Modeling Security Weaknesses to Enable Practical Run-time Defenses

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

William Melicher

B.S., Computer Engineering, University of Virginia
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

August, 2019

Abstract

Security weaknesses are sometimes caused by patterns in human behaviors. However, it can be difficult to identify such patterns in a practical, yet accurate way. In order to fix security weaknesses, it is crucial to identify them. Useful systems to identify security weaknesses must be accurate enough to guide users' decisions, but also be lightweight enough to produce results in a reasonable time frame. In this thesis, we show how machine-learning techniques allow us to detect security weaknesses that result from patterns in human behavior faster and more efficiently than current approaches, enabling new, practical run-time defenses. We present two applications to support this thesis.

First, we use neural networks to identify users' weak passwords and show how to make this approach practical for fully client-side password feedback. One problem with current password feedback is that users can get either quick but often incorrect feedback by using heuristics or accurate but slow feedback by simulating adversarial guessing. In contrast, we found that our approach to password guessing is both more accurate and more compact in implementation than previous ones, which enables us to more practically estimate resistance to password-guessing attacks in real time on client machines.

Second, we use deep learning models to identify client-side cross-site scripting vulnerabilities in JavaScript code. We collected JavaScript functions from hundreds of thousands of web pages and using a taint-tracking-enabled browser labeled them according to whether they were vulnerable to cross-site scripting. We trained deep neural networks to classify source code as safe or as potentially vulnerable. We demonstrate how our models can be used as a lightweight building block to selectively enable other defenses, e.g., taint tracking.

Acknowledgments

I am very grateful to the many people who have supported me during my PhD. I would particularly like to thank my advisor, Lujo Bauer, and my committee members: Limin Jia, Nicolas Christin, and Mihai Christodorescu, for their help and guidance. I am also thankful for the many other colleagues who I had the pleasure of working with at CMU.

I would like to thank the National Science Foundation, John and Claire Bertucci, PNC Center for Financial Services Innovation, Microsoft Research, Carnegie Mellon Cylab, NVIDIA, and Google for funding and other support. In particular, work in this thesis was supported in part by gifts from NVIDIA, Google, Microsoft Research, John and Claire Bertucci, and PNC Center for Financial Services Innovation; by Carnegie Mellon Cylab through a Cylab Presidential Fellowship; and by the National Science Foundation via grant CNS1704542.

Thesis Committee:

Prof. Lujo Bauer, Carnegie Mellon University, Chair

Prof. Nicolas Christin, Carnegie Mellon University

Prof. Limin Jia, Carnegie Mellon University

Dr. Mihai Christodorescu, Visa Research

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security weaknesses are sometimes caused by patterns in human behaviors, e.g., picking easily guessable passwords or making predictable mistakes while designing systems. One challenge with current security practices is that accurately identifying security weaknesses in real time is often impractical. For weakness identification to be useful for protecting users, it must be accurate, as well as lightweight enough to be run on a normal machine and give low-latency results. Machine-learning models hold promise for identifying security weaknesses and, at the same time, for being more lightweight than other methods. In this thesis, we study how to develop machine-learning models to detect security weaknesses that arise from patterns in human behavior. We focus on two instances of this problem: identifying weak passwords and identifying code injection vulnerabilities in JavaScript.

For our first application, we trained a neural network that can accurately guess passwords for the purpose of estimating password strength. Users often create predictable passwords based on patterns in language. Neural networks have been shown to be useful for modeling natural language, and guessing passwords is conceptually similar to natural language processing. We demonstrate that our neural network password model is more accurate than previous models and, in addition, an order of magnitude more compressible—making it possible to deliver a small model to client machines for practical strength estimation during

password creation. For our second application, we use neural networks to detect JavaScript code-injection vulnerabilities. In our prior work, we discovered a large number of JavaScript code-injection vulnerabilities on the Internet. After examining those vulnerabilities, we found their abstract syntax trees to be conceptually similar, perhaps due to patterns in programmers' behavior. Leveraging this observation, We use machine learning to detect code that contains injection vulnerabilities and show that our method is accurate and efficient enough to enable new run-time defenses.

Thesis statement

> *Machine learning techniques allow us to detect security weaknesses that result from patterns in human behavior faster and more efficiently than current approaches, enabling practical, new run-time defenses.*

1 Password strength estimation

Text passwords are currently the most common form of authentication, and they promise to continue to be so for the foreseeable future [62]. Unfortunately, users often choose predictable passwords, enabling password-guessing attacks. In response, proactive password checking is used to evaluate password strength [11].

A common way to evaluate the strength of a password is by running or simulating password-guessing techniques [33, 71, 158]. A suite of well-configured guessing techniques, encompassing both probabilistic approaches [37, 86, 159] and off-the-shelf password-recovery tools [112, 133], can accurately model the vulnerability of passwords to guessing by expert attackers [149]. Unfortunately, these techniques are often very computationally intensive, requiring hundreds of megabytes to gigabytes of disk space, and taking days to execute. Therefore, they are typically unsuitable for real-time evaluation of password strength, and sometimes for any practically useful evaluation of password strength.

To gauge the strength of human-chosen text passwords both more accurately and more practically, we propose using artificial neural networks to guess passwords. Artificial neural networks (hereafter referred to as "neural networks") are a machine-learning technique designed to approximate highly dimensional functions. They have been shown to be very effective at generating novel sequences similar to those humans create [55, 138], suggesting a natural fit for generating password guesses.

In this thesis, we first comprehensively test the impact of varying the neural network model size, model architecture, training data, and training technique on the network's ability to guess different types of passwords. We compare our implementation of neural networks to state-of-the-art password-guessing models, including widely studied Markov models [86] and probabilistic context-free grammars [71, 159], as well as software tools that guess passwords based on dictionaries and mangling rules [112, 133]. We find that neural networks can guess passwords more successfully than other password-guessing methods in general, especially so beyond 10^{10} guesses and on non-traditional password policies. These cases are interesting because password-guessing attacks often proceed far beyond 10^{10} guesses [50, 52] and because existing password-guessing attacks underperform on new, non-traditional password policies [127, 128].

Although more effective password guessing using neural networks is an important contribution on its own, we also show that the neural networks we use can be highly compressed with minimal loss of guessing effectiveness. In addition to guessing passwords when simulating an attack, one application of password guessing models is providing a client-side measure of password strength. Because of our approach's compressibility, our approach is far more suitable than existing password-guessing methods for client-side strength estimation. Most existing client-side password checkers are inaccurate because they rely on simple, easily compressible heuristics, such as counting the number of characters or character classes in a password [31]. In contrast, we show that a highly compressed neural network more accurately measures password strength than existing client-side checkers. We can compress

such a neural network into hundreds of kilobytes, which is small enough to be included in an app for mobile devices, bundled with encryption software, or used in a web page password meter.

2 Detecting JavaScript injection vulnerabilities

Cross-site scripting (XSS) is the most frequently reported class of web-application vulnerabilities, constituting 25% of web vulnerabilities in 2014, and are becoming more and more common [21, 114]. By compromising client-side browser security using XSS, attackers can gain control over login cookies, passwords, and authentication tokens, and perform application-level actions as users, for example, sending emails or making financial transactions [101]. Preventing XSS typically requires website owners to not only sanitize all untrusted inputs to their web application, but also sanitize all input that could be received by the client's JavaScript interpreter—a task that can be error-prone due to the complexity of web applications and the widespread use of sensitive functions in JavaScript code. Document Object Model cross-site scripting (DOM XSS)—a particular type of XSS vulnerability that occurs entirely in the client-side JavaScript—has recently also gained attention. Traditional methods for detecting and defending against XSS vulnerabilities in server-side code—for example, server-side taint-tracking or web application firewalls—typically do not apply to DOM XSS because the vulnerability lies entirely in client code and servers may not even have logs to detect when an attack occurs [10, 19, 67, 117].

Our goal is to create practical, effective methods for detecting JavaScript injection vulnerabilities using machine learning. In Section 2.1, we discuss how we collect ground truth data. We use this ground truth data in our proposed approach, introduced in Section 2.2.

2.1 Collecting ground truth

Accurate ground truth data is necessary for models to create accurate predictions. Therefore, for our purpose, we build an automated tool for detecting certain types of JavaScript injection vulnerabilities with a low false positive rate. Prior work showed how to detect DOM XSS vulnerabilities using taint-tracking to track flows of attacker controllable information to sensitive functions (e.g., `eval` and `document.write`) [79, 105]. One challenge of this method is that once a flow that is a potential DOM XSS vulnerability is observed, the flow must be confirmed to be exploitable. In this work, we show how to more accurately detect whether a flow that is potentially vulnerable is capable of being exploited. Prior work used a method that correctly identified many possible flows, finding many real bugs. However, by using a different method of confirming vulnerabilities in tandem, we were able to find 83% more vulnerabilities than using prior methodology [79].

We used our improved methodology to detect DOM XSS vulnerabilities on the Internet. We found that, relative to the number of pages crawled, there are more observed flows than prior work [79]; additionally, we found a higher proportion of those flows are confirmed to be vulnerable than were confirmed to be vulnerable in prior work using the same methodology [79]. We believe this indicates that DOM XSS vulnerabilities are becoming more common in the four years since the previous study was undertaken. In addition, we qualitatively examined the reasons behind the vulnerabilities. For example, we observed that many of the vulnerabilities did not share the same code, implying that the vulnerabilities we found are due to custom code, rather than the inclusion of buggy shared libraries. Finally, using our collected dataset of DOM XSS vulnerabilities, we evaluated static-analysis tools that are designed to detect DOM XSS. We found that static-analysis tools performed poorly at detecting the same vulnerabilities found by the dynamic analysis. Our findings on static-analysis tools suggest that testing using both dynamic and static approaches may be necessary to secure web applications from DOM XSS.

2.2 Detecting injection vulnerabilities using machine learning

Client-side XSS vulnerabilities are particularly difficult to identify or defend against without human involvement because of the high degree of accuracy required for practical purposes, and the lack of insight by server-side operators into client-side execution. At the same time, manual analysis of source code for such vulnerabilities does not scale. We present a system for using machine learning to detect client-side XSS in JavaScript by examining source code. Because of the challenges of scale in our work—billions of individual JavaScript functions—and because web browsers are particularly sensitive to performance degradation, we aimed for vulnerability detection that is light-weight. At the same time, we must be robust to massive class imbalance—the vast majority of the code is not vulnerable. We find that our system can detect 80% of true potential vulnerabilities while maintaining a ratio of 78% of predicted vulnerabilities being true vulnerabilities. Our system could be used for triaging code to identify potential XSS vulnerabilities for further manual analysis, or for selectively implementing automated defensive intervention before vulnerable code is executed.

Chapter 2

Background and Related Work

Machine learning has been studied for applications in security contexts. We review related work on modeling passwords in Section 1 and on software vulnerability detection at the source-code level in Section 2.

1 Modeling password guessing

To highlight when modeling password strength matters, we first summarize password-guessing attacks. We then discuss metrics and models for evaluating password strength, including lightweight methods for estimating password strength during password creation. Finally, we summarize prior work on generating text using neural networks.

1.1 Password-guessing attacks

The extent to which passwords are vulnerable to guessing attacks is highly situational. For phishing attacks, keyloggers, or shoulder surfing, password strength does not matter because the attacker would already have access to the user's plaintext password. Similarly, some systems implement rate-limiting policies, locking an online account or a device after a small number of incorrect password entry attempts. In these cases, passwords other than perhaps

the million most predictable are unlikely to be guessed, preventing most guessing attacks [44].

Guessing attacks are a threat, however, in three scenarios. First, if rate limiting is not properly implemented, as is believed to have been the case in the 2014 theft of celebrities' personal photos from Apple's iCloud [56], large-scale guessing becomes possible. Second, if a database of hashed passwords is stolen, which sadly occurs frequently [13,16,22,51,52,88,111, 115,146], an offline attack is possible. An attacker chooses likely candidate passwords, hashes them, and searches the database for a matching hash. When a match is found, attackers can rely on the high likelihood of password reuse across accounts and try the same credentials on other systems [30]. Attacks leveraging password reuse have real-world consequences, including the recent compromise of Mozilla's Bugzilla database due to an administrator reusing a password [119] and the compromise of 20 million accounts on Taobao, a Chinese online shopping website similar to eBay, due to password reuse [36].

Third, common scenarios in which cryptographic key material is derived from, or protected by, a password are vulnerable to large-scale guessing in the same way as hashed password databases for online accounts. For instance, for password managers that sync across devices [61] or privacy-preserving cloud backup tools (e.g., SpiderOak [131]), the security of files stored in the cloud depends directly on password strength. Furthermore, cryptographic keys used for asymmetric secure messaging (e.g., GPG private keys), disk-encryption tools (e.g., TrueCrypt), and Windows Domain Kerberos Tickets [27] are protected by human-generated passwords. If the file containing this key material is compromised, the strength of the password is critical for security. The importance of this final scenario is likely to grow with the adoption of password managers and encryption tools.

1.2 Measuring password strength

Models of password strength often take one of two conceptual forms. The first relies on purely statistical methods, such as Shannon entropy or other advanced statistical approaches [14,15]. However, because of the large sample sizes required—on the order of tens of millions—we

consider these types of model out of scope for our use case, in which it would be ideal to measure the strength of a single password in isolation. Shannon entropy has traditionally been used, yet also requires large sized sample to model human-chosen passwords because of the large amount of very infrequent passwords [14].

The second conceptual approach to model password strength is to simulate adversarial password guessing [32, 86, 149], an approach which is capable of estimating the strength of a single password in isolation. Our application of neural networks follows this method. Below, we describe the password-guessing approaches that have been widely studied in academia and used in adversarial password cracking, all of which we compare to neural networks in our analyses. Academic studies of password guessing have focused on probabilistic methods that take as input large password sets, then output guesses in descending probability order. In contrast, password cracking tools used in practice rely on efficient heuristics to model common password characteristics.

Many research works have been built around creating new or improving existing password strength metrics [37, 73, 86, 94, 159]. There are statistical models of passwords such as Markov models [37, 86, 94] and probabalistic context-free grammars (PCFGs) [73, 159]. In addition, there are dictionary and mangling rule methods implemented by the popular tools Hashcat [133] and John the Ripper [112]. Probabilistic methods often make better guesses than mangling word list approaches; however, they are typically computationally less efficient at making those guesses. In practice, mangling word list methods—often guided by humans—are used to actually crack passwords, but probabilistic methods can give better estimates for password strength because of their better accuracy [149]. In addition, probabilistic methods typically perform relatively more consistently during guessing, while mangling word list methods may have sharper jumps when a specific, high frequency, mangling rule or dictionary word occurs during guessing [149]. Furthermore, mangling word list methods are often more optimized for the simple password policies that are often used in practice (i.e., passwords with at least eight chariacters), rather than more complex policies [149]. We next

describe the probabilistic methods in more detail.

1.3 Probabilistic methods

Probabilistic methods extract a statistical distribution from password training sources. The goal is to input large amounts of passwords into an algorithm, and ask the algorithm to output password guesses in order of most probable passwords.

Probabilistic context-free grammars One probabilistic method uses probabilistic context-free grammars (PCFGs) [159]. The intuition behind PCFGs is that passwords are built with template structures (e.g., six letters followed by two digits) and terminals that fit into those structures. A password's probability is the probability of its structure multiplied by those of its terminals.

Researchers have found that using separate training sources for structures and terminals improves guessing [71]. It is also beneficial to assign probabilities to unseen terminals by smoothing, as well as to augment guesses generated by the grammar with passwords taken verbatim from the training data without abstracting them into the grammar [73]. Furthermore, using natural-language dictionaries to instantiate terminals improves guessing, particularly for long passwords [153].

Markov models Using Markov models to guess passwords, first proposed in 2005 [94], has recently been studied more comprehensively [37, 86]. Conceptually, Markov models predict the probability of the next character in a password based on the previous characters, or context characters. Using more context characters can allow for better guesses, yet risks overfitting. Smoothing and backoff methods compensate for overfitting. Researchers have found that a 6-gram Markov model with additive smoothing is often optimal for modeling English-language passwords [86]. We use that configuration in our analyses.

1.4 Mangled word list methods

In adversarial password cracking, software tools are commonly used to generate password guesses [50]. The most popular tools transform a word list (passwords and dictionary entries) using mangling rules, or transformations intended to model common behaviors in how humans craft passwords. For example, a mangling rule may append a digit and change each 'a' to '@'. Two popular tools of this type are Hashcat [133] and John the Ripper (JtR) [112]. While these approaches are not directly based on statistical modeling, they produce somewhat accurate guesses [149] quickly, which has led to their wide use [50]. Hashcat, due to its efficient GPU implementation, is especially well known for being able to crack passwords in practice by hashing passwords [149]. John the Ripper—which also has a GPU implementation, albeit less well supported—has the ability to incrementally crack passwords, leveraging previous guesses in a password set to improve future guesses.

1.5 Proactive password checking

Although the previously discussed password-guessing models can accurately model human-created passwords [149], they take hours or days and megabytes or gigabytes of disk space [73, 86, 112, 133], making them too resource-intensive to provide real-time feedback to users. For example: a PCFG implementation requires nearly 5GB to represent the learned structure and terminal probabilities, and can take hours to look up probabilities [73]; Markov models require roughly 1GB to store frequency counts in our implementations [86]; JTR and Hashcat each requires large word lists, which can be roughly 1GB for accurate configurations and can require days to enumerate guesses [149]. Prior client-side password checkers, such as those running entirely in a web browser, rely on heuristics that can be easily encoded. Many common meters rate passwords based on their length or inclusion of different character classes [31, 148]. Unfortunately, in comprehensive tests of both client- and server-side password meters, all but one meter was highly inaccurate [31]. Only zxcvbn [160, 161], which uses dozens of more advanced heuristics, gave reasonably accurate strength estimations [31].

Such meters, however, do not directly model adversarial guessing because of their inability to succinctly encode models and calculate real-time results. In contrast, our approach, which we present in Section 3, models adversarial guessing entirely on the client side using principled simulation of guessing, rather than heuristic methods.

Given the challenges of implementing accurate password-strength measurement on resource-constrained clients, it might be tempting to use a system architecture where the password model is stored on a server and only measurement results are communicated to the client. However, in many situations the user's password should never be sent to the server for security reasons, for example, in the case of device encryption software, keys that protect cryptographic credentials, or the master password for a password manager. Even in cases where the user's password is eventually sent to an external server, using a remote password-strength measurement mechanism may allow powerful side channels based on keyboard timing, message size, and caching [7].

Neural networks Neural networks, which we use to model passwords, are a machine-learning technique for approximating highly dimensional functions [100]. Designed to model human neurons, they are particularly adept at fuzzy classification problems and generating novel sequences. Our method of generating candidate password guesses draws heavily on previous work that generated the probability of the next element in a string based on the preceding elements [55,138]. For example, in generating the string *password*, a neural network might be given *passwor* and output that *d* has a high probability of occurring next.

Although password creation and text generation are conceptually similar, little research has attempted to use insights from text generation to model passwords. A decade ago, neural networks were proposed as a method for classifying passwords into two very broad categories (weak or strong) [26], but that work did not seek to model the order in which passwords would be guessed or other aspects of a guessing attack. To our knowledge, the only proposal to use neural networks in a password-guessing attack was a recent blog post [95]. In sharp

contrast to our extensive testing of different parameters to make neural networks effective in practice, that work made few refinements to the application of neural networks, leading the author to doubt that the approach has "any practical relevance." Additionally, that work sought only to model a few likely password guesses, as opposed to our use of Monte Carlo methods to simulate an arbitrary number of guesses.

Conceptually, neural networks have advantages over other methods. In contrast to PCFGs and Markov models, the sequences generated by neural networks can be inexact, novel sequences [55], which led to our intuition that neural networks might be appropriate for password guessing. Prior approaches to probabilistic password guessing (e.g., Markov models [20]) require too much bandwidth for sending a representation of the model such that they are impractical to implement client-side. However, neural networks can model natural language in far less space than Markov models [90]. Neural networks have also been shown to transfer knowledge about one task to related tasks [166]. This is crucial for targeting novel password-composition policies, for which training data is sparse at best. Many approaches for modeling text use Long short-term memory (LSTM) models, a special type of recurrent neural network, which has feedback elements to be able to process sequences of inputs [55, 138]; however, recently, approaches using convolutional neural networks have been applied to text classification [77]. Our approach presented here uses LSTM models, though we have explored the use of convolutional models as well.

2 Software vulnerability detection in source code

Here we review work related to detecting JavaScript code injection vulnerabilities. We first provide general background on DOM XSS vulnerabilities on the web in Section 2.1. Then we cover related work characterizing the state of DOM XSS vulnerabilities in Section 2.2. Then, we describe a range of defenses against XSS in general, and DOM XSS vulnerabilities specifically. We describe content filtering techniques in Section 2.3, that attempt to filter

```
document.write(
  '<a href="' + document.location +
  '">Link</a>');
```

Figure 2.1: Example of a DOM XSS vulnerability. An attacker could inject arbitrary markup using `document.location` as an attack vector by crafting a link that injects an attacker-controlled script into the page. An attacker may execute code by crafting a link like: `http://[website]/[page]#"><script>CODE</script><!-`.

active exploits. We also discuss analysis techniques that attempt to identify vulnerabilities, describing dynamic analysis in Section 2.4, and static analysis in Section 2.5. Finally, we describe work relevant to our proposed approach that uses machine learning for program analysis in Section 2.6.

2.1 DOM XSS vulnerabilities

Code injection is a class of vulnerabilities where an attacker exploits a bug that enables the attacker to execute arbitrary code for the purpose of, for example, exfiltrating secret data, such as login tokens inside cookies, compromise the victim's machine by redirecting to a website that hosts malicious content. Cross-site-scripting (XSS) vulnerabilities in general are a type of injection vulnerability in which an attacker can inject arbitrary code into a running web application to, for example, take control of the data and credentials used in the application. For example, attackers may get access to the website's cookies (which potentially contain login tokens) or may execute user actions with respect to the compromised website [101]. In XSS, the injected code is JavaScript that runs in a web application with the privileges of the compromised website. Unlike traditional XSS attacks, in which an attacker's injection might be the result of a server-side failure to sanitize input, DOM XSS, also known as client-side XSS, is a relatively new type of XSS vulnerability that occurs as a result of JavaScript manipulation of the DOM on the client.

For a DOM XSS vulnerability to be present, there must be a flow of information from a potentially attacker-controlled source to a sensitive sink function. Examples of potentially attacker-controlled sources include: the URL of the document, accessed via the

`document.location` JavaScript object; data passed in cross-origin messages using the `postMessage` API; cookies accessed via the `document.cookie` object; and the HTTP referrer accessed by the `document.referrer` JavaScript object and other methods. Sinks can include any mechanism to execute arbitrary code, for example: the `eval` function, `document.write`, JavaScript event handlers (e.g., the "onclick" attribute), and URLs that have a JavaScript scheme (e.g., `<a href='javascript:...'>`). Figure 2.1 shows an example of a DOM XSS vulnerability. In the example, an attacker could craft a link that breaks out of the `href`'s single-quoted attribute and injects an arbitrary script; for example, `http://[website]/[page]#"><script>CODE</script><!-`. This link, when clicked, would execute the attacker-controlled code (`CODE`). An attacker may convince their victims to click on the link using social engineering, or may embed the link in an iframe on a website that the attacker controls. Like traditional XSS bugs, the specifics of crafting an exploit depend heavily on the website and the victim's browser.

2.2 Empirical studies of DOM XSS

Prior work by Lekies et al. crawled the internet searching for DOM XSS vulnerabilities on the Alexa Top 5,000 websites [79]. Their work showed that automatically generated exploits can be created and that DOM XSS vulnerabilities affect 9.6% of domains on the Alexa Top 5,000 websites. Our work for ground-truth detection uses a similar methodology as Lekies et al.'s work for identifying tainted flows, but we use a new and novel method for confirming whether flows are indicative of DOM XSS vulnerabilities. We describe in detail the similarities and differences between our methodologies and results in Section 2.2 in Chapter 4. Other work quantitatively examined DOM XSS vulnerabilities, finding that while many vulnerabilities are of low complexity, some are the result of highly complex JavaScript interactions [135]. The researchers found that many DOM XSS bugs are the result of vulnerable third-party scripts, missing knowledge about browser-provided APIs, unaware developers, or incompatible first- and third- party code. Our findings about vulnerability complexity and the role

of third-party code are similar; we compare them in detail in Sections 3.1 and 3.3. Differently from previous work, we also explore the role of advertising domains, the effectiveness of static-analysis tools, the distribution of vulnerabilities across and within domains, and design-level prevention mechanisms such as HTML templating.

2.3 XSS defenses: content filtering

Browsers and web servers have sought to employ filters for XSS exploits. Web site engineers can restrict the scripts that are allowed to run on websites with the use of content security policies (CSP) [66]. However, such policies are often not configured, misconfigured, or configured in such a way as to not substantially limit XSS exploits [19]. In early 2019, an experimental browser API based on the idea of trusted types was introduced. When the API is activated by a special CSP header, JavaScript code must use the API to access sensitive sinks; the API ensures that these interactions are safe. However, using the API requires that programmers modify code and is not compatible with legacy code [75].

Web application firewall filters are a common defense against general XSS vulnerabilities that work at the network level by filtering requests that contain potential exploits (e.g., the presence of the `script` tag in the URL or cookie). For general XSS exploits, attackers can routinely bypass such defenses by tweaking their exploit to evade the detection patterns used by the filter, for example, by using different encodings, such as HTML encoding, XML encoding [10, 67]. Furthermore, because DOM XSS vulnerabilities are often completely on the client, server-side filtering may not have the opportunity to remove malicious content. Client-side filters have also been used, such as the XSS auditor in browsers derived from WebKit [140], which often suffer from similar evasion attacks [117]. Researchers examined a list of known DOM XSS vulnerabilities, and showed that in 73% of cases the XSS Auditor fails to filter an attack [134]. The paper concluded that many domains make use of partly-tainted HTML markup injection, and that blocking all such cases would not be feasible.

```
var tainted_string = location.href; // http://example.com/#content
var b = tainted_string.substring(tainted_string.indexOf("#"));
var c = b + ":untainted";
document.write(c.substring(c.indexOf(":") + 1);
```

Figure 2.2: An example of potential overtainting, unless precise taint is applied.

2.4 Dynamic analysis of JavaScript

Dynamic analyses operate on observations (e.g. function calls) collected during program executions. To collect observations either the language runtime or the source code may need to be modified. Dynamic analyses typically are limited to information collected by specific executions and do not have insight into non-executed code. Furthermore, such methods incur runtime overhead [134], and can require significant engineering work to modify a complex runtime environment—in our case, the JavaScript engine. Despite the limitations of execution overhead and code coverage, dynamic analyses are often used because of their low rates of false positives [79]. One reason for the low false positive rate is because there is a concrete execution path that led to the observed behaviors.

Taint tracking The most relevant dynamic analysis for identifying injection vulnerabilities is taint tracking. This technique marks data from potentially attacker-controlled sources as tainted and propagates information about tainted values throughout the program. For example, the taint-tracking engine marks the concatenation of two pieces of data tainted if one of them is. When a tainted string is used in a sensitive sink, the taint-tracking engine could flag this flow of information as a potential DOM XSS vulnerability. In general, taint tracking can suffer from false positives when data is incorrectly marked as tainted (over tainting) and can suffer from false negatives when either tainted information is lost during execution (under tainting) or vulnerable code is not executed.

