such as static security analysis, where the numerical values of the features may not have directly interpretable meaning, and methods from non-geometric categories may be a better choice. Our numeric experiments support this hypothesis.

Consider the list of features in Section 3. Two reports with significantly different source line numbers may not be significantly different (whether they are or not depends on the code). While examples that took similar time to complete may be similar, examples that took different but long times to complete may be similar as well. These features of the data may limit the performance of geometric classifiers, and so we may expect that tree and rule-based methods may outperform functional learning. The performance of instance-based learning is harder to predict, since a a generalized notion of distance may not directly rely on numerical quantities.

4.6 Performance Measurement

The performance of classification methods can be understood through a range of statistics, such as precision, recall, and accuracy. Every method may have a whole range of values associated to it. For example, precision is the ratio of positives correctly identified to all the instances labeled positive, while recall is the proportion of positive instances correctly identified. There is a natural tradeoff between recall and false positives. If one tries to catch a higher portion of positive instances, one will necessarily also mislabel a greater portion of negative ones. Thus, increasing recall leads to decreased precision.

This suggests two ways of comparing binary classification algorithms. One way is to compute an aggregate measure of quality across a range of possible policies. This is typically done using receiver operator characteristic (ROC) curves, see, *e.g.*, [19]. ROC curves plot true positive rate (recall) as a function of false positive rate for each classifier. The area under and ROC curve, called AUC, serves as an aggregate measure of quality; the higher the AUC, the better the classifier (overall). An AUC of 1 means that the classifier can perfectly separate the two classes; an AUC of 0.5 means that the classifier is essentially no better than a random guess. Two classifiers with the same AUC may perform differently in different regions of the 2D space defined by true positives and false positives.

A second way is to prescribe a policy choice, for example, to require high recall. This is appropriate for static security analysis, since we want to miss as few real issues as possible. Then, for high recall, one can compare classifiers using their false positive rates (the lower the better), or similar measures such as precision (the higher the better). Different classifiers may be superior for different policy choices.

5. IMPLEMENTATION AND EVALUATION

In this section, we describe our prototype implementation of the ALETHEIA system. We then present experiments that we have conducted to evaluate the efficacy of ALETHEIA and the viability of its underlying usage scenario.

5.1 Prototype Implementation

ALETHEIA is implemented as a Java library. Its public interface consists of a main operation that accepts as input (i) a set of security warnings, (ii) a partial mapping from warnings to the label assigned to them by the user (true vs false) and (iii) a filtering policy (specifying a precision/recall tradeoff). To compute a highest-ranking classifier per the policy, ALETHEIA partitions the mapped warnings into training and testing subsets. (By default, these are obtained by splitting the set in half at random.) Each of the candidate classification algorithms is then trained on the training subset. A rank is assigned to the resulting classifier based on the test instances. The

output is the highest-ranking classifier. It accepts a warning as its argument and returns a boolean value. The lifetime of the filter is decided by the client analysis tool, which may compute the filter from scratch across analysis tasks, different applications, etc.

The current ALETHEIA prototype is built atop version 3.6.10 of the Weka data-mining library [30].[5] Weka provides a rich set of machine-learning algorithms, which are all implemented in Java and expose a consistent interface. ALETHEIA utilizes both the classification algorithms provided by Weka and the statistical analysis tools built into it. In particular, ALETHEIA makes use of the ZeroR and OneR rule-based classifiers; the NaiveBayes and BayesNetwork Bayesian classifiers; the K-Star lazy classification algorithm; the NB-Tree and J48 tree-based classifiers; and the SVM functional classification method.

For the filtering policy, ALETHEIA lets the user express the tradeoff between *precision* and *recall*, which are calculated as follows:

$$p = \frac{tp}{tp + fp} \quad (precision) \tag{2}$$

$$r = \frac{tp}{tp + fn} \quad (recall) \tag{3}$$

where *tp*, *fp*, and *fn* indicate the number of true positives, false positives, and false negatives reported by the analysis, respectively. The balance between precision and recall is achieved via a normalized weight w. This leads to the following formula:

$$w \times r + (1 - w) \times p \tag{4}$$

where the current ALETHEIA implementation fixes the discrete values $w \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$ to simplify user interaction.

5.2 Experimental Setup

To evaluate ALETHEIA, we collected an extensive set of 3,758 security warnings output by a commercial JavaScript security checker when applied to 1,706 HTML pages taken from 675 top-popular Web sites. These Web sites include all Fortune 500 companies, the top 100 Web sites,[6] as well as several handpicked sites of IT and security vendors. Overall, the source files are highly diversified, reflecting a variety of software categories and coding idioms. All 3,758 of the reported warnings were carefully reviewed by a professional ethical hacker, who determined for each of the issues whether or not it is an actual vulnerability.

Based on this set of classified findings, we have created an experimental harness that performs the following series of actions for a given policy w and training-set size n:

- A sample of n warnings is obtained at random out of the 3,758 classified findings.
- Next, ALETHEIA is applied: Each of the candidate classifiers (ZeroR, OneR, etc) is trained on the $\frac{n}{2}$ training-set instances, and then tested on the remaining $\frac{n}{2}$ test instances.
- The filter output by ALETHEIA is then applied to the remaining $3,758 - n$ warnings. Since the entire set has already been classified in advance, we can compute precision and recall per w and n.

