Defending Against Web Application Attacks: Approaches, Challenges and Implications

Dimitris Mitropoulos, Panos Louridas, Michalis Polychronakis, and Angelos Dennis Keromytis

Abstract—Some of the most dangerous web attacks, such as Cross-Site Scripting and SQL injection, exploit vulnerabilities in web applications that may accept and process data of uncertain origin without proper validation or filtering, allowing the injection and execution of dynamic or domain-specific language code. These attacks have been constantly topping the lists of various security bulletin providers despite the numerous countermeasures that have been proposed over the past 15 years. In this paper, we provide an analysis on various defense mechanisms against web code injection attacks. We propose a model that highlights the key weaknesses enabling these attacks, and that provides a common perspective for studying the available defenses. We then categorize and analyze a set of 41 previously proposed defenses based on their accuracy, performance, deployment, security, and availability characteristics. Detection accuracy is of particular importance, as our findings show that many defense mechanisms have been tested in a poor manner. In addition, we observe that some mechanisms can be bypassed by attackers with knowledge of how the mechanisms work. Finally, we discuss the results of our analysis, with emphasis on factors that may hinder the widespread adoption of defenses in practice.

Index Terms—Web application security, protection mechanisms, exploitation models, software testing, SQL injection, XSS

1 Introduction

Web application attacks may involve security misconfigurations, broken authentication and session management, or other issues. Some of the most dangerous and prevalent web application attacks, however, exploit vulnerabilities associated with improper validation or filtering of untrusted inputs, resulting in the injection of malicious script or domain-specific language code. Attacks of this type include Cross-Site Scripting (XSS) [1], and SQL injection attacks [2], among others.

For the past several years, these attacks have been topping the lists of the most dangerous vulnerabilities published by OWASP,[1] MITRE,[2] and other organizations. For instance, consider the case of OWASP's popular Top Ten project, which aims to raise awareness about web application security by identifying some of the most critical risks organizations may face. In its three consecutive Top Ten lists (2007, 2010, 2013), different injection attacks dominate the top five positions.

At the same time, attackers find new ways [3], [4] to bypass defense mechanisms using a variety of techniques, despite the numerous countermeasures that are being introduced. As an example, already by 2006, there were more than 20 proposed defenses against SQL injection attacks [5]. Since then, the number has doubled, while researchers have indicated that the number of SQL injection attacks has been steadily increasing in recent years [6].

In this paper, we explore how different attacks associated with the exploitation of untrusted input validation errors can be modeled under a *common perspective*. To that end, we propose an exploitation model which highlights that most of the steps needed to mount different types of code injection attacks are common.[3] This is validated by the fact that some protection mechanisms defend against more than one of these types of attacks.

Then, we *categorize* a selection of representative protection mechanisms. In our selection we include protection mechanisms that counter web attacks when they take place, while we do not consider countermeasures that identify vulnerabilities using static program analysis [7] (which takes place during the development or testing phases). Similarly, dynamic analysis techniques that examine applications to identify vulnerabilities that may lead to the attacks that we described earlier are out of scope as well. Note also, that we did not consider mechanisms that have not been presented in research papers.

Furthermore, we *analyze* each mechanism across the following dimensions:

- *Accuracy:* protection mechanisms are as good as their detection capability; this requires low false positive and false negative rates.

1. https://www.owasp.org/index.php/Top_10_2013-Top_10
2. http://cwe.mitre.org/top25/

3. Note that we do not consider lower-level attacks based on the exploitation of memory corruption vulnerabilities and the injection of binary code—see Section 2.

- *D. Mitropoulos and A.D. Keromytis are with the Department of Computer Science, Columbia University, New York, NY 10027. E-mail: {dimitro, angelos}@cs.columbia.edu.*
- *P. Louridas is with the Department of Management Science and Technology, Athens University of Economics and Business, Athina 104 34, Greece. E-mail: louridas@aueb.gr.*
- *M. Polychronakis is with the Computer Science Department, Stony Brook University, Stony Brook, NY 11794. E-mail: mikepo@cs.stonybrook.edu.*

Manuscript received 18 Mar. 2016; revised 19 Sept. 2016; accepted 29 Jan. 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2017.2665620

- *Availability:* whether the protection mechanism and its testbed are publicly available.
- *Performance overhead:* the overhead imposed by the mechanisms at their points of deployment.
- *Ease of use:* whether the mechanism is practical in terms of deployment and can be easily adopted by security experts.
- *Security:* the robustness of the protection mechanism against attackers with knowledge of its internals who attempt to circumvent it.
- *Detection Point:* the location where a mechanism detects an attack based on our exploitation model.

All the above requirements are considered important when building mechanisms for the protection of applications [8], [9], [10]. Based on this analysis, we identify the advantages and disadvantages of the various mechanisms and enumerate some of their common characteristics. We also draw useful conclusions about the various protection categories and see how they compare to each other. In addition, we attempt to shed light on the factors that may impede the adoption of defenses in practice. Finally, we provide some lessons and recommendations that developers of new defenses may find helpful.

The main contributions of this paper are the following:

1) We provide a unified exploitation model for different types of web application attacks based on code injection.
2) We categorize and analyze proposed defenses using a set of criteria that are important for building protection mechanisms.
3) We provide insight based on the issues that arise from our analysis. We put emphasis on factors that may hinder the widespread deployment of protection mechanisms, and the transition of tools from research to practice.

The rest of the paper is organized as follows: Section 2 provides some insights on web code injection attacks and Section 3 presents our proposed model. Section 4 introduces the dimensions across which we analyze the various defenses. Our categorization and analysis is presented in Section 5 and our observations are provided in Section 6. Finally, Section 7 highlights some lessons learned from our observations and Section 8 concludes the paper.

2 Code Injection Attacks in Web Applications

Lack of input validation is a major vulnerability behind dangerous web application attacks. By taking advantage of this, attackers can inject their code into applications to perform malicious tasks. Exploits of this kind can have different forms depending on the execution context of the application and the location of the programming flaw that leads to the attack.

Bratus et al. [11] portray the issue in a more generic way: *"unexpected (and unexpectedly powerful) computational models inside targeted systems, which turn a part of the target into a so-called 'weird machine' programmable by the attacker via crafted inputs (a.k.a. 'exploits')."* In particular, *"every application that copies untrusted input verbatim into an output program is vulnerable to code injection."* Ray and Ligatti [2] have proved this claim based on formal language theory.

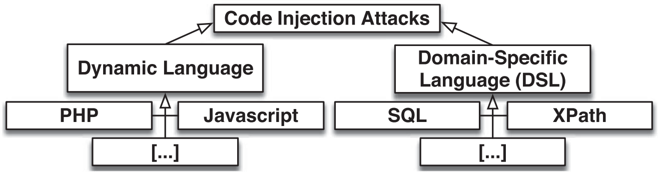

Fig. 1. A taxonomy of dynamic and domain-specific language code injection attacks against web applications.

Code injection attacks can be divided in two categories. The first involves binary code and the second higher-level language code. An extensive survey on binary code injection attacks was conducted by Lhee and Chapin [12]. Advances in memory corruption vulnerability exploitation have been studied extensively [9] and countermeasures to such attacks have already been analyzed [13]. In this work we do not consider binary code injection, focusing instead on defenses that protect web applications against attacks based on the injection of higher-level language code.

Fig. 1 presents a taxonomy of source code injection attacks against web applications. Such attacks may involve high-level language code, written in either a *Domain Specific Language* (DSL) or a *Dynamic Language*. To illustrate, we discuss examples from both categories that will be used throughout the paper.

Injection attacks that involve DSLs constitute an important subset of the code injection problem, as DSLs such as SQL and XML play a significant role in the development of both web and mobile applications. For example, many applications have interfaces through which a user enters input to interact with the application, thereby interacting with the underlying database. This input can become part of an SQL query and gets executed on the target database. Code injection attacks that exploit vulnerabilities in database interfaces by taking advantage of input validation issues, such as incorrectly passed parameters or incorrect type handling, are called SQL injection attacks [1], [5]. Consider a trivial exploit that takes advantage of incorrectly filtered quotation characters in an application that shows the password of a forgetful user by executing the following query:

```
SELECT password FROM userdata WHERE id = 'Alice'
```

Attackers that would input the string `anything' OR 'x'='x` could view every item in the table. Savvy programmers can use certain API functions, such as PHP's `mysql_real_escape_string()`, to detect malformed input, or, better, use prepared SQL statements instead of statement templates. Unfortunately, the increasing number of SQL injection attacks suggests that programmers are not always that careful. Using similar techniques, malicious users can mount other exploits based on DSLs such as XPath [1], XML and JSON [14]. The effects can be wide-ranging. A malicious user can view sensitive information, destroy or modify protected data, or even crash the entire application.