The first tool that uses taint tracking for discovering DOM XSS vulnerabilities is Firefox-based DOMinator [105], which marks data in the JavaScript engine as tainted and observes when tainted data entered sensitive sink functions. Later, researchers improved the precision

of the method by adding byte-precise taint tracking, which attaches taint information to specific bytes in the JavaScript engine [79, 134]. Precise taint tracking enables the taint analysis to have fewer instances of over tainting, especially in cases where only specific bytes of a tainted input are used. For example, like in Figure 2.2, when performing the substring and concatenation operations, it could be that no tainted bytes actually end up into a resulting string, but without byte-precise taint information, a taint analysis may be forced to over taint output. While these works have focused on taint tracking of string data types, other work combined static data-flow analysis with dynamic taint tracking to enable tracking of arbitrary JavaScript objects [23].

Confirming potential flows Taint tracking only identifies a flow from attacker-controllable input to sensitive sink functions; it does not guarantee that a flow is exploitable, because tainted data may be sanitized by the programmer. For example, a programmer may match tainted data to the regular expression for a digit before using it in a sensitive sink function, therefore rending the vulnerability unable to be injected. Researchers often use heuristic methods to generate automated exploits. In one prior work, researchers used a context-specific exploit generation methodology designed to create a workable exploit by analyzing the context in which the tainted string occurs in the sink [79]. In a separate line of work, potentially vulnerable flows were confirmed with a preconfigured list of injections that may lead to exploits [107]. While these approaches are capable of confirming many specific flows as vulnerable, in general, such heuristic approaches may have false negatives—flows that are not confirmed could potentially be confirmed using other injections. Currently, labor-intensive manual analysis by experts, or complex symbolic analysis (which also may have false negatives) [141], are the only alternatives to confirming flows with test injections.

Runtime-agnostic taint tracking To ameliorate the engineering burden of modifying the JavaScript runtime, other work used browser-agnostic taint tracking by implementing their taint engine in JavaScript [107]. They created a method to attach track taint informa-

tion using an extension that is injected into webpages, which can be used in multiple different JavaScript runtime environments (e.g., Chrome, Firefox). The browser-agnostic framework allows detecting vulnerabilities that are specific to certain browsers; however, such vulnerabilities account for a small fraction of all vulnerabilities. In addition, because such taint tracking is not implemented in the more efficient native code, it can cause large performance degradation. Furthermore, such runtime-agnostic taint tracking is unable to track taint information through the integrated DOM and other HTML APIs that are implemented in native code that are used in nearly all JavaScript code.

Dynamic runtime defenses While taint tracking can be used to identify vulnerabilities, it can also be used to defend against such attacks at run time. Researchers have also shown, for example, how to use taint tracking to defend against DOM XSS vulnerabilities at run time [134]. The work proposes the use of browser-based taint tracking to more precisely prevent XSS vulnerabilities. However, this requires modifying the browser engine and, as mentioned earlier, can cause performance degradation of between 7% and 17% in certain benchmarks. While this may seem low, browser vendors are exceptionally sensitive to performance degradation. The researchers conclude that many domains make use of partly-tainted HTML markup injection, and that blocking all such cases would not be feasible, instead recommending a specific heuristic policy to separate safe cases from dangerous cases. Other researchers used taint tracking to sanitize injected strings at run time just before those strings are used by the sensitive sink functions [106]. While this approach is promising, it requires that the vulnerable script be available ahead of time to a remote server, which is often not practical due to the frequent use of dynamically generated scripts or scripts that are only accessible behind login pages.

2.5 Static analysis of JavaScript

Static analyses, analyze source code to detect properties of interest without executing the program. Consequently, the analysis has insight to all the possible execution paths that a program may take and can reason about all paths. Static analyses are often used to detect issues before code deployment as part of development [57]. Static analysis does not incur runtime overhead, but may add friction to the development process. Scaling up to handle large code bases is a known issue for static analyses. It is particularly challenging to statically analyze Javascript, as it is a dynamic language and lacks strict typing information. Furthermore, some language constructs such as `eval` are difficult to reason about. Such challenges are well-documented [57, 143]. To improve scalability, static analyses often give up soundness and completeness and suffer from false-positive and false-negative errors [57, 143, 154].

There are a variety of different implementations of and techniques for static analysis for JavaScript available in commercial tools including: IBM Security AppScan, Acunetix, Trustwave App Scanner, Retina web application scanner, Qualys web inspect, HP Fortify static code analyzer, Coverity's JavaScript scanner, and Burp Suite Pro. However, it is typical for individual tools to make their own specific tradeoffs, and consequently it is rarely the case that one tool is strictly better than another, though it is common for specific tools to have unique findings [8,137]. In our work, we measure the false negative and false positive rate of specific tools, described in Section 3 in Chapter 4, and find that they do not identify the specific vulnerabilities that we found using dynamic methods.

Static analysis techniques Prior work has developed WALA libraries, which implement a method for static taint analysis in JavaScript and other languages. As part of a commercial product, the system was evaluated on 50 real websites [57]. It uses source and sink taint awareness to statically detect bugs and describes techniques to statically deal with some dynamic issues—prototypes, etc. In the paper, they limited runtime to four minutes on four

CPUs. Most time seems to be spent in search that starts at sources and attempts to find an un-sanitized path to a sink.

Taint Analysis for Java (TAJ) was developed to mitigate the compute time constraints of static analysis by heuristically focusing the analysis on parts of the program that are likely to have vulnerabilities [143]. Built on top of WALA, TAJ is static taint analysis engine for use in Java web applications. TAJ is designed to analyze applications of any size and produce useful answers given limited time a space. Uses a set of bounded analysis techniques to maximize utility when given limited runtime.

Mixed static and dynamic analyses To mitigate the downsides of static analysis, namely the amount of false positives, work has explored mixing elements of static and dynamic analyses.

To eliminate false positives in the static analysis, prior work has also explored mixing static and dynamic analysis by partially concretizing objects in the DOM—focusing on the `document.location` or `referrer` objects—based on the context in which they were viewed during the static analysis. The analysis then performs symbolic string analysis of the program to predict whether specific flows are vulnerable more accurately—e.g., when using `indexOf` on the `document.location` object, the analysis will know the return [141].

Other work has also explored blending static and dynamic analyses. One work dynamically detected what code was actually executed, by instrumenting uses of eval and dynamically included scripts, and then applied a static analysis tool to check that code; the only effect of this, is that the static analysis may have insight into dynamically executed code (e.g., with `eval` or by adding a `script` tag at runtime) [156]. Yet other work has started with static analysis, and selectively concretized the document URL and referrer based on the context to apply symbolic string analysis. The primary benefit being that the static analysis engine could distinguish which vulnerabilities are capable of being exploited [141]. Work in the Node.js ecosystem uses a simple static check for potential sink functions, and then applies

a run-time enforcement that attempts to infer templated arguments instead of concatenating code, finding many opportunities for automatically patching Node.js programs [132].

2.6 Machine learning in program analysis

As machine-learning algorithms and tools are getting more practical, there has been a rise in interest in applying machine learning to aid in program analysis. The idea of extracting source code into vectors and labeling for security purposes to analyze with statistical models has been used in a variety of contexts [29, 82, 165].

A number of projects have looked into using machine learning to analyze programs in JavaScript. One prior work on identifying vulnerabilities in source code used statistical models to reduce the number of false positives, by training a classifier on the user's manual labels [142]. A simple statistical model is used to classify detected flows as false positives or true positives based on user feedback. The model took into account hand-engineered features such as the location of the flow, the source and sink of the flow, and other metadata, such as the time required to identify the flow, and number of functions involved in the flow. These features were designed to remove noisy, repeated errors in common libraries or other complex functions. We avoid hand-engineered features so our approach would be generally applicable to other types of program analysis tasks. Zozzle uses machine learning to detect malicious JavaScript code [29]. Zozzle also used hand-engineered features of source code to analyze source code and then used a Bayesian model to classify source code samples, successfully identifying many instances of malicious JavaScript.

Researchers have also investigated how to learn basic program analysis building blocks from data [9]. A decision tree model is used to learn how to perform certain, low-level inference rules for program analysis, such as points-to analysis and how data flows through a program. The decision tree was trained using an iterative approach in which a small initial training data set is modified to create counter-examples that progressively teach more nuanced rules. In contrast, we will use deep neural networks, which obviates the need for

creating hand-engineered features. This has the benefit of being more general to different contexts, and would allow a system to adapt to changing source code idioms.

Most closely related to ours, is recent work on using neural network models to detect vulnerabilities in C code [82]. The researchers use LSTM models to detect vulnerabilities by identifying key points, which are domain-specific points in the program to analyze, typically specific library/API function calls. These key points are selected by extracting the parts of the program that relate to the expressions used in the arguments. One characteristic of this approach is that it leverages the highly static nature of the C programs that it analyzes. We experimented with using similar approaches for JavaScript, but found it difficult to statically determine with confidence the key points of a program and to extract data dependencies. Finally, their dataset does not have the significant class imbalance between rare vulnerable code and common safe code that is found in our work, likely because they focus only on a subset of code that they are able to statically extract. In their work, they are able to achieve results ranging from 5–18% false-negative rate while achieving between 3–5% false-positive rate.

Neural network vector representations of programs In using neural networks for program analysis, program representation remains an open issue. Prior work has explored a handfull of different representations [3, 4, 82, 92]; however, there is little consensus about what type of representation is appropriate for different tasks. Using random walks through program abstract syntax trees (ASTs), researchers created `code2vec`, which translates ASTs into vectors for consumption by machine learning models by linking starting and terminal nodes with a series of movements up and down the AST tree [4]. Tree convolutions base the classification of each node on the nodes that are close to it using neural network convolutions. Tree convolutions have been used previously to analyze source code to detect algorithm performance bugs [92]. Tree convolutions work by convolving node information over the tree-like structure, in a similar way as a neural network processes images. Work has also

been done on analyzing graph-structured data for use in program analysis. We also explored an approach using Gated-graph neural networks [81], a type of network that classifies graph structures, and has broad applications in modeling chemical bonds, natural language and social network graphs. These types of networks have recently been used to model certain properties of source code, such as idiomatic coding style [3]. Gated-graph neural networks model a graph classification problem by using the idea of message sending. Each node sends a message—a real-valued vector—describing its current state to all adjacent nodes. Nodes that receive messages combine the messages with their own current state to move to a new state. This process is repeated for a set number of iterations. In this model, different types of edges can be distinguished by the network in that each edge type will have a different function for combining inputs. Additionally, each node can be specifically classified, or the model can be modified to classify an entire graph. In practice, to classify an entire graph, it is suggested to add a synthetic node to the graph that is connected to all other nodes with a special edge type, which is used for classifying the entire graph.

Chapter 3

Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks[1]

1 Introduction

Text passwords are currently the most common form of authentication, and they promise to continue to be so for the foreseeable future [62]. Unfortunately, users often choose predictable passwords, enabling password-guessing attacks. In response, proactive password checking is used to evaluate password strength [11].

A common way to evaluate the strength of a password is by running or simulating password-guessing techniques [33, 71, 158]. A suite of well-configured guessing techniques, encompassing both probabilistic approaches [37, 86, 159] and off-the-shelf password-recovery tools [112, 133], can accurately model the vulnerability of passwords to guessing by expert attackers [149]. Unfortunately, these techniques are often very computationally intensive,

[1]The majority of this chapter was previously published as: William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks. In Proc. USENIX Security, 2016.

requiring hundreds of megabytes to gigabytes of disk space, and taking days to execute. Therefore, they are typically unsuitable for real-time evaluation of password strength, and sometimes for any practically useful evaluation of password strength.

With the goal of gauging the strength of human-chosen text passwords both more accurately and more practically, we propose using artificial neural networks to guess passwords. Artificial neural networks (hereafter referred to as "neural networks") are a machine-learning technique designed to approximate highly dimensional functions. They have been shown to be very effective at generating novel sequences [55,138], suggesting a natural fit for generating password guesses.

In this chapter, we first comprehensively test the impact of varying the neural network model size, model architecture, training data, and training technique on the network's ability to guess different types of passwords. We compare our implementation of neural networks to state-of-the-art password-guessing models, including widely studied Markov models [86] and probabilistic context-free grammars [71, 159], as well as software tools using mangled dictionary entries [112,133]. In our tests, we evaluate the performance of probabilistic models to large numbers of guesses using recently proposed Monte Carlo methods [32]. We find that neural networks guess passwords more successfully than other password-guessing methods in general, especially so beyond 10^{10} guesses and on non-traditional password policies. These cases are interesting because password-guessing attacks often proceed far beyond 10^{10} guesses [50, 52] and because existing password-guessing attacks underperform on new, non-traditional password policies [127, 128].

Although more effective password guessing using neural networks is an important contribution on its own, we also show that the neural networks we use can be highly compressed with minimal loss of guessing effectiveness. Our approach is thus far more suitable than existing password-guessing methods for client-side password checking. Most existing client-side password checkers are inaccurate [31] because they rely on simple, easily compressible heuristics, such as counting the number of characters or character classes in a password. In

contrast, we show that a highly compressed neural network more accurately measures password strength than existing client-side checkers. We can compress such a neural network into hundreds of kilobytes, which is small enough to be included in an app for mobile devices, bundled with encryption software, or used in a web page password meter.

To demonstrate the practical suitability of neural networks for client-side password checking, we implement and benchmark a neural-network password checker in JavaScript. This implementation, which we have released as open-source software,[2] is immediately suitable for use in mobile apps, browser extensions, and web page password meters. Our implementation gives real-time feedback on password strength in fractions of a second, and it more accurately measures resistance to guessing than existing client-side methods.

In summary, this chapter makes three main contributions that together substantially increase our ability to detect and help eliminate weak passwords. First, we propose neural networks as a model for guessing human-chosen passwords and comprehensively evaluate how varying their training, parameters, and compression impacts guessing effectiveness. In many circumstances, neural networks guess more accurately than state-of-art techniques. Second, leveraging neural networks, we create a password-guessing model sufficiently compressible and efficient for client-side proactive password checking. Third, we build and benchmark a JavaScript implementation of such a checker. In common web browsers running on commodity hardware, this implementation models an arbitrarily high number of adversarial guesses with sub-second latency, while requiring only hundreds of kilobytes of data to be transferred to a client. Together, our contributions enable more accurate proactive password checking, in a far broader range of common scenarios, than was previously possible.

2 System design

We experimented with a broad range of options in a large design space and eventually arrived at a system design that 1) leverages neural networks for password guessing, and 2) provides

[2] `https://github.com/cupslab/neural_network_cracking`

a client-side guess estimation method.

2.1 Measuring password strength

Similarly to Markov models, neural networks in our system are trained to generate the next character of a password given the preceding characters of a password. Figure 3.1 illustrates our construction. Like in Markov models [32, 86], we rely on a special password-ending symbol to model the probability of ending a password after a sequence of characters. For example, to calculate the probability of the entire password 'bad', we would start with an empty password, and query the network for the probability of seeing a 'b', then seeing an 'a' after 'b', and then of seeing a 'd' after 'ba', then of seeing a complete password after 'bad'. To generate passwords from a neural network model, we enumerate all possible passwords whose probability is above a given threshold using a modified beam-search [84], a hybrid of depth-first and breadth-first search. If necessary, we can suppress the generation of non-desirable passwords (e.g., those against the target password policy) by filtering those passwords. Then, we sort passwords by their probability. We use beam-search because breadth-first's memory requirements do not scale, and because it allows us to take better advantage of GPU parallel processing power than depth-first search. Fundamentally, this method of guess enumeration is similar to that used in Markov models, and it could benefit from the same optimizations, such as approximate sorting [37]. A major advantage over Markov models is that the neural network model can be efficiently implemented on the GPU.

Calculating guess numbers In evaluating password strength by modeling a guessing attack, we calculate a password's *guess number*, or how many guesses it would take an attacker to arrive at that password if guessing passwords in descending order of likelihood. The traditional method of calculating guess numbers by enumeration is computationally intensive. For example, enumerating more than 10^{10} passwords would take roughly 16 days in our unoptimized implementation on an NVIDIA GeForce GTX 980 Ti. However, in

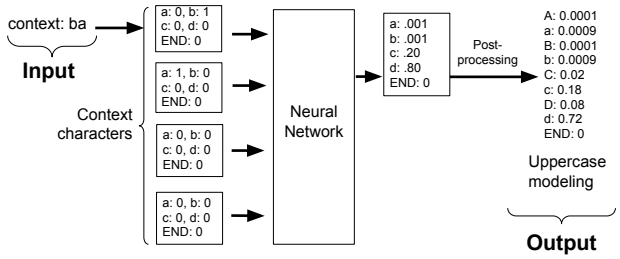


Figure 3.1: An example of using a neural network to predict the next character of a password fragment. The network is being used to predict a 'd' given the context 'ba'. This network uses four characters of context. The probabilities of each next character are the output of the network. Post processing on the network can infer probabilities of uppercase characters.

addition to guess number enumeration, we can also estimate guess numbers accurately and efficiently using Monte Carlo simulations, as proposed by Dell'Amico and Filippone [32].

2.2 Our approach

There are many design decisions necessary to train neural networks. The design space forces us to decide on the modeling alphabet, context size, type of neural network architecture, training data, and training methodology. We experiment along these dimensions.

Model architectures In this work, we use recurrent neural networks because they have been shown to be useful for generating text in the context of character-level natural language [55, 138]. Recurrent neural networks are a specific type of neural network where connections in the network can process elements in sequences and use an internal memory to remember information about previous elements in the sequence. We experiment with two different recurrent architectures in Section 4.1.

Alphabet size We focus on character-level models, rather than more common word-level models, because there is no established dictionary of words for password generation. We also complement our analysis with exploratory experiments using syllable-level models in Section 4.1. We decided to explore hybrid models based on prior work in machine learning [90]. In the hybrid construction, in addition to characters, the neural network is allowed

to model sub-word units, such as syllables or tokens. We chose to model 2,000 different tokens based on prior work [90] and represent those tokens the same way we would characters. A more thorough study of tokenized models would explore both more and fewer tokens. Using tokenized structures, the model can then output the probability of the next character being an 'a' or the token 'pass'. We generated the list of tokens by tokenizing words in our training set along character-class boundaries and selecting the 2,000 most frequent ones.

Like prior work [20], we observed empirically that modeling all characters unnecessarily burdens the model and that some characters, like uppercase letters and rare symbols, are better modeled outside of the neural network. We can still create passwords with these characters by interpreting the model's output as templates. For example, when the neural network predicts an 'A' character, we post-process the prediction to predict both 'a' and 'A' by allocating their respective probabilities based on the number of occurrences of 'a' and 'A' in the training data—as shown in Figure 3.1. The intuition here is that we can reduce the amount of resources consumed by the neural network when alternate heuristic approaches can efficiently model certain phenomena (e.g., shifts between lowercase and uppercase letters).

Password context Predictions rely on the context characters. For example, in Figure 3.1, the context characters are 'ba' and the target prediction is 'd'. Increasing the number of context characters increases the training time, while decreasing the number of context characters could potentially decrease guessing success.

We experimented with using all previous characters in the password as context and with only using the previous ten characters. We found in preliminary tests that using ten characters was as successful at guessing and trained up to an order of magnitude faster, and thus settled on this choice. When there are fewer than ten context characters, we pad the input with zeros. In comparison, best-performing Markov models typically use five characters of context [32, 86]. While Markov models can overfit if given too much context, neural networks typically overfit when there are too many parameters.

Providing context characters in reverse order—e.g., predicting 'd' from 'rowssap' instead of 'passwor'—has been shown to sometimes improve performance [54]. We empirically evaluate this technique in Section 4.1.

Model size We must also decide how many parameters to include in models. To gauge the effect of changing the model size on guessing success, we test a large neural network with 15,700,675 parameters and a smaller network with 682,851 parameters. The larger size was chosen to limit the amount of time and GPU memory used by the model, which required one and a half weeks to fully train on our larger training set. The smaller size was chosen for use in our browser implementation because it could realistically be sent over the Internet; compressed, this network is a few hundred kilobytes. We evaluate the two sizes of models with a variety of password policies, since each policy may respond differently to size constraints, and describe the results in Section 4.1.

Transference learning We experimented with a specialized method of training neural networks that takes advantage of *transference learning*, in which different parts of a neural network learn to recognize different phenomena during training [166]. One of the key problems with targeting non-traditional password policies is that there is little training data. For example, in our larger training set, there are 105 million passwords, but only 2.6 million satisfy a password policy that requires a minimum of 16 characters. The sparsity of training samples limits guessing approaches' effectiveness against such non-traditional policies. However, if trained on all passwords, the learned model is non-optimal because it generates passwords that are not accurate for our target policy even if one ignores passwords that do not satisfy the policy. Transference learning lets us train a model on all passwords, yet tailor its guessing to only longer passwords.

When using transference learning, the model is first trained on all passwords in the training set. Then, the lower layers of the model are frozen. Finally, the model is retrained only on passwords in the training set that fit the policy. The intuition is that the lower layers

in the model learn low-level features about the data (e.g., that 'a' is a vowel), and the higher layers learn higher-level features about the data (e.g., that vowels often follow consonants). Similarly, the lower layers in the model may develop the ability to count the number of characters in a password, while the higher level layers may recognize that passwords are typically eight characters long. By fine-tuning the higher-level parameters, we can leverage what the model learned about all passwords and retarget it to a policy for which training data is sparse.

Training data We experimented with different sets of training data; we describe experiments with two sets of passwords in Sections 3.1 and 4.2, and also with including natural language in training data in Section 4.1. For machine-learning algorithms in general, more training data is better, but only if the training data is a close match for the passwords we test on.

2.3 Client-side models

Deploying client-side (e.g., browser-based) password-strength-measuring tools presents severe challenges. To minimize the latency experienced by users, these tools should execute quickly and transfer as little data as possible over the network. Advanced guessing tools (e.g., PCFG, Markov models, and tools like JtR and Hashcat) run on massively parallel servers and require on the order of hundreds of megabytes or gigabytes of disk space. Typically, these models also take hours or days to return results of strength-metric tests, even with recent advances in efficient calculation [32], which is unsuitable for real-time feedback. In contrast, by combining a number of optimizations with the use of neural networks, we can build accurate password-strength-measuring tools that are sufficiently fast for real-time feedback and small enough to be included in a web page.

Optimizing for model size

To deploy our prototype implementation in a browser, we developed methods for succinctly encoding it. We leveraged techniques from graphics for encoding 3D models for browser-based games and visualizations [25]. Our encoding pipeline contains four different steps: weight quantization, fixed-point encoding, ZigZag encoding, and lossless compression. Our overall strategy is to send fewer bits and leverage existing lossless compression methods that are natively supported by browser implementations, such as `gzip` compression [47]. We describe the effect that each step in the pipeline has on compression in Section 4.3. We also describe encoding a short word list of passwords in Bloom filters.

Weight quantization First, we quantized the weights of the neural network to represent them with fewer digits. Rather than sending all digits of the 32-bit floating-point numbers that describe weights, we only send the most significant digits. Weight quantization is routinely used for decreasing model size, but can increase error [90]. We show the effect of quantization on error rates in Section 4.3. We experimentally find that quantizing weights up to three decimal digits leads to minimal error.

Fixed-point encoding Second, instead of representing weights using floating-point encoding, we used fixed-point encoding. Due to the weight-quantization step, many of the weight values are quantized to the same values. Fixed-point encoding allows us to more succinctly describe the quantized values using unsigned integers rather than floating point numbers on the wire: one could internally represent a quantized weight between -5.0 and 5.0 with a minimum precision of 0.005, as between -1000 and 1000 with a precision of 1. Avoiding the floating-point value would save four bytes. While lossless compression like `gzip` partially reduces the need for fixed-point encoding, we found that such scaling still provides an improvement in practice.

ZigZag encoding Third, negative values are generally more expensive to send on the wire. To avoid sending negative values, we use ZigZag encoding [116]. In ZigZag encoding, signed values are encoded by using the last bit as the sign bit. So, the value of 0 is encoded as 0, but the value of -1 is encoded as 1, 1 is encoded as 2, -2 is encoded as 3, and so on.

Lossless compression We use regular `gzip` or `deflate` encoding as the final stage of the compression pipeline. Both `gzip` and `deflate` produce similar results in terms of model size and both are widely supported natively by browsers and servers. We did not consider other compression tools, like LZMA, because their native support by browsers is not as widespread, even though they typically result in slightly smaller models.

Bloom filter word list To increase the success of client-side guessing, we also store a word list of frequently guessed passwords. As in previous work [149], we found that for some types of password-cracking methods, prepending training passwords improves guessing effectiveness. We stored the first two million most frequently occurring passwords in our training set in a series of compressed Bloom filters [91].

Because Bloom filters cannot map passwords to the number of guesses required to crack, and only compute existence in a set, we use multiple Bloom filters in different groups: in one Bloom filter, we include passwords that require fewer than 10 guesses; in another, all passwords that require fewer than 100 guesses; and so on. On the client, a password is looked up in each filter and assigned a guess number corresponding to the filter with the smallest set of passwords. This allows us to roughly approximate the guess number of a password without increasing the error bounds of the Bloom filter. To drastically decrease the number of bits required to encode these Bloom filters, we only send passwords that meet the requirements of the policy and would have neural-network-computed guess numbers more than three orders of magnitude different from their actual guess numbers. We limited this word list to be about 150KB after compression in order to limit the size of our total model. We found that significantly more space would be needed to substantially improve guessing

success.

Optimizing for latency

We rely on precomputation and caching to make our prototype sufficiently fast for real-time feedback. Our target latency is near 100 ms because that is the threshold below which updates appear instantaneous [99].

Precomputation We precompute guess numbers instead of calculating guess numbers on demand because all methods of computing guess numbers on demand are too slow to give real-time feedback. For example, even with recent advances in calculation efficiency [32], our fastest executing model, the Markov model, requires over an hour to estimate guess numbers of our test set passwords, with other methods taking days. Precomputation decreases the latency of converting a password probability to a guess number: it becomes a quick lookup in a table on the client.

The drawback of this type of precomputation is that guess numbers become inexact due to the quantization of the probability-to-guess-number mapping. We experimentally measure (see Section 4.3) the accuracy of our estimates, finding the effect on accuracy to be low. For the purpose of password-strength estimation, we believe the drawback to be negligible, in part because results are typically presented to users in more heavily quantized form. For instance, users may be told their password is "weak" or "strong." In addition, the inaccuracies introduced by precomputation can be tuned to result in safe errors, in that any individual password's guess number may be an underestimate, but not an overestimate.

Caching intermediate results We also cache results from intermediate computations. Calculating the probability of a 10-character password requires 11 full computations of the neural network, one for each character and one for the end symbol. By caching probabilities of each substring, we significantly speed up the common case in which a candidate password changes by having a character added to or deleted from its end. We experimentally show

the benefits of caching in Section 4.3.

Multiple threads On the client side, we run the neural network computation in a separate thread from the user interface for better responsiveness of the user interface.

2.4 Implementation

We build our server-side implementation on the Keras library [24] and the client-side implementation on the neocortex browser implementation [97] of neural networks. We use the Theano back-end library for Keras, which trains neural networks faster by using a GPU rather than a CPU [6, 7]. Our implementation trains networks and guesses passwords in the Python programming language. Guess number calculation in the browser is performed in JavaScript. Our models typically used three long short-term memory (LSTM) recurrent layers and two densely connected layers for a total of five layers. On the client side, we use the WebWorker browser API to run neural network computations in their own thread [150].