This experimental setup provides an accurate and measurable simulation of user interaction with ALETHEIA. In reality, n and w are determined by the user.

We report in Table 1 (under Appendix A) on the results we obtained across the five policies built into ALETHEIA as well as $100 \leq n \leq 900$. In the interest of robust methodology and reproducible results, we have executed the harness in each of the w/n configurations a total of 15 times, where the reported data is the average across all runs.

5.3 Experimental Hypotheses

Before we present the raw experimental data output by our testbed, we lay out the experimental hypotheses guiding our research.

H1: Feasibility.

Our first research hypothesis asserts the overall viability of our approach. Concretely, we consider manual classification of up to 200 warnings as *tolerable* and a filter that achieves at least 95% accuracy as *effective*. This leasds to the following statement of viability:

> H1: Based on tolerable user effort, it is possible — for a given precision/recall policy — to filter the remaining warnings effectively with respect to the specified policy.

H2: Learning Framework.

The second hypothesis concerns the need for an extensive and diversified set of classifiers. The claim is that there isn't a "one size fits all" filter that could replace the learning approach embodied in ALETHEIA. A space of candidate classifiers needs to be searched for an optimal filter per the reported warnings and policy at hand:

> H2: None of the classification algorithms in our suite satisfies that across all the different filtering policies (i.e., precision/recall tradeoffs), it achieves a score that is at most 10% below the best classifier for the given policy.

That is, the choice of which classifier to use is context sensitive, and so a general filtering strategy would not be appropriate.

H3: Diminishing Returns.

The third and final hypothesis with which we went into the experiments is that filter quality, as a function of user burden, gradually plateaus. If so, then this is reason for encouragement, because the filter computed in return to tolerable user effort is a nearly optimal one given the diminishing return thanks to classification of additional warnings:

> H3: Improvement in filter quality, measured in terms of policy score, diminishes as the user contributes increasingly more warning classifications.

The hope, beyond merely confirming H3, is for the plateauing trend to arise already at a tolerable threshold of 200 warnings per H1.

5.4 Experimental Results

The results of the experiments are listed in numeric form in Table 1 (under Appendix A). To make the trends easier to appreciate, we present the results pertaining to data points of interest — 100 and 200 user classifications, which we perceive as reasonable manual effort — in visual form in Figures 3 through 10 and 13.

In Figures 3 and 4, we visualize the scores of different classifiers as a continuous function of the policy — which is defined as the linear expression $w \times r + (1 - w) \times p$ — for 100 and 200 classified instances respectively. Recall that r denotes recall and p denotes precision. The policy is, therefore, a linear function of w (which determines the relative weight of precision vs recall), and hence

[5] http://www.cs.waikato.ac.nz/ml/weka/

[6] http://www.web100.com/category/web-100/

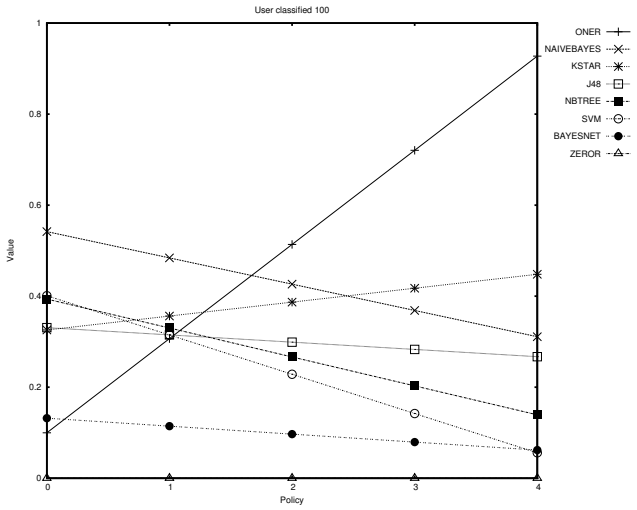

Figure 3: Scores Achieved by the Different Classifiers As a Function of the Policy Given 100 Classified Warnings

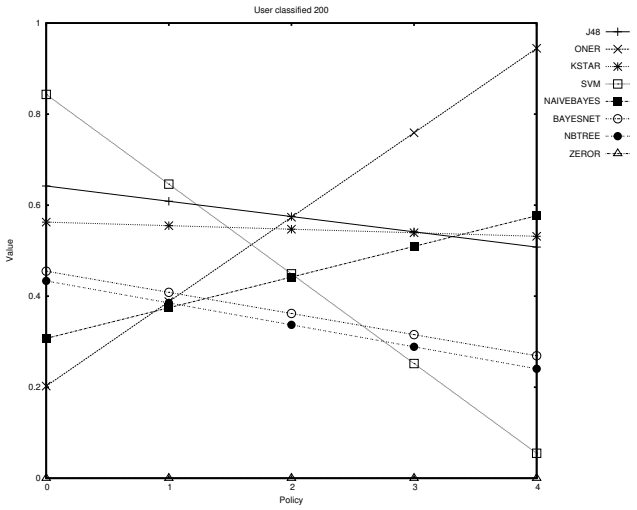

Figure 4: Scores Achieved by the Different Classifiers As a Function of the Policy Given 200 Classified Warnings