HTML is another DSL that can be used for malicious purposes when an application does not properly handle user-supplied data. Based on this vulnerability attackers can supply valid HTML, typically via a parameter value, and inject their own content into the application's page. HTML injection is mainly associated with XSS attacks [15]. However, HTML injection can also be used as a vehicle for Cross-Site Request

Forgery (CSRF) attacks [16] (even though a common CSRF attack does not necessarily involve code injection). Consider a bulletin board system where img tags are allowed. A malicious user could embed a CSRF request within an img tag in the following manner:

```
<img src='http://www.vulnerable.com/admin.php?
edituserwithID=13&addgroup=admin'/>
```

When the page with this injected code is accessed by an administrator, the attacker (with ID 13) will gain administrative privileges over the vulnerable.com web page, while the administrator of the bulletin board system will have no immediate indication that there has been an attack.

A recent class of code injection attacks involve dynamic languages such as JavaScript and PHP [14]. JavaScript injection attacks make up a large subset of dynamic language code injection attacks and are considered a critical issue in web application security mainly because they are associated with major vulnerabilities such as XSS attacks and Cross-Channel Scripting (XCS) attacks [17]. Such attacks are enabled when a web application accepts and redisplays data of uncertain origin without proper validation and filtering. Based on this flaw, an attacker can manage to inject a script in the JavaScript engine of a browser and alter its execution flow. For a typical XSS example that involves JavaScript injection consider a web page that prints the value of a query parameter (query) from the page's URL as part of the page's content without escaping the value. Attackers can take advantage of this and inject an iframe tag into the page to steal a user's cookie and send it via an image request to a web site under their control (malicious.com). This could be achieved by including the following link to the malicious web site (or sending it via phishing email) and inducing the user to click on it:

```
http://example.com/vulnerable.html?query=
      <iframe src="javascript:document.body.
      innerHTML=+'<img src="http://malicious.
      com/?c='+encodeURIComponent(document.
      cookie)+'\">"'></iframe>
```

Note that in many cases XSS attacks involve the injection of both HTML and JavaScript code.

A simple example of a PHP injection attack is an input string that is fed into an eval() function call, e.g.:

```
$variable = $_GET['var'];
$input = $_GET['value'];
eval('$variable = ' . $input . ';')
```

The user may pass into the value parameter code that will be executed on the server. Hence, if an attacker provides as input the following string: `10 ; system("touch foo");` then a file will be created on the server—it is easy to imagine more detrimental scenarios.

A recent attack called PHP Object Injection (POI) [4] does not directly involve the injection of code, but still achieves arbitrary code execution in the context of a PHP application through the injection of specially crafted objects (e.g., as part of cookies). When deserialized by the application, these objects result in arbitrary code execution. Note that the exploitation model we propose in the following section also captures such attacks.

3 Exploitation Model

We provide a step-by-step exploitation model to aid in understanding the process of carrying out code injection attacks against web applications. Fig. 2 illustrates the required steps for different classes of attacks, such as SQL injection, XSS, and CSRF. Links between steps are labeled with the different attacks that use that path, while the numbers next to attack labels denote the sequence of steps for a particular attack. Most steps are common in all attacks, as different attack types follow similar exploitation paths, as mentioned in the previous section. Large 'X' marks on transitions correspond to the points where the defenses we consider in this work detect or prevent attacks.

An attacker can initiate an injection attack through two main routes. One way is to use the browser of a victim as an attack vehicle, through which the code will be injected in the application. For example, the attacker could embed a malicious script into a URL and then trick a user to click on it through social engineering, e.g., by sending a phishing email (transition P-XSS 1.1, N-XSS 1.1, CSRF 1). Alternatively, the attacker may be able to inject directly the malicious code on the server through an HTTP request (DSL 1, P-XSS 1, N-XSS 1, I-CSRF 1). This would happen in a web application that accepts and processes user input without appropriate validation. An attacker could upload data containing a specially crafted script to steal the cookies of the visiting users (P-XSS 1, N-XSS 1), an img tag including a malicious HTTP request (CSRF 1, I-CSRF 1), or embed malicious SQL code to retrieve private data from a database (DSL 1). Note that XSS and CSRF attacks can start from both routes. Once the injected code reaches the vulnerable application, it becomes a part of a value represented by a program variable. The target of the attack determines the route from that point and on. In an SQL injection attack, the injected code becomes part of a query that eventually reaches the database where it is executed.

Cross-site scripting attacks fall into three categories, *non-persistent* (also known as *reflected*) XSS, *persistent* (also known as *stored*) XSS and *Document Object Model* (DOM)*-based* XSS [15]. Non-persistent XSS attacks take place when the data provided by a user is processed on-the-fly by server-side application logic and ends up without proper sanitization into a dynamically generated response (P-XSS 7, N-XSS 2) that is eventually rendered by the user's browser. Thus, in a non-persistent XSS attack, the injected code is not saved on the server, but immediately becomes part of the content that is sent back to a user. In persistent XSS attacks, on the other hand, malicious code is permanently stored on the server (P-XSS 5). The injected code residing at server-side then re-enters the application's execution flow and becomes part of the content that is eventually sent to the user as part of a future response. DOM-based XSS attacks involve the modification of the DOM of a webpage. The DOM treats an HTML document as a tree structure where each node is an object representing a part of the document. Each object can be accessed and manipulated programmatically and any visible changes may then be reflected in the browser. In a DOM-based XSS attack, the malicious payload (e.g., hidden in a well-crafted URL that is sent to a user via phishing: N-XSS 1.1) is executed as a result of the manipulation of the DOM

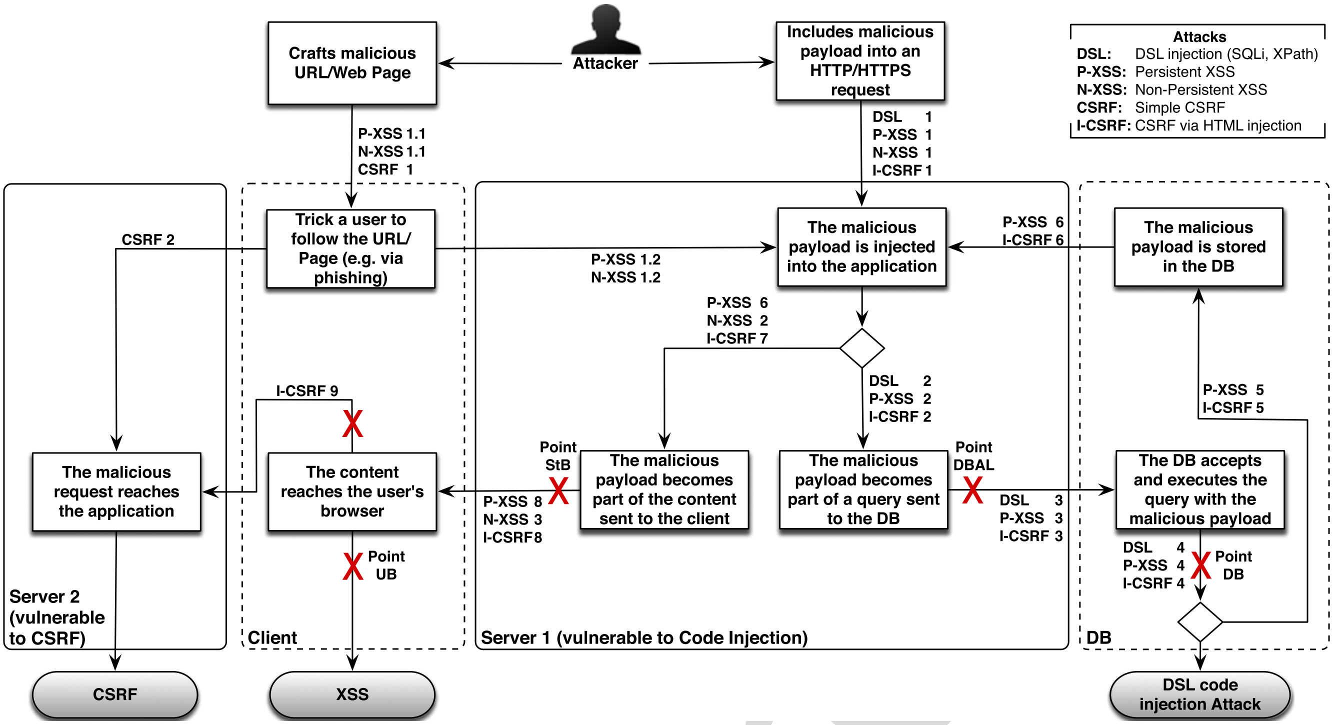

Fig. 2. Attack model for dynamic and domain specific language code injection in web applications. Transitions are labeled with the different attacks that use that path, while the numbers next to attack labels denote the sequence of steps for a particular attack. Points where different types of defenses detect or prevent attacks are marked with an 'X' symbol: (1) UB (at the browser): [15], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], (2) StB (en route from the server to the browser): [36], [37], [38], [39], [40], [41], [42], [43], (3) DBAL (at the database abstraction layer): [1], [14], [44], [45], [46], [47], [48], [49], [50], (4) DB (at the database): [35], [51], [52], [53], [54], [55].

environment (e.g., when another flawed script accesses the modified DOM object) and is not contained in the HTTP response. That is, the malicious payload never reaches the server. Such an attack can be performed purely client-side across HTML frames. Hence, server-side defenses might not be effective in this case.