For some applications, such as in a password meter, it is desirable to conservatively estimate password strength. Although we also want to minimize errors overall, on the client we prefer to underestimate a password's resistance to guessing, rather than overestimate it. To get a stricter underestimate of guess numbers on our client-side implementation, we compute the guess number without respect to capitalization. We find in practice that our model is able to calculate a stricter underestimate this way, without overestimating many passwords' strength. We don't do this for the server-side models because those models are used to generate candidate password guesses, rather than estimating a guess number. After computing guess numbers, we apply to them a constant scaling factor, which acts as a security parameter, to make the model more conservative at the cost of making more errors. We discuss this tradeoff more in Section 4.3.

3 Testing methodology

To evaluate our implementation of neural networks, we compare it to multiple other password cracking methods, including PCFGs, Markov models, JtR, and Hashcat. Our primary metric for guessing accuracy is the guessability of our test set of human-created passwords. The guessability of an individual password is measured by how many guesses a guesser would take to crack a password. We experiment with two sets of training data and with five sets of test data. For each set of test data, we compute the percentage of passwords that would be cracked after a particular number of guesses. More accurate guessing methods correctly guess a higher percentage of passwords in our test set.

For probabilistic methods—PCFG, Markov models, and neural networks—we use recent work to efficiently compute guess numbers using Monte Carlo methods [32]. For Monte Carlo simulations, we generate and compute probabilities for at least one million random passwords to provide accurate estimates. While the exact error of this technique depends heavily on each method, guess number, and individual password, typically we observed 95% confidence intervals of less than 10% of the value of the guess-number estimate; passwords for which the error exceeded 10% tended to be guessed only after more than 10^{18} guesses. For all Monte Carlo simulations, we model up to 10^{25} guesses for completeness. This is likely an overestimate of the number of guesses that even a well-resourced attacker could be able to or would be incentivized to make against one password.

To calculate guessability of passwords using mangling-rule-based methods—JtR and Hashcat—we enumerate all guesses that these methods make. This provides exact guess numbers, but fewer guesses than we simulate with other methods. Across our different test sets, the mangling-rule-based methods make between about 10^{13} and 10^{15} guesses.

3.1 Training data

To train our algorithms, we used a mixture of leaked and cracked password sets. We believe this is ethical because these password sets are already publicly available and we cause no additional harm with their use.

We explore two different sets of training data. We term the first set the Password Guessability Service (*PGS*) training set, used by prior work [149]. It contains the Rockyou [151] and Yahoo! [49] leaked password sets. For guessing methods that use natural language, it also includes the web2 list [155], Google web corpus [53], and an inflection dictionary [125]. This set totals 33 million passwords and 5.9 million natural-language words.

The second set (the *PGS++* training set) augments the PGS training set with additional leaked and cracked password sets [5, 13, 16, 18, 28, 40, 43, 48, 49, 58, 64, 65, 68, 76, 78, 88, 110, 113, 115, 123, 130, 136, 144, 151, 163, 167]. For methods that use natural language, we include the same natural-language sources as the PGS set. This set totals 105 million passwords and 5.9 million natural-language words.

3.2 Testing data

For our testing data we used passwords collected from Mechanical Turk (MTurk) in the context of prior research studies, as well as a set sampled from the leak of plaintext passwords from 000webhost [46]. In addition to a common policy requiring only eight characters, we study three less common password policies shown to be more resistant to guessing [87, 128]: 4class8, 3class12, and 1class16, all described below. We chose the MTurk sets to get passwords created under more password policies than were represented in leaked data. Passwords generated using MTurk have been found to be similar to real-world, high-value passwords [42, 87]. Nonetheless, we chose the 000webhost leak to additionally compare our results to real passwords from a recently leaked password set. In summary, we used five testing datasets:

- **1class8**: 3,062 passwords longer than eight characters collected for a research study [71]
- **1class16**: 2,054 passwords longer than sixteen characters collected for a research study [71]
- **3class12**: 990 passwords that must contain at least three character classes (uppercase letters, lowercase letters, symbols, digits) and be at least twelve characters long collected for a research study [128]
- **4class8**: 2,997 passwords that must contain all four character classes and be at least eight characters long collected for a research study [87]
- **webhost**: 30,000 passwords randomly sampled from among passwords containing at least eight characters in the 000webhost leak [46]

3.3 Guessing configuration

PCFG We used a version of PCFG with terminal smoothing and hybrid structures [73], and included natural-language dictionaries in the training data, weighted for each word to count as one tenth of a password. We also separated training for structures and terminals, and trained structures only on passwords that conform to the target policy. This method does not generate passwords that do not match the target policy.

For PCFG, Monte Carlo methods are not able to estimate unique guess numbers for passwords that have the same probability. This phenomenon manifests in the Monte Carlo graphs with jagged edges, where many different passwords are assigned the same guess number (e.g., in Figure 3.5c before 10^{23}). We assume that an optimal attacker could order these guesses in any order, since they all have the same likelihood according to the model. Hence, we assign the lowest guess number to all of these guesses. This is a strict overestimate of PCFG's guessing effectiveness, but in practice does not change the results.

Markov models We trained 4-, 5-, and 6-gram models. Prior work found the 6-gram models and additive smoothing of 0.01 to be an effective configuration for most password

sets [86]. Our results agree, and we use the 6-gram model with additive smoothing in our tests. We discard guesses that do not match the target policy.

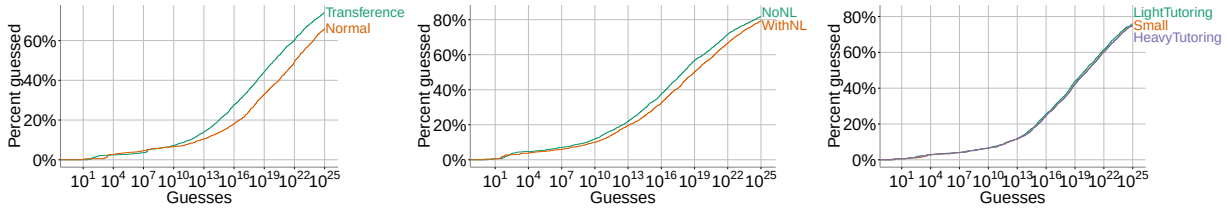
Mangling word list methods We compute guess numbers using the popular cracking tools Hashcat and John the Ripper (JtR). For Hashcat, we use the best64 and gen2 rule sets that are included with the software [133]. For JtR, we use the SpiderLabs mangling rules [145]. We chose these sets of rules because prior work found them effective in guessing general-purpose passwords [149]. To create the input for each tool, we uniqued and sorted the respective training set by descending frequency. For JtR, we remove guesses that do not match the target policy. For Hashcat, however, we do not do so because Hashcat's GPU implementation can suffer a significant performance penalty. We believe that this models a real-world scenario where this penalty would also be inflicted.

4 Evaluation

We performed a series of experiments to tune the training of our neural networks and compare them to existing guessing methods. In Section 4.1, we describe experiments to optimize the guessing effectiveness of neural networks by using different training methods. These experiments were chosen primarily to guide our decisions about model parameters and training along the design space we describe in Section 2.2, including training methods, model size, training data, and network architecture. In Section 4.2, we compare the effectiveness of the neural network's guessing to other guessing algorithms. Finally, in Section 4.3, we describe our browser implementation's effectiveness, speed, and size, and we compare it to other browser password-measuring tools.

4.1 Training neural networks

We conducted experiments exploring how to tune neural network training, including modifying the network size, using sub-word models, including natural-language dictionaries in



(a) Transference learning **(b)** Natural-language dictionaries **(c)** Tutoring

Figure 3.2: Alternative training methods for neural networks. The x-axes represent the number of guesses in log scale. The y-axes show the corresponding percentage of 1class16 passwords guessed. In (b), *WithNL* is a neural network trained with natural-language dictionaries, and *NoNL* is a neural network trained without natural-language dictionaries.

training, and exploring alternative architectures. We do not claim that these experiments are a complete exploration of the space. Indeed, improving neural networks is an active area of research.

Transference learning We find that the transference learning training, described in Section 2.2, improves guessing effectiveness. Figure 3.2a shows in log scale the effect of transference learning. For example, at 10^{15} guesses, 22% of the test set has been guessed with transference learning, as opposed to 15% without transference learning. Using a 16 MB network, we performed this experiment on our 1class16 passwords because they are particularly different from the majority of our training set. Here, transference learning improves password guessing mostly at higher guess numbers.

Including natural-language dictionaries We experimented with including natural-language dictionaries in the neural network training data, hypothesizing that doing so would improve guessing effectiveness. We performed this experiment with 1class16 passwords because they are particularly likely to benefit from training on natural-language dictionaries [153]. Networks both with and without natural language data were trained using the transference learning method on long passwords. Natural language was included with the primary batch of training data. Figure 3.2b shows that, contrary to our hypotheses, training on natural language decreases the neural network's guessing effectiveness. We believe neural networks do not benefit from natural language, in contrast to other methods like PCFG,

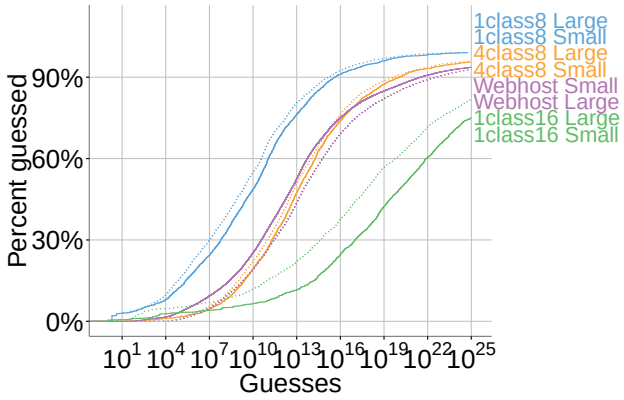


Figure 3.3: Neural network size and password guessability. Dotted lines are large networks; solid lines are small networks.

because this method of training does not differentiate between natural-language dictionaries and password training. However, training data could be enhanced with natural language in other ways, perhaps yielding better results.

Password tokenization We find that using hybrid, sub-word level password models does not significantly increase guessing performance at low guess numbers. Hybrid models may represent the same word in multiple different ways. For example, the model may capture a word as one token, 'pass', or as the letters 'p', 'a', 's', 's'. Because Monte Carlo simulations assume that passwords are uniquely represented, instead of using Monte Carlo methods to estimate guess numbers, we calculated guess numbers by enumerating the most probable 10^7 guesses. However, at this low number of guesses, we show this tokenization has only a minor effect, as shown in Figure 3.4b. We conducted this experiment on long passwords because we believed that they would benefit most from tokenization. This experiment shows that there may be an early benefit, but otherwise the models learn similarly. We consider this result to be exploratory both due to our low guessing cutoff and because other options for tuning the tokenization could produce better results.

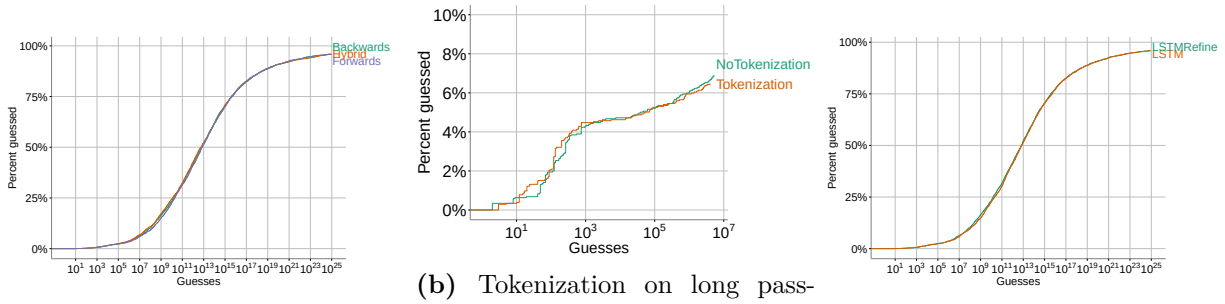
Model size We find that, for at least some password sets, neural network models can be orders of magnitude smaller than other models with little effect on guessing effectiveness. We tested how the following two model sizes impact guessing effectivess: a large model with

1,000 LSTM cells or 15,700,675 parameters that uses 60 MB, and a small model with 200 LSTM cells or 682,851 parameters that takes 2.7 MB.

The results of these experiments are shown in Figure 3.3. For 1class8 and 4class8 policies, the effect of decreasing model size is minor but noticeable. However, for 1class16 passwords, the effect is more dramatic. We attribute differences between the longer and shorter policies with respect to model size to fundamental differences in password composition between those policies. Long passwords are more similar to English language phrases, and modeling them may require more parameters, and hence larger networks, than modeling shorter passwords. The webhost test set is the only set for which the larger model performed worse. We believe that this is due to the lack of suitability of the particular training data we used for this model. We discuss the differences in training data more in Section 4.2.

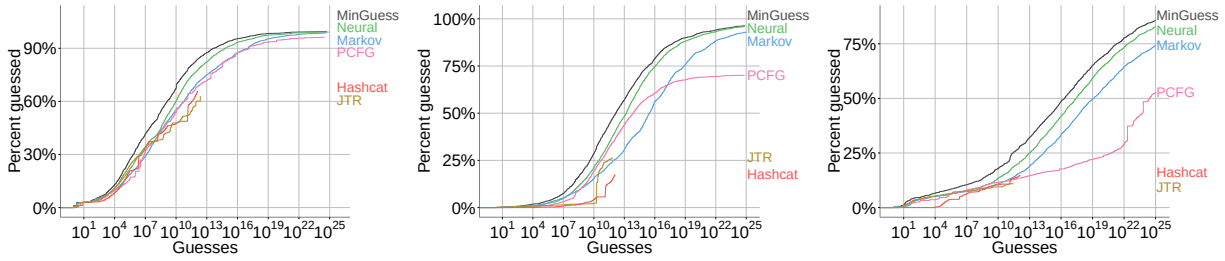
Tutored networks To improve the effectiveness of our small model at guessing long passwords, we attempted to tutor our small neural network with randomly generated passwords from the larger network. While this had a mild positive effect with light tutoring, at a roughly one to two ratio of random data to real data, the effect does not seem to scale to heavier tutoring. Figure 3.2c shows minimal difference in guessing accuracy when tutoring is used, and regardless of whether it is light or heavy.

Backwards vs. forwards training As described in Section 2.2, processing input backwards rather than forwards can be more effective in some applications of neural networks [54]. We experiment with guessing passwords backwards, forwards, and using a hybrid approach where half of the network examines passwords forwards and the other half backwards. We observed only marginal differences overall. At the point of greatest difference, near 10^9 guesses, the hybrid approach guessed 17.2% of the test set, backwards guessed 16.4% of the test set and forwards guessed 15.1% of the test set. Figure 3.4a shows the result of this experiment. Since the hybrid approach increases the amount of time required to train with only small improvement in accuracy, for other experiments we use backwards training.



(a) Training direction. **(b)** Tokenization on long passwords. **(c)** LSTM vs. refined LSTM.

Figure 3.4: Additional tuning experiments. Our LSTM experiments tested on complex passwords with 16M parameters. We found very little difference in performance. Our experiments on tokenization examined long passwords. Our experiments on training direction involved training backwards, forwards, and both backwards and forwards with 16M parameters on complex passwords.



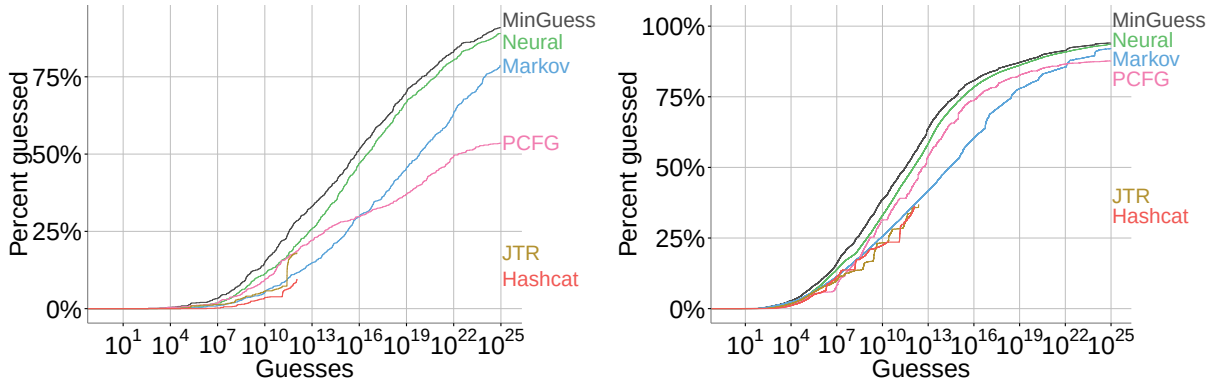
(a) 1class8 passwords **(b)** 4class8 passwords **(c)** 1class16 passwords

Figure 3.5: Guessability of our password sets for different guessing methods using the PGS data set. *MinGuess* stands for the minimum number of guesses for any approach. Y-axes are differently scaled to best show comparative performance.

Recurrent architectures We experimented with two different types of recurrent neural-network architectures: long short-term memory (LSTM) models [63] and a refinement on LSTM models [70]. We found that this choice had little effect on the overall output of the network, with the refined LSTM model being slightly more accurate, as shown in Figure 3.4c.

4.2 Guessing effectiveness

Compared to other individual password-guessing methods, we find that neural networks are better at guessing passwords at a higher number of guesses and when targeting more complex or longer password policies, like our 4class8, 1class16, and 3class12 data sets. For example, as shown in Figure 3.5b, neural networks guessed 70% of 4class8 passwords by 10^{15} guesses, while the next best performing guessing method guesses 57%.

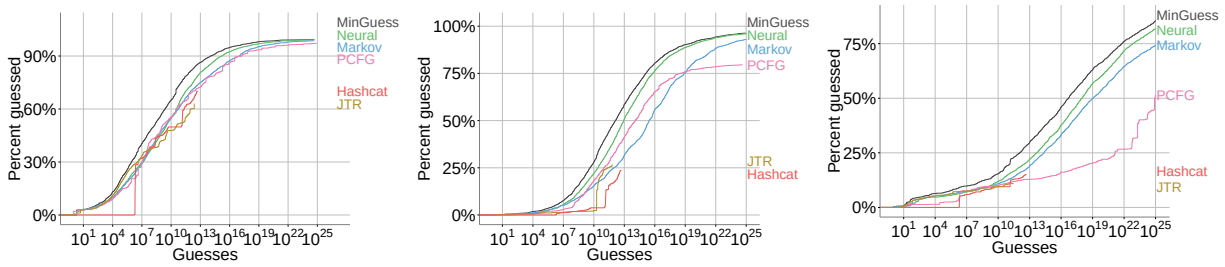


(a) 3class12 passwords **(b)** Webhost passwords

Figure 3.6: Guessability of our password sets for different guessing methods using the PGS data set (continued).

Models differ in how effectively they guess specific passwords. *MinGuess*, shown in Figure 3.5, represents an idealized guessing approach in which a password is considered guessed as soon as it is guessed by any of our guessing approaches, including neural networks, Markov models, PCFG, JtR, and Hashcat. That MinGuess outperforms neural networks suggests that using multiple guessing methods should still be preferred to using any single guessing method for accurate strength estimation, despite the fact that neural networks generally outperform other models individually.

For all the password sets we tested, neural networks outperformed other models beginning at around 10^{10} guesses, and matched or beat the other most effective methods before that point. Figures 3.5-3.6 show the performance of the different guessing methods trained with the PGS data set, and Figures 3.7-3.8 show the same guessing methods trained with the PGS++ data set. Both data sets are described in more detail in Section 3.1. In this section, we used our large, 15.7 million parameter neural network, trained with transference learning on two training sets. While performance varies across guessing method and training set, in general we find that the neural networks' performance at high guess numbers and across policies holds for both sets of training data with one exception, discussed below. Because these results hold for multiple training and test sets, we hypothesize that neural networks would also performe well in guessing passwords created under many policies that we did not



(a) 1class8 passwords **(b)** 4class8 passwords **(c)** 1class16 passwords

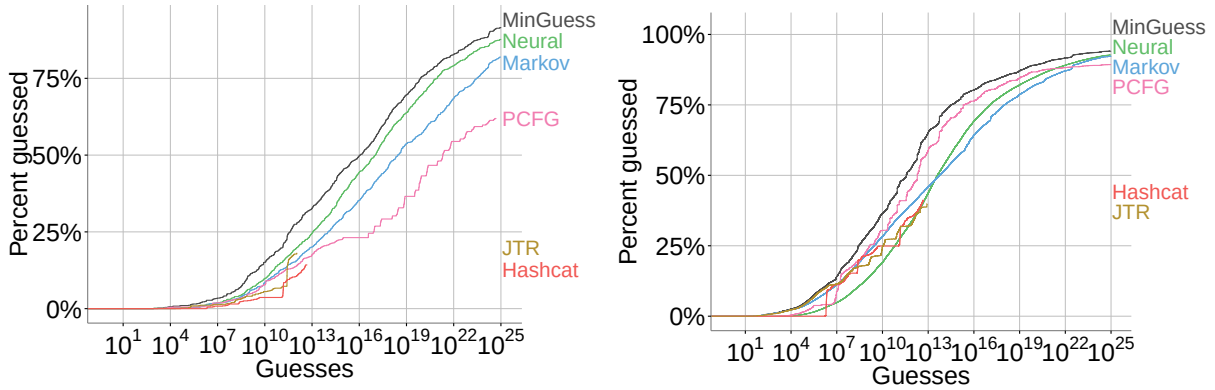
Figure 3.7: Guessability of our password sets for different guessing methods using the PGS++ data set. *MinGuess* stands for the minimum number of guesses for any approach.

test.

In the webhost test set using the PGS++ training data, neural networks performed worse than other methods. For webhost, all guessing methods using the PGS++ data set were less effective than the PGS data set, though some methods, such as PCFG, were only slightly affected. Because all methods perform worse, and because, when using the PGS training data, neural networks do better than other methods—similar to other test sets—we believe that the PGS++ training data is particularly ineffective for this test set. As Figure 3.3 shows, this is the only data set where a smaller neural network performs significantly better than the larger neural network, which suggests that the larger neural network model is fitting itself more strictly to low-quality data, which limits the larger network's ability to generalize.

Qualitatively, the types of passwords that our implementation of neural networks guessed before other methods were novel passwords that were dissimilar to passwords in the training set. The types of passwords that our implementation of neural networks were late to guess but that were easily guessable by other methods often were similar to words in the natural-language dictionaries, or were low-frequency occurrences in the training data.

Resource requirements In general, PCFGs require the most disk, memory, and computational resources. Our PCFG implementation stored its grammar in 4.7GB of disk space. Markov models are the second largest of our implementations, requiring 1.1GB of disk space. Hashcat and JtR do not require large amounts of space for their rules, but do require storing the entire training set, which is 756MB. In contrast, our server-side neural network requires



(a) 3class12 passwords **(b)** Webhost passwords

Figure 3.8: Guessability of our password sets for different guessing methods using the PGS++ data set (continued).

Pipeline stage	Size	`gzip`-ed Size
Original JSON format	6.9M	2.4M
Quantization	4.1M	716K
Fixed point	3.1M	668K
ZigZag encoding	3.0M	664K
Removing spaces	2.4M	640K

Table 3.1: The effect of different pipeline stages on model size. This table shows the small model that targets the 1class8 password policy, with 682,851 parameters. Each stage includes the previous stage, e.g., the fixed-point stage includes the quantization stage. We use `gzip` at the highest compression level.

only 60MB of disk space. While 60MB is still larger than what could effectively be transferred to a client without compression, it is a substantial improvement over the other models.

4.3 Browser implementation

While effective models can fit into 60MB, this is still too large for real-time password feedback in the browser. In this section, we evaluate our techniques for compressing neural network models, discussed in Section 2.3, by comparing the guessing effectiveness of the compressed models to all server-side models—our large neural network, PCFG, Markov models, JtR, and Hashcat.

Model encoding Our primary size metric is the `gzip`-ed model size. Our compression stages use the JSON format because of its native support in JavaScript platforms. We

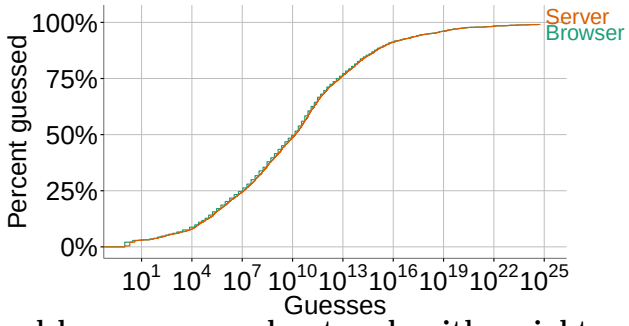
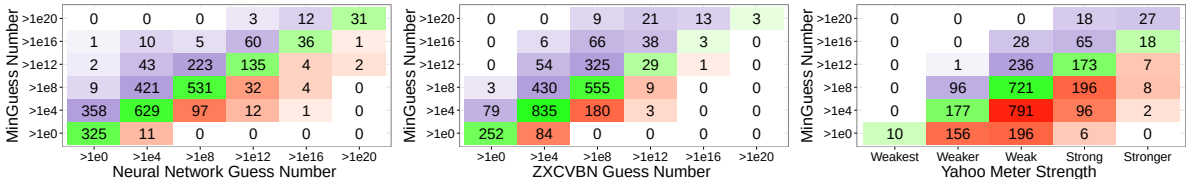


Figure 3.9: Compressed browser neural network with weight and curve quantization compared an unquantized network. *Browser* is our browser network with weight and curve quantization. *Server* is the same small neural network without weight and curve quantization.

explored using the MsgPack binary format [93], but found that after `gzip` compression, there was no benefit for encoding size and minor drawbacks for decoding speed. The effects of different pipeline stages on compression are shown in Table 3.1.

Weight and probability curve quantization Because current methods of calculating guess numbers from probabilities are too slow, taking hours or days to return results, we precompute a mapping from password probability to guess number and send the mapping to the client, as described in Section 2.3. Such a mapping can be efficiently encoded by quantizing the probability-to-guess-number curve. Quantizing the curve incurs safe errors—i.e., we underestimate the strength of passwords. We also quantize the model's parameters in the browser implementation to further decrease the size of the model. Both weight and curve quantization are lossy operations, whose effect on guessing we show in Figure 3.9. Curve quantization manifests in a saw-tooth shape to the guessing curve, but the overall shape of the guessing curve is largely unchanged.

Evaluating feedback speed Despite the large amount of computation necessary for computing a password's guessability, our prototype implementation is efficient enough to give real-time user feedback. In general, feedback quicker than 100 ms is perceived as instantaneous [99]; hence, this was our benchmark. We performed two tests to measure the speed of calculating guess numbers: the first measures the time to produce guess numbers with a



(a) Client-side Neural Network **(b)** `zxcvbn` **(c)** Yahoo!

Figure 3.10: Client-side guess numbers compared to the minimum guess number of all server-side methods. The number in the bin represents the number of passwords in that bin. For example, neural networks rated 358 passwords as being guessed with between 10^0 and 10^4 guesses, while server-side approaches rate them as taking between 10^4 and 10^8 guesses. The test passwords are our 1class8 set. The Yahoo! meter does not provide guess numbers and, as such, has a different x-axis. Overestimates of strength are shown in shades of red, underestimates in shades of purple, and accurate estimates in shades of green. Color intensity rises with the number of passwords in a bin.

		Total	Unsafe
1class8	Neural Network	1311	164
	`zxcvbn`	1331	270
	Yahoo!	1900	984
4class8	Neural Network	1826	115
	`zxcvbn`	1853	231
	Yahoo!	1328	647

Table 3.2: The number of total and unsafe misclassifications for different client-side meters. Because the Yahoo! meter provides different binning, we pre-process its output for fairer comparison, as described in Section 4.3.

semi-cached password; the second computes the total time per password. The semi-cached test measures the time to compute a guess number when adding a character to the end of a password. We believe this is representative of what a user would experience in practice because a user typically creates a password by typing it in character by character.