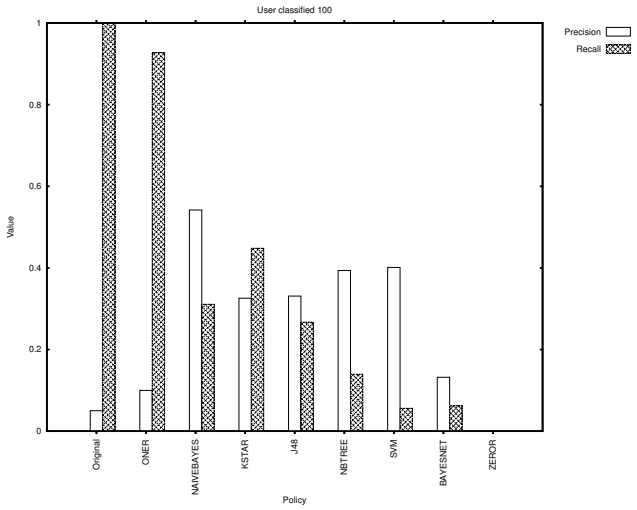

Figure 5: Precision and Recall for the Different Classifiers Given 100 Classified Warnings

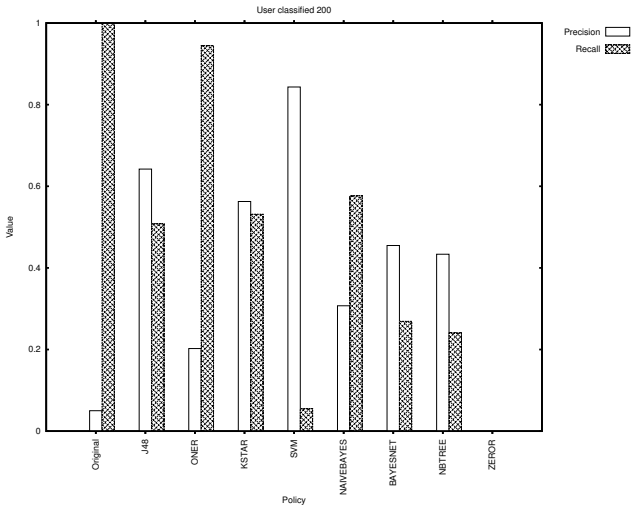

Figure 6: Precision and Recall for the Different Classifiers Given 200 Classified Warnings