A Cross Frame Scripting (XFS) attack is a recent threat that combines a malicious script with an `iframe` that loads a legitimate page in an effort to steal data from a user. Consider an attacker that lures via social engineering a user to navigate to a web page the attacker controls. The attacker's page then loads JavaScript and an HTML `iframe` pointing to a legitimate site. Once the user enters his or her credentials into the legitimate site, the malicious script records all keystrokes.

CSRF attacks typically involve just a URL or page that contains a malicious request towards a site vulnerable to CSRF. An attacker can entice a victim to click on the URL or visit the page through social engineering (CSRF 1), which will the result in a malicious request towards the vulnerable site (CSRF 2). Note that this scenario does not involve the injection of any code through the exploitation of some server-side vulnerability (as is the case with Server 1). On the other hand, CSRF is also possible through injection (I-CSRF—recall our example in the previous section). In this attack, an attacker manages to inject HTML code that contains a malicious CSRF request in the way we described in Section 2. This code will be stored in the database (I-CSRF 5) and then it will re-enter the application when a user visits a page (I-CSRF 6). When the content reaches the browser, a malicious request will be generated towards the site that is vulnerable to CSRF.

4 Analysis Dimensions

Research on web application attack defense mechanisms has a dual character. First, a defense mechanism may be important, and therefore publishable, because it shows that an attack can be detected or prevented in a reliable manner. Second, a defense mechanism may be important not just as a research contribution, but as a practical tool, if it can be used by administrators and users to shield their applications against the supported attacks.

The detection and prevention of attacks in a reliable manner can be analyzed using criteria common with other research fields:

- Statistical measurements that show how reliable the detection really is.
- Research practices that promote replication and validation of the findings.

Whether a reliable defense mechanism has value in a practical setting rests on a different set of criteria:

- What are the overheads imposed by the mechanism?
- How easy is it to deploy and use the mechanism?
- How robust is it against ways to circumvent it?
- At which point throughout the system does it block an attack?

The first two criteria correspond to *Accuracy* and *Availability*. The other four criteria correspond to *Runtime Performance*, *Ease of Use*, *Security*, and *Point of Detection*.

4.1 Accuracy

Web application defenses crucially have to capture the presence of an attack. This, however, does not make a defense mechanism immediately useful. A detection mechanism must also be reliable. Detection accuracy is gauged with the following metrics [8], [56]:

- *Sensitivity*: the probability that an attack will be caught.
- *Specificity*: the probability that a normal interaction will not be flagged.
- *Positive Predictive Value* (PPV): the probability that a reported attack is a real attack. It is the conditional probability that an event is an attack if the detection mechanism flags it as such.
- *Negative Predictive Value* (NPV): the probability that if nothing is reported, no attack has taken place. It is the conditional probability that an event is not an attack given that the detection mechanism flags it as normal.

We will focus on sensitivity and specificity; we will come back to PPV and NPV in Section 7. Sensitivity and specifity are defined using the following [57]:

- True Positive (TP): an attack that raises an alarm.
- True Negative (TN): an event that is not an attack and that does not raise an alarm.
- False Positive (FP): an event that although it is not an attack, raises an alarm.
- False Negative (FN): an event that although is an attack, does not raise an alarm.

So that we can calculate:

$$\text{SE} = \text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1}$$

$$\text{SP} = \text{Specificity} = \frac{\text{TN}}{\text{FP} + \text{TN}}. \tag{2}$$

Sensitivity and specificity can be calculated based on test data alone. To calculate sensitivity, we run the test on a controlled environment where we allow only attack events to reach the system. The ratio of reported attacks over all attacks will give us the sensitivity. Similarly, to calculate the specificity we can run the test on a controlled environment where we allow only innocuous events to reach the system. The ratio of non-reported events over all events will give us the specificity. Note that the use of sensitivity and specificity in this context has been advocated before [58].

4.2 Availability

To ensure reproducibility of research results, the source code implementing a detection mechanism should be available to researchers. Ideally the code should be available under an open source license, so that it is easy to modify and improve an approach; even when this is not possible, for whatever reason, it is important to make sure that all computer code and test data is available, noting any restrictions on accessibility. Availability of computer code has been recognized as an important issue outside the computer science field: the journal *Nature* has adopted a publication policy stressing access to code and data [10]. At a time and date that the merits of open access to code are discussed in non computer science journals [59], we should expect computer scientists to lead the way.

4.3 Performance Overhead

Detection mechanisms may impose a cost due to their use, as they typically introduce some amount of extra computation on existing applications. The overhead depends on the specifics of each mechanism. For example, it may be due to some form of run-time checking, or some form of obfuscation. Also, depending on the approach, the cost may be incurred on different places: it may affect a server (e.g., its memory or CPU usage, processing throughput, or response latency), it may affect the client, or both. The usefulness of a mechanism depends therefore on the computational cost it requires and on where it imposes it, as different overheads may be acceptable at the server-side than the client-side. What kind of numbers is reported matters as well. Reporting absolute measurements gives little information on the actual overhead, unless separate measurements are given for the system under study with and without the proposed mechanism. Percentage measurements are normally better.

Performance evaluation rests on strong foundations [60] and is a vibrant field as new technologies emerge [61]. Although it may not be necessary to conduct a comprehensive performance evaluation analysis for a defense mechanism, the more evaluation results are provided for a mechanism, the more valuable it becomes as a practical approach.

4.4 Ease of Use

The value of a detection mechanism as a practical tool depends on how easy it is to deploy it in a production setting. This aspect is orthogonal to the value of a detection mechanism as a research finding. Devising a technique to detect a hitherto undetectable class of attacks may be an excellent research contribution that merits publication; it may also be heavily cited and open the road to other, practical implementations in the future.

Ease of use depends on the deployment process required for the mechanism. The detection mechanism may be deployed at either or both the server and the client-side. Deployments on just the server or the client are easier to handle than deployments on both of them. The mechanism may be an add-on or plugin for existing software, client or server, or it may be tightly integrated with existing software, requiring rebuilding from source code.

No matter where it is installed, the means of installation influences ease of use. A detection mechanism that is available as a ready-to-install package will trump others that only exist in the form of source code.

4.5 Security

Defenders and attackers are often caught in a cat-and-mouse game, where countermeasures are bypassed by savvy attacks, which are caught by more sophisticated countermeasures, yet again bypassed by savvier attacks, and so on. We use security to refer to the ability of a detection mechanism to resist circumvention.

A mechanism that has not been bypassed is not eternally secure, as it is possible that a bypass method will be discovered in the future. We examine the various approaches based on the knowledge we have so far, that is, whether there are any known ways to bypass the detection mechanism today.

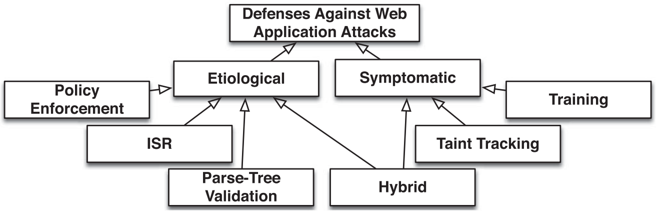

Fig. 3. The basic categories of countermeasures against web application attacks based on code injection. For each approach we provide the references that present corresponding mechanisms: *Policy enforcement*: [18], [20], [21], [23], [24], [25], [26], [27], [28], [30], [31], [36], [37], [43], ISR: [22], [38], [39], [51], *Parse-tree validation*: [1], [44], *Taint tracking*: [15], [29], [40], [45], [46], [47], [48], *Training*: [41], [42], [49], [50], [52], [53], [62], *Hybrid*: [14], [19], [30], [32], [33], [34], [35], [54], [55].

4.6 Point of Detection

Detection mechanisms vary on the location where they detect an attack. There are four different points where an attack can be caught, as seen in Fig. 2:

1) At the user's browser (Point UB).
2) En route from the server to the user's browser—in most cases within a proxy (Point STB).
3) At the database abstraction layer, before it reaches the server's database (Point DBAL).
4) After the malicious code reaches the server's database (Point DB).

5 DEFENSES

We categorize and analyze a large set of defenses developed to prevent the attacks described in Sections 2 and 3. We perform our analysis across the dimensions discussed in the previous section, except from availability, which we treat separately in Section 6. Due to the immense number of published works in the area, we only consider defenses that have been proposed in publications cited more than 20 times according to Google Scholar. We also include some recent works that have been presented in top security conferences, even though they have been cited less than 20 times, so that recent research is not penalized.