We perform both tests on a laptop running OSX with a 2.7 GHz i7 processor and using the Chrome web browser (version 48). We randomly selected a subset of 500 passwords from our 1class8 training set for these tests. In the semi-cached test, the average time to compute a guess number is 17 ms (stdev: 4 ms); in the full-password test, the average time is 124 ms (stdev: 48 ms). However, both the semi-cached test and the uncached test perform fast enough to give quick feedback to users.

Comparison to other password meters We compared the accuracy of our client-side neural network implementation to other client-side password-strength estimators. Approx-

imations of password strength can be under- or overestimates. We call overestimates of password strength unsafe errors, since they represent passwords as harder to guess than they actually are. We show that our meter can more precisely measure passwords' resistance to guessing with up to half as many unsafe errors as existing client-side models, which are based on heuristics. Our ground truth for this section is the idealized MinGuess method, described in Section 4.2.

Prior work found nearly all proactive password-strength estimators to be inconsistent and to poorly estimate passwords' resistance to guessing [31]. The most promising estimator was Dropbox's `zxcvbn` meter [160, 161], which relies on hand-crafted heuristics, statistical methods, and plaintext dictionaries as training data to estimate guess numbers. Notably, these plaintext dictionaries are not the same as those used for our training data, limiting our ability to fully generalize from these comparisons. Exploring other ways of configuring `zxcvbn` is beyond the scope of this evaluation. We compare our results to both `zxcvbn` and the Yahoo! meter, which is an example of using far less sophisticated heuristics to estimate password strength.

The Yahoo! meter does not produce guess numbers but bins passwords as weakest, weaker, weak, strong, and stronger. We ignore the semantic values of the bin names, and examine the accuracy with which the meter classified passwords with different guess numbers (as computed by the MinGuess of all guessing methods) into the five bins. To compare the Yahoo! meter to our minimum guess number (Table 3.2), we take the median actual guess number of each bin (e.g., the "weaker" bin) and then map the minimum guess number for each password to the bin that it is closest to on a log scale. For example, in the Yahoo! meter, the guess number of $5.4 \cdot 10^4$ is the median of the "weaker" bin; any password closer to $5.4 \cdot 10^4$ than to the medians of other bins on a log scale we consider as belonging in the "weaker" bin. We intend for this to be an overestimate of the accuracy of the Yahoo! meter. Nonetheless, both our work and prior work [31] find the Yahoo! meter to be less accurate than other approaches, including the `zxcvbn` meter.

We find that our client-side neural network approach is more accurate than the other approaches we test, with up to two times fewer unsafe errors and comparable safe errors, as shown in Figure 3.10 and Table 3.2. Here, we used our neural network meter implementation with the tuning described in Section 2.4. We performed the 1class8 test with the client-side Bloom filter, described in Section 2.3, while the 4class8 test did not use the Bloom filter because it did not significantly impact accuracy. Both tests scale the network output down by a factor of 300 and ignore case to give more conservative guess numbers. We chose the scaling factor to tune the network to make about as many safe errors as `zxcvbn`. In addition, we find that, compared to our neural network implementation, the `zxcvbn` meter's errors are often at very low guess numbers, which can be particularly unsafe. For example, for the 10,000 most likely passwords, `zxcvbn` makes 84 unsafe errors, while our neural network only makes 11 unsafe errors.

Besides being more accurate, we believe the neural network approach is easier to apply to other password policies. The best existing meter, `zxcvbn`, is hand-crafted to target one specific password policy. On the other hand, neural networks enable easy retargeting to other policies simply by retraining.

5 Summary

This chapter describes how to use neural networks to model human-chosen passwords and measure password strength. We show how to build and train neural networks that outperform state-of-the-art password-guessing approaches in efficiency and effectiveness, particularly for non-traditional password policies and at guess numbers above 10^{10}. We also demonstrate how to compress neural network password models so that they can be downloaded as part of a web page. This makes it possible to build client-side password meters that provide a good measure of password strength. In Chapter 6, Section 1, we discuss future work in the area of password guessing models.

Chapter 4

Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting[1]

1 Introduction

In this chapter our goal is to collect ground-truth data about DOM XSS vulnerabilities in order to create a dataset on which to train machine-learning classifiers, described in Chapter 5. We also aim to answer the following questions about DOM XSS as part of an independent effort to detect DOM XSS vulnerabilities using taint tracking. Are DOM XSS vulnerabilities becoming more or less common? How do state-of-the-art methods for detecting DOM XSS vulnerabilities compare? Are web developers learning to avoid such vulnerabilities through good coding practices, for example, using encoding schemes or design patterns such as HTML templating? What are the causes of DOM XSS? Do shared libraries or web-content-generation frameworks propagate DOM XSS vulnerabilities across a large

[1]The majority of this chapter was previously published as: William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In Proc. NDSS, 2018.

number of sites?

To answer these questions, we use a dynamic approach to detect DOM XSS vulnerabilities on the Internet. Prior work showed how to detect DOM XSS vulnerabilities using taint tracking to track flows of attacker-controllable information sources to sensitive sink functions (e.g., `eval` and `document.write`) [79, 105]. The existence of such flows only indicates that data from a source can reach a sink, but does not account for whether the data has been sanitized by the programmer. Thus, once a flow with a potential DOM XSS vulnerability is observed, the flow must be confirmed to be exploitable. In this chapter, we show how to more accurately detect whether a flow that is potentially vulnerable is capable of being exploited. Although an attacker can use several types of sources (e.g., cross-origin messages and cookies), we focus, similarly to prior work [79], on confirming flows from URL-based sources. These are of particular interest because, compared to other flows, they are easy for attackers to exploit.

We used this methodology to detect DOM XSS vulnerabilities on Internet. We crawled the homepages and five random subpages of websites on the Alexa Top 10,000 most popular websites list [2]. Compared to previous work [79], we observed both more flows per web page and determined a higher proportion of those flows to be vulnerable, even when using the same methodology as previous work to determine which flows are vulnerable. Using our improved method for determining which flows are vulnerable, we found 83% more vulnerabilities than by using prior methodology [79]. We believe this indicates that DOM XSS vulnerabilities are becoming more common in the four years since the previous study was undertaken.

In addition, we qualitatively examined the code paths that led to the vulnerabilities. We observed, for example, that most of the vulnerabilities did not share code, implying that the vulnerabilities we found are due to custom code, rather than the inclusion of buggy shared libraries. We also observed errors in the implementation of HTML templating that allowed XSS vulnerabilities. Templating can be an effective way to prevent DOM XSS vulnerabilities, and is similar to using parameterized SQL queries. We found cases where bespoke templated

HTML designs failed to properly encode template values, which attackers could then inject code into.

Finally, using our collected dataset of DOM XSS vulnerabilities, we evaluated static-analysis tools that are designed to detect DOM XSS. In the past, researchers have compared the effectiveness of vulnerability scanners on synthetic datasets [34, 137], whereas we used real-world vulnerabilities. We found that static-analysis tools performed poorly at detecting the vulnerabilities found by the dynamic analysis. However, some static tools were shown to have low false-negative rates and at the same time identify DOM XSS issues not found by the dynamic analysis, suggesting that dynamic analyses and static analyses are finding qualitatively different types of vulnerabilities. Our findings on static-analysis tools suggest that testing using both dynamic and static approaches may be necessary to secure web applications from DOM XSS.

In Section 2, we detail our methodology for crawling the Internet for DOM XSS vulnerabilities, and our improved technique for confirming potentially vulnerable flows. In Section 3 we describe the results of our experiments for detecting DOM XSS vulnerabilities and evaluating static-analysis tools for detecting DOM XSS. We describe the limitations of our work in Section 4. We discuss the implications of our findings in Section 5, and conclude in Section 6.

2 Methodology

Next, we describe the methodology for our experiments to detect DOM XSS vulnerabilities on the Internet. In Section 2.1, we describe how we crawled websites and which web pages we visited. In Section 2.2, we discuss the specifics of the taint-tracking engine we developed. In Section 2.3, we describe how we confirmed vulnerabilities. Finally, we detail the methodology for testing static-analysis tools in Section 2.4.

2.1 Crawling for DOM XSS vulnerabilities

We first crawled the Internet using a browser instrumented to perform taint tracking. The browser collected information about what data flows occurred in the page, and output a log file detailing the flows and the encoding methods applied. Then, for a subset of flows, we tested whether the flow was exploitable by generating example inputs crafted to deliver a payload to the sensitive sink. This methodology builds on the methodology used in prior work for detecting DOM XSS vulnerabilities at scale [79]. We describe the differences from this prior work in this Section and in Section 2.3.

To crawl websites, we started by visiting the Alexa Top 10,000 websites, a list of the globally most popular websites [2]. Then, we collected all the links to other web pages on the home page of each website, and randomly selected five web pages that were hosted on the same domain as the original domain to limit our crawl to a manageable size given our resource constraints. This crawl is broader, but more shallow, than Lekies et al.'s [79], and the difference between the two offers the opportunity for new insights about the incidence of DOM XSS vulnerabilities (see Section 5.1). We also obeyed the `robots.txt` directives, which direct automated programs—robots—as to which pages may be traversed [74]. We automated the process of visiting web pages and extracting the links on a page by developing and using a browser plugin. Whenever any web page did not load correctly—for example, because of a timeout—we attempted to load the same page three times. If the failed page was not the top-level page of a domain, we attempted to load a different web page in the same domain. Crawling occurred during the summer of 2017, roughly four years after Lekies et al.'s work was published [79].

2.2 Dynamic taint analysis

Like prior work [79], we instrumented Chromium to perform byte-precise tracking of the provenance of each byte of strings in JavaScript. We will not focus on the design of our taint-aware browser because it is not a core contribution of our work and the design is

```
document.write(
  '<a href="' + encodeURI(document.location) +
  '">Link</a>');
```

Figure 4.1: Modified example of Figure 2.1, in which the code would have a DOM XSS vulnerability if the encoding function was not applied. In this example, the `encodeURI` function encodes the location so that the double-quote character cannot be injected.

similar to prior work. We have released the source code for our modified, taint-aware version of Chromium and V8, the JavaScript engine used in Chromium (see `https://github.com/wrmelicher/ChromiumTaintTracking`).

To summarize the design of the taint-aware browser: We first allocated space in each JavaScript string primitive for a one-byte taint value that stores the provenance of each byte of the string. This allows taint information to be precisely propagated during string concatenation or slicing. In addition to the provenance of each string byte, each bookkeeping byte also records which built-in encoding methods have been applied. For example, using the `encodeURIComponent` JavaScript function will modify the taint information to reflect that the string has been encoded using the `encodeURIComponent` function. During taint propagation, matching encoding-decoding pairs will cancel each other. For example, if a string is encoded using `encodeURIComponent` and later decoded using `decodeURIComponent`, then the string will be identified as having no encoding applied. The taint information is only stored for string types and not for arbitrary JavaScript objects. This prevents tracking across different data types: for example, parsing a string into an integer and then writing the integer to a string would remove all taint information.

Our browser checks the arguments of sensitive functions (e.g., the `eval` function or `document.write`; see the Table 4.2 for an exhaustive list) for tainted bytes. If an argument contains tainted bytes, then a record is written to a log file describing the flow, including: the type of taint, the locations of the tainted bytes, the sensitive sink function, and a stack trace. Afterwards, we analyze the logs to determine which flows are *potentially* vulnerable to DOM XSS attacks and which flows are not.

- `document.write`, and `document.writeln`
- Assignment to the `src` attribute of a script, embed, iframe, or img. Includes JavaScript assignment (`element.src` = "..."), and assignment using `setAttribute`.
- Assignment to the `href` attribute of a anchor element. Includes JavaScript assignment and `setAttribute`.
- `eval`
- Assignment to the inner text of a `script` node.
- Implicit string-to-function conversion inside `setTimeout` and `setInterval`
- Assignment to `innerHTML`, and `outerHTML`, and `insertAdjacentHTML` properties
- Assignment to `document.cookie`
- Assignment to `document.location`
- Assignment to the `style` attribute. Includes JavaScript assignment and `setAttribute`.
- Assignment to all event handler attributes. Includes JavaScript assignment and `setAttribute`.

Figure 4.2: List of instrumented sink functions

Whether a flow is vulnerable depends on the context of the injection in the HTML or JavaScript, the encoding functions that have been applied, and the source and sink types. For example, if we detect that a tainted value is not encoded and begins in the context of an HTML double-quoted attribute, then that flow is potentially vulnerable. However, if the string is encoded using the `encodeURIComponent` built-in function, then a double quoted attribute is not vulnerable because the `encodeURIComponent` function encodes the double quote as "%22". Figure 4.1 shows code that would have a DOM XSS vulnerability if an encoding function was not applied. This list of potentially vulnerable flows is then tested to decide whether the flow is actually vulnerable to XSS attacks using a process we describe in Section 2.3. One example of a potentially vulnerable flow that is not actually vulnerable to DOM XSS attacks is when the application performs custom sanitization of inputs that is not detected by the taint-tracking engine—for example, by halting execution if the input does not match a certain form that is known to be safe. The log files also contain the stack trace for the sink call of each flow. In addition to making the flow more repeatable for post-hoc

Method A: injection at end of URL
Observed URL:
`example.url.com/path?param=test&a=b`
Generated injection URL:
`example.url.com/path?param=test&a=b#INJECT`

Method B: injection into parameter
Observed URL:
`example.url.com/path?param=test&a=b`
Observed `eval`-ed string:
`var a = 'test';`
Observed taint location:
The 9th through 13th bytes of the string—starting with the first 't' in test and ending with the last 't' in test.
Generated injection URL:
`example.url.com/path?a=b#&param=INJECT`

Figure 4.3: Explanation of injection methods using an artificial example. In Method A, the injection is inserted at the end of the string. In Method B, we attempt to insert the injection into the parameter value that matches the tainted string in the text of the observed argument to the sensitive sink. **INJECT** marks the point of injection.

manual analysis, this allows us to examine the code path that led to the vulnerability (see Section 3.1).

2.3 Attack confirmation

By crawling web pages using the taint-aware browser described in Section 2.2, we generate a list of potentially vulnerable flows. We then simulate an exploit to test those potentially vulnerable flows to decide whether they are actually vulnerable. We experimented with two methods of automatically crafting injections to test: one used by prior work, which appends the injection to the end of the string [79]; and a novel method that attempts to more accurately pinpoint the specific bytes of the string in which to inject a payload. For the purposes of automatically crafting injections we limited ourselves to URL-based sources (e.g., the `document.location.href` object and derivatives like `document.location.search`, `document.location.hash`, etc.). Those types of potential vulnerabilities are straightforward to generate potential exploits for, and therefore can be

easily verified to be actual vulnerabilities. For the same reasons, they are the flows commonly targeted by attackers [103].

The first method (termed method A), used in prior work [79], appended the exploit to the end of the URL. The new method (termed method B), which more accurately pinpoints where in the string to inject the payload, attempts to insert the exploit into the bytes of the source string that match the tainted bytes in the sink. An example is shown in Figure 4.3. The log files contain the information about which bytes of the string are tainted and the semantic source label for those bytes (e.g., from the URL). Therefore, we can infer which bytes of the source will make their way into the sink by examining the string that is injected into the sink and comparing it to the source string. The insight behind this method comes from the observation that many of the values injected into sinks are values of parameters provided in the URL. Our method is designed to capture URL parsing in client-side code. It is relatively commonplace for JavaScript code to manually parse query parameters on the client, for example, by parsing the URL looking for the special characters that signal parameters: ?, &, and =. In this way, the URL is often used to pass parameters to other links or to control the display of the web page. While this method of confirming that a flow is vulnerable is extremely simple, in practice we find the combination of both methods to generate 83% more exploits than just the first method (method A).

To test candidate exploits purely in the browser, i.e., without affecting the website, we limit our candidate exploits to the part of the URL string after the hash (the '#' character), as this segment of the URL string is not sent to the website hosting the page, but only processed internally by the JavaScript running in the browser.

We also did not craft actual valid HTML and JavaScript exploits for attack confirmation, but rather crafted a unique string that included characters necessary for an exploit (e.g., the single quote character if injecting a value into a single quoted HTML attribute). In our payload, we injected the string `marker<>'"` and then examined our sink injection log files for this string.

We believe that avoiding the use of valid HTML and JavaScript in simulated exploits and targeting only the portion of the URL string after the hash—beyond limiting risk to web servers—leads to simulated exploits that are both easier to generate and less likely to be caught by client or server-side filters. Such filters are notorious for being easily bypassable by humans [10, 67]. However, for an automated injection, we wanted our approach to scale to many websites and detect when an exploit could likely be crafted, instead of being filtered by an easily bypassable defense mechanism. To confirm that a flow was vulnerable to DOM XSS attacks, we searched the logs for the unique injection string in the output. To confirm that our methodology did not yield false positives, we randomly sampled 40 flows that our process flagged as vulnerable and manually developed a working exploit. We found that all 40 instances were vulnerable; therefore, we believe that the vast majority of cases found by our automated method were actual vulnerabilities.

After confirming vulnerabilities, we qualitatively examined a subset of these vulnerabilities for insights into the root cases of DOM XSS vulnerabilities. For each vulnerability that we manually analyzed, a researcher manually reproduced the vulnerability based on the saved stack trace in our log files. Then, we distilled the vulnerability to a small amount of code that could describe the flow of data in the vulnerability. These code snippets were then analyzed to extract the themes common to vulnerabilities. We classified vulnerabilities by complexity and also noted other interesting aspects of the code that had the vulnerability. Our results for this analysis are presented in Section 3.3.

2.4 Static analysis

After we collected a list of confirmed DOM XSS vulnerabilities, one of the analyses we performed was to evaluate the effectiveness of static-analysis tools to detect these vulnerabilities. We sampled our dataset in two ways to create test sets to evaluate the false-positive rate and the rate with which the tested static-analysis tools detect these vulnerabilities. First, we sampled websites that have known vulnerabilities from our dataset of confirmed DOM

		Sinks														
		Anchor src	Cookie	CSS	CSS style attribute	Embed src	HTML	Iframe src	IMG src	JavaScript	Event handler	set-Timeout	Location	Script src	**Total**
Sources	Cookie	11,269	256,784	297	297	0	61,164	2,098	115,363	20,469	114	28	582	50,176	518,641
	Message	16,704	18,373	311	311	0	20,974	3,475	70,517	1,182,456	98	73	535	24,393	1,338,220
	Multiple	4	0	0	0	0	9	3	35	0	0	0	0	15	66
	Referrer	62,476	3,670	31	31	0	55,796	3,657	42,193	645	11	11	537	16,659	185,717
	Storage	11,023	4,590	112	112	0	3,712	396	7,146	3,541	9	1	23	9,494	40,159
	URL	226,214	31,150	418	418	15	237,714	137,364	193,200	2,446	914	140	2,711	238,354	1,071,058
	URL hash	1,601	171	2	2	0	1,938	148	2,322	173	0	101	33	2,400	8,891
	URL host	3,383	116,967	19	19	0	17,147	10,035	25,394	389	6	3	308	5,716	179,386
	URL hostname	21,494	612,759	127	127	0	44,903	24,761	104,664	1,001	269	74	400	16,218	826,797
	URL origin	21,225	46	1	1	0	1,801	47,887	3,273	336	0	2	64	1,762	76,398
	URL pathname	20,235	9,807	15	15	0	3,913	1,301	102,945	1,457	628	12	193	13,326	153,847
	URL search	4,549	2,922	0	0	0	5,665	474	13,425	63	0	0	48	2,759	29,905
	URL port	0	0	0	0	0	0	0	2	0	0	0	0	0	2
	URL protocol	82,953	661	92	92	1	94,538	20,746	152,501	123	11	33	356	72,075	424,182
	window.name	2,109	4,504	8	8	0	24,845	160	3,826	12,621	0	3	67	2,374	50,525
	Total	485,239	1,062,404	1,433	1,433	16	574,119	252,505	836,806	1,225,720	2,060	481	5,857	455,721	

Table 4.1: Source-to-sink flow counts for different source-sink pairs. Rows in the table are sources and columns are sinks. We focus on the shaded columns and rows in this work. "Cookie" as a sink means assignment to the `document.cookie` object; as a source means data originating from `document.cookie`. "Location" refers to assignment to `document.location`.

XSS vulnerabilities, found using methodology described in Section 2.3. Then, we sampled from all websites that we visited to measure the false-positive rate. Note that for measuring the false-positive rate, we sampled from all websites, not only from websites where we did not detect a vulnerability. We sampled in this way so that our sampling would not be biased towards sites that might be less buggy. Sampling from our dataset of known vulnerabilities, rather than using manufactured vulnerabilities, has the benefit that we are using real-world bugs.

Description of static-analysis tools We evaluated three tools for detecting DOM XSS vulnerabilities: ScanJS [122], esflow [41], and the static-analysis tools in Burp Suite Pro [126]. We also attempted to test jsprime [108], but were unable to get it to work without crashing. We focused on open-source or inexpensive proprietary tools that statically detect DOM XSS vulnerabilities. There are variety of other, more expensive proprietary vulnerability scanning tools, including: IBM Security AppScan, Acunetix, Trustwave App Scanner, Retina web application scanner, Qualys web inspect, HP Fortify static code analyzer, and Coverity's JavaScript scanner. However, for our application of scanning a large number of domains,

these were prohibitively expensive. Prior work has compared a wide variety of these proprietary tools for general purpose vulnerability detection (i.e., not restricted to DOM XSS vulnerabilities), and found them to have comparable error rates to each other [137].

The static-analysis tools that we chose appear to have different tradeoffs. ScanJS is a tool meant to help people avoid coding practices that lead to, among other things, DOM XSS vulnerabilities. As such, it flags code that could be unsafe without aiming to identify whether the code leads to an exploitable bug. For example, it may point out all locations where the `document.write` function was used with a non-static string as an argument. While this is a good practice to avoid, it is not always indicative of a vulnerability. In fact, the majority of cases are benign. Burp Suite attaches a confidence rating to each potential vulnerability that it flags, giving guidance about which findings are most reliable. Burp Suite also receives code from the website by acting as a proxy between the browser and the website, meaning that it has access to code that is dynamically loaded (e.g., by a <`script`> tag added during execution) unlike the other tools; however, it still is not able to analyze code that is dynamically *generated* (e.g., by using the `eval` function). Esflow is unique in that it often attaches source and sink information to its issue reports for easier debugging.

3 Results

We used the taint-tracking and crawling methodology described in Section 2 to collect a dataset of tainted flows. We visited 44,722 web pages, which had in total 319,481 frames. One would expect that trying to visit five subpages on each domain, we would have visited 60,000 web pages: 10,000 top level pages and 50,000 subpages. However, we skipped loading 1,761 web pages due to `robots.txt` directives; and we were unable to load 4,094 web pages after three attempts due to timeouts, 462 because Chromium would not load the page (most often due to SSL warnings), and 26 because Chromium crashed when rendering them. Some of the pages unable to be loaded were top-level pages; in that case we also did not visit other

pages on that domain.

We describe how we detected DOM XSS vulnerabilities using our dynamic analyses in Section 3.1. Then, in Section 3.2, we use the results from our dynamic analysis to evaluate different static-analysis tools for detecting DOM XSS vulnerabilities. Finally, in Section 3.3, we describe the qualitative trends that we observed from manually analyzing a sample of our dataset.

3.1 DOM XSS vulnerabilities detected using dynamic analysis

After crawling our set of web pages, we post-processed the generated taint-tracking logs to generate a list of observed data flows. Each flow has a source, through which an attacker could inject code, and a sink, a sensitive function that consumes data derived from the source of the flow. We tracked flows that have sources that could be potentially manipulated by an attacker, and sinks that could potentially execute JavaScript, including functions that directly execute JavaScript (e.g., `eval`), functions that inject HTML (e.g., assigning to innerHTML or calling `document.write`), and JavaScript event handlers. A summary of the sources and sinks that we tracked can be found in Table 4.1.

Overall, visiting 44,722 pages resulted in 4,140,873 detected flows. We focus on flows with URL sources and HTML or JavaScript sinks, as these are the most straightforward to exploit. Consistently with that, research has generally focused on examining this subset of flows or found it to account for the majority of exploitable flows (e.g., [79, 107]).

Other flows have preconditions that make automatically exploiting them more difficult at scale. For example, to exploit a cookie flow, an attacker must find a way to manipulate the victim's cookies; for message flows, an attacker must find a potential flow whose code does not check the message origin and also send the message at the proper time, when the receiver is expecting it. With the exception of message flows, URL-based sources account for the largest number of flows to sinks that can execute arbitrary JavaScript (HTML and JavaScript sinks). Hence, these are the flows we analyze, and we show that they lead to

Step #		# this work	# 25m flows [79]	as % of total flows this work	as % of total flows 25m flows [79]
	Seed domains	10,000	5,000		
	Web pages	44,722	504,275		
	Frames	319,481	4,358,031		
1	Total flows	4,140,873	24,474,306		
2	URL*, referrer, window.name sources to JS, HTML sinks	363,034	1,825,598	8.77%	7.46%
	URL* sources to JS, HTML sinks	285,147	‡	6.89%	
3	Flows from step 2 excluding those blocked by encoding methods	97,924	313,794	2.36%	1.28%
4	Flows from step 3 excluding those blocked by natural encoding in Chromium	93,481	‡	2.26%	
5	Flows from step 4 including only URL-based sources	54,954	‡	1.33%	
6	Unique† flows from step 5	5,217	‡	0.13%	
7a	Unique† vulnerabilities from step 6 after exploit step using method A§	1,754	6,167‡‡	0.04%	0.03%
7b	Unique† vulnerabilities from step 6 after exploit step using method A and B††	3,219		0.08%	
	Vulnerable iframe URLs	4,668	‡		
	Vulnerable domains	364	480		
	Unique vulnerabilities as percent of pages visited using method A	4%	1.2%		
	Unique vulnerabilities as percent of pages visited using method A + B	7.3%			

Table 4.2: Break down of flows comparing replication of prior work with the same methodology [79]. *) Excludes the JavaScript `location.protocol` property as it is not readily exploitable. †) Applying the uniqueness filter of hosting domain, code location, breakout sequence. ‡) not reported in that work. §) Method A appends the injection to the end of the source string. ††) Method B inserts the injection into the bytes of the source string which match the tainted bytes in the sink after encodings and decodings have been applied. ‡‡) Includes flows from `window.name` sources because that work includes those exploits.

many instances of DOM XSS vulnerabilities. Of the 4,140,873 flows we detected, 285,147 (7%) had a URL-based source.

Confirmed vulnerable flows We determine whether a tainted flow is vulnerable as follows. We first discard flows in which the tainted value is encoded using a built-in encoding method, for example, the `encodeURIComponent` function; we are certain that such flows would ordinarily not be exploitable. This eliminates 66% of the flows we focus on (URL sources to JavaScript or HTML sinks). We next remove from consideration flows that could not be exploited in Chromium due to Chromium's natural encoding of some URL variables (for example, Chromium automatically encodes the content of `document.location.search` to prevent the occurrence of any character that would not be allowed in a URL). After removing those types of flows, we determine which of the remaining 1.33% (54,954 flows) of flows are actually vulnerable by attempting injections. A summary of the number of flows removed at each stage compared to previous work with similar methodology [79] can be seen

Method	# of unique vulnerabilities
Only injection at end	715
Only injection in key-value pair	1,465
Both methods	1,039
Total	3,219

Table 4.3: Summary of the injection methods used to confirm different vulnerabilities. "Only at end" refers to the injection method that inserts the injection at the end of the source (Method A). "Only key-value pair" refers to the injection that inserts the injection in the value of a tainted query key-value pair (Method B). "Both methods" refers to cases where either method would have identified the flow.

in Table 4.2.