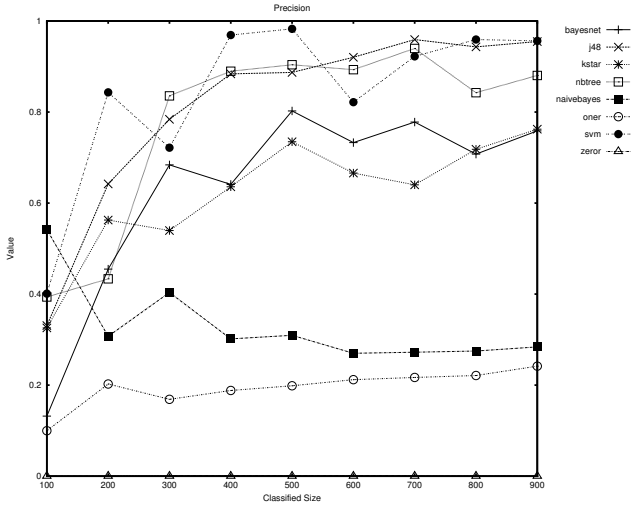

Figure 7: Precision As a Function of Classified-set Size

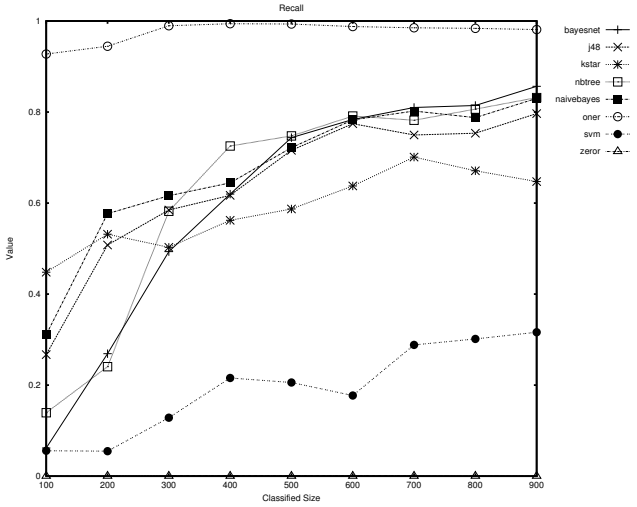

Figure 8: Recall As a Function of Classified-set Size

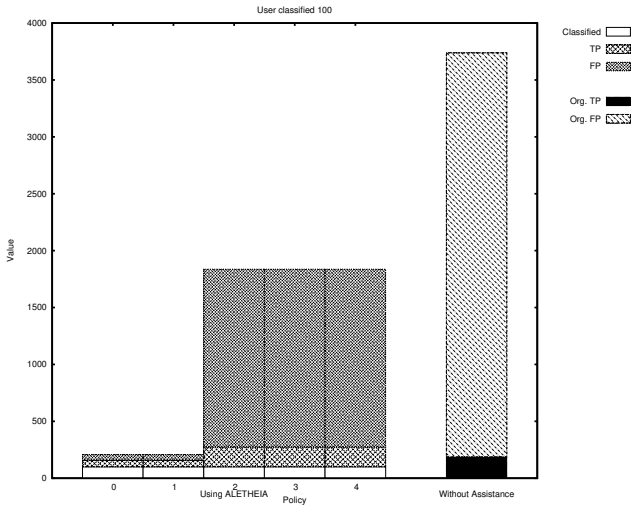

Figure 9: Number of Findings the User Has to Review with ALETHEIA (by Policy: 1-4) and without ALETHEIA Given 100 Initial Classifications

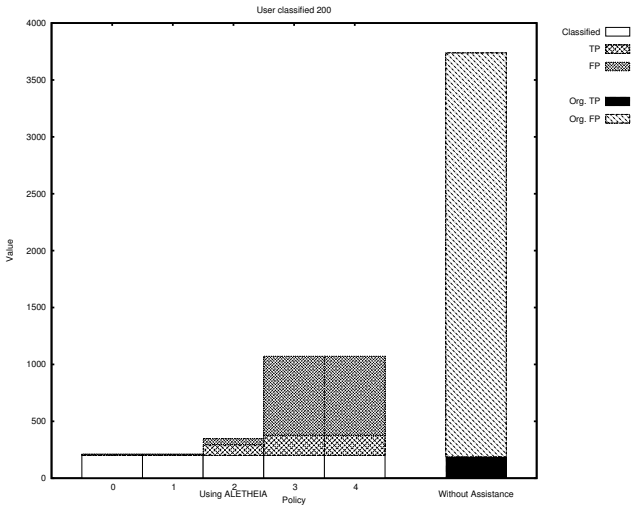

Figure 10: Number of Findings the User Has to Review with ALETHEIA (by Policy: 1-4) and without ALETHEIA Given 200 Initial Classifications

the straight trend lines in 3 and 4 (and in general). The horizontal labels, $i \in \{0, \ldots, 4\}$, are the discrete policy points $\frac{i}{4}$, as discussed in Section 5.1.

The bar graphs in Figures 5 and 6 depict precision and recall information for the different classifiers, again given 100 and 200 classified instances respectively. To clarify, these were calculated for the remaining $3,758 - n$ warnings (i.e., all warnings excluding those classified by the user).

Notice that the ZeroR classifier has both precision and recall values of 0. This results from the simplicity of this classifier, which merely predicts the majority category. Since true warnings constitute only 5% of all the reports, ZeroR predicts that all warnings are false, the result being 0 precision and recall values. Also note that the original algorithm (without filtering) has perfect recall. Thus, the relative improvement in precision and degradation in recall due to different classifiers can be quantified by comparison with the respective "Original" bars.

The graphs in Figures 7 and 8 plot changes in precision and recall, respectively, as a function of the classified-set size. Intuitively, classifier quality improves as the number of classified instances increases. The graphs in Figures 9 and 10 depict user effort, measured as number of manually classified issues, under different ALETHEIA configurations as well as without ALETHEIA. Last, Figure 13 displays the trend line for policy score for the five discrete policy points above a function of classified-set size.

In the following, we evaluate our experimental hypotheses.

Evaluation of H1.

First, for H1, we begin by reviewing Figure 5 that corresponds to a relatively modest effort by the user of classifying only 100 out of the full set of 3,758 warnings reported by the security checker. Driven by this data alone, ALETHEIA achieves precision improvements ranging from 417% (OneR) all the way up to 1274% (NaiveBayes). If the user prefers to preserve true alarms, then ALETHEIA is able to comply with this requirement with 76% recall alongside $\times 4.17$ improvement in accuracy (OneR).

We now switch to Figure 6, which reflects data for 200 classified instances (still a tolerable amount of work by the user in our view and experience). Here we observe across all policies significant reduction in false alarms, ranging between $\times 3.4$ (OneR) and $\times 18.4$ (SVM) according to the bias induced by the policy. In particular, if the user has a strong bias toward preserving true warnings, then ALETHEIA satisfies this bias with recall as high as 97% while simultaneously improving precision by 340% (OneR), which is a very dramatic improvement in user experience. Similar trends can be gleaned from review of the additional data in Table 1 (for 300-900 classified instances).

Viewed from the perspective of user effort, ALETHEIA enables visible saving thanks to filtering out false alarms, as we depict in Figures 9 and 10. In these figures, we quantify developer effort in "warning units", counting how many reports the user has to review under the different policies after the filtering applying by ALETHEIA given classified-set sizes of 100 and 200, respectively. The reference, appearing as the rightmost column, is the 3,758 warnings reported by default. We visualize within each column the proportion of true vs false positives. We also visualize the proportion of reports classified by the user where appropriate.