Fig. 3 presents a taxonomy of web application defenses against injection attacks. We can identify three broad categories: *etiological*, *symptomatic*, and *hybrid*. The etiological category involves mechanisms designed to block attacks based on their causes and origins. The symptomatic category incorporates a variety of schemes that inspect the behavior of applications and detect attacks based on their undesirable symptoms [8], [63]. Hybrid mechanisms borrow characteristics from both categories. Table 1 lists the specific mechanisms we consider in this work, grouped according to the subcategories shown in Fig. 3, and for each mechanism provides the following information:

1) Number of citations of the corresponding publication(s).
2) Accuracy and computational overhead measurements.
3) Types of attacks handled.

Recall that the point of operation for each mechanism is provided in the caption of Fig. 2.

5.1 Etiological

There are three main categories of etiological approaches used to protect web applications against injection attacks: *Parse-Tree Validation*, *Policy Enforcement*, and *Instruction Set Randomization*.

5.1.1 Parse-Tree Validation

The key idea behind parse-tree validation is to compare the tree representation of the abstract syntactic structure of the code that is about to be executed with the one that was originally intended. If the trees diverge, the application is probably under attack.

For DSL code injection attacks, mechanisms check the query before the inclusion of user input with the one resulting after the inclusion of user input. Two mechanisms that implement this approach for protection against SQL injection attacks, SQLGuard [44] and SQLcheck [1], are quite similar and detect the attack before a query reaches the database (Point DBAL). Contrary to SQLGuard, SQLcheck has been extensively tested in terms of accuracy, as can be seen in Table 1. A disadvantage of these mechanisms is that the application must be modified in every code fragment that sends an SQL query to the database for execution.

Parse-tree validation is an effective approach for the detection of DSL code injection attacks, especially as implemented in SQLcheck. This is not however the case for mechanisms that borrow elements from this approach and examine the syntax trees of scripts to detect JavaScript-driven XSS attacks, as we will see in Section 5.3.

5.1.2 Policy Enforcement

This approach is used to prevent XSS and CSRF attacks. When using a framework that implements policy enforcement, developers must define specific security policies on the server-side. Policies can be expressed through JavaScript extensions, pattern matching, or syntax-specific settings. The policies are then enforced either in the user's browser at runtime, or on a server-side proxy that intercepts server responses.

Noxes [18] is the only framework that partially allows users to specify policies for the prevention of XSS attacks. The key idea behind Noxes is to parse the HTML response that reaches the browser and find static URL references. Then, based on a set of policies, Noxes allows or blocks any generated requests (Point UB). Such policies can also be provided by the server (i.e., "never follow a link that leads to the malicious.com web site"). The main issue with Noxes is that URLS can be dynamically assembled by scripts, which may lead to false alarms.

Some frameworks define policies based on information and features provided by the DOM of a web page. Specifically, developers must place all legitimate scripts inside HTML elements like `div`. The web browser (Point UB) parses the DOM tree and executes scripts only when they are contained in such elements. All other scripts are treated according to the policies defined on the server. Frameworks that support this functionality include BEEP [20] and DSI [21]. The main problem with these mechanisms is that they do not examine the script's location inside the web document.

TABLE 1
Summary of Mechanisms Developed to Counter Application Attacks Based on Code Injection

Approach	Mechanism	# of Citations	Requirements[1]		Attack[4]
			TP,TN,FP,FN[2]	Performance Overhead[3]	
Parse-Tree Validation	SQLguard [44]	368	(NA,NA,NA,NA)_?	3% (S)	SQLi
	SQLcheck [1]	547	(36848,7648,0,0)_r	3 ms per query (S)	SQLi
Policy Enforcement	DSI [21]	188	(5268,NQ,NQ,85)_r	1.85% (C)	XSS
	NoForge [37]	166	(7,NQ,NQ,0)_r	NA (S)	CSRF
	Noxes [18]	69	(3,NA,NA,0)_r	NA (C)	XSS
	BEEP [20]	362	(61,NA,NA,0)_r	14.4% (C)	XSS
	BrowserShield [36]	273	(19,NQ,0,0,)_r	8% (S)	XSS
	CoreScript [24]	212	(NQ,NQ,NQ,NA)_s	NQ (C)	XSS
	SOMA [27]	65	(5,NA,NA,0)_s	5.58% (C)	XSS, CSRF
	Phung et al. [25]	124	(37,NA,NA,4)_r	5.37% (C)	XSS
	ConScript [23]	152	(NA,NA,NA,NA)_?	7% (C)	XSS
	CsFire [28]	52	(419582,1141807,0,3)_r[5]	NA (C)	CSRF
	CSP [31]	143	(NA,NA,NA,NA)_?	NQ (C)	XSS, CSRF
	jCSRF [43]	4	(2,NA,NA,0)_r	2 ms (S)	CSRF
	WebJail [26]	53	(2,NA,NA,1)_?	~6.89 ms (C)	XSS
ISR	SQLrand [51]	428	(3,NA,NA,0)_a	≤6.5 ms (S)	SQLi
	SMask [39]	31	(5,NQ,NQ,NQ)_r	NA (S)	SQLi, XSS
	Noncespaces [38]	146	(6,NA,NA,0)_r	2% (S)	XSS
	XJS [22]	29	(1380,NA,NA,1)_r	1.6–40 ms (C)	XSS
Taint Tracking	Haldar et al. [45]	234	(2,NA,NA,0)_s	NQ (S)	SQLi, XSS
	CSSE [47]	387	(7,NQ,NQ,NQ)_r	2–10% (S)	SQLi, XPathi, XSS
	Xu et al. [46]	368	(9,NQ,0,NQ)_r	average 76% (S)	SQLi, XSS
	WASC [40]	37	(NQ,NQ,NQ,NQ)_r	up to 30% (S)	SQLi, XSS
	Vogt et al. [29]	490	(NQ,NQ,NQ,NA)_r	NQ (C)	XSS
	PHP Aspis [48]	26	(12,NQ,NQ,2)_r	2.2× (S)	SQLi and PHPi, XSS
	Stock et al. [15]	23	(1169,NA,NA,0)_r	7–17% (C)	DOM-based XSS
Training	DIDAFIT [52]	208	(NA,NA,NA,NA)_?	NA (S)	SQLi
	AMNESIA [50]	551	(1470,NQ,0,0)_a	NQ (S)	SQLi
	libAnomaly [53]	310	(9,15987,60,0)_r	0.20–1 ms per query (S)	SQLi
	XSSDS [42]	96	(NQ,NQ,NQ,0)_r	NQ (S)	XSS
	SWAP [41]	83	(NQ,NQ,NQ,NQ)_r	up to 261 ms (S)	XSS
	SDriver [49], [62]	36	(241,NQ,0,0)_a	39% (S)	SQLi and XPathi
Hybrid	XSS-GUARD [30]	153	(8,NQ,NQ,NQ)_r	5–24% (C)	XSS
	Blueprint [19]	187	(94,NA,NA,0)_r	13.6% (C)	XSS
	Diglossia [14]	23	(25,NQ,NQ,NQ)_r	13% (S)	SQLi and JSONi
	JSFlow [32]	72	(NQ,NQ,NQ,NQ)_r	2× (C)	XSS
	COWL [33]	41	(NQ,NQ,NQ,NQ)_r	16% (C)	XSS
	Bauer et al. [34]	10	(NA,NA,NA,NA)_?	average 55% (C)	XSS
	SIF [54]	155	(NA,NA,NA,NA)_?	26% (S)	SQLi, XSS
	Hails [35]	92	(NQ,NQ,NQ,NQ)_r	28% (S, C)	SQLi, XSS
	Aeolus [55]	58	(NA,NA,NA,NA)_?	323.5 ms per request (S)	SQLi, XSS

[1]*NA (Not Available) indicates that a requirement is not mentioned in the paper. NQ (Not Quantified) indicates that a requirement is mentioned in the publication but is not quantified.*
[2]*Tuples contain numbers given for True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). For every tuple there is a corresponding suffix that indicates whether the testbed was based on: real-world applications known to be vulnerable* (r)*, synthetic benchmarks* (s)*, or both* (a)*. A question mark* (?) *denotes that no test results are reported.*
[3]*Whether the overhead is incurred on the server (S) or the client (C).*
[4]i *stands for injection.*
[5]*The numbers in this particular case involve requests.*
The different results does not necessarily indicate that one mechanism is more effective than the other. This is because most of them were evaluated under different assumptions and settings.

Attackers can take advantage of this fact to perform mimicry attacks [64]. Specifically, they can execute legitimate scripts, but not as intended by the original design of the developers. This is extensively described by Athanasopoulos et al. [22], who also describe another recent variation of JavaScript injection attacks, known as return-to-JavaScript attacks, which can be used to bypass the above mechanisms.