In our taint-tracking system, we specially mark flows that have multiple, incompatible encodings with a flag that represents the use of multiple encodings, but not which specific encodings or in what order. Such flows accounted for 2% of flows overall. While we did not attempt to determine whether these flows were actually vulnerable, there were 716 unique flows of this type that may have been potentially vulnerable (i.e., that could have been included in row 6 of Table II). If they had been included, they would account for 12% of potentially vulnerable flows.

We used two methods to confirm vulnerabilities—each described in Section 2.3—based on where to insert the injected payload: inserting the payload at the end of the URL or inserting the payload into the key-value pair from which we observed a flow to a sink. We found that 45% of the confirmed vulnerabilities we detected were due to flows from key-value pairs, 22% of the vulnerabilities were only the result of inserting the payload at the end of the source, and 32% of the vulnerabilities were observed to work with both methods.[2] Table 4.3 shows the breakdown of how many of the 3,219 unique vulnerabilities came from which injection method. Both methods of injection work in cases where the entire URL is concatenated with markup (i.e., `document.location`, rather than a specific substring, is included in markup). Our key-value pair injection method identifies vulnerabilities that involve parsing URL parameters, while inserting the injection at the end identifies vulnerabilities in which part of

[2]Numbers do not add up to 100% due to rounding.

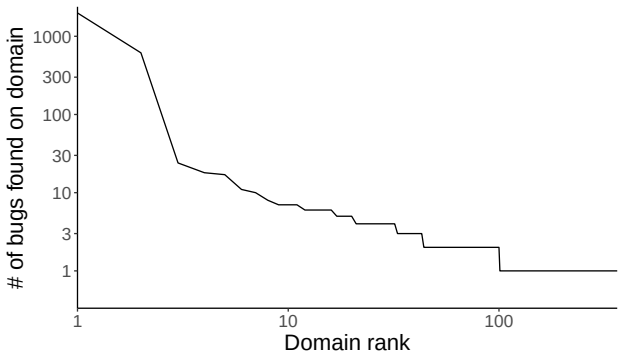


Figure 4.4: Distribution of unique vulnerabilities across domains. The y-axis shows the number of unique vulnerabilities found on a particular domain in log scale. The x-axis shows domains sorted by frequency; for example, ten on the axis shows the domain with the 10th most vulnerabilities. For example, the domain with the most vulnerabilities had nearly 1,987 unique vulnerabilities.

the path or URL besides the URL parameters is used as part of the parameter to the sink.

To count unique vulnerabilities, we removed duplicates using the same method used as Lekies et al. [79]: unique bugs are identified by their domain, their location in the script, and the context (e.g., inside a double-quoted attribute or the name of an element attribute) of the tainted section of the string argument to the sensitive sink.

We also computed the number of unique vulnerabilities across different domains, as shown in Figure 4.4. We found that the majority of vulnerabilities come from a handful of domains, and that many domains had only a few unique vulnerabilities or one vulnerability: the ten domains with the most vulnerabilities had in total 2703 unique vulnerabilities; the remaining 354 domains accounted for the remaining 516 vulnerabilities.

Interestingly, when performing the vulnerability confirmation crawl we observed vulnerabilities in six iframe URLs that were not previously seen in our first crawl. These iframe URLs were part of the confirmation crawl because either they or the top-level pages that included them had previously been marked as potentially containing a vulnerability. The difference in time between collecting data and confirming vulnerabilities was nine days.

Vulnerability attribution by domain and domain category We next attempted to shed light on the cause of the vulnerabilities that we observed by examining where they occurred. Were they due to third-party scripts, old versions of popular libraries, custom

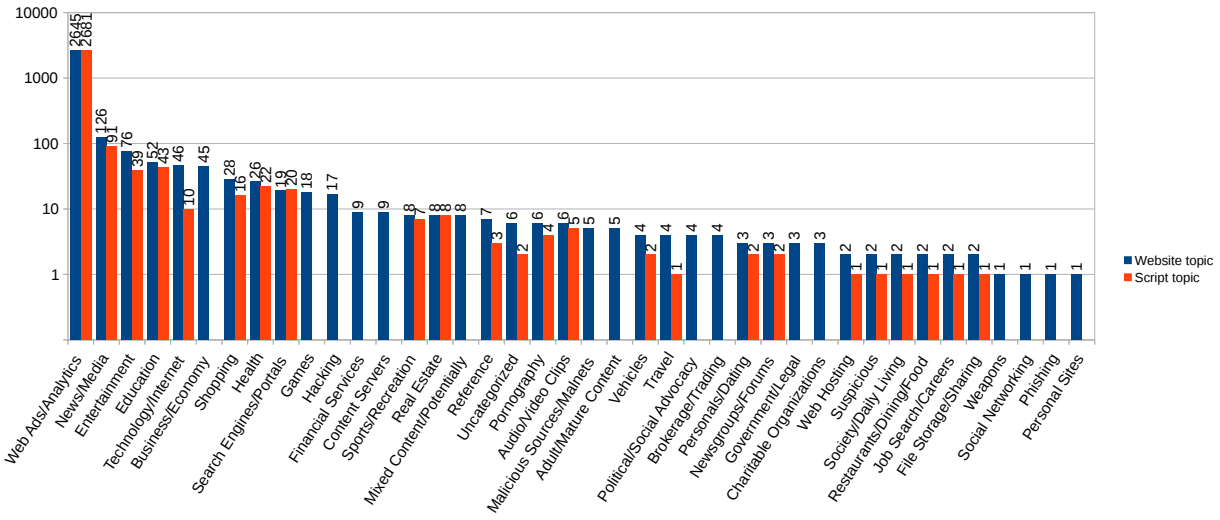


Figure 4.5: The count of URL domains and script domains in different website categories. The bar height shows the number of script domains or frame domains with the corresponding category in our dataset of unique vulnerabilities. The y-axis is in logarithic scale. Script domains are the domains that the vulnerable scripts were hosted on. Frame domains are the domain of the frame where the vulnerability was located. Note: the numbers for script domains do not add to 3,219 because some sinks did not have a script URL. This may happen when the sink location is in dynamically generated code.

code for each website, or other causes? For this measurement, we used the URL of the frame where the vulnerability was found, since this is the context in which an attacker would be able to execute JavaScript, rather than using the URL of the top-level frame. We used the location of the sink as a starting point for determining to which entity to attribute the vulnerability. In particular, we examined the distribution of vulnerabilities in three ways: (1) the domain on the iframe in which the script executed; (2) the domain on which the script was hosted (web pages often import scripts from other domains); and (3) the domain of the top-level page that the user was visiting. Rather than reporting results about individual domains, we report them by the topic category of the domains. We use the Blue Coat K9 classification of domains into topics [12] for this purpose.

We found that the vast majority, 2,645 of 3,219 of our unique vulnerabilities (82%), were found to execute inside iframes with domains that were known to serve web advertisements or perform analytics. Other domain types that accounted for many vulnerabilities included shopping and news.

We also analyzed what type of domains hosted the scripts in which we found vulnerabilities. Similarly to the above result, we found that 2,681 of 3,219 vulnerabilities (83%) were in scripts hosted on advertising and analytics domains. Figure 4.5 shows the analysis of the types of script domains and website domains with confirmed vulnerabilities. For this measurement, we used the domain of the script where the sink function call was found. While advertising domains were the most popular source of vulnerable scripts, our data-collection infrastructure did not capture enough information to similarly categorize scripts that did not have potentially vulnerable flows. Hence, while we can report that, in web pages that had at least one flow, 38% of the time the flow originated in a script that was categorized as an advertising script, we cannot determine whether the fraction of advertising-domain scripts that was vulnerable was greater than the fraction of scripts from other domain categories.

We matched the unique vulnerabilities that we found with the top-level web pages that contained those vulnerabilities in our dataset. Many of the vulnerabilities that we found were on subframes of other web pages, and we wanted to understand how much exposure users would have if they visited the top-level pages in our dataset. Table 4.4 shows the categories of these top-level web pages. Note that there are significantly more data points than unique vulnerabilities because some vulnerabilities were present in web pages that were subframes of multiple web pages. In contrast to Figure 4.5, where the most popular category was web ads and analytics, here the popular topics are news/media (27.7%) and entertainment (12.9%).

In total, for 282 (8.8%) of the vulnerabilities we found the domain the script was hosted on was different from the domain of the iframe in which the script executed. This suggests that while a non-trivial fraction (8.8%) of vulnerabilities may be caused by developers relying on third-party scripts, the vast majority of vulnerabilities are in the developers' own scripts (or at least scripts hosted locally on their domains). The fraction is smaller than reported in prior work, which found that 22% of code attributable purely to an error by a third party [135]. This difference could be the result of our different methodology for confirming DOM XSS vulnerabilities.

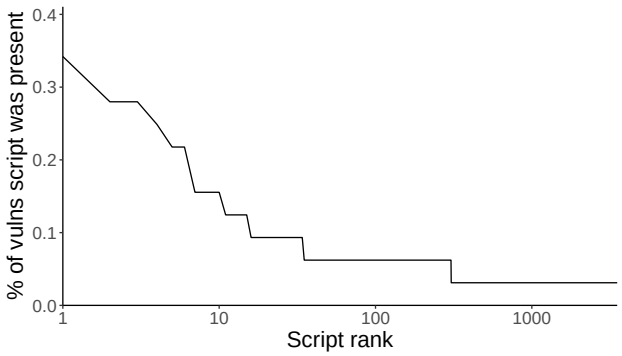


Figure 4.6: The percent of stack traces from the dataset of unique vulnerabilities that scripts were found in. The x-axis shows the rank in log scale of the scripts in a sorted list; for example, 10 shows the script that was 10th most frequently present in stack traces. The y-axis is the percent of script URLs that were less than that rank. For example, the first script was present in the stack traces of 0.34% (11 of 3,219) unique vulnerabilities.

Vulnerability attribution by script We additionally examined the scripts in the entire stack trace for each vulnerability. The goal of this analysis was to determine whether some scripts occurred in stack traces of vulnerabilities particularly often. Such scripts could be good candidates for adding encoding functions, or for other remediation, as that would prevent many vulnerabilities. At the same time, if some scripts occur in many stack traces of vulnerabilities, this could indicate that developers misunderstand how to correctly use that script.

For our 3,219 confirmed vulnerabilities, we identified the scripts in the stack trace of each vulnerability, and then counted how many stack traces each script was present in. For this analysis we removed jQuery from our results because many websites use various HTML rendering functions (e.g., the `html` or `append` jQuery methods) that are working as intended, but misused by the caller. Hence, we removed any script name that contained the string "jquery" without respect to character case.

We found that the majority of vulnerable scripts was present in only one unique vulnerability stack trace—implying that the causes of vulnerabilities are unique. Figure 4.6 shows the percentage of stack traces that each script is seen in.

Vulnerabilities in commonly blocked content Internet folklore has often claimed that ad blocking software protects your computer from XSS vulnerabilities common in ad networks [69]. Because we found that many of both the target web pages and the vulnerable scripts were located on domains that hosted advertising, we tested how much protection a normal user who uses an ad blocker would have from such vulnerabilities. For this analysis, we used the adblockparser Python library [118] to simulate what scripts would not be executed if the user was running ad blocking software that obeyed the rules defined in the Adblock EasyList, a popular rule list of advertising content to block [38]. We counted a vulnerability as being blocked if the Adblock EasyList would block either the script or the entire target URL when the target URL is loaded inside an iframe and not as the main frame. We found that, of the 3,219 unique vulnerabilities, 2,039 (63%) would have been blocked by this simulated configuration of Adblock.

3.2 Effectiveness of static-analysis tools

We next examine whether the vulnerabilities we found could be detected at development time using off-the-shelf static-analysis tools. We find that most could not, although static-analysis tools sometimes found additional bugs.

More specifically, with our dataset of confirmed vulnerabilities 3.1, we tested static-analysis tools to evaluate their ability to find the same vulnerabilities that the dynamic analysis found. To target JavaScript, dynamic analysis traditionally is seen as having fewer false positives [104]; however, static analysis is often more helpful for programmers during development because of the lack of customized analysis for adding new code—developers can set up a static-analysis toolchain once to automatically check new code. In addition, static analysis can be more complete—able to detect vulnerabilities in code that was not executed on a particular run of the program. The majority of the vulnerabilities that were caught in our experiment only by static-analysis tools were in this category.

Overlapping vulnerabilities We compared the rates at which different tools, described in Section 2.4, found the vulnerabilities that we had previously compiled using the dynamic analysis. We found that the tools we tested usually failed to detect the DOM XSS vulnerabilities from our dataset. The full results comparing different tools are provided in Table 4.5. Notably, while Burp Suite, the most promising tool, had a low rate of finding the same errors as the dynamic analysis, it pointed out many potential issues not found by the dynamic analysis. We next describe these findings in more detail.

We randomly sampled 50 of the 3,219 unique bugs we found using our dynamic analysis. Then, for the two tools that scanned JavaScript code, we downloaded the web page where the vulnerability was located, and all the accompanying JavaScript, and used the downloaded JavaScript as input to the tools. We included any JavaScript embedded in HTML in addition to externally loaded JavaScript. Because the tool was examining the web page statically, we were not able to include JavaScript that is dynamically loaded during the execution of the page. For Burp Suite, we loaded the web page in the Chromium web browser and connected Burp Suite to the browser via a proxy. In this way, all the JavaScript requested by the browser was analyzed by Burp Suite's static-analysis tool. Therefore, it was able to access external scripts that were dynamically loaded during execution.

We counted a tool as successfully identifing one of these vulnerabilities if it output any error message related to DOM XSS referring to the code where the sink for the vulnerability was located. For Burp Suite and esflow, we manually reviewed all messages and counted how many referred to the exact line number and character offset of the sink that was vulnerable. For ScanJS, we counted messages that referred to the same script location (line number and script name) as the vulnerable sink, provided that they warned of a potential XSS vulnerability. We explicitly terminated the analysis of any program that took longer than one hour. This affected four of the 50 web pages tested with esflow. In practice, we believe that this is realistic in that static-analysis tools must give results within a reasonable time to be useful to developers. We do not count as matches cases in which the tool detects a

vulnerability that was not a part of our dataset.

False positive rates We compared the false positive rates of the tools by sampling tool output and manually deciding whether the particular snippets of code that the tool flagged could be exploitable. We found Burp Suite to have no false positives, while the other tools had many false positives. Table 4.6 shows the results for all tools.

We randomly sampled 50 out of all the URLs that we visited in our dataset. We randomly sampled from all URLs, and not just URLs for which we did not confirm vulnerabilities, so that our results for false positive rates would not be biased towards websites that are more secure, and therefore more likely to have false positives than true positives. We then analyzed the scripts on those pages with each of the tested static-analysis tools. Each tool would report findings on the pages related to DOM XSS. We randomly sampled 20 of those findings, except for esflow, which only reported 19 findings. We then manually examined each finding, and the piece of code that it referred to, to determine whether that piece of code could be exploitable. In doing so we aimed to simulate how a developer assessing the same code and reviewing the output of the tool would categorize the bug.

Our guiding criteria for manually counting true and false positives in the tools' output was to look for flows from exploitable sources (e.g., URLs, cross-origin messages) to exploitable sinks (e.g., `document.write`, `eval`) without encoding that would render the flow benign. Thus, we counted flows as true positives even if the identified block of code was not executed during the page load during which our dynamic analysis detected a vulnerability. This may happen because the function is dead code, or because that particular page load did not happen to execute the vulnerable function. We believe this method of measuring false positives is a conservative estimate, because in reality some of the identified vulnerabilities may be in dead code. We also did not include findings that were not related to DOM XSS vulnerabilities—for example, warnings about bad coding practices—as either false positives or false negatives. We aimed to measure the number of actionable vulnerabilities that could

be detected by our tested static-analysis tools.

Deciding whether a "bug" reported by a static-analysis tool is exploitable was a judgment call. We believe that any bias in the judgment is likely to be biased towards marking something as non-exploitable when it could be exploited, because it is easier to show how a piece of code might be dangerous, but much harder to confirm that code is safe in all cases. In practice, we believe this closely matches how an engineer reading the output of such a tool would label the output.

3.3 Qualitative trends in DOM XSS vulnerabilities

To gain greater insight into the causes that give rise to DOM XSS vulnerabilities, we manually, qualitatively analyzed two subsets of the vulnerabilities detected by our dynamic analysis. First, we randomly selected from the unique vulnerabilities; however, we noticed that a large portion of these vulnerabilities was semantically very similar, despite being unique bugs according to our uniqueness criteria (script location, hosting domain, and context; also used in previous work [79]). Therefore, we also selected a separate subsample of vulnerabilities, in which we first randomly selected 20 domains on which vulnerabilities had been found, and then selected a random vulnerability on each domain. This allowed us to get a sample that is conceptually more representative of the types of bugs that occur across different domains, and hence of the types of problems that are likely to be encountered by different organizations. A summary of the trends we observed is located in Table 4.7.

Vulnerability complexity First, we found that some bugs were extremely simple, such as concatenating the entire URL into an HTML or JavaScript execution function. Examples of why this happened were creating a form where the form submission attached a return URL for the current page, or passing the web page's URL as a query parameter for the source of an iframe. In addition to simple concatenation, we also found cases where the URL was stored in a non-local variable that could be assigned to in code that was far away from the

sink (e.g., in a different file or function).

Failed mitigation behaviors In addition, for eight of the forty vulnerabilities the relevant code was more complex and spanned multiple functions. Three of those eight utilized custom template processing that did not perform encoding based on the context in the template, resulting in insecure templating code. Another vulnerability was due to an attempt to perform custom, but highly incomplete, filtering—removing all instances of <`script`> tags, but still leaving open many other ways for an exploit to occur, for example, by using event handler code like <`img src=x onerror='INJECTION'/`>. For two other vulnerabilities, code involved in the flow was dynamically generated using the `eval` function, meaning that such code would typically not be visible to static-analysis tools.

We did not observe any failed attempts to use custom encoding functions. Combined with the fact that many of the bugs were shallow—a finding echoed by [135]—this suggests that perhaps engineers were not aware that URLs could contain characters that could be used to inject markup.

When manually reproducing vulnerabilities, we also observed cases where complex control-flow paths must be followed to execute the vulnerable piece of code. For example, we could not reproduce one specific vulnerability until realizing that the vulnerable code was only executed if the screen width was larger than 1,024 pixels, as it had been in our original data collection. In another case, the vulnerable section of code was executed on some page loads but not on others. We discuss code coverage further in Section 5.

4 Limitations

Despite being less straightforward to automatically exploit in the context of a live website, other types of flows besides the ones we focused on (URL to HTML and JavaScript flows) may also be vulnerable. For example, we observed (during manual analysis) a flow from a cookie source to an HTML sink that could be exploitable by a second flow from a URL source

into a cookie sink on the same web page. The page could be exploited by crafting a special URL, from which content would flow through the document's cookie into the HTML sink. Other work has observed XSS vulnerabilities that derive from cookie sources and could be exploitable by web attackers [168]. Vulnerabilities that exploit the JavaScript `postMessage` API have also been reported [129]. Location sinks can be leveraged to create more potent phishing websites, in which an attacker may craft a URL that points to the target website but is redirected to a phishing website via assignment to `document.location` in JavaScript. A victim might assume that they were on a benign website because the hostname in the URL they clicked on was benign. We speculate that these other types of flows might be similar to the types of flows we study here and would be a good avenue for future work.

We sampled only a subset of the web pages on the Internet, and on the pages we sampled, we did not exercise much dynamic functionality—for example, by clicking on web-page elements or entering text—nor were we able to visit web pages behind log-in barriers. On one hand, this allowed our analysis to scale to large numbers of vulnerabilities and websites, but on the other, the vulnerabilities we detected may not be representative of all vulnerabilities. Nonetheless, we found many vulnerabilities through our analyses. Our manual analysis of vulnerabilities may also exhibit similar biases: We performed a more in-depth analysis only of a subset of our results. This subset was by necessity small so that it would be feasible to manually analyze. We do not suggest that the examined vulnerabilities are representative of our dataset, but we analyzed them in depth to give greater insight into at least some vulnerabilities.

Due to slight differences in methodology, the comparison of our results to previous work may not be perfectly accurate. Differences in results may be due to implementation differences, although for the parts of methodology that are shared between our work and Lekies et al.'s [79], we tried to reproduce previous methodology faithfully.

5 Discussion

We performed this study to measure the prevalence of DOM XSS vulnerabilities, evaluate and inform the design of static-analysis tools, and assess the viability of other methods for preventing DOM XSS vulnerabilities. We first discuss how the raw results of our measurement study compare to previous work that used similar methodology to measure XSS vulnerabilities (Section 5.1), teasing out which differences are the result of methodology and which reflect a change in the prevalence of DOM XSS vulnerabilities. We then leverage our quantitative and qualitative analyses of the sources and nature of DOM XSS vulnerabilities to discuss the weaknesses of some suggested countermeasures (Section 5.2). Finally, we further interpret the results of our examination of static-analysis tools and suggest how these tools could be improved to catch more DOM XSS vulnerabilities (Section 5.3).

5.1 Comparing measurements on DOM XSS vulnerabilities

Our methodology for detecting DOM XSS vulnerabilities replicates and builds on Lekies et al.'s [79]. We extend Lekies et al.'s methodology by adding another method for determining whether a bug is exploitable, namely, inserting a potential exploit into query key-value pairs rather than just at the end of the URL. When inserting the injection at the end of the URL, we find that a roughly similar fraction of flows is vulnerable as reported by Lekies et al. [79]. However, using both methods of inserting the injection, we identify 83% more confirmed vulnerabilities than when just inserting the exploit at the end. This suggests that previous work, as well as our own, may substantially undercount the number of vulnerable flows.

Our methodology differed from that of Lekies et al., in that we visited twice as many top-level domains, but fewer subpages for each domain (see Section 2.1). We believe this is the main cause of different findings for the number of domains that have at least one vulnerability. Previous work found 9.6% of domains to have at least one vulnerability, while we found 3.8% of domains to have one. Interesting, this shows that vulnerabilities are

not systemic, i.e., a domain that has at least one vulnerability is not likely to have that vulnerability (or different ones) on a preponderance of pages. Since some parts of pages hosted on the same domain are often shared across most pages, this implies that DOM XSS vulnerabilities are usually not in this shared content.

Where our methodology and that of Lekies et al. are most directly comparable—when relying only on the simpler method of confirming vulnerabilities and examining the ratio of vulnerabilities to number tainted flows or to number of pages visited—our results are generally similar, although overall our results suggest an increase in the number of vulnerabilities over time. In our work, we find more flows per page—on average, 92.6 flows per page compared to an average of 48.5 flows per page in Lekies et al.'s work. Additionally, normalizing by the number of flows we found more vulnerabilities: We found 0.04% of flows to be vulnerable, while Lekies et al. reported 0.03%. Normalizing by the number of pages visited, we also found more vulnerabilities: 1,754 vulnerabilities on 44,722 pages (3.9%); previous work found 6,167 vulnerabilities on 504,275 web pages (1.2%). We speculate that this difference is because JavaScript programs are becoming more complex, and as a consequence DOM XSS vulnerabilities are becoming more frequent.

5.2 Preventing DOM XSS

In Section 3.1, we showed that the unique vulnerabilities typically did not involve many of the same scripts: the stack traces of the exploits of different vulnerabilities were generally composed of different scripts. We interpret this to mean that most DOM XSS vulnerabilities are due to custom code, and not library code that is shared by many domains.

One way to prevent DOM XSS vulnerabilities is to detect them before the software is released. We believe that a promising direction for finding DOM XSS vulnerabilities at scale is using techniques that analyze larger portions of the program space. We explore one method of doing so in Chapter 5. The problem of code coverage of dynamic analysis techniques is not new or specific to DOM XSS vulnerabilities; however, it can be a bigger

hurdle for large-scale analysis of web applications than for traditional programs. Running many versions of a web application may require a large amount of network bandwidth for reloading web pages, which can make it difficult to scale. Solutions that avoid reloading the page to explore more sections of the program should be explored, as well as methods to force execution down alternate program paths. In particular, work on fuzzing parameters for traditional XSS [35] and on forcing JavaScript execution through different code paths [72] holds promise.

Our analysis also provides additional evidence of the risks of developing custom versions of common design patterns. While using design patterns for templated HTML—a practice analogous to parameterized SQL queries—is generally a good approach to preventing DOM XSS vulnerabilities, it is important to correctly implement the details. For example, we observed three instances of bespoke HTML template implementations that did not apply encoding functions to the values of the templates. In general, custom templating implementations can be error prone, because differences in context can be easy to overlook. For example, to be safe from XSS, a value in a templated HTML statement that is inside a script tag must first have encoding applied for the HTML parser and then for the JavaScript parser.

5.3 Measuring the effectiveness of static-analysis tools

We next further interpret the results of running the static-analysis tools on a sample of vulnerable scripts (see Section 3.2). These results suggest that many vulnerabilities may currently escape both static and dynamic analyses. We also leverage our results to suggest ways to extend static-analysis tools to catch more bugs.

In Section 3.2, we measured the false positive rate of different static-analysis tools and the rate at which tools correctly identified vulnerabilities from our dataset of known vulnerabilities using dynamic taint tracking. For the false positive rate, we empirically sampled the tools output and manually decided whether a tool's finding was a false positive or a true

positive. However, we were unable to empirically measure the false negative rate overall. This is because it was not feasible for us to know all possible vulnerabilities in non-contrived application. Instead, we measure the rate at which static-analysis tools can detect known bugs that are detected with a different methodology.

Our analysis of Burp Suite, the best-performing static-analysis tool we tested, showed low false-positive rates but also an inability to detect most of the vulnerabilities identified by the dynamic analysis. Together, these two measurements imply that the static-analysis tools were detecting largely different vulnerabilities that our analysis. The dataset of vulnerabilities with which we tested static-analysis tools, however, was limited to vulnerabilities detected through our dynamic analysis. In our test, we visited a large number of web pages but did not attempt to exercise much of the web application's functionality, for example, by clicking on fields, entering data into forms, or sending messages to pages. It is likely that such activities would reveal more vulnerabilities. In addition, in our dataset, we found many of the vulnerabilities to be shallow, in that they involved a straightforward concatenation of data from a source into the parameter to a sensitive sink function. This is similar to prior findings [135]. Indeed, it could be that the majority of vulnerabilities are more complex and would be better detected by static-analysis tools. Given that we know that there are many bugs that escape either analysis, we speculate that there may be many more bugs that escape both analyses.

ScanJS generally appears to identify poor coding practices that lead to DOM XSS vulnerabilities, rather than detecting such vulnerabilities. Indeed, many of the suggestions that the tool gives revolve around the use of certain functions being dangerous (e.g., `document.write` or `eval`). While ScanJS unfortunately did not detect many vulnerabilities on our dataset, we believe it to be useful for, e.g., enforcing coding standards.