The difference in height from the reference column, which captures the gap in user effort, is significant. This is true even of the more conservative policies given only 100 user-classified instances (Figure 9), but moving to 200 user-classified instances has a much more dramatic effect (Figure 10): For a conservative user, approximately 75% of the initial workload is eliminated. For a less conservative user, the saving becomes about 90% with very few issues to review beyond the initial 200 classifications.

The conclusion we derive from the data is the H1 is confirmed. That is, given tolerable classification effort by the user (up to 200 warnings), ALETHEIA is able to significantly decrease the rate of false warnings while also achieving high recall rates if a bias for preservation of true positives is expressed. As a consequence, there is substantial reduction in user effort.

As a natural extension, given a rich dataset like ours, an offline learning setting — whereby per-policy universal classifiers are computed once for all users and software systems — could completely eliminate the end user's classification effort. Though seemingly attractive, this solution is complicated by the fact that static security checkers are often configured by the user (e.g., by modifying the security rules, severity levels, analysis bounds, etc).

Such customizations bear direct impact on the resulting classifier, and so offline learning is less adequate.

In our evaluation, this was not a concern since we retained the same (default) tool configuration across all benchmarks. Indeed, if a given configuration is used across an entire organization or development team, then classifiers can be computed and shared based on the effort of a single user (or by distributing the 200 classifications across multiple users).

Evaluation of H2.

Moving to H2, we refer to Figures 3 and 4. These indicate the trend followed by each of the classifier across the continuum of policies for $w \in [0, 1]$. For the discrete data points corresponding to $w \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$, we provide in the second column of Table 1, titled "Model", the classifier achieving the highest score.

Concentrating on Figures 3 and 4, we derive the clear observation that none of the classifiers dominates, or is even close to dominating, the other classifiers. This would have corresponded to that hypothetical classifier's graph being above (or near to) all the other graphs. In reality, all the graphs intersect with each other, mostly far from the boundaries, which indicates that different policies require different classifiers and there is no "one size fits all" candidate. The trend lines for other classified-set sizes, which we omit for lack of space, echo the same pattern.

The data in Table 1 across the entire range of classified-set sizes is also consistent with this analysis. Across all policies and sections, five out of the eight algorithms plugged into ALETHEIA come out as best at least once. Somewhat unsurprisingly, OneR — which generates a one-level decision tree, thus being a simple rule-based classifier — fits well with the "extreme" policy that biases toward preservation of true positives. At the other extreme, there is more variance with J48 and SVM being the most effective candidates overall. The asymmetry between the two extremes ($w = 0$ and $w = 1$) is due to the disproportion between true and false alarms in the dataset (which is an accurate representation of warnings in the wild).

The in-between policies are enforced effectively by the tree-based classifiers. Our conjecture is that this follows from the non-geometric interpretation of several important witness features, such as the lexical location and line number of sources and sinks. SVM and other functional classifiers are confused by such features, and rule-based techniques are often too naive to capture the complexity of the decision process. (See also Section 4.5.) To illustrate this, we refer the reader to Figures 11 and 12. The former depicts an NB-Tree instance learned by ALETHEIA, whereas the latter presents a representative fragment of a OneR model. The difference between the two in granularity of judgment is obvious, favoring tree-based classifiers for weakly biased policies.

Our conclusion, in summary, is that the decision of which classification algorithm to apply to a given filtering scenario is context sensitive, depending both on the policy expressed by the user and on the warnings at hand. This confirms H2, motivating why simpler filtering strategies are unlikely to achieve comparable performance results.

Evaluation of H3.

Finally, to evaluate H3 we examine Figures 7 and 8. These represent data for up to 900 classified instances, which is well beyond reasonable user effort in our opinion, the only purpose being to validate whether there is indeed noticeable gain in classifying more instances than the user can reasonably be asked to classify.

The graphs — for both precision and recall — provide a clear negative answer: Roughly speaking, up to 200 classified instances

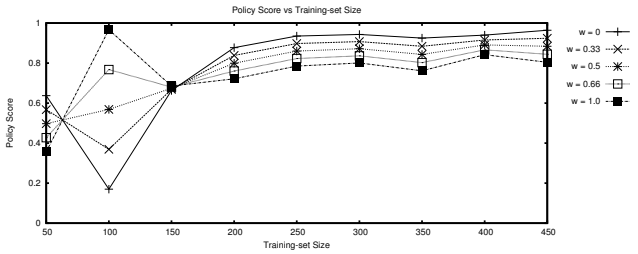

Figure 13: Policy Score as a Function of the Training-set Size, where Policies Are Represented as Their Respective w Value

there is sharp improvement in classifier quality, where the improvement trend drops sharply in the range of 300 to 900 user classifications. While there is still gain from additional classified instances in some of the cases, returns are clearly diminishing.