A policy enforcement technique developed by Mozilla, called CSP (Content Security Policy)[4] [31] is currently supported by many browsers to prevent XSS and CSRF attacks. To eliminate such attacks, web site administrators can specify which domains the browser should treat as valid sources of

4. https://developer.mozilla.org/en-US/docs/Web/Security/CSP

script and which not. These policies are communicated via the HTTP headers. Then, the browser (Point UB) will only execute scripts that exist in source files from white-listed domains. Note that, if an application involves embedded scripts, developers must utilize the CSP's *nonce* concept. This is an unpredictable, random value indicated in the `script-src` directive, which in turn is applied as a nonce attribute to `<script>` elements. As a result, only those elements that have the correct nonce will execute. Even if an attacker is able to inject markup into the page, the attack will be prevented by the attacker's inability to guess the nonce value. However, attackers may still bypass this feature and invoke a script from a non-whitelisted source. To do so, their injected code must be crafted in a way that the nonce is handled by the browser as an attribute of the payload [65].

Another policy enforcement approach introduces policies directly either in HTML or JavaScript code to confine their behavior. BrowserShield [36] acts as a proxy on the server-side (Point StB) to parse the HTML of server responses and identify scripts. Then, it rewrites them into safe equivalents and protects the web user from exploits that are based on reported browser vulnerabilities. ConScript [23], CoreScript [24], and the framework by Phung et al. [25] extend JavaScript with new primitive functions that provide safe methods to protect potentially vulnerable JavaScript functions. In both cases, policy enforcement takes place at client-side, in the JavaScript engine of the browser (Point UB). In this way, XSS attacks that take advantage of functions such as `write` and `eval`, which are used to assemble innocuous-looking parts into harmful strings,[5] would fail.

An issue regarding the above frameworks involves features like script inclusion and `iframe` tags. Even though they allow developers to decide if they will disable them or not, this is impractical because such features are quite popular and widely used. If developers choose to use them, these frameworks cannot define policies that restrict the behavior of third-party scripts introduced by such features. Thus, they would be vulnerable to attacks that use `iframes` in the way described in Section 2. WebJail [26], and SOMA [27] are two frameworks that can actually detect such attacks (Point UB). To achieve this, SOMA requires site administrators to specify legitimate, external domains for sending or receiving information in order to approve interactions between them and the protected web site. As a result, SOMA can also detect CSRF attacks. WebJail contains the functionality of third-party scripts by introducing a web component integrator that restricts the access that these scripts may have to either the data or the functionality of other components.

Policy enforcement mechanisms that detect CSRF attacks are usually implemented in the form of a server-side proxy (Point StB) interposed at the client-server communication path. NoForge [37] parses the HTML server responses and adds a token to every URL referring to that particular server. Then, it associates the token with the cookie representing the session ID for the application. When a request is received, the mechanism checks if the request contains the token related to the session ID. A disadvantage of NoForge is that dynamically created HTML within the browser will not include the token. Thus, sites that create part of their HTML code at client-side will remain vulnerable. In addition, it does not support cross-origin requests. The above problems are addressed by jCSRF [43], which shares similar functionality. Finally, CsFire [28] examines cross-domain interactions to design a cross-domain policy at the client-side (Point UB). The policy is based on the concept of a relaxed same-origin policy that allows communication between sub-domains of the same registered domain. Most of the above frameworks involve several deployment hurdles, as they require significant source code modifications by the developers on the server-side to introduce and enforce the applied policies.

5. The infamous Sammy worm that infected MySpace in 2005 utilized the `eval` function to assemble a malicious script.

5.1.3 Instruction Set Randomization (ISR)

ISR is a method that has been applied to counter different kinds of application attacks [66], and was originally applied for the prevention of binary code injection attacks [67]. The main idea behind ISR is to change the representation of code based on a randomly chosen transformation, and randomize the execution environment accordingly. In this way, any malicious code injected as part of untrusted input data, by attackers who do not know the randomization algorithm, will not be executed.

SQLrand [51] applies the concept of ISR for the prevention of SQL injection attacks. It allows programmers to create SQL statements using randomized instructions instead of standard keywords. The modified queries are reconstructed at runtime using the same key used for randomization, which is inaccessible to a malicious user. SQLrand is one of the few mechanisms that prevent SQL injection attacks at the database level (Point DB).

The same concept can be applied for protection against XSS attacks that inject JavaScript or HTML code. Initially, the trusted code of a web page can be transformed to a random representation using a simple function such as XOR. Before being sent to the client (Point StB), or being processed by the browser (Point UB), the legitimate code is transformed back to its original form, while any additional injected code will be transformed into junk code. Variations of this approach include Noncespaces [38] and XJS [22], which randomize the instruction set of HTML and JavaScript, respectively. Contrary to XJS, in Noncespaces administrators must set specific policies in a manner similar to a firewall configuration language. SMask [39] is another framework that was inspired by ISR. To detect XSS attacks, it searches for HTML and JavaScript keywords within the application's legitimate code. This is done before the processing of any HTTP request. When a keyword is found, it adds a token to it, resulting in a "code mask." Then, before sending the resulting HTML data to the user, the framework searches the data for illegal code using the same keywords (Point StB). Since all legitimate code has been "masked," the injected code can be identified. The need for pre-processing and post-processing the code, however, may add a significant overhead to the application. Unfortunately, the authors of SMask did not provide measurements regarding the runtime overhead of the tool (see Table 1).

ISR is a deterministic approach that can be applied to prevent different attacks in an effective manner. However, Sovarel et al. [68] have investigated thoroughly the effectiveness of ISR and showed that a malicious user may be able to circumvent it by determining the randomization key. Their results indicate that applying ISR in a way that provides a certain degree of security against a motivated attacker is more difficult than previously thought. Furthermore, developers who wish to use such mechanisms must follow good coding practices and make sure that randomized code statements are never leaked (e.g., as part of an exception error), as this may be used to reveal the encoding key.

Even though the above implementations impose a low computational overhead, they require significant deployment effort. In particular, SQLrand [51] requires the integration of a proxy within the database server, while Noncespaces and XJS [22] require modifications on both the server and the client.

5.2 Symptomatic

Symptomatic techniques follow two main approaches. They either track untrusted input and ban certain operations on it, or they first learn what code to trust and then approve for execution code that they recognize as safe.

5.2.1 Taint Tracking

A taint tracking scheme marks untrusted ("tainted") data, such as a variable set by a field in a web form, and traces its propagation throughout the program. If the variable is used in an expression that sets another variable, that variable is also marked as untrusted and so on. If any of these variables is used in a potentially risky operation (e.g., sending the data to a vulnerable "sink," such as a database, a file, or the network), the scheme may act accordingly.

Taint tracking is provided as a feature in some programming languages, such as Perl and Ruby. By enabling this feature, Perl would refuse to run code vulnerable to an SQL injection attack (consider a tainted variable being used in a query) and would exit with an error message.

There are different implementations of this approach in terms of how the tainted data is marked and tracked, and how attacks are detected. For example, Haldar et al. [45] have implemented their scheme for the Java Virtual Machine (JVM), where they instrument various classes. When a tainted string is used as an argument to a sink method an exception is raised (Point DBAL).

It is possible to apply further checks when it is established that tainted data have reached a sink (Point DBAL). Xu et al. [46] track taint information at the level of bytes in memory. To distinguish between legitimate and malicious uses of untrusted data that reach a sink, they search the data for suspicious symbols using regular expressions. CSSE [47] associates tainted data with specific metadata. Such metadata include the origins of tainted data, its propagation within the application, and others. When tainted data reaches a sink, CSSE performs syntactic checks based on its metadata (Point DBAL). PHP Aspis [48] works in a similar way. To obtain metadata, it takes advantage of the PHP array data structure. Finally, WASC [40] analyzes HTML responses to check if there is any tainted data that contains scripts (Point StB).

A recent study [69] showed that there are ways to circumvent the majority of the above schemes. Furthermore, most of them are not easy to deploy since the majority of input vectors, string operations, and output vectors of the application must be instrumented.

Vogt et al. [29] have developed a tainting scheme that follows a different approach. In contrast to the above schemes, which operate on the server-side, their technique tracks sensitive information at the client-side (Point UB). This is a form of positive data flow tracking, where tagged data is considered to be legitimate. Their scheme detects JavaScript-driven XSS attacks by ensuring that a script can send sensitive user data only to the site from which it came from. Stock et al. [15] propose a scheme that also operates in the browser (Point UB). The scheme focuses on the detection of DOM-based XSS attacks. This scheme is different from the previous one because it marks and observes data that are considered harmful. Specifically, it employs a taint-enhanced JavaScript engine that tracks the flow of attacker-controlled data. To detect potential attacks, the scheme uses HTML and JavaScript parsers that can identify the generation of code coming from tainted data.

An issue that involves all taint tracking schemes involves the difficulty of maintaining accurate taints [70] (e.g., implicit flows [71]). In such cases, certain, tainted inputs can escape the tracking mechanism. Keeping track of such input may be impractical not only because of the various technical difficulties, but also because it would raise false alarms.