Based on our experiments, we can make some recommendations for improving static-analysis tools. One area where static-analysis tools could improve is the ability to track flows across function boundaries. We found a non-negligible number of such vulnerabilities

Measurement	
• Our key-value pair injection method in conjunction with prior method found 83% more vulnerabilities than found using only prior method of injection [79].	Sec. 3.1
• We identified what has changed and what remains the same in DOM XSS over a 4-year span by building on top of a prior experiment.	Sec. 5.1
XSS trends	
• We found more tainted flows overall and a higher rate of vulnerable flows than previous work, which suggests that DOM XSS is getting worse.	Sec. 5.1
• Vulnerabilities are concentrated on a small number of iframe owners and script hosting sites.	Sec. 3.1
• 83% of vulnerabilities are due to code hosted on advertising and analytics domains.	Sec. 3.1
What contributes to XSS	
• DOM XSS vulnerabilities are likely not systemic within domains.	Sec. 5.1
• Vulnerabilities are often in unique, custom code, not in shared libraries.	Sec. 3.1
• Incorrectly implemented bespoke HTML templating, a defense against XSS, introduces XSS vulnerabilities.	Sec. 3.3
XSS prevention	
• Ad blocking would block many of the vulnerabilities and is an effective client-side protection tool.	Sec. 3.1
• Incorrectly implemented templating leads to vulnerabilities and possibly false sense of security.	Sec. 5.2
• The three popular (low-cost or free) static-analysis tools we tested are not effective at finding the vulnerabilities found using our dynamic tool; however, Burp often finds vulnerabilities not found by our tool.	Sec. 5.3

Figure 4.7: Summary of findings.

(20% in the domain sampling setting in Section 3.3) and tracking such flows can be difficult statically, especially when there are many branches. Another aspect of static-analysis tools that could use improvement is the ability to track flows that go through objects. For example, a tainted string is sometimes stored as the key or value in a JavaScript object and later used in a computation. Finally, one constraint of static-analysis tools that is especially limiting in JavaScript is the inability to analyze dynamically generated code. A hybrid static-analysis tool that analyzes new code before it is executed in the browser might be better able to detect such vulnerabilities.

6 Summary

We studied how to detect and prevent DOM XSS vulnerabilities in JavaScript code. In this work, we improved on the methodology to confirm DOM XSS vulnerabilities, finding 83% more vulnerabilities than by using previous methodology applied to the same dataset. We used our methodology for detecting DOM XSS vulnerabilities to empirically measure the prevalence of DOM XSS vulnerabilities on the Internet, finding them to be more common now than when previously measured in 2013. With our collected dataset of DOM XSS vulnerabilities, we also compared the ability of static-analysis tools to detect the same bugs that dynamic analysis techniques found, finding static-analysis tools to detect different types of bugs, with little overlap. A summary of our findings can be found in Figure 4.7.

%	Count	Category of top level website
27.7%	2856	News/Media
12.9%	1337	Entertainment
9.9%	1026	Technology/Internet
5.1%	523	Games
4.4%	453	Education
4.1%	424	Sports/Recreation
3.6%	376	Reference
3.5%	362	Shopping
2.8%	289	Hacking
2.7%	280	Business/Economy
2.4%	246	Society/Daily Living
2.0%	202	Mixed Content/Potentially Adult
1.7%	178	Newsgroups/Forums
1.6%	164	Health
1.4%	143	Search Engines/Portals
1.2%	119	Brokerage/Trading
1.1%	114	Political/Social Advocacy
1.1%	113	Financial Services
1.0%	107	Travel
0.9%	98	Vehicles
0.8%	80	Restaurants/Dining/Food
0.7%	76	Uncategorized
0.7%	71	Real Estate
0.6%	62	Job Search/Careers
0.5%	55	File Storage/Sharing
0.5%	51	Audio/Video Clips
0.5%	50	Government/Legal
0.4%	46	Software Downloads
0.4%	43	Religion
0.4%	43	Pornography
0.4%	39	Adult/Mature Content
0.4%	37	Alternative Spirituality/Belief
0.3%	31	Email
0.3%	28	Social Networking
0.3%	28	Personal Sites
0.2%	24	Malicious Sources/Malnets
0.2%	21	Office/Business Applications
0.2%	21	Auctions
0.2%	18	Phishing
0.1%	14	Humor/Jokes
0.1%	13	Suspicious
0.1%	13	Charitable Organizations
0.1%	8	Scam/Questionable/Illegal
0.1%	7	Web Hosting
0.1%	6	Intimate Apparel/Swimsuit
<0.1%	5	Weapons
<0.1%	5	Placeholders
<0.1%	5	Gambling
<0.1%	4	Web Ads/Analytics
<0.1%	4	Personals/Dating
<0.1%	3	Nudity
<0.1%	2	Military
<0.1%	2	Chat (IM)/SMS
	10325	**Total**

Table 4.4: Categories of top level domains that contain an iframe with a DOM XSS vulnerability. The count column shows the number of top level pages in a category that contained a frame with a vulnerability. The percent shows the percent of top level pages with that category.

	% of detected vulnerabilities	# of reported issues
Esflow	0%	4
ScanJS	8%	2700
Burp Suite	10%	39

Table 4.5: The percent of vulnerabilities detected by the dynamic analysis that were detected by different static-analysis tools out of a total of 50 web pages with known vulnerabilities.

Tool	False positive %	# of reported issues
Esflow	95%	19
ScanJS	100%	3764
Burp Suite	0%	36

Table 4.6: Empirical false positive rate computed from a random sample of 20 reported errors over 50 randomly sampled web pages.

	By domain	By unique bug
Simple concatenation	8	1
Simple except for variable usage	4	18
Spans multiple functions	8	1
Custom templating	3	0
Custom filtering	1	0
Dynamically generated code	2	0

Table 4.7: Description of types of vulnerability qualities observed during qualitative coding of bugs. We randomly sampled our vulnerability dataset in two ways, by domain and by unique vulnerabilities; each sample contained a total of 20 vulnerabilities. "Simple concatenation" refers to bugs that were a simple concatenation of the entire source with HTML or JavaScript markup. "Multiple functions" refers to vulnerabilities that spanned multiple functions. "Simple except for variable usage" refers to bugs where a variable with the source value was concatenated with HTML or JavaScript code. Custom templating refers to code that attempted to use a custom templating library, but without encoding. Dynamically generated code refers to instances where code involved in the bug was dynamically generated, and would generally be outside of the abilities of static analysis.

Chapter 5

Statically Analyzing JavaScript Using Machine Learning

1 Introduction

To mitigate the effect of DOM XSS vulnerabilities, a variety of defenses have been proposed. Filtering specific content—e.g., at server using web application firewalls [102], on the client using content security policies [66], or by the client-side XSS auditor in WebKit [140]—is frequently brittle and can be evaded by attackers [10,19,67,117]. Dynamic analysis methods, such as taint tracking, to detect DOM XSS vulnerabilities [79] have limited code coverage—they can only make claims about code paths that were executed. Using taint tracking at run time also requires significant changes to browser infrastructure and reduces browser performance [134].

Another approach to preventing XSS vulnerabilities is through static analysis, which can in principle lead to vulnerabilities being fixed before they are exploited, or even before code is released. Unlike run-time mitigation mechanisms, static analyses can also reason about all possible executions, not just the executions observed by the tool. However, traditional static-analysis tools often have difficulty reasoning about the dynamic features of JavaScript [57,

120, 139], have high false-positive and false-negative rates [80, 89], and generally do not scale [57].

The use of machine learning (ML) could potentially enable more accurate and performant static analysis. Researchers have recently started to explore ML as a method for analyzing the properties of programs [3,4] and for vulnerability detection in C programs [82]. ML classifiers have several properties that suggest that they could be effective at identifying vulnerable code: First, they can perform classification tasks more quickly than current static (or run-time) analyses. Second, ML classifiers could potentially be trained to identify patterns indicative of vulnerabilities that static analyses would miss, and, unlike run-time analyses, could identify vulnerabilities even if the code is not executed. Prior work observed that many of the vulnerabilities are syntactically similar and of low complexity [89, 135], which makes them likely to be captured by patterns that a classifier can learn from labeled training data, providing further support that ML classifiers could be effective at identifying client-side XSS vulnerabilities.

In this chapter, we explore the effectiveness of ML models to detect DOM XSS vulnerabilities statically by examining source code. While work on using ML models to examine source code exists [3, 4, 82], there is as yet no consensus about what ML models are effective or how best to represent code in a way amenable to ML classification. In our work, we experiment with several approaches, eventually focusing on classifying programs at the level of functions, using a bag-of-words representation to encode source code (after experimenting with more complex representations), and using deep neural network models (after experimenting with simpler ones). Our approach is novel in the space of using ML to analyze programs and may generalize to other program analysis applications beyond detecting XSS vulnerabilities in JavaScript.

In addition to exploring different input granularities, input representations, and model types, two challenges inherent in applying ML to vulnerability identification are collecting sufficient labeled ground truth data and dealing with its inherent one-sidedness. To col-

lect sufficient ground truth data—instances of JavaScript functions labeled as vulnerable or not—we leverage previous work that developed an open-sourced taint-tracking-enabled web browser [89], described in Chapter 4. Using their tool, we crawl over 300,000 web pages, identifying over 2,300 definitely vulnerable functions and over 180,000 potentially vulnerable functions. The resulting data is significantly one-sided, both in the ratio of confirmed to potential vulnerabilities and in the ratio of vulnerable to non-vulnerable functions, presenting challenges both in training classifiers and in evaluating them effectively.

The rest of the chapter unfolds as follows. We describe how we collect and label ground truth data, collecting and labeling over 18 billion JavaScript functions for vulnerabilities and how we use our dataset to train and evaluate models for statically detecting DOM XSS vulnerabilities based only on source code (Section 2). We then present the results of our experiments (Section 3) and, finally, discuss their implications as well as how our models could enable run-time defenses or development-time tools (Section 4).

2 Methodology

We first describe the design space that we experimentally explore to construct an effective ML model for detecting DOM XSS vulnerabilities (Section 2.1). We then describe how we process ground-truth data—labeled instances of JavaScript functions—to extract features and prepare it for use in training an ML classifier (Section 2.2). Finally, we describe how we collect ground-truth data to train our models (Section 2.3).

2.1 Machine-learning classifier design

The intuition that machine learning can help with identifying DOM XSS vulnerabilities is that prior work has identified that many vulnerabilities, and other interesting aspects of source code, are *syntactically* similar [82, 89]. We believe this is because developers often write code to perform certain tasks—and vulnerable code—in similar patterns, e.g., using

the same variable names or calling the same API methods. We aimed therefore to apply machine-learning methods to detect these patterns in an automated way.

In designing a classifier to detect vulnerabilities, we had to narrow down the design space of machine-learning algorithms. As we report in Section 3.2, we experiment with: model types (logistic classifiers, and deep neural network models); model sizes; training time; and using transference learning during training.

Balancing errors First, we had to decide on the relative weighting of misclassifications between vulnerable training instances and non-vulnerable training instances. Because our data is significantly one-sided, a naive classifier might be able to achieve nearly perfect accuracy by always predicting that code is non-vulnerable. However, we want the classifier to be more conservative and to prefer to misclassify a safe piece of code as vulnerable, rather than to misclassify a vulnerable piece of code as safe. Therefore, we more heavily weighted vulnerable code during training. In early experiments, we experimented weighting ratios between vulnerable and safe functions of 1, 10, 100, and 1,000. We found a weighting of 100 to be optimal—with lower ratios the classifier would never predict something to be vulnerable, and at a relative weighting of 1,000, the classifier would not train stably.

Vectorizing features Second, when converting features to ones that our classifier can understand, we used feature hashing [157] to represent our sparse data. Feature hashing allows us to represent unbounded vocabularies by using sparse vectors in which our words are hashed to specific buckets. The tradeoff of this technique is that it introduces ambiguity to the classifier when the hash function has collisions. In order to minimize the chance of collisions, we use a feature size of 2^{18}, as recommended to balance memory and collisions [124]. These features are then represented as a sparse vector to minimize memory usage. To use dense vectors for our neural network models, we apply an embedding transformation to convert the sparse bag of words into a dense vector space. This transformation learns a mapping of sparse features into the dense vector space that is optimized during training

along with other model parameters.

Implementation Our classifier is built using TensorFlow [1]. Training time depends on the amount of data and the specific model parameters used. For one of our smaller models (Figure 5.5a), training time is 11K functions per second, which works out to roughly 20 hours to train on 5% of our total data on our hardware with one NVIDIA Tesla P100. Our implementation can run prediction at a rate of 15K items per second; in our unoptimized implementation, we believe much time is spent reading and writing results to disk.

2.2 Feature extraction and data preparation

To classify code snippets, we first must translate them into a form that can be consumed by a neural network. We first take JavaScript source code, and then parse the source code with the V8 JavaScript parser. We then annotate specific AST nodes with positive and negative classification labels. Positive labels correspond to AST nodes that are either potential or confirmed sensitive sink functions, depending on whether we are training on our potential or confirmed vulnerability datasets; all other nodes are annotated with negative labels. Finally, we shuffle and partition data into training, testing, and validation datasets.

Unitization After we label the source code, we must unitize the source code into code snippets that are used for classification. For the experiments that we present here, we unitize code by individual functions. In early experiments, we tried using whole script unitization, and with unitizing nodes by extracting surrounding AST nodes within a fixed semantic distance, but found the approach of using whole function slicing to be most useful. Whole script slicing tended to select too much code, limiting the ability of the classifier to pinpoint specific features to correlate with labels, while the fixed semantic distance would often select too little code.

Extracting features After parsed code has been transformed into snippets, we then extract input features for use in our ML model. For the experiments that we show here, we used a bag of words model by extracting all of the relevant symbols, and operations inside the parsed AST tokens—for example, variable names, operation names (plus, minus, etc.), method names, property names. We believe that this representation is somewhat robust in the face of small changes in source code, e.g., between different versions of libraries, because often such changes can keep a significant amount of the variable names and function names the same.

We experimented with methodologies that attempt to extract program slices based on prior work [82], but found that for our application of analyzing JavaScript, a highly dynamic language, rather than C, we could not confidently statically identify the key points of the program that are required for slicing. We also experimented with models based on gated-graph neural networks [3], but found such models to not learn stably.

Experimental setup We divided our dataset into training, testing, and validation subsets: the training dataset was used to train our models, the testing dataset was used to evaluate the performance of competing models during hyper-parameter tuning, and the validation dataset was used for measuring the performance of the final models we selected after exploring the hyper-parameter space. We split our data into 80% training, 10% testing, and 10% validation on the basis of unique scripts: for each script in our collected dataset, there is a 80% chance it would be used during training, 10% chance in testing, and a 10% chance for validation. Then after partitioning by script, we unitized our scripts by extracting individual functions. We partition our data by unique scripts to better understand our model's performance on complete scripts that it has never seen before, and more closely match a real use situation in which the model will be presented with complete scripts. Furthermore, when we experimented with splitting data by non-unique scripts, the model would often memorize the training data and become too accurate for us to fairly gauge its performance.

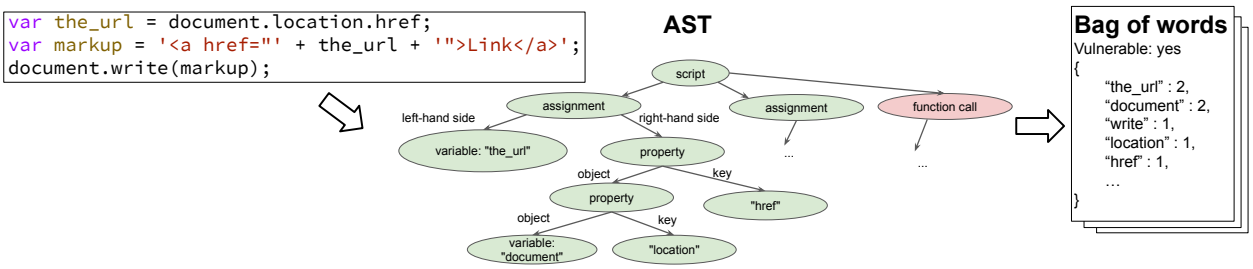


Figure 5.1: Diagram showing ground-truth data collection. The red circle represents a vulnerable label.

For confirmed vulnerabilities, which have many fewer positive examples, we use cross fold validation to determine the effect that such random splits have on our model performance. Our aim is to remove the differences resulted by random chance. We use five folds for cross validation and then combine results from the separate folds by concatenating [45].

We also apply a weight to each function based on the number of times that the function was observed in our crawls. We apply this weighting because functions defined in very common libraries (e.g., jQuery) are more important to classify correctly than code that is comparatively uncommon. For training, weighting takes the form of repeating code instances multiple times in the training set, which is then shuffled. We repeat training instances rather than simply applying a weight during training because in our experiments, the models would not stably train because of the presence of extremely common functions which would outweigh the gradients of less common functions. However, by repeating instances multiple times, the effects are smoothed out over the training period.

Our training dataset—after unitizing code, repeating training instances, and shuffling—is roughly 1.8TB when compressed with GZIP. To further lower the size of the training dataset, we represent it as a series of file pairs in which one file contains unique instances and is loaded entirely into memory before training starts, and the other file represents indexes into that unique dataset. Because this representation removes duplicates, and because we can optimize compression by sorting our source data, this representation is 576GB compressed with GZIP.

2.3 Ground-truth data collection

Here we describe our infrastructure for collecting ground-truth data for training and testing our vulnerability classifiers, as described in Chapter 4. First, we use a modified browser with taint tracking enabled to collect a series of execution traces of websites. Our modified browser is driven by a browser extension that interacts with a server-side database, which directs crawling activities, via HTTP interactions. Those traces identify code that is potentially vulnerable to DOM XSS injections. The generated log files, which contain records of tainted flows, are stored in the database. Next, we attempt to confirm whether potentially vulnerable flows are exploitable. We then label source code based on the results of our execution traces and confirmation passes by aggregating results of multiple executions. The ASTs, with sensitive sinks labeled according to whether they were involved in potentially vulnerable or confirmed vulnerable flows, are used as training data for later steps. An overview of how ground-truth data is created is shown in Figure 5.1. The parts of the infrastructure that are common to Chapter 4 are: the modified web browser, parts of the crawling infrastructure, the logic for detecting potentially vulnerable flows, and operations for generating potential vulnerabilities.

Taint tracking To collect ground-truth labels, we leverage prior work for detecting DOM cross-site scripting vulnerabilities [89]. This methodology is described in detail in Chapter 4. First we perform a large scale crawl on the web using a specially modified version of the Chromium browser, which detects potential DOM cross-site scripting vulnerabilities. Specifically, the browser's V8 engine and WebKit infrastructure were modified to use taint tracking to detect when potentially attacker controlled data is used as an argument in sensitive sink functions and operations—for example, `document.write`, or assigning to `innerHTML`. During execution of each webpage's JavaScript inside the browser, the modified browser writes a log file which contains: a record of all source code executed by the browser, the parsed representation of that source code according to V8, and all sink executions that contained tainted

data, in addition to other bookkeeping information. For each execution of a sink with tainted data, we additionally know: the value of the tainted argument, the specific characters which are tainted, whether any specific built-in encoding methods had been applied (e.g., `escape`, `encodeURI`, or `encodeURIComponent`), and a full trace of the JavaScript call stack.

Crawl methodology In total, we crawled the Alexa top 10,000 [2] most popular websites and visited 289,392 web pages on those websites. The crawl occurred during January of 2019. We began by visiting the root webpage of the website, and then sampled 40 sublinks on the same domain as the root webpage, for a total of 410,000 attempted webpage visits. We obeyed robots.txt [74] directives during our crawl, preventing us from loading some pages, and additionally some webpages did not correctly load during our crawl. If an individual webpage did not load successfully, we attempted to load a different random webpage on the same domain, where possible. While loading webpages, we first waited for the page ready event to signal that the page had loaded, then we waited for 90 seconds for each page to complete execution. We chose 90 seconds because we empirically observed that the vast majority of sinks which contained tainted arguments were observed within this time frame.

Compressed log files from our crawl are stored using the Cap'n Proto file format [152] and take roughly 26TB of disk space. The majority of space is dedicated to the source code and parsed representation of executed scripts. Many scripts are repeated across multiple log files. The databases for aggregating log files take 382GB of disk space after removing source code and only preserving parsed JavaScript, compressing, and removing duplicates.

Labeling After we collect these log files of web page execution, we process each log file and insert it into a database to aggregate results for labeling. We detect which scripts contained tainted arguments to potentially vulnerable sinks during execution, and where those sinks are located. We aggregate the results of multiple runs, to tell if a specific piece of code contained a potential vulnerability across any page execution. To locate the specific call to sink functions in source code, we output an annotated stack trace of the function call

during execution, which contains references to specific AST nodes in the parsed JavaScript, in addition to the parsed AST of all JavaScript that is executed. We use AST node of the JavaScript stack frame closest to the bottom of the call stack as indication of the positive node. We do not use other functions in the stack trace. We do not have information about the location of the source of the tainted flow, so we are unable to label such nodes.

To tell whether a specific instance of a sensitive sink is potentially vulnerable, we make a decision based on the specific sink in question (ones that are capable of executing arbitrary JavaScript), and whether the encoding methods that were applied to tainted data match the context of where the taint is applied. For example, if our log file indicates that the `document.write` function was called with a tainted argument of `<a href="http://example.com#hash">Link</a>` in which the ninth through 32^{nd} bytes are tainted (from the `h` in `http` to the `h` in `hash`) by webpage's URL without any applied encoding functions, we would mark that AST node as potentially vulnerable. However, if the `encodeURI` function were applied to the tainted bytes before being used in the sink, then we would mark that function as not vulnerable, because the encodeURI function escapes the double quote character that is necessary to escape from the context in which the tainted characters are located. This is similar logic to detecting potentially vulnerable flows to logic used in prior work [79, 89, 134].

For code that is potentially vulnerable according to our log processing, we take a further step of generating simple confirmation test injections, similar to prior work [79, 89]. We combine our knowledge of the specific encoding functions applied to generate a test injection for each potentially vulnerable flow, and then re-execute the webpage with our test injection to see if the flow is indeed vulnerable. This step is necessary because programmers have broad ability to do ad-hoc sanitization of flows that does not use the built-in encoding methods. We present two types of experiments in this work based on labeling of code that is potentially vulnerable and code that is confirmed to be vulnerable. We discuss applications for both models in Section 4.4.

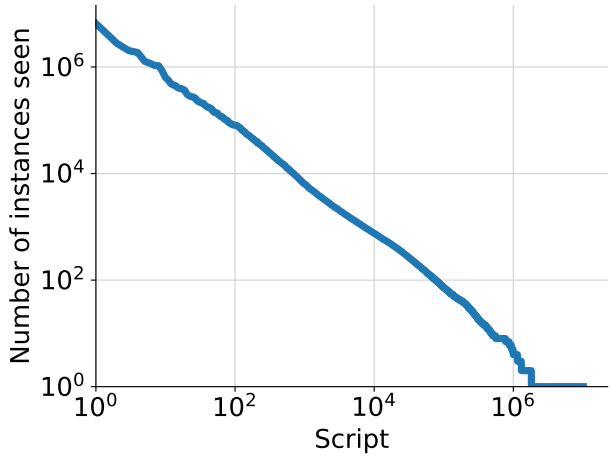


Figure 5.2: Frequency of occurrence of scripts in our dataset. The y-axis shows the number of times a script occurred in our dataset in log scale. The x-axis shows individual scripts sorted from most frequent to least in log scale.

Because our ground-truth data is created by leveraging the results of a dynamic analysis approach, our ground-truth data is limited by the biases inherent to such an approach—namely issues of program coverage. Our dataset can only label AST nodes as being vulnerable (or potentially vulnerable) when those nodes are executed. We cannot make claims about code that is not executed. Furthermore, because of the dynamic nature of JavaScript execution, even code that is executed could be labeled as negative after one execution and positive after a different execution. While our dataset may have missed vulnerabilities—either potential or confirmed—for all nodes that we mark as vulnerable, we are confident that they are in fact vulnerable at least on one execution of the program. We discuss in more detail the limitations of our dataset in Section 4.3.

2.4 Attacker model

The use of machine learning in security tasks can expose systems to new attacks. For example, in *poisoning attacks* attackers try to subvert the training data used in a system [96, 98, 109] or attempt and in *evasion attacks* attackers attempt to construct inputs that appear normal but evade correct classification [83, 121, 162].

We do not consider such attacks. Our attacker model assumes that JavaScript code

is benign but potentially vulnerable, and that attackers may provide malicious input to websites—such as malicious URLs—that may exploit an XSS vulnerability. For evasion attacks, our system is designed to detect *accidental* vulnerabilities, e.g., in order to help protect website owners who, we assume, have control over the JavaScript on their website. If site owners already do not have control over the JavaScript on their website, an XSS or other attack could likely already occur without the need to specifically evade our detection infrastructure. For poisoning attacks, an attacker would have to control a substantially large enough portion of training data that it would limit the practicality of such an attack.

2.5 Metrics

Due to the large class imbalance in our training data, we find the tradeoff between precision and recall to be most useful for evaluating our models' performance. Precision is the fraction of true positives divided by the number of instances that the model claims are positive. Recall is the true-positive rate, which is the number of true positives divided by the number of positive labeled instances. We use precision and recall as our primary metrics because notably, the false positive rate for all of our classifiers is very low (e.g., $< 1 \cdot 10^{-5}$ depending on the experiment, and the threshold for classification chosen), but this could still translate to a low precision score because the base rate of true negatives in our dataset is very high. Furthermore, accuracy—the number of correct predictions divided by all predictions—is not a substantially useful metric, because a classifier could have more than 99% accuracy (for example 99.83% for potential vulnerabilities) by always predicting that a piece of code is not vulnerable, but this would not be useful.

In addition to recall, which is weighted by the occurrence frequency of a specific vulnerability, we also consider coverage of distinct vulnerabilities. It is important for a classifier to identify all vulnerabilities.

We are also interested in the execution time required to perform inference with our model. We want to understand it in relation to other complementary methods, as we discuss

		Training	Testing	Validation	All data
Confirmed	total # functions	15B	2.0B	1.8B	19B
	# of vulnerabilities	3.8M	357K	354K	4.5M
	% of total that are vulnerable	0.025%	0.018%	0.019%	0.024%
	distinct # functions	383M	48M	48M	478M
	# of distinct vulnerabilities	1,853	235	238	2,326
	% of distinct functions that are vulnerable	0.00048%	0.00049%	0.00050%	0.00049%
Potential	total # functions	15B	1.7B	1.8B	19B
	# of vulnerabilities	27M	2.8M	2.2M	32M
	% of total that are vulnerable	0.18%	0.17%	0.12%	0.17%
	distinct # functions	382M	48M	48M	478M
	# of distinct vulnerabilities	144K	19K	18K	180K
	% of distinct functions that are vulnerable	0.038%	0.039%	0.037%	0.038%

Table 5.1: Summary statistics of our dataset of vulnerabilities. We show statistics for both our confirmed and potential vulnerability datasets. For confirmed vulnerabilities, we present the average of all folds. Some numbers in the all data column do not add to the sum of other columns due to rounding. Vulnerabilities refer to the section of the table that they are in, for example, *% vuln* in the *Confirmed* section refers to confirmed vulnerabilities. Distinct refers to functions after applying our distinct criteria on script content.