As another source of confirmation, we examine vertical slices of Table 1. The answer, according to the data, is strictly negative. As one data point, for $w = 1$ recall remains within the range of $0.97 - 0.99$ and precision is restricted to the range $3.4 - 4.81$ all the way between 200 and 900 classified instances. The trend of improvement is sublinear at best. Indeed, 200 is figuring as the sweet spot, which also reaffirms H1.

Finally, we refer the reader to the graph in Figure 13. This graph visualizes the highest score achieved per each of the policies as a function of sample size. Somewhat unintuitively, the trend is not monotonic, let alone linear. A key reason for this, in our analysis, is that the data is unbalanced (many more false warnings compared to true alarms). Hence, a naive classifier can perform well by simply classifying all of the alarms as false. Still, aside from the nonlinear trends for small samples, all of the graphs in Figure 13 are visibly plateauing beyond the cutoff value of 200. This again suggests that returns (i.e., filter quality) fast diminish thanks to enlarging the sample size.

6. RELATED WORK

The challenge of eliminating as many false positives as possible from the report of a static analysis, without introducing an excessive number of false negatives, has been the subject of numerous research studies.

Junker, *et al.* [12] cast static-analysis clients as syntactic model-checking problems. Violations of the verification property result in a counterexample (or path), which is checked for feasibility using an SMT solver. If the path is infeasible, then the causes for its infeasibility are extracted via path slicing and an observer automaton is constructed to exclude all paths sharing the same cause. Junker, *et al.* have integrated their approach into the Goanna tool for static analysis of C/C++ programs. In another study [6], Fehnker, *et al.* share their experience in applying Goanna to large software systems consisting of 10^6 and more lines of code, including the Firefox browser. They report on an excessive number of false alarms, and address this usability issue by refining the security rules of Goanna. In contrast with many other analyses, the Goanna rules are phrased as automata, which are translated into a Kripke structure for model checking. Ulike ALETHEIA, these works are strictly concerned with reducing the number of false positives, without giving the user of the analysis the flexibility to express a preference that reflects a bias over true versus false positives.

Muske, *et al.* [15] tackle the problem of false warnings via a partitioning approach. They assume two partitioning phases: The first

```
time <= 5042
|   ruleName = jsCrossSiteRequestForgery : NB 2
|   ruleName = jsDOMXSSandOpenRedirect
|   |   time <= 1862.5: NB 4
|   |   time > 1862.5
|   |   |   time <= 4252 ? NB 6 : NB 7
|   ruleName = jsDOMCrossSiteScripting
|   |   time <= 1216.5 ? NB 9 : NB10
|   ruleName = MethodOverwrite: NB 11
|   ruleName = jsOpenRedirect: NB 12
|   ruleName = jsCodeInjection : NB 13
|   ruleName = jsPortManipulation : NB 14
|   ruleName = jsProtocolManipulation : NB 15
time > 5042
|   numSteps <= 14.5
|   |   sinkLineNo <= 89.5
|   |   |   sourceLineNo <= 27.5: NB 19
|   |   |   sourceLineNo > 27.5
|   |   |   |   numSteps <= 8.5
|   |   |   |   |   time <= 27839.5 ? NB 22 : NB 23
|   |   |   |   numSteps > 8.5: NB 24
|   |   sinkLineNo > 89.5: NB 25
|   numSteps > 14.5
|   |   time <= 393223 ? NB 27 : NB 28
```

Figure 11: An NB-Tree Instance Computed by ALETHEIA as a Candidate Filter (in Textual Weka Format)

```
sinkURL:
        file :/.../52311/152673. js        -> NO
        file :/.../3170/646. js    -> NO
        file :/.../1065/183. js    -> YES
        file :/.../644/51458. js   -> YES
        file :/.../2751/183. js    -> YES
        file :/.../4978/4978. html         -> YES
        file :/.../449/449. html   -> YES
        file :/.../35320/35320. html       -> NO
        file :/.../7967/7967. html         -> NO
        file :/.../402/1501. js    -> YES
        file :/.../1451995/1451995. html   -> NO
        ...
```

Figure 12: Fragment of a OneR Instance Computed by ALETHEIA as a Candidate Filter (in Textual Weka Format)

divides the warnings into equivalence classes, assigning a *leader* to each class, such that if the leader is a false warning, then all other warnings are also false. The second step is to partition the leader warnings. This is done based on the variables modified along the path and their modification points. The entire process is meant to facilitate manual review of the warnings generated by a static analysis tool. The authors report on 50%-60% reduction in review effort corresponding to 60% redundant warnings on average, which are more readily eliminated thanks to their technique. Just like for ALETHEIA, the purpose of this work is to reduce the burden on the analyst reviewing the results of a static analysis. The difference is that this work does that by partitioning the analysis results, in such a way that results with the same characteristics are grouped in the same equivalence class, whereas ALETHEIA is based on an algorithm that learns the characteristics of false positives and prevents from presenting to the analyst flows that are highly likely to lead to other false positives.