5.2.2 Training

Training techniques are based on the ideas of Denning's original intrusion detection framework [72]. In particular, a training mechanism learns all valid legitimate code statements during a training phase (mostly in the form of signatures). This can be done in various ways depending on the implementation. Then, only those statements will be recognized and approved for execution during production.

Training methods that detect DSL-driven injection attacks generate and store valid code statements (e.g., SQL or XPath queries) in various forms, and detect attacks as outliers from the set of valid code statements. An early approach, DIDAFIT [52], detects SQL injection attacks (Point DBAL) by recording all database transactions stripped from user input. Subsequent refinements by Valeur et al. [53] tag each transaction with the corresponding application as an extension of their anomaly detection framework called libAnomaly. SDriver [49], [62] is a signature-based mechanism that prevents SQL and XPath injection attacks. The signatures generated during a training phase are based on features that can depend either on code statements or on their execution environment (e.g., the stack trace). Then, at runtime, the mechanism checks all statements for compliance and can block code statements containing injected elements (Point DBAL). AMNESIA [50] is a tool that also detects SQL injection attacks (Point DBAL) by associating a query model with the location of every SQL statement within the application. Then, at runtime, it monitors the application's execution to detect when SQL statements diverge from the expected model.

Various countermeasures against XSS attacks follow a similar pattern. SWAP [41] creates a unique identifier (script ID) for every legitimate script on the server. Then, a JavaScript detection component placed in a web proxy (Point StB) searches for injected scripts with no corresponding ID in the server's responses. If no injected scripts are found, the proxy forwards the response to the client. This mechanism is relatively inflexible since it does not support dynamic scripts. In addition, it imposes a significant overhead (see Table 1). The authors of XSSDS [42] have implemented a similar mechanism that also supports dynamic and external scripts. Specifically, during the training phase, they build a list of all benign scripts. For external scripts, they keep a whitelist of all the valid domain names that contain scripts used by the application.

Defenses based on training include some mechanisms that can be easily circumvented. For example, DIDAFIT [52] and libAnomaly [53] do not tag transactions with their corresponding call sites. This can lead easily to false negatives. For instance, recall the application mentioned in Section 2, which will show the password for a forgetful user by executing the following query:

```
SELECT password FROM userdata WHERE id = 'Alice'
```

This same application could allow users to login to the site using just their password via a custom login form (like the "Password only login" plugin of Wordpress[6]) but allow the login either with the user's password or with the administrator password. The corresponding query to verify the password on the login form would be as follows:

```
SELECT password FROM userdata WHERE id = 'Alice'
OR id = 'admin'
```

Even if the application's administrators have chosen to use either DIDAFIT [52] or libAnomaly [53] as protection, an attacker could bypass them and obtain the administrator's password via email, by entering on the form the standard string: `nosuchuser' OR id = 'admin`. The infected query matches the signature of the second one above and is therefore accepted. The problem lies with the call location of the query, not the query alone.

Mechanisms based on signatures that involve elements not only associated with code statements (e.g., AMNESIA [50] and SDriver [49]) could detect such attacks. Specifically, SDriver associates a complete stack trace with the root of an SQL statement, thus it can correlate queries with their call sites and detect attacks like the above. Furthermore, training tools that detect XSS attacks based on JavaScript injection would fail to detect mimicry attacks where legitimate scripts can be executed by attackers, but not in the way intended by the developers [22].

In general, the detection accuracy of training approaches is heavily influenced by the coverage that is achieved during the training phase. If the coverage is insufficient, false alarms are very likely. In addition, when a code statement is altered, a new training phase is necessary.

Most of the training approaches are relatively easy to deploy. DSL injection attack countermeasures can be retrofitted to a system typically by changing some configuration files (i.e., SDriver). This does not apply to AMNESIA though, since significant source code modifications are required for every query that exists in the application. Finally, SWAP and XSSDS are implemented within a proxy on the server-side.

6. wordpress.org/plugins/password-only-login/

5.3 Hybrid

This category includes mechanisms that borrow characteristics from both etiological and symptomatic approaches. Five of them focus on the detection of XSS attacks and one focuses on DSL code injection attacks.

XSS-GUARD [30] is a training scheme that employs parse-tree validation. During training, the scheme maps legitimate scripts to HTTP responses. During production XSS-GUARD retrieves for every script included in a response its parsed tree and checks if it is one of those previously mapped to this response. Apart from the comparison of the parsed trees, XSS-GUARD checks also for an exact match of lexical entities. To achieve this, the scheme utilizes that data structures of Firefox's JavaScript engine (Point UB). However, string literals are not compared literally, which can lead to false negatives. For instance, consider a banner rotator that every time it runs it creates a value that depends on the current date and the length of the array that contains the references of the various images to be displayed. Then, based on this value, it shows a specific image to a user. In a vulnerable web site that allows users to post data and contains this banner rotator, a malicious user could create and store a script that has the same code structure, with the same JavaScript keywords contained in the rotator script. In this script, attackers could also include references to tiny images hosted on a web server that is maintained by them in order to retrieve the IP addresses of the users that visit the vulnerable site.

Blueprint [19] is a policy enforcement framework that uses parsed trees to detect XSS attacks. To guarantee that untrusted content is not executed, Blueprint generates at the server-side a parsed tree from untrusted HTML to ensure that it does not contain any dynamic content. Then, the parsed tree is transfered to the document generator of the browser (Point UB), where untrusted browser parsing behavior is ruled out. Blueprint is an efficient countermeasure but imposes non-negligible overhead due to its extensive parsing (see Table 1).

Diglossia [14] combines positive taint tracking together with parse-tree validation. Diglossia was based on the theory of Ray and Ligatti [2] (which we saw in Section 2) to detect DSL code injection attacks. In addition, it is actually the first, and so far the only framework that detects JSON injection attacks. When an application computes an output string (query), Diglossia computes a "shadow" of that string. Specifically, it maps all characters introduced by the application to a shadow character set. This set does not contain any characters coming from the tainted input. Then, the scheme creates the tree of the query that is about to be executed and compares it with the parsed tree of the "shadow" (Point DBAL). If the trees do not match, the application is probably under attack. Note that Diglossia can be bypassed in the same way as other taint tracking approaches [69].

Information Flow Control (IFC) mechanisms combine positive taint tracking and policy enforcement to prevent XSS attacks on the client-side (Point UB). Representative implementations such as JSFlow [32], COWL [33] and the framework by Bauer et al. [34] allow developers to express information flow policies by extending the type system of JavaScript. Then, the policies are enforced by the JavaScript interpreter through dynamic checks. IFC frameworks do not focus explicitly on JavaScript and they have a broader scope. They can be used to build secure applications, thus preventing different attacks. Policies can be provided either as compile-time program annotations, or as run-time requirements defined by the user. Such frameworks include SIF (Servlet Information Flow) [54], Hails [35], and Aeolus [55].

6 Observations

We group our key observations on the 41 publications that we consider in this work along the dimensions we identified in Section 4.

6.1 Accuracy

Table 1 indicates that the authors of 3 out of 41 (7.3 percent) publications provided a complete tuple of TP, TN, FP, and FN. On the other hand, we see that 7 out of 41 (17 percent) were not tested at all. In these cases, all four elements of the tuple are not available (NA). An interesting observation is that in 10 out of the 41 (24.3 percent) publications, the evaluation focused on only on attack detection, and there was no test focusing on false positives. The corresponding tuples contain TP and FN results, but TN and FP results are not available (NA). This stresses the fact that authors may be more interested in making sure that their mechanisms can detect known attacks rather than seeing how they respond under normal conditions. In some cases, where the number of TP results are not quantified, authors did not mention how many or which attacks they performed. Moreover, in some cases we observe that even if the authors report the existence of possible false positive and negative results, they have not quantified them (NQ).

Even when some numbers are given, they may not be adequate. Sensitivity and specificity are statistical measures, and as such, they should be interpreted with suitable confidence intervals. There are various methods to calculate confidence intervals, even without the need to have large sets of samples in order to be able to use the central limit theorem [73]. However, sample sizes in the single digits are not enough to produce good intervals.