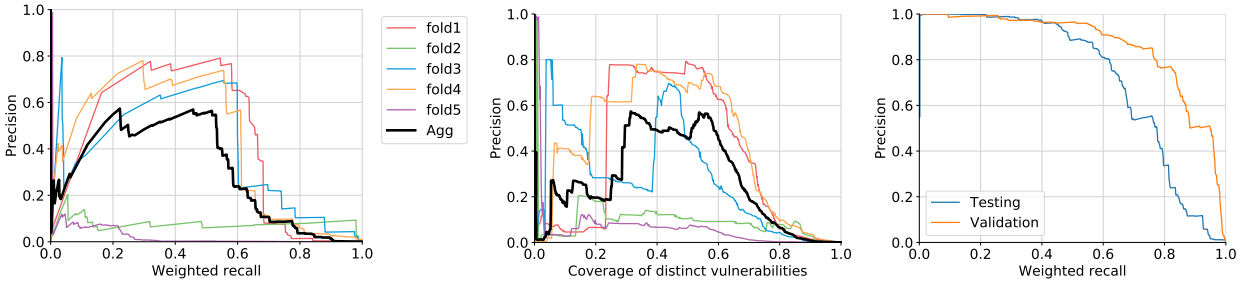
in Section 4.4.

3 Results

Here we present the results of our models for detecting vulnerabilities based on source code. First, we discuss attributes of our ground-truth data (Section 3.1). Then, we discuss our findings about the training process and tuning training parameters using our test dataset, such as the model type, the model size, and amount of training data (Section 3.2). Finally, we evaluate our best-performing models on our validation data set (Section 3.3).

3.1 Ground-truth dataset

After collecting ground-truth data using the methodology described in Section 2.3, we had a dataset that could be used for training a classifier. We wanted to understand the degree to which frequently used scripts, for example jQuery, would impact our dataset. For example, if the frequency of a handful of individual script accounts for a significant amount of the



(a) Confirmed vulnerabilities **(b)** Confirmed vulnerabilities **(c)** Potential vulnerabilities

Figure 5.3: Effect of the random split of our dataset on model performance. *Agg* is the aggregate combination of all five folds. The x-axis shows metrics for the true-positive rate. The difference between coverage of distinct vulnerabilities and weighted recall is that weighted recall is weighted by the occurrence frequency of a piece of code in our dataset, while coverage of distinct vulnerabilities shows the fraction of distinct true vulnerabilities that are correctly labeled.

dataset, then it could be that the model's performance would be dominated by their ability to recognize certain specific scripts. We find that, while some of these high frequency scripts exist, there is also a significant long tail of unique scripts. Because of this, we also look at the metric of our model's coverage of detecting vulnerabilities in unique scripts, which are compared by (a hash of) their content, in addition to the model's recall on scripts weighted by their frequency. Our dataset included 23,013,705 unique scripts. Those scripts were observed a total of 240,830,867 times. These 23 million unique scripts were collected from 992,644 iFrames on our crawl of 10,000 domains. A histogram of the number of times each script was seen is shown in Figure 5.2.

Our dataset is significantly one-sided in that positive labels, i.e., vulnerabilities, are rare compared to negatives, which are safe, non-vulnerable functions. For example, overall, only 0.024% of all functions were confirmed vulnerable, while 0.00049% of unique functions had been confirmed as vulnerable. Furthermore, because of the scale of our crawl, we were able to collect data about over 18 billion functions, corresponding to nearly 500 million unique functions. Our training data set has 15 billion functions. Summary statistics for our datasets are shown in Table 5.1. The table shows the results aggregated across folds for the confirmed dataset. As described in Section 2.3, we consider datasets for both confirmed and potential vulnerabilities.

3.2 Training experiments

We experimented with a variety of different parameters to tune our models. We describe the different experiments to select parameters for training our model here. For these experiments, we report results using our test set.

Effect of crossfold validation For our dataset of confirmed vulnerabilities, because of the small number of positive samples, relative to the number of total samples—only 0.024% of functions were vulnerable—we wanted to understand the role that the random split into training, testing, and validation has on our results. We plotted the results from multiple folds of our experiments for detecting confirmed vulnerabilities in Figure 5.3. We find that there is a large variation in the individual folds. Specifically, folds two and five in Figure 5.3 perform particularly poorly. One potential reason for this is that those folds happened to have particularly few true vulnerabilities in the random split—for example, in fold two, there were 155,328 positive labels in the test set, compared to 627,753 in fold one. This difference in the base rate between the folds means that there is a roughly four times greater chance for the model to have lower precision for the same recall in fold one than in fold two, regardless of what the model learns. Another explanation could be because the positive examples in the training set happened to better match the positive samples in the testing set. All results for confirmed vulnerabilities that we discuss from now on are the aggregate across five folds.

For potential vulnerabilities, there are more positive examples compared to the number of total samples—0.17% of functions are vulnerable, nearly seven times more than for confirmed vulnerabilities—which smooths out the effect of the randomized splitting of our dataset. This can be seen in that the variation between the training and validation folds, for example in Figure 5.3c, is smaller than the variation between the folds in our confirmed experiments.

Amount of training data Because of the huge amount of available training data (roughly 15 billion functions), we wanted to understand at what point additional training was not

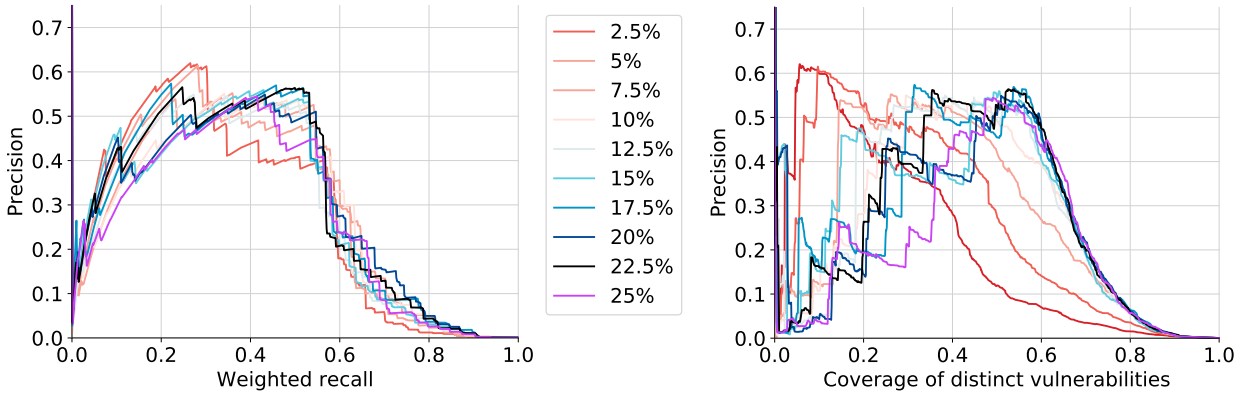
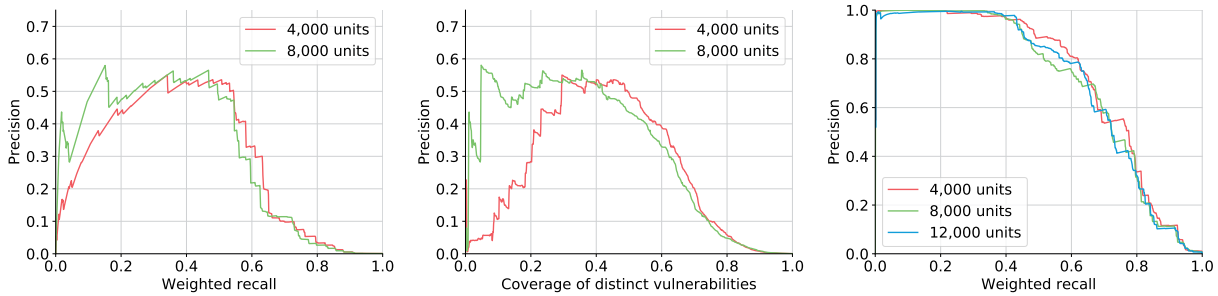


Figure 5.4: Effect of the amount of training data on predicting confirmed vulnerabilities. The amount of training data varies between 2.5% and 25% of available training data. Results show the aggregate across five folds.



(a) Confirmed vulnerabilities **(b)** Confirmed vulnerabilities **(c)** Potential vulnerabilities

Figure 5.5: Effect of varying model size for confirmed and potential vulnerabilities. For confirmed data, results show the aggregate across five folds.

useful. We trained models using progressively more of our dataset and measured the effect on prediction performance. The results can be seen in Figure 5.4. We found that after training on 20% of the training data, there was no significant change in performance, as shown by the difference between the lines for 20%, 22.5%, and 25% of data in Figure 5.4. We believe this is because the model may have already seen a significant number of the distinct functions by this point, given that there are on average roughly 40 instances of every distinct function that are shuffled through out the training data for both confirmed and potential datasets. For subsequent experiments, we use only 20% of our data because it was the value that performed best on our test dataset.

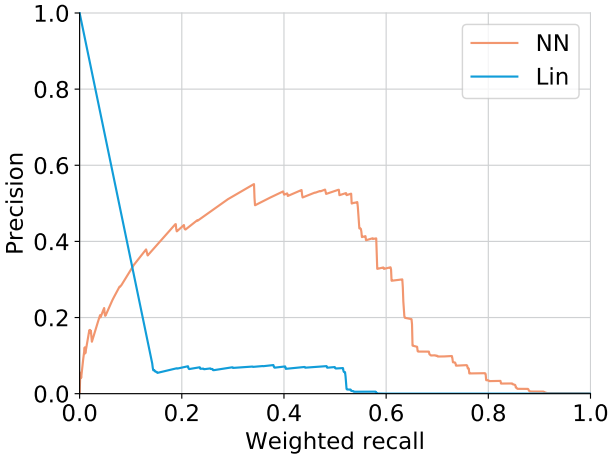


Figure 5.6: Performance of linear and deep models on confirmed vulnerabilities. Results show the aggregate across five folds.

Model size We experimented with different sizes of deep neural network models. For these experiments, we used a three-layer architecture, similar to that used in prior work [82]. We trained one model with 4,000, 2,000, and 500 units, respectively, in the three layers; a model with 8,000, 4,000, and 1,000 units; and, for potential vulnerabilities only, a larger model with 12,000, 6,000, and 1,500 units. Because of the amount of compute resources necessary for training models for confirmed vulnerabilities (because of the need to cross validate), we only experimented with the first two variations; however, for potential vulnerabilities, because we did not see a significant difference between first two variations, we also experimented with the third, largest model.

The results for this experiment are shown in Figure 5.5. For confirmed vulnerabilities in Figures 5.5a, and 5.5b, we find that the smaller variation performed slightly better than the larger one. For potential vulnerabilities, we find similarly that model size is not a large factor, at least for the model sizes that we experimented with, as shown in Figure 5.5c. For both confirmed and potential vulnerabilities, we use our smaller model with 4,000 units in the first layer going forward.

Model types We experimented with two different types of models: a logistic regression model and a deep neural network model. We chose to test logistic regression to verify our

hypothesis that the features were significantly more complex than could be modeled with linear models.

Results are shown in Figure 5.6. We find that the linear model performs very poorly compared to the neural network model. For example, when the models detect 50% of vulnerabilities, the logistic regression model has a precision of 7%, while the neural network has a precision of 55%.

We experimented with linear models because to test whether vulnerability prediction could be based simply on the use of certain APIs or specific variable names. If a linear model were able to predict our data accurately, it would obviate the need for using a more complex deep neural network model. However, we found that our deep models were more accurate than our simpler linear models, giving us some indication that our deep neural network models are extracting more meaningful patterns from the data.

3.3 Results for best-performing models

After the experiments described in Section 3.2, we had trained variations of our models and determined which performed best on the test data. Here we report on the performance of our best-performing models for confirmed and potential vulnerabilities using the final version of our models on our validation dataset. We discuss qualitative properties of the missed predictions for our models in Section 4.1. We used our smaller deep neural network models, using 4,000 units in the first layer, trained on 20% of the available data, as larger models and more training data seemed to offer no substantial advantage in performance but were more computationally expensive when running prediction.

Detecting vulnerabilities Using the best configuration of models that we found in our experiments described in Section 3.2, we tested our models on validation data. Results are shown in Figure 5.7. For potential vulnerabilities, for example, when detecting 80% of true potential vulnerabilities, 77% of the vulnerabilities that the model claims are vulnerable are

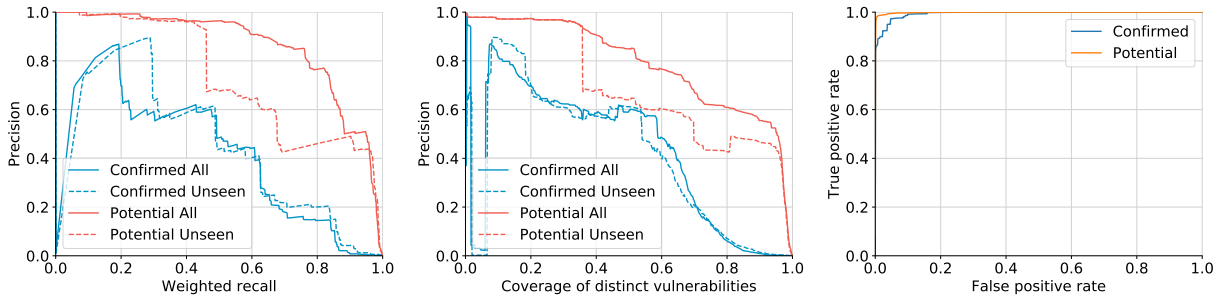


Figure 5.7: Model performance for validation data.

in fact vulnerable. While for confirmed vulnerabilities, the models correctly identify 58% of distinct confirmed vulnerabilities while having a precision of 57%.

We were interested in the tradeoff of the model's false positive and false negative rate for applications that are tolerant of low precision as long as the false positive rate is low. We show the ROC curve for our model, showing the tradeoff between the raw false-positive and true-positive rate, in Figure 5.7. Because of the huge class imbalance, we achieve recall of 80% with an extremely low false positive rate of 0.1%. However, because of the base rate of negatives in our dataset, this can still translate to poor precision. Models for our confirmed vulnerabilities have comparatively worse performance than our models that detect potential vulnerabilities. We speculate that this is because detecting confirmed vulnerabilities is a more difficult problem—all confirmed vulnerabilities are also potential vulnerabilities, but a model must additionally attempt to reason about whether specific code is definitely vulnerable. Additionally, the base rate of confirmed vulnerabilities is also significantly lower than for potential vulnerabilities, as shown in Table 5.1, further challenging our models.

Previously unseen functions Because our training and testing scenario involves splitting our data into training and testing groups by scripts, as described in Section 3.1, it could be that the same function that was present in the training data is duplicated in the testing data. This happens when different scripts contain the same function, for example, when scripts internally bundle other common libraries, like jQuery. We wanted to be sure that our model was not simply memorizing answers, so we tested our model's performance on functions that

it has never seen before. We removed from our validation data, any function that was an exact match for a function that existed in some other script to create a list of previously unseen functions. Our validation data overall contains 48 million distinct functions. After removing previously seen functions, our validation set of previously unseen functions contains 12 million functions (average across all folds). When manually looking at results, we did not find evidence that jQuery or other popular libraries were dominating our results or errors.

Results for our validation set and the subset of previously unseen functions are shown in Figure 5.7. For confirmed vulnerabilities, the performance of the model is only slightly different when considering only previously unseen data. For example, for confirmed vulnerabilities, overall our model has 58% coverage when detecting distinct vulnerabilities and has 57% precision, while for previously unseen vulnerabilities, our model has 53% coverage when detecting distinct vulnerabilities and has 60% precision. This leads us to believe that, for confirmed vulnerabilities, there is little effect from duplicated functions in different scripts, since the performance characteristics are similar. However, for potential vulnerabilities, there was a somewhat more pronounced effect, though the performance for potential vulnerabilities is still better than for confirmed vulnerabilities. For example, when considering all vulnerabilities, the model has 60% precision and 80% recall, while for only previously unseen vulnerabilities, the model has 42% precision at 80% recall. It could be that the model for predicting potential vulnerabilities is memorizing more of the data than the models for confirmed vulnerabilities.

4 Discussion

We analyze our results further, discuss several topics important for applying machine learning to vulnerability detection in general, and examine how our vulnerability detection approach could be used to enable practical defenses. First, we examine seemingly incorrect predictions made by our classifier and show that many predictions labeled as false positives are in fact

correctly identified vulnerabilities (Section 4.1). Then we discuss the importance of choosing the right metrics when analyzing one-sided data (Section 4.2) and the biases in training data that our approach is affected by but largely overcomes (Section 4.3). Finally, we describe several ways in which our approach to detecting vulnerabilities could enable other practical defenses and vulnerability detection mechanisms (Section 4.4).

4.1 Explaining mispredictions

We manually examined some of predictions of confirmed vulnerabilities that were categorized as false positives and found evidence that some were in fact actual vulnerabilities that are mislabeled in our training data.

When examining Figures 5.7, and 5.5b, we observed that, surprisingly, precision of confirmed vulnerabilities was sometimes low even to the left of the curve, where one would expect high precision at the cost of low coverage. This signals that some of the model's most confident positive predictions were incorrect. We hypothesized that this may be the result of mislabeled data, as we discuss in Section 4.3. We manually investigated ten of the model's most confident false positives and found that, of the ten items: three were instances of mislabeled ground-truth data; four were functions that were vulnerable (depending on the calling context), but were not executed; and three were legitimate false positives (i.e., certainly not vulnerable code).

The three false positives that we ascribe to incorrect ground-truth data are caused by the fact that the collection of ground truth has separate phases to detect potential vulnerabilities and to confirm them. For highly dynamic web sites, page content may change between the two phases. In this case, a script that was initially labeled as potentially vulnerable may fail to get labeled as confirmed vulnerable, simply because it was no longer available on the site during the confirmation phase. Hence, these three false positives were actually correct predictions of vulnerabilities that would have been confirmed had it not been for this imprecision in ground-truth data collection.

In four of the ten cases, the false positives were functions that were not executed during the analysis that was used to collect ground-truth data. That they were identified by our classifier shows the potential advantage it may have over dynamic taint tracking.

For the remaining three of ten false positives that we examined, the functions incorrectly classified as vulnerable involved calls to sensitive sinks but in (uncommon) ways that were not vulnerable.

That seven of the ten high-confidence predictions categorized as false positives should have in fact been classified as true positives suggests that the classifier's ability to identify vulnerable code may be better than the evaluation shows.

We also manually examined ten instances of randomly sampled false negative results. We found little conceptual overlap between the examples; however, one common characteristic is that all of the false negatives we observed were large and complex functions. We were expecting that false negatives could potentially be caused by minifiers that obfuscate function and variable names, removing semantic information used by the model, but did not find any evidence for this. It could be that longer functions are further from the middle of the distribution and we are less likely to have good training data (i.e., that vulnerable instances of similar functions are unlikely to be encountered in training).

4.2 Importance of choosing the right metrics

Reporting classification performance using false-positive (e.g., in Figure 5.7) and true-positive rates often falls afoul of the base rate fallacy—performance can appear exceedingly good, but only because the data is significantly one-sided. Hence, we focus on *precision*, which leads to a more robust evaluation of our classifiers' performance.

Because of the very low base rate of functions labeled as vulnerable in our dataset, it can be challenging to create a classifier with acceptable levels of precision—even with an extremely small false-positive rate, the number of false positives could greatly outweigh the number of true positives. For our best performing model for identifying confirmed vulnera-

bilities, we found that when our classifier is detecting 58% of distinct true vulnerabilities it has a precision of 57%, as shown in Figure 5.7. At that same point, the false-positive rate is 0.0057%. Furthermore, when our coverage of distinct vulnerabilities is 70%, our false-positive rate is still only 0.39%, despite precision dropping to 22.3%. Because our coverage of distinct vulnerabilities performs similarly to the performance for recall weighted by occurrence, we believe that our model is not simply memorizing extremely common scripts (e.g., jQuery). As we discuss in Section 4.4, we believe this combination of precision and low false-positive rate may be sufficient for several real-world applications.

4.3 Bias in ground-truth data

Although our training data is biased, our models are able generalize to detect vulnerabilities, both ones that did not appear in training data (Section 3.3) and ones that appeared but were mislabled (as non-vulnerable code) in the training data (Section 4.1). Less biased training data would likely yield even better performance, and so here we discuss the main ways in which our training data is biased.

As we describe in Section 2.3, our ground truth data is derived using dynamic taint tracking. One feature of this methodology is that our dataset will have many functions that are labeled as safe but that are in fact vulnerable. In particular, functions that are dead code and on code paths that were not executed during the data collection would be mislabeled as safe, since taint tracking would have had no opportunity to detect that they are vulnerable. An alternative method for addressing uncertainty in ground-truth data is by using soft labels, in which ground-truth labels are given a probability lower than 100% during training [39]. We experimented with soft labeling strategies, though results were inconclusive. Nonetheless, we show that our models can usefully detect vulnerabilities in our dataset, even ones unseen during training.

In addition, our browser infrastructure is based on an old version of the Chromium browser, based on version 57 (a version from August 2016). The vulnerabilities that we

observe in principle may not apply to other browsers. Notably, that version of Chromium handles encoding of values after the hash differently than the current version of Chromium, which may affect which vulnerabilities are exploitable on newer versions. However, the vulnerabilities we detect would certainly affect many browsers. Independently, we believe our experiments to be useful for demonstrating that machine learning can be used to detect vulnerabilities and that they yield insights into applying machine learning to program analysis tasks.

Finally, if any instance of a function is found vulnerable by taint tracking, then our labeling marks all instances of that function as vulnerable. This means that we may label a function as vulnerable when, in fact, exploiting that vulnerability would require cross-function interactions that are present on some web pages that use that function, but not on others. Arguably, it is appropriate to always flag such functions as vulnerable, since they are not safe in all contexts.

4.4 Applications

We believe that our experiments show that DNNs can help detect DOM XSS vulnerabilities in several practical settings. In addition, our approach is agnostic to the specific language or program analysis task, and so similar results may be achievable for other languages or program analysis tasks.

Selectively enabling run-time defenses The models that identify confirmed or potential vulnerabilities could be useful as a lightweight method to identify potential issues before JavaScript code is run, e.g., in a web browser. Because classification has false positives, we do not suggest preventing execution of JavaScript function if it is identified as vulnerable. Instead, a more promising approach might be to use the classifier's output to selectively enable taint tracking or other more heavyweight, but more accurate, techniques.

In particular, prior work demonstrated how a precise taint-tracking enabled browser can

be used as an effective run-time defense for DOM XSS vulnerabilities, but at the cost of 7–17% CPU overhead [134]. As browser vendors are extremely sensitive to performance degradation, this amount of overhead is too high for deployment to be likely. In contrast, for our non-optimized implementation and on our dataset, the average time to classify a function is 66μs, which works out to 11ms to classify all the functions in a script (scripts contain 161 functions on average, and median of two).

When our classifier (for confirmed vulnerabilities) is tuned to achieve 80% recall, it makes as vulnerable 0.11% of individual functions, and about 0.17% of scripts (which often have multiple vulnerable functions). Hence, if our classifier were used as a first filter before applying, e.g., taint tracking, to determine whether a flagged vulenerability is a false positive, only 0.17% of scripts would have to be analyzed using the more heavyweight method.

Because the false positive rate for our neural networks is low, the overhead in this approach would be dominated by the time spent on neural network filtering, and not the more heavyweight method like taint tracking.

Although our measurements are carried out on a machine with a GPU, many client machines also have access to some accelerator. The cost of classification could be further lowered by taking advantage of processing already performed, such as the browser parsing JavaScript code, which is a precursor to both executing the code and classifying it. Finally, models could be minimized or optimized using previously suggested techniques [60], and classification could be performed in parallel with other tasks to hide latency.

Analyzing large codebases Another potential use of our classifiers is to enable analyzing large bodies of code, e.g., software repositories, for which more mainstream methods would be prohibitively expensive. For researchers or practitioners seeking to quickly characterize code quality—e.g., to measure changes in quality over time, assess the risk of relying on a particular codebase, or cheaply provide feedback to code maintainers—the precision and recall of our classifiers may be high enough that they could be applied directly.

To eliminate false positives, classifiers like the ones we develop—for either confirmed or potential vulnerabilities—could also be the first step in a pipeline that later involved additional analysis of the functions the classifiers identify as vulnerable. For example, our classifier for potential vulnerabilites could be used to remove from further consideration 99.97% of functions, while sacrificing 20% of true vulnerabilities. If it required one second to analyze each of the scripts in our dataset without using our models—an estimate that more realistically could be on the order of minutes for static analysis [57] or tens of seconds for dynamic analysis when including execution time inside a web browser [134]—the analysis would take 266 days. In contrast, if the majority of code were filtered out by our classifiers before more heavyweight analysis was applied, the whole process would require slightly less than three days. The majority of the three days would be spent on applying the neural network to the 23 million scripts, which each have 161 functions on average, and only two hours on applying the manual analysis to the scripts identified as vulnerable by our classifier.

Static code linting Rapid, if imprecise, identification of vulnerable functions may be useful in settings where only a small amount of code is analyzed, but performance overhead is particulary undersirable. For instance, our classifiers could be used to identify vulnerable functions during code development, inside an IDE, with no penalty to the user experience. As discussed above, vulnerabilities identified by a classifier could be further investigated using more expensive methods.

5 Summary

We demonstrated an approach for using lightweight machine-learning techniques to detect XSS vulnerabilities in JavaScript source code. We showed that our approach is effective in identifying vulnerable scripts at the cost of relatively few incorrect predictions, even when hampered by faulty or incomplete ground-truth data. We further discussed how our approach can be coupled with additional static or dynamic analyses to dramatically improve

on the execution time of the static or dynamic analyses themselves, potentially enabling more efficient run-time or development-time defenses. In Chapter 6, Section 2, we discuss future work in the area of using machine-learning to detect DOM XSS.

Our results show that deep neural networks using a bag-of-words representation of JavaScript functions are a promising approach for identifying DOM XSS vulnerabilities. One of our models, for example, detects 80% of potential vulnerabilities with a false-positive rate of 0.03%. Compared to other prior work, albeit on different datasets and use cases [82], this represents a large improvement in the false-positive rate while achieving the same true-positive rate. More indicatively of the performance of our classifier, this false-positive rate corresponds to a precision of 78%—which means that for every 100 functions identified as vulnerable, 78 actually are vulnerable. We also empirically show that their performance and accuracy make our classifiers a good building block for even more accurate vulnerability detection mechanisms. For example, if we use our classifiers to selectively enable run-time defenses that have a larger impact on execution time (e.g., dynamic taint tracking [134]), then only 0.17% of scripts would be impacted with a decrease in performance while detecting 80% of confirmed vulnerabilities. For triaging potential vulnerabilities, using our model could decrease the amount of time required to analyze code by 99%.

Chapter 6

Conclusions and Future Work

Here we discuss promising avenues for future work in both of our application areas: guessing passwords using neural networks (Section 1) and using machine learning to detect DOM XSS vulnerabilities (Section 2). In both projects, there are application-specific approaches for furthering the research presented here, in addition to approaches to better leverage ongoing research in the machine-learning community.

1 Password guessing models

Adjusting estimation of password strength Tuning neural networks for password guessing and developing accurate client-side password-strength metrics both remain fertile grounds for further research.

Though our goal was to simulate a smart attacker who is motivated to guess as many passwords as possible in a fixed budget of guessing, attackers may not always guess in such a way. For example, attackers may choose to use commonly available tools, such as Hashcat or John the Ripper, which may guess passwords in a different order—even if strictly less optimal when given a fixed budget of guesses. Because such tools can enable attackers to make more guesses than they otherwise would be able to, this can be a rational strategy for many attackers. Furthermore, attackers may configure such tools manually. It is an open

problem of how to simulate such strategies in an automated way.

While our neural-network password guesser is often better at guessing passwords than other models, it still makes errors. Ideally, we would prefer that those errors are more conservative—underestimating a password's strength rather than overestimating it. It could be that alternate methods of training neural networks could help; one approach would be to train a neural network to directly estimate the guess number of a password, rather than our approach of building a password guesser using neural networks to predict the next character of a password. Such an approach could allow tuning the model to prefer certain types of errors.