An orthogonal approach that Muske, *et al.* propose [16] is to first apply a scalable yet imprecise abstract interpretation, and then remove false warnings using a bounded model-checking technique, which features the opposite tradeoff. The main goal is the elimination of false reports. This approach consists of two distinct techniques to identify equivalence between assertions, thereby obviating the need to verify all but one of the assertions, and a third technique for skipping verification of assertions that do not influence any of the false alarms. However, this work does not allow the analyst to choose whether the filter applied to the result of an analysis should err more towards the false-positive or true-positive side.

Johnson, *et al.* report on interviews with developers to learn the reasons why static analysis tools are underused [11]. They conclude that the two main barriers to adoption of such tools are (i) false positives and (ii) the way in which warnings are presented to the developer. These are related: if the warning is hard to understand, then false alarms become harder to identify and eliminate.

EFindBugs, developed by Shen, *et al.* [25], is an improved version of the popular FindBugs tool [2] that addresses the excess of false positives commonly reported by FindBugs. This is achieved via a two-staged error ranking strategy. First, EFindBugs is applied to a sample program. The resulting warnings are classified manually, and an approximate *defect likelihood* is assigned to each bug category and bug kind. This determines the initial ranking of reported bugs. Defect likelihood is later tuned in a self-adaptively manner when EFindBugs is run on the user's application thanks to users' feedback on reported warnings. This optimization process is executed automatically and based on the correlations among error reports with the same bug pattern. Unlike ALETHEIA, this approach does not offer the analyst the ability to finely tune the results according to the specific needs of the analyst.

7. CONCLUSION AND FUTURE WORK

We have taken a step toward bridging the usability gap separating between developers and automated security checkers based on static program analysis. The strongest detractor for developers is the excess of false warnings and the difficulty to understand the findings reported by the analysis tool (often because they are spurious and represent infeasible program behaviors). Thus, in aiming for completeness, static security checkers end up discouraging the user to the point that the tool is not used at all even at the cost of missing actual vulnerabilities that the tool is capable of finding.

In our approach, the report from ALETHEIA is cleansed by combining user interaction with statistical learning techniques. The user specifies a policy, or preference, that reflects a bias over true versus false warnings. The user also contributes classifications for a small fragment of the overall warnings. These trigger the creation and evaluation of multiple candidate filters, which are each evaluated according to the user-provided policy. The best filter is then applied to the remaining warnings, leaving the user to review only a (small) subset of the raw warnings output by the tool. Experiments that we have performed over ALETHEIA are highly encouraging. For example, given only 200 classified warnings, if the user expresses a preference toward preservation of true positives, ALETHEIA is able to improve precision by a factor of 2.868 while reducing recall by a negligible factor ($\times$1.006). Conversely, if the user favors elimination of false alarms, still based on only 200 classified warnings, ALETHEIA achieves a recall-degradation factor of only 2.212 and a precision-improvement factor of 9.014. Other

policies are enforced in an equality effective manner. In all cases, and across all policies, ALETHEIA is able to improve precision by a factor ranging between $\times 2.868$ and $\times 16.556$ in return to user classification of only 200 warnings.

In the future, we intend to extend ALETHEIA to consider not only different classification algorithms but also different instantiations of the same algorithm. This should fine tune the resulting filter, making it more effective. The challenge is to efficiently search through the unbounded space of parameter configurations for each of the candidate algorithms. We hope to address this challenge by applying local search strategies guided by the shape of the AUC curve computed for each of the algorithms. Another enhancement that we plan to investigate is online refinement: Instead of asking the user to eagerly tag several hundred warnings, the goal is to apply filtering incrementally, on the fly, as the user classifies warnings. This offers immediate benefit from every user labeling. An important research question that the online setting raises is how to order the warnings presented to the user for effective online filtering and how sensitive the computation of the online filter is to the choice and order of warnings.

8. REFERENCES

[1] N. Aronszajn. Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68, 1950.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using findbugs on production software. In *OOPSLA Companion*, 2007.

[3] C. M. Bishop. *Pattern recognition and machine learning*, volume 1. Springer, 2006.

[4] J. G. Cleary, L. E. Trigg, et al. K^*: An instance-based learner using an entropic distance measure. In *ICML*, 1995.

[5] D. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7), 1977.

[6] A. Fehnker, R. Huuck, S. Seefried, and M. Tapp. Fade to grey: Tuning static program analysis. *ENTCS*, 266, 2010.

[7] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA*, 2011.

[8] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *WWW*, 2009.

[9] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[10] D. Heckerman, D. Geiger, and D. M. Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3), 1995.

[11] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, 2013.

[12] M. Junker, R. Huuck, A. Fehnker, and A. Knapp. Smt-based false positive elimination in static program analysis. In *ICFEM*, 2012.

[13] B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security*, 2005.

[14] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, 2008.

[15] T. B. Muske, A. Baid, and T. Sanas. Review efforts reduction by partitioning of static analysis warnings. In *SCAM*, 2013.

[16] T. B. Muske, A. Datar, M. Khanzode, and K. Madhukar. Efficient elimination of false positives using bounded model checking. In *VALID*, 2013.

[17] C. G. Nevill-Manning, G. Holmes, and I. H. Witten. The development of holte's 1r classifier. In *Artificial Neural Networks and Expert Systems, 1995. Proceedings., Second New Zealand International Two-Stream Conference on*. IEEE, 1995.