Regarding the subjects of tests, we see that 30 out of 41 (73.1 percent) mechanisms were tested with real-world applications known to be vulnerable. The vulnerabilities associated with these applications are enlisted in the following providers: the Common Vulnerabilities and Exposures (CVE) database,[7] the Bugtraq[8] security mailing list, the xssed.com security bulletin provider,[9] Microsoft's security bulletin[10] and the ha.ckers.org[11] security bulletin provider. Fig. 4 presents how many, and which publications referred to which source. In numerous occasions authors performed tests based on the same applications. For example, 12 mechanisms were tested on a vulnerable version of the PHPBB bulletin board software.[12]

There are also authors that took a different approach. For instance, during their initial tests, Stock et al. [15] managed to bypass the browser-based XSS filters of 73 percent out of 1,602

7. https://cve.mitre.org/
8. http://seclists.org/bugtraq/
9. www.xssed.com
10. https://technet.microsoft.com/en-us/security/bulletin
11. http://ha.ckers.org/
12. https://www.phpbb.com/

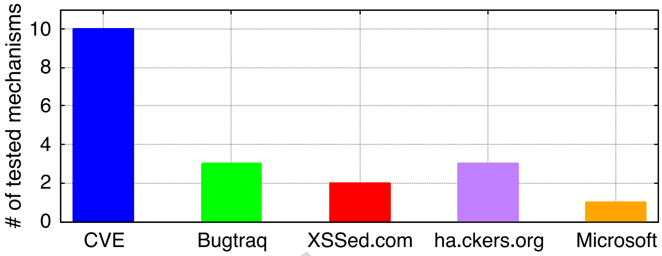

Fig. 4. Most common sources of real-world vulnerabilities used for testing of web attack defenses. Specifically, the publications that used a particular source are as follows: CVE [14], [30], [37], [39], [40], [41], [43], [46], [48], [51], Bugtraq [18], [42], [47], XSSED.COM [21], [22], ha.ckers.org [19], [20], [25], Microsoft [36].

real-world DOM-based XSS vulnerabilities. These vulnerabilities were actually found as part of their previous research [74]. Finally, in the AMNESIA [50] and SDriver [49] publications, the authors managed to break existing application suites and then test the accuracy of their tools on them.

6.2 Availability

Table 2 presents our findings regarding the availability of each mechanism in terms of source code and corresponding executables. We also examined the availability of the testbeds mentioned in each paper. We see that only 8 out of 41 (19.5 percent) of the publications provided a link to their mechanism and, 2 from these 6 web pages are currently not available. In 7 cases the authors made either the source code or their executables available after their paper got published. In one case (CSP [31]), the source was available before the publication through the Mozilla community. Regarding the availability of test materials, we see that only 4 out of 41 (9.7 percent) publications have currently their testbeds available.

6.3 Performance Overhead

Table 1 shows the performance overhead of each mechanism. In addition, it indicates if the overhead is incurred at the server (S) or at the client-side (C). In almost half of the approaches, 20 out of 41 (48.7 percent), the overhead is provided as a percentage, while in other cases, 8 our of 41 (19.5 percent), the authors provide the latency (in ms) that their mechanisms add to the normal execution time of the protected application. Note that in some cases, the times of normal executions are not provided so it is not possible to convert absolute time measurements to percentage overheads. In 10 cases (24.3 percent), the overhead was either not available (NA) or not quantified (NQ). For PHP Aspis [48] and JSFlow [32] the authors indicate that with their mechanisms the execution time is doubled.

6.4 Ease of Use

Mechanisms in different categories face different deployment issues. Consider the majority of mechanisms in the policy enforcement subcategory. In most cases, developers should modify multiple components to use each mechanism. Specifically, mechanisms like BrowserShield [36] and BEEP [20] require modifications both on the server and at the client. Thus, it would be difficult to be adopted by both browser vendors and application developers. On the other hand, there are cases where the policies introduced at the server are enforced at the client-side, via a library embedded

TABLE 2
Availability of the Studied Defenses

Approach	Mechanism	Availability[1]		
		Source Code	Executable	Testbed
Parse-Tree Validation	SQLGuard [44]	**AO**	**AO**	NA
	SQLcheck [1]	NA	NA	NA
Policy Enforcement	DSI [21]	NA	NA	NA
	NoForge [37]	**AO**	**AO**	NA
	Noxes [18]	NA	NA	NA
	BEEP [20]	✓	✓	✓
	BrowserShield [36]	NA	NA	NA
	CoreScript [24]	**AO**	NA	NA
	SOMA [27]	NA	NA	NA
	Phung et al. [25]	✓	✓	✓
	ConScript [23]	**AO**	NA	NA
	CsFire [28]	✓	NA	NA
	CSP [31]	**AO**	**AO**	NA
	jCSRF [43]	NA	NA	NA
	WebJail [26]	NA	NA	NA
ISR	SQLrand [51]	NA	NA	NA
	SMask [39]	NA	NA	NA
	Noncespaces [38]	NA	NA	NA
	xJS [22]	NA	NA	NA
Taint Tracking	Haldar et al. [45]	NA	NA	NA
	CSSE [47]	NA	NA	NA
	Xu et al. [46]	NA	NA	NA
	WASC [40]	NA	NA	NA
	Vogt et al. [29]	NA	NA	NA
	PHP Aspis [48]	**AO**	NA	**AO**
	Stock et al. [15]	NA	NA	NA
Training	DIDAFIT [52]	NA	NA	NA
	AMNESIA [50]	NA	**AO**	NA
	libAnomaly [53]	**?**	**?**	**?**
	XSSDS [42]	NA	NA	NA
	SWAP [41]	NA	NA	NA
	SDriver [49], [62]	✓	✓	NA
Hybrid	XSS-GUARD [30]	NA	NA	NA
	Blueprint [19]	**?**	**?**	**?**
	Diglossia [14]	NA	NA	NA
	JSFlow [32]	✓	✓	NA
	COWL [33]	NA	NA	NA
	Bauer et al. [34]	NA	NA	NA
	SIF [54]	**AO**	NA	NA
	Hails [35]	**AO**	NA	**AO**
	Aeolus [55]	✓	✓	NA

[1] *A check mark (✓) indicates that the publication includes a link to a page where the software is available.* ***AO*** *(Available On-line) indicates that the software is available on-line but the address is not mentioned in the paper, which probably means that the it was made available after the publication. A question mark (**?**) indicates that a link to the software was included in the publication but is now inaccessible.*

in the server's response (i.e., Blueprint [19]). Such an approach is convenient as no modifications are needed at the client.

Extensive modifications in the application's source code can also be a reason that can make a mechanism difficult to use. SQLrand [51], AMNESIA [50], mechanisms of the parse-tree validation subcategory, and mechanisms of the taint tracking subcategory are such examples. In the first three cases, programmers should modify every code fragment that involves the execution of a query. In the latter case they should also change all the code fragments that involve user input handling.

By design, IFC frameworks provide limited support for securing legacy applications and they are not always easy to adopt because developers need to learn new constructs to use them. However, there are attempts to overcome such issues. For instance, the authors of Aeolus [55] provide a simple security model that tries to match the way programmers understand authorization and access control with the tracking of information flow.

Finally, even though many of the mechanisms that involve training are easy to deploy, they have a distinct disadvantage. When the application is altered, mechanisms like SDriver [49] and XSS-GUARD [30] require a new training phase. However, with the increased adoption of automated testing and continuous integration frameworks, this phase could be easily repeated.

6.5 Security

In our discussion of the various mechanisms in Section 5, we observed that some of them can be bypassed by attackers that know the internals of their operation. Still, the design of some mechanisms allows them to be extended and become immune to the circumvention attacks they are currently vulnerable to. For example, policy enforcement frameworks based on JavaScript or HTML rewriting can be extended to detect attacks that leverage `iframe` tags. This does not apply to all mechanisms though. In particular, training mechanisms like DIDAFIT [52] and XSS-GUARD [30] must be redesigned to detect the atacks we described in the related sections.

Taking the attackers' point of view, mimicry attacks can affect a range of different mechanisms: they can be used to bypass mechanisms in the hybrid category, as well as the policy enforcement and training subcategories.

6.6 Point of Detection

Given the fact that the final steps of DSL injection and CSRF attacks take place at the server-side, all mechanisms that detect such attacks are placed at some point within the server-side infrastructure. From the mechanisms that detect DSL code injection attacks, 3 out of 16 (18.7 percent) do so at the level of the RDBMS (Point DB). The other 13 are based on interposing on API calls related to output vectors (Point DBAL).

For frameworks that deal with XSS attacks, we see that attack prevention may take place either at the server or the client-side. In the first case, a proxy is typically placed in front of the server to examine the server's responses before they reach a user's browser. In the second case, a modified browser or a library that is securely downloaded from the server checks the responses for potential attacks. We find that the tendency so far is to create frameworks that perform detection on the client-side: in particular, 18 out of 30 frameworks (60 percent) detect XSS attacks at the client-side. Note that the majority of the mechanisms that detect such attacks on the server can also detect DSL code injection (9 out of 13).

7 Recommendations and Lessons Learned

Our observations lead to some lessons and recommendations that developers of new mechanisms may find helpful. In particular, our observations call for improvements in the accuracy of experimental testing and code availability, while aiming to reduce performance overheads and deployment hurdles.

7.1 Improving Testing Accuracy

One of our key findings indicates that many proposed defenses are tested in a poor manner. In many cases, researchers tend to not provide results on false positives or false negatives for their mechanisms (where applicable). Mere discussion on the existence of such results without quantifying them also blurs the picture.

A reasonable argument would be that many defenses (e.g., many mechanisms coming from the etiological category, or the IFC frameworks), do not need to be validated purely through testing, since they provide systematic arguments as to why their design is secure against attacks. In order for this to hold, however, their implementation should be flawless and precisely follow its specification, which may not be the case in practice. Moreover, even mechanisms that detect attacks based on their root cause, instead of their observed behavior, may still be circumvented by evasive attacks.