In addition, for our meter, we did not proscribe how strong a password must be to be considered safe. Such a determination would need to consider specific details of the deployment that are not known to us, such as: How are passwords stored? What is the risk of database compromise? What proportion of users secure their account with multiple authentication factors? In a guessing attack, compromise is binary—attackers either guess the users' password or not. This binary determination implies that there are certain cutoffs that matter—for example, more guesses than an online attacker could make, or more guesses than an offline attacker could make—and that between cutoffs, there is little or no marginal security benefit for stronger passwords. This could mean that the accuracy of our password meter approach would be better measured by observing when passwords would be correctly classified given this framework (e.g., resistant to an offline attack, or only an online attack, or neither). However, for the general case, the precise value of guessing cutoffs are not obvious.

Leveraging research in machine learning Model compression and distillation are active areas of machine-learning research. We applied relatively straightforward techniques, but more sophisticated techniques for model compression could perform better, such as using soft-weight sharing [147], pruning networks [85], and other various compression techniques [59]. Other work achieves higher compression ratios for neural networks than we do

by using matrix factorization or specialized training methods [59, 164]. Prior work has used neural networks to learn the output of a larger ensemble of models [17] and obtained better results than our network tutoring (Section 4.1).

2 Using machine learning to detect DOM XSS

Detecting vulnerabilities using taint tracking Our methodology for detecting vulnerabilities using taint tracking, described in Chapter 4, can detect many vulnerabilities, but also fails to detect some classes of vulnerabilities. We did not make an attempt to achieve significant code coverage. It is likely that higher code coverage would discover more vulnerabilities. Techniques like dynamic symbolic analysis could be used to determine how an attacker's input could modify execution paths and may be useful for gaining more coverage; however, such techniques are currently of limited effectiveness, especially for reasoning about complex string operations that are common in JavaScript. Other, less sophisticated methods, such as fuzzing, are likely to be more straightforwardly useful in the short term. Finally, our methodology for confirming potentially vulnerable flows is also an area in which further research could improve results. We currently confirm flows by automatically generating test injections in an ad-hoc manner; ideally, we would use a more principled approach to distinguishing potentially vulnerable flows from those that are actually vulnerable.

Considering malicious code We did not consider attacker models involving classifier evasion or poisoning attacks, though such attacks may be possible. Though an attacker would need to control large portions of the training data used, poisoning attacks may be possible, perhaps by leveraging ad networks that could allow scripts to be deployed on a wide variety of sites. Evasion attacks may also be possible, though it is less clear what an attacker would gain. The attacker would need to have enough control over scripts to tweak them such that they evade the classifier, but at the same time, the attacker would seem to have little to gain by evading detection when they already have the ability to exert

control over the script, which could presumably be used to inject their own payload already. Investigating both types of attacks would be useful to better understanding the properties of our approach.

Leveraging research from machine learning Our models for detecting DOM XSS, while promising, are limited by the program representation; our representation was not particularly expressive and did not model complex program relations. More complex representations, such as those that take into account the order of or relationships among different tokens, would allow models to make decisions on more features and could improve our results [3, 4, 92]. Particularly promising are gated-graph neural networks due to their ability to annotate programs with more information, such as that derived from static analysis [3], though such heavyweight annotations would limit the effectiveness of performing classification at run time.

If machine-learning models for detecting injection vulnerabilities were more accurate and had better metrics for precision, such models could be used as part of the development process, rather than a run-time defense—particularly because, even in the presence of a vulnerability, users are unlikely to be under active attack.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems. `https://www.tensorflow.org/`, 2015. Software available from tensorflow.org.

[2] Alexa. Top sites globally. `http://www.alexa.com/topsites/countries/US`, 2017.

[3] Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *Proc. International Conference on Learning Representations* (2018).

[4] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. In *Proc. Symposium on Principles of Programming Languages* (2019).

[5] Password leaks: Alypaa. `https://wiki.skullsecurity.org/Passwords`, 2010.

[6] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. Theano: New features and speed improvements. In *Proc. Neural Information Processing Systems 2012 Deep Learning workshop* (2012).

[7] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: A CPU and GPU math expression compiler. In *Proc. Neural Information Processing Systems 2012 Deep Learning workshop* (2010).

[8] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM 53*, 2 (2010).

[9] Bielik, P., Raychev, V., and Vechev, M. Learning a static analyzer from data. In *Proc. Conference on Computer-Aided Verification* (2017).

[10] Bijjou, K. Web application firewall bypassing—how to defeat the blue team. OWASP open web application security project, 2015.

[11] Bishop, M., and Klein, D. V. Improving system security via proactive password checking. *Computers & Security 14*, 3 (1995).

[12] Blue coat k9. `http://www1.k9webprotection.com/`, 2017.

[13] Bonneau, J. The Gawker hack: How a million passwords were lost. *Light Blue Touchpaper* Blog, December 2010. `http://www.lightbluetouchpaper.org/2010/12/15/the-gawker-hack-how-a-million-passwords-were-lost/`.

[14] Bonneau, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proc. IEEE Symposium on Security and Privacy* (2012).

[15] Bonneau, J. Statistical metrics for individual password strength. In *Proc. International Workshop on Security Protocols* (2012).

[16] Brodkin, J. 10 (or so) of the worst passwords exposed by the LinkedIn hack. *Ars Technica*, June 6, 2012. `http://arstechnica.com/security/2012/06/10-or-so-of-the-worst-passwords-exposed-by-the-linkedin-hack/`.

[17] Buciluǎ, C., Caruana, R., and Niculescu-Mizil, A. Model compression. In *Proc. International Conference on Knowledge Discovery and Data Mining* (2006).

[18] Burnett, M. Xato password set. `https://xato.net/`.

[19] Calzavara, S., Rabitti, A., and Bugliesi, M. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *Proc. Conference on Computer and Communications Security* (2016).

[20] Castelluccia, C., Dürmuth, M., and Perito, D. Adaptive password-strength meters from Markov models. In *Proc. Network and Distributed System Security Symposium* (2012).

[21] Cenzic. Cenzic application vulnerability trends report 2014, 2014.

[22] Chang, J. M. Passwords and email addresses leaked in Kickstarter hack attack. *ABC News*, Feb 17, 2014. `http://abcnews.go.com/Technology/passwords-email-addresses-leaked-kickstarter-hack/story?id=22553952`.

[23] Chen, Q., and Kapravelos, A. Mystique: Uncovering information leakage from browser extensions. In *Proc. Conference on Computer and Communications Security* (2018).

[24] Chollet, F. Keras Github repository. `https://github.com/fchollet/keras`.

[25] Chun, W. WebGL Models: End-to-End. In *OpenGL Insights*. 2012.

[26] Ciaramella, A., D'Arco, P., De Santis, A., Galdi, C., and Tagliaferri, R. Neural network techniques for proactive password checking. *IEEE Transactions on Dependable and Secure Computing 3*, 4 (2006).

[27] Clercq, J. D. Resetting the password of the KRBTGT active directory account, 2014. `http://windowsitpro.com/security/resetting-password-krbtgt-active-directory-account`.

[28] CSDN password leak. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`.

[29] Curtsinger, C., Livshits, B., Zorn, B. G., and Seifert, C. Zozzle: fast and precise in-browser javascript malware detection. In *Proc. USENIX Security* (2011).

[30] Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. The tangled web of password reuse. In *Proc. Network and Distributed System Security Symposium* (2014).

[31] de Carné de Carnavalet, X., and Mannan, M. From very weak to very strong: Analyzing password-strength meters. In *Proc. Network and Distributed System Security Symposium* (2014).

[32] Dell'Amico, M., and Filippone, M. Monte Carlo strength evaluation: Fast and reliable password checking. In *Proc. Conference on Computer and Communications Security* (2015).

[33] Dell'Amico, M., Michiardi, P., and Roudier, Y. Password strength: An empirical analysis. In *Proc. IEEE International Conference on Computer Communications* (2010).

[34] Doupé, A., Cova, M., and Vigna, G. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proc. Conference on Detection of Intrusions and Malware Vulnerability Assessment* (2010).

[35] Duchene, F., Rawat, S., Richier, J.-L., and Groz, R. KameleonFuzz: evolutionary fuzzing for black-box XSS detection. In *Proc. ACM Conference on Data and Application Security and Privacy* (2014).

[36] Duckett, C. Login duplication allows 20m Alibaba accounts to be attacked. *ZDNet*, February 5, 2016. `http://www.zdnet.com/article/login-duplication-allows-20m-alibaba-accounts-to-be-attacked/`.

[37] Dürmuth, M., Angelstorf, F., Castelluccia, C., Perito, D., and Chaabane, A. OMEN: Faster password guessing using an ordered markov enumerator. In *Proc. International Symposium on Engineering Secure Software and Systems* (2015).

[38] Easylist filter for AdBlock. `https://easylist.to/`, 2017.

[39] El Gayar, N., Schwenker, F., and Palm, G. A study of the robustness of knn classifiers trained using soft labels. In *IAPR Workshop on Artificial Neural Networks in Pattern Recognition* (2006).

[40] Password leaks: Elitehacker. `https://wiki.skullsecurity.org/Passwords`, 2009.

[41] esflow: Elegant, fast JavaScript static security analyzer for finding issues like DOM XSS. `https://www.npmjs.com/package/esflow`.

[42] Fahl, S., Harbach, M., Acar, Y., and Smith, M. On the ecological validity of a password study. In *Proc. Symposium on Usable Privacy and Security* (2013).

[43] Faith writer leak. `https://wiki.skullsecurity.org/Passwords#Leaked_passwords`.

[44] Florêncio, D., Herley, C., and van Oorschot, P. C. An administrator's guide to internet password research. In *Proc. USENIX Large Installation System Administration Conference* (2014).

[45] Forman, G., and Scholz, M. Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. *ACM SIGKDD Explorations Newsletter 12*, 1 (2010).

[46] Fox-Brewster, T. 13 million passwords appear to have leaked from this free web host. *Forbes*, October 28, 2015. `http://www.forbes.com/sites/thomasbrewster/2015/10/28/000webhost-database-leak/`.

[47] Gailly, J.-L. gzip. `http://www.gzip.org/`.

[48] Goodin, D. 10,000 Hotmail passwords mysteriously leaked to web. *The Register*, October 5, 2009. `http://www.theregister.co.uk/2009/10/05/hotmail_passwords_leaked/`.

[49] Goodin, D. Hackers expose 453,000 credentials allegedly taken from Yahoo service. *Ars Technica*, July 12, 2012. `http://arstechnica.com/security/2012/07/yahoo-service-hacked/`.

[50] Goodin, D. Anatomy of a hack: How crackers ransack passwords like "qeadzcwrsfxv1331". *Ars Technica*, May 27, 2013. `http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/`.

[51] Goodin, D. Why LivingSocial's 50-million password breach is graver than you may think. *Ars Technica*, April 27, 2013. `http://arstechnica.com/security/2013/04/why-livingsocials-50-million-password-breach-is-graver-than-you-may-think/`.

[52] Goodin, D. Once seen as bulletproof, 11 million+ Ashley Madison passwords already cracked. *Ars Technica*, September 10, 2015. `http://arstechnica.com/security/2015/09/once-seen-as-bulletproof-11-million-ashley-madison-passwords-already-cracked/`.

[53] Google. Web 1T 5-gram version 1, 2006. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13.

[54] Graves, A. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012.

[55] Graves, A. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850, 2013.

[56] Greenberg, A. The police tool that pervs use to steal nude pics from Apple's iCloud. *Wired*, September 2, 2014. https://www.wired.com/2014/09/eppb-icloud/.

[57] Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *Proc. International Symposium on Software Testing and Analysis* (2011).

[58] Hak5 leak. https://wiki.skullsecurity.org/Passwords#Leaked_passwords.

[59] Han, S., Mao, H., and Dally, W. J. A deep neural network compression pipeline: Pruning, quantization, Huffman encoding. arXiv preprint arXiv:1510.00149, 2015.

[60] Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Proc. Neural Information Processing Systems* (2015).

[61] Henry, A. Five best password managers. *LifeHacker*, January 11, 2015. http://lifehacker.com/5529133/.

[62] Herley, C., and van Oorschot, P. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy Magazine 10*, 1 (Jan. 2012).

[63] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997).

[64] Hunt, T. A brief Sony password analysis. http://www.troyhunt.com/2011/06/brief-sony-password-analysis.html, 2011.

[65] Huynh, T. ABC Australia hacked – nearly 50,000 user credentials posted online, half cracked in 45 secs. *Techgeek*, February 27, 2013. http://techgeek.com.au/2013/02/27/abc-australia-hacked-nearly-50000-user-credentials-posted-online/.

[66] Inc., F. Content security policy reference. https://content-security-policy.com/, 2016.

[67] Ivanov, V. Web application firewalls: Attacking detection logic mechanisms. Blackhat USA, 2016.

[68] Johnstone, L. 9,885 user accounts leaked from Intercessors for America by Anonymous. http://www.cyberwarnews.info/2013/07/24/9885-user-accounts-leaked-from-intercessors-for-america-by-anonymous/, 2013.

[69] Jones, A. On widespread XSS in ad networks. https://blogs.msmvps.com/alunj/2016/04/09/on-widespread-xss-in-ad-networks/, 2017.

[70] Jozefowicz, R., Zaremba, W., and Sutskever, I. An empirical exploration of recurrent network architectures. In *Proc. International Conference on Machine Learning* (2015).

[71] Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., and Lopez, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. IEEE Symposium on Security and Privacy* (2012).

[72] Kim, K., Kim, I. L., Kim, C. H., Kwon, Y., Zheng, Y., Zhang, X., and Xu, D. J-Force: Forced execution on JavaScript. In *Proc. The Web Conference (WWW)* (2017).

[73] Komanduri, S. *Modeling the adversary to evaluate password strengh with limited samples*. PhD thesis, Carnegie Mellon University, 2016.

[74] Koster, M. The web robots pages. http://www.robotstxt.org/, 2017.

[75] Kotowicz, K. Trusted types help prevent cross-site scripting. https://developers.google.com/web/updates/2019/02/trusted-types, 2019.

[76] Krebs, B. Fraud bazaar carders.cc hacked. http://krebsonsecurity.com/2010/05/fraud-bazaar-carders-cc-hacked/.

[77] Lai, S., Xu, L., Liu, K., and Zhao, J. Recurrent convolutional neural networks for text classification. In *AAAI conference on artificial intelligence* (2015).

[78] Lee, M. Hackers have released what they claim are the details of over 21,000 user accounts belonging to Billabong customers. *ZDNet*, July 13, 2012. http://www.zdnet.com/article/over-21000-plain-text-passwords-stolen-from-billabong/.

[79] Lekies, S., Stock, B., and Johns, M. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proc. Conference on Computer and Communications Security* (2013).

[80] Lekies, S., Stock, B., and Johns, M. Research talk: 25 million flows later: Detection and exploitation of DOM-based XSS. https://www.youtube.com/watch?v=FxJAeWOop1Y, 2015.

[81] Li, Y., Zemel, R., Brockschmidt, M., and Tarlow, D. Gated graph sequence neural networks. In *Proc. International Conference on Learning Representations* (2016).

[82] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proc. Network and Distributed System Security Symposium* (2018).

[83] Liang, B., Su, M., You, W., Shi, W., and Yang, G. Cracking classifiers for evasion: A case study on the Google's phishing pages filter. In *Proc. The Web Conference (WWW)* (2016).

[84] Lowerre, B. T. *The HARPY speech recognition system*. PhD thesis, Carnegie Mellon University, 1976.

[85] Luo, J.-H., Wu, J., and Lin, W. Thinet: A filter level pruning method for deep neural network compression. In *Proc. IEEE International Conference on Computer Vision* (2017).

[86] Ma, J., Yang, W., Luo, M., and Li, N. A study of probabilistic password models. In *Proc. IEEE Symposium on Security and Privacy* (2014).

[87] Mazurek, M. L., Komanduri, S., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., Kelley, P. G., Shay, R., and Ur, B. Measuring password guessability for an entire university. In *Proc. Conference on Computer and Communications Security* (2013).

[88] McAllister, N. Twitter breach leaks emails, passwords of 250,000 users. *The Register*, Feb 2, 2013.

[89] Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In *Proc. Network and Distributed System Security Symposium* (2018).

[90] Mikolov, T., Sutskever, I., Deoras, A., Le, H.-S., Kombrink, S., and Cernocky, J. Subword language modeling with neural networks. Preprint (`http://www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf`), 2012.

[91] Mitzenmacher, M. Compressed Bloom filters. *IEEE/ACM Transactions on Networking (TON) 10*, 5 (2002).

[92] Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *AAAI Conference on Artificial Intelligence* (2016).

[93] Msgpack: It's like JSON but fast and small. `http://msgpack.org/index.html`.

[94] Narayanan, A., and Shmatikov, V. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. Conference on Computer and Communications Security* (2005).

[95] Neef, S. Using neural networks for password cracking. Blog post. `https://0day.work/using-neural-networks-for-password-cracking/`, 2016.

[96] Nelson, B., Barreno, M., Chi, F. J., Joseph, A. D., Rubinstein, B. I. P., Saini, U., Sutton, C., Tygar, J. D., and Xia, K. Exploiting machine learning to subvert your spam filter. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats* (2008).

[97] Neocortex Github repository. https://github.com/scienceai/neocortex.

[98] Newell, A., Potharaju, R., Xiang, L., and Nita-Rotaru, C. On the practicality of integrity attacks on document-level sentiment analysis. In *Proc. Workshop on Artificial Intelligence and Security* (2014), AISec '14.

[99] Nielsen, J., and Hackos, J. T. *Usability engineering*. Academic press Boston, 1993.

[100] Nielsen, M. A. *Neural networks and deep learning*. Determination Press, 2015.

[101] OWASP. Cross-site scripting. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[102] OWASP. Web application firewall. https://www.owasp.org/index.php/Web_Application_Firewall, 2016.

[103] OWASP. DOM based XSS. https://www.owasp.org/index.php/DOM_Based_XSS, 2017.

[104] OWASP. Static code analysis. https://www.owasp.org/index.php/Static_Code_Analysis, 2017.

[105] Paola, S. D. DOMinator. https://github.com/wisec/DOMinator, 2011.

[106] Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., and Saxena, P. Auto-patching dom-based xss at scale. In *Proc. ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2015).

[107] Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., and Saxena, P. DexterJS: robust testing platform for DOM-based XSS vulnerabilities. In *Proc. ESEC/FSE* (2015).

[108] Patnaik, N., and Sahoo, S. JavaScript static security analysis made easy with JSPrime. *Black Hat USA* (2013).

[109] Perdisci, R., Dagon, D., Lee, W., Fogla, P., and Sharif, M. Misleading worm signature generators using deliberate noise injection. In *Proc. IEEE S&P* (2006).

[110] Perl monks password leak. http://news.softpedia.com/news/PerlMonks-ZF0-Hack-Has-Wider-Implications-118225.shtml, 2009.

[111] Perlroth, N. Adobe hacking attack was bigger than previously thought. *The New York Times Bits Blog*, October 29, 2013. http://bits.blogs.nytimes.com/2013/10/29/adobe-online-attack-was-bigger-than-previously-thought/.

[112] Peslyak, A. John the Ripper. http://www.openwall.com/john/, 1996.

[113] PHPBB password leak. https://wiki.skullsecurity.org/Passwords, 2009.

[114] Podjarny, G. Snyk blog: XSS attacks: The next wave. https://snyk.io/blog/xss-attacks-the-next-wave/, 2017.

[115] Protalinski, E. 8.24 million Gamigo passwords leaked after hack. *ZDNet*, July 23, 2012. http://www.zdnet.com/article/8-24-million-gamigo-passwords-leaked-after-hack/.

[116] Protocol buffer encoding. https://developers.google.com/protocol-buffers/docs/encoding, 2008.

[117] pythech. Yet another Chrome XSS auditor bypass. https://turkmenog.lu/blog/2017/11/06/yet-another-chrome-xss-auditor-bypass/, 2017.

[118] Python parser for Adblock Plus filters. https://github.com/scrapinghub/adblockparser, 2017.

[119] Ragan, S. Mozilla's bug tracking portal compromised, reused passwords to blame. *CSO*, September 4, 2015. http://www.csoonline.com/article/2980758/.

[120] Richards, G., Lebresne, S., Burg, B., and Vitek, J. An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices* (2010), vol. 45, ACM.

[121] Rndic, N., and Laskov, P. Practical evasion of a learning-based classifier: A case study. In *Proc. IEEE S&P* (2014).

[122] ScanJS. https://github.com/mozfreddyb/eslint-config-scanjs, 2017.

[123] Schneier, B. Myspace passwords aren't so dumb. http://www.wired.com/politics/security/commentary/securitymatters/2006/12/72300, 2006.

[124] scikit learn. scikit learn: Feature extraction. https://scikit-learn.org/stable/modules/feature_extraction.html, 2019.

[125] SCOWL. Spell checker oriented word lists. http://wordlist.sourceforge.net, 2015.

[126] Security, P. W. Burp Suite. https://portswigger.net/burp.

[127] Shay, R., Bauer, L., Christin, N., Cranor, L. F., Forget, A., Komanduri, S., Mazurek, M. L., Melicher, W., Segreti, S. M., and Ur, B. A spoonful of sugar? The impact of guidance and feedback on password-creation behavior. In *Proc. Conference on Human Factors in Computing Systems* (2015).

[128] Shay, R., Komanduri, S., Durity, A. L., Huh, P. S., Mazurek, M. L., Segreti, S. M., Ur, B., Bauer, L., Christin, N., and Cranor, L. F. Can long passwords be secure and usable? In *Proc. Conference on Human Factors in Computing Systems* (2014).

[129] Son, S., and Shmatikov, V. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *Proc. Network and Distributed System Security Symposium* (2013).

[130] Specialforces.com password leak. `http://www.databreaches.net/update-specialforces-com-hackers-acquired-8000-credit-card-numbers/`, 2011.

[131] SpiderOak. Zero knowledge cloud solutions. `https://spideroak.com/`, 2016.

[132] Staicu, C.-A., Pradel, M., and Livshits, B. Synode: Understanding and automatically preventing injection attacks on node. js. In *Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science* (2016).

[133] Steube, J. Hashcat. `https://hashcat.net/oclhashcat/`, 2009-.

[134] Stock, B., Lekies, S., Mueller, T., Spiegel, P., and Johns, M. Precise client-side protection against DOM-based cross-site scripting. In *Proc. USENIX Security* (2014).

[135] Stock, B., Pfistner, S., Kaiser, B., Lekies, S., and Johns, M. From facepalm to brain bender: exploring client-side cross-site scripting. In *Proc. Conference on Computer and Communications Security* (2015).

[136] Stratfor leak. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`, 2011.

[137] Suto, L. Analyzing the accuracy and time costs of web application security scanners.

[138] Sutskever, I., Martens, J., and Hinton, G. E. Generating text with recurrent neural networks. In *Proc. International Conference on Machine Learning* (2011).

[139] Taly, A., Erlingsson, Ú., Mitchell, J. C., Miller, M. S., and Nagra, J. Automated analysis of security-critical JavaScript APIs. In *Proc. IEEE S&P* (2011).

[140] The chromium project: Xss auditor. `https://www.chromium.org/developers/design-documents/xss-auditor`, 2010.

[141] Tripp, O., Ferrara, P., and Pistoia, M. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proc. International Symposium on Software Testing and Analysis* (2014).

[142] Tripp, O., Guarnieri, S., Pistoia, M., and Aravkin, A. Aletheia: Improving the usability of static security analysis. In *Proc. Conference on Computer and Communications Security* (2014).

[143] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. Taj: effective taint analysis of web applications. In *ACM Sigplan Notices* (2009), vol. 44.

[144] Trustwave. eHarmony password dump analysis, June 2012. http://blog.spiderlabs.com/2012/06/eharmony-password-dump-analysis.html.

[145] Trustwave Spiderlabs. SpiderLabs/KoreLogic-Rules. https://github.com/SpiderLabs/KoreLogic-Rules, 2012.

[146] Tsukayama, H. Evernote hacked; millions must change passwords. *Washington Post*, March 4, 2013. https://www.washingtonpost.com/8279306c-84c7-11e2-98a3-b3db6b9ac586_story.html.

[147] Ullrich, K., Meeds, E., and Welling, M. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008* (2017).

[148] Ur, B., Kelley, P. G., Komanduri, S., Lee, J., Maass, M., Mazurek, M., Passaro, T., Shay, R., Vidas, T., Bauer, L., Christin, N., and Cranor, L. F. How does your password measure up? The effect of strength meters on password creation. In *Proc. USENIX Security* (2012).

[149] Ur, B., Segreti, S. M., Bauer, L., Christin, N., Cranor, L. F., Komanduri, S., Kurilova, D., Mazurek, M. L., Melicher, W., and Shay, R. Measuring real-world accuracies and biases in modeling password guessability. In *Proc. USENIX Security* (2015).

[150] Using Web workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. Accessed:2016.

[151] Vance, A. If your password is 123456, just make it hackme. *New York Times*, January 20, 2010. http://www.nytimes.com/2010/01/21/technology/21password.html.

[152] Varda, K. Cap'n proto. https://capnproto.org/, 2019.

[153] Veras, R., Collins, C., and Thorpe, J. On the semantic patterns of passwords and their security impact. In *Proc. Network and Distributed System Security Symposium* (2014).

[154] Wassermann, G., and Su, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices* (2007), vol. 42, ACM.

[155] The "web2" file of English words. http://www.bee-man.us/computer/grep/grep.htm#web2, 2004.

[156] Wei, S., and Ryder, B. G. Practical blended taint analysis for JavaScript. In *Proc. International Symposium on Software Testing and Analysis* (2013), ACM.

[157] Weinberger, K., Dasgupta, A., Attenberg, J., Langford, J., and Smola, A. Feature hashing for large scale multitask learning. *arXiv preprint arXiv:0902.2206* (2009).

[158] Weir, M., Aggarwal, S., Collins, M., and Stern, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proc. Conference on Computer and Communications Security* (2010).

[159] Weir, M., Aggarwal, S., Medeiros, B. d., and Glodek, B. Password cracking using probabilistic context-free grammars. In *Proc. IEEE Symp. Security & Privacy* (2009).

[160] Wheeler, D. zxcvbn: Realistic password strength estimation. `https://blogs.dropbox.com/tech/2012/04/zxcvbn-realistic-password-strength-estimation/`, 2012.

[161] Wheeler, D. L. zxcvbn: Low-budget password strength estimation. In *Proc. USENIX Security* (2016).

[162] Wittel, G. L., and Wu, S. F. On attacking statistical spam filters. In *Proc. Conference on Email and Anti-Spam* (2004).

[163] WOM Vegas password leak, 2013. `https://www.hackread.com/wom-vegas-breached-10000-user-accounts-leaked-by-darkweb-goons/`.

[164] Xue, J., Li, J., Yu, D., Seltzer, M., and Gong, Y. Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *Proc. International Conference on Acoustics, Speech and Signal Processing* (2014).

[165] Yamaguchi, F., Lindner, F., and Rieck, K. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proc. USENIX Workshop on Offensive Technologies* (2011).

[166] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. How transferable are features in deep neural networks? In *Proc. Neural Information Processing Systems* (2014).

[167] YouPorn password leak, 2012. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`.

[168] Zheng, X., Jiang, J., Liang, J., and Duan, H.-X. Cookies lack integrity: Real-world implications. In *Proc. USENIX Security* (2015).