[18] A. Y. Ng and M. I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in neural information processing systems*, 2, 2002.

[19] M. S. Pepe. *The statistical evaluation of medical tests for classification and prediction*. Oxford University Press, 2003.

[20] M. Pontil and A. Verri. Properties of support vector machines. *Neural Computation*, 10, 1998.

[21] J. R. Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *J-SAC*, 21(1), 2006.

[23] S. Saitoh. *Theory of reproducing kernels and its applications*. Longman, 1988.

[24] B. Schölkopf, A. J. Smola, R. C. Williamson, and P. L. Bartlett. New support vector algorithms. *Neural Computation*, 12, 2000.

[25] H. Shen, J. Fang, and J. Zhao. Efindbugs: Effective error ranking for findbugs. In *ICST*, 2011.

[26] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *TOSEM*, 22(4), 2013.

[27] O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *ISSTA*, 2014.

[28] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, 2013.

[29] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, 2009.

[30] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

APPENDIX

A. DETAILED EXPERIMENTAL RESULTS

The raw results of the experiments are summarized in Table 1. For readability, the table is partitioned into sections according to the number n of classified instances used by ALETHEIA to converge on the best filter. These correspond to user effort (in performing manual classification of warnings), and so the table is organized around this parameter. Precision and recall improvement are measured relative to those of the original security checker, without ALETHEIA, which has perfect recall (being that it is conservative) yet low precision of 5% on the set of 3,758 warnings.

In a given section of Table 1, each row reflects the statistics for a particular policy, governed by the value of w. The "Test Recall" and "Test Precision" columns provide recall and precision statistics for the tested instances (forming half of the classified instances made available to ALETHEIA), whereas "Full Recall" and "Full Precision" reflect these measures for the remaining $3,758 - n$ warnings. We also indicate the highest-ranking algorithm per policy (under the "Model" column).

w (Policy: $w \times r + (1-w) \times p$)	Model	Test Recall	Test Precision	Full Recall	Full Precision
Size of Classified Set: 100					
0	NAIVEBAYES	0.31	10.83	0.36	12.74
1/4	NAIVEBAYES	0.31	10.83	0.36	12.74
1/2	ONER	0.93	2	0.76	4.17
3/4	ONER	0.93	2	0.76	4.17
1	ONER	0.93	2	0.76	4.17
Size of Classified Set: 200					
0	SVM	0.05	16.86	0.12	18.4
1/4	SVM	0.05	16.86	0.12	18.4
1/2	J48	0.51	12.84	0.42	9.44
3/4	ONER	0.94	4.05	0.97	3.4
1	ONER	0.94	4.05	0.97	3.4
Size of Classified Set: 300					
0	NBTREE	0.58	16.7	0.58	15.16
1/4	NBTREE	0.58	16.7	0.58	15.16
1/2	NBTREE	0.58	16.7	0.58	15.16
3/4	ONER	0.99	3.37	0.99	3.34
1	ONER	0.99	3.37	0.99	3.34
Size of Classified Set: 400					
0	J48	0.75	19.17	0.77	18.33
1/4	J48	0.75	19.17	0.77	18.33
1/2	NBTREE	0.78	18.78	0.76	18.49
3/4	NBTREE	0.78	18.78	0.76	18.49
1	ONER	0.99	4.33	0.98	4.36
Size of Classified Set: 500					
0	SVM	0.21	19.65	0.23	18.79
1/4	NBTREE	0.75	18.07	0.79	18.69
1/2	NBTREE	0.75	18.07	0.79	18.69
3/4	ONER	0.99	3.97	0.99	3.95
1	ONER	0.99	3.97	0.99	3.95
Size of Classified Set: 600					
0	J48	0.77	18.4	0.69	18.05
1/4	J48	0.77	18.4	0.69	18.05
1/2	J48	0.77	18.4	0.69	18.05
3/4	NBTREE	0.79	17.85	0.8	18.85
1	ONER	0.99	4.24	0.98	4.22
Size of Classified Set: 700					
0	J48	0.75	19.17	0.77	18.33
1/4	J48	0.75	19.17	0.77	18.33
1/2	NBTREE	0.78	18.78	0.76	18.49
3/4	NBTREE	0.78	18.78	0.76	18.49
1	ONER	0.99	4.33	0.98	4.36
Size of Classified Set: 800					
0	SVM	0.3	19.18	0.29	18.4
1/4	J48	0.75	18.85	0.84	18.79
1/2	J48	0.75	18.85	0.84	18.79
3/4	NBTREE	0.81	16.84	0.83	17.65
1	ONER	0.98	4.42	0.98	4.53
Size of Classified Set: 900					
0	SVM	0.32	19.12	0.34	19.05
1/4	J48	0.8	19.09	0.8	19.27
1/2	J48	0.8	19.09	0.8	19.27
3/4	NBTREE	0.83	17.6	0.77	18.09
1	ONER	0.98	4.83	0.98	4.81

Table 1: Per-policy Precision and Recall Statistics as a Function of Number of Classified Instances