When we introduced specificity and sensitivity in Section 4.1, we deferred discussion of PPV and NPV to this point. These two relate to the effectiveness of a detection mechanism in an actual production setting, instead of a testbed. If an attack is detected in a production environment, how much should we be worried? The answer is provided by PPV. If no attack is detected in a production environment, how relaxed should we be that no attack has indeed taken place? The answer is provided by NPV. We can calculate PPV and NPV with the following equations:

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{3}$$

$$\mathrm{NPV} = \frac{\mathrm{TN}}{\mathrm{FN} + \mathrm{TN}}. \tag{4}$$

Equations (5) and (6) use true and false positives and negatives, like Equations (1) and (2). However, TP, TN, FP, and FN are not qualitatively the same in these two cases: whereas sensitivity and specificity are measured on a testing environment, PPV and NPV are measured on a real, production environment. In fact, if PR is the probability of a particular class of attacks in the real world, i.e., its prevalence, then we have [57]:

$$\mathrm{PPV} = \frac{\mathrm{SE} \times \mathrm{PR}}{\mathrm{SE} \times \mathrm{PR} + (1 - \mathrm{SP}) \times (1 - \mathrm{PR})} \tag{5}$$

$$\mathrm{NPV} = \frac{\mathrm{SP} \times (1 - \mathrm{PR})}{(1 - \mathrm{SE}) \times \mathrm{PR} + \mathrm{SP} \times (1 - \mathrm{PR})}. \tag{6}$$

Prevalence is the prior probability that an event might be an attack, based on our understanding of the volume and frequency of a particular class of attacks; PPV and NPV are the revised estimates of that probability based on the results of the detection mechanism. The lower the prevalence of an attack, the more confident we can be that a negative test result indicates that no attack has taken place and the less sure we can be that a positive test result indicates a real attack.

Of course it is not easy, and it may not even be possible, to know how prevalent an attack class is. Also, attacks against a system may depend on factors such as its visibility and popularity, and thus the same software may be subject to varying attack intensity depending on where it is actually deployed. With this in mind, it may be unfair to ask researchers to provide PPV and NPV values for their mechanisms.

This does not mean though that deriving PPV and NPV is altogether impossible. One could deploy a system armoured with an attack detection mechanism on a honeypot to study what happens over a time period. This could give an indication about the performance of the mechanism in a realistic setting. Alternatively, one could deploy a target, unarmoured system, on a honeypot to study the prevalence of the class of attacks to be detected. Studying the prevalence of classes of attacks is an interesting area of study on its own, and could feed directly on the evaluation of attack detection mechanisms as practical tools.

Apart from demonstrating the value of a mechanism, good accuracy tests may be beneficial per se, leading to more well-designed and robust defenses. Mechanisms that can be circumvented were not extensively tested in terms of accuracy. For instance, DIDAFIT [52] was not tested at all and XSS-GUARD's [30] testing involved six known attacks. This also applies to some frameworks coming from the taint tracking subcategory. It is possible that more tests during development would have led the authors to larger design improvements [75].

The accuracy of a tool may be related to the scope of the attacks it aims to detect. A more limited scope may allow the development of more accurate tools. In this vein, the system by Stock et al. [15] is the only one that targets exclusively DOM-based XSS attacks and detects them in an accurate manner, as seen in Table 1. It is also extensively tested with real-world attacks.

7.2 Code Availability

Apart from testing practices, another area that merits improvement is the availability of prototypes and testbeds. We are not aware of specific reasons why authors of detection mechanisms seem to be wary of publicly releasing their code and tests. Our finding may reflect the status in the current point in time, when authors are urged to publish their code and tests, and may start, or may have already started doing so, but this does not show yet in the research we examined, which goes several years back. We also saw instances where material was published, but does not seem to be available any more. That points to the importance of reproducibility of research materials, an issue arising in all scientific fields. It is not enough to publish the underlying code and data, but to make sure that it remains accessible, and to provide the means to test it even as technology advances and operating systems, file formats, and software libraries change. That may be too much to ask right now; what seems reasonable, though, is to ask of researchers working on defenses not to buck the trend and to take steps to increase the availability of their research.

7.3 Performance Overhead, Deployment and Security Remarks

Performance overhead comes up as a non-trivial issue. We observe that there are some mechanisms that introduce an overhead that is more than 10 percent. Relative research on protection mechanisms [9] indicates that mechanisms that introduce an overhead larger than 10 percent do not tend to gain wide adoption in production environments. Hence, the computational overhead of some mechanisms could be a reason why they have not been adopted.

The deployment difficulties we have found with many mechanisms are not a reason not to adopt them, but they may hinder their widespread use. Since code injection attacks are complex, it may be logical to expect that mechanisms to detect them would be complex too. The effort to install and use a tool should be weighed against the expected benefits. That is one more reason why it is important to report tool accuracy, as this provides an immediate indicator to the expected benefits.

The issue is more acute when the point of detection is at the client. We saw that many policy enforcement mechanisms that detect attacks at the browser are not easy to deploy because modifications are needed both on the server and the client-side. Conversely, it may be possible to deliver the tools to the user unobtrusively; for example, we saw that Blueprint [19] enforces policies at the client-side by embedding a library in the server's response to the client. In general, when developing a new countermeasure, researchers should consider where it will detect attacks, observe the corresponding deployment challenges, and try to mitigate them.

From a security perspective, we saw that there are cases where mechanisms coming from the same category can be bypassed by similar attack patterns. This denotes that there are issues found in the design of each approach. Recall, for instance, that implicit flows can be used against many taint tracking solutions, and mimicry attacks can be launched to bypass training and policy enforcement defenses. IFC mechanisms though, can deal with such attacks. This indicates that when solutions borrow elements from other categories (IFC mechanisms belong to the hybrid category) could be more effective. Also, we could say that different mechanisms could be used together to defend different attacks. For instance, developers could employ both CSP [31] and SQLrand [51] to deal with XSS and SQL injection respectively.

8 Conclusion

Despite many approaches that have been developed, attacks based on code injection against web applications have been consistently present for the last 15 years, and it appears that they will continue to be. Attackers seem to find new ways to introduce malicious code to applications using a variety of languages and techniques [3], [4]. Meanwhile, during the last decade, there have been numerous mechanisms designed to detect one or more of types of such attacks. Although some deployed and widely used frameworks, such as CSP [31], share characteristics (for instance, HTML sanitization and `eval` handling) with previous proposals, most research works are still not used in practice.

In order for a security tool to be used in practice, it must provide some value to the user. In particular, the value should outweigh the cost of its use. The cost is not necessarily monetary, but may be incurred from the time required to use the tool, any inconvenience caused, false alarms that may raise, and so on. These costs are related to the issues we have been investigating here: poor testing, high overhead, lack of publicly available prototypes, deployment difficulties, compromised security.

Improving any of these aspects would not just increase the value of a research work as a practical tool, but it would also increase its research value as well. Accurate detection reporting would help in evaluating different approaches. Extensive performance measurements can reveal impractical designs and focus effort elsewhere. Availability of source code enhances basic scientific tasks like verification and reproducibility. Ease of deployment brings ease of experimentation. Secure methods can form the basis for developing methods with more extensive coverage.

On the positive side, some defenses have been extensively tested in terms of accuracy (SQLCheck [1], AMNESIA [50], libAnomaly [53]), solve specific problems in an effective way (Blueprint [19] and the system by Stock et al. [15]), or have a low computational overhead (DSI [21] and the system by Phung et al. [25]). There are also cases where researchers have made their code available (BEEP [20] and SDriver [49]), which can promote the development of better mechanisms. This argument has been raised by others researchers too [9]. Furthermore, testing defenses in a production setting could also propel their adoption.

We hope that the exploitation model, analysis, and observations that emerged from our research can be a reference point for researchers who aim to develop new, practical countermeasures against web application attacks.

Acknowledgments

We would like to thank Roxana Geambasu and the anonymous reviewers for their suggestions and comments. This work was supported by US National Science Foundation Grant No. CNS-13-18415.

References

[1] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Proc. 33rd ACM Symp. Principles Program. Languages*, 2006, pp. 372–382.
[2] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proc. 39th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2012, pp. 179–190.
[3] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: Stealing the pie without touching the sill," in *Proc. 19th Conf. Comput. Commun. Secur.*, 2012, pp. 760–771.
[4] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in PHP: Automated POP chain generation," in *Proc. 21st ACM Conf. Comput. Commun. Secur.*, 2014, pp. 42–53.
[5] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proc. Int. Symp. Secure Softw. Eng.*, Mar. 2006, pp. 13–15.
[6] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 771–781.
[7] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surveys*, vol. 44, no. 3, pp. 11:1–11:46, Jun. 2012.
[8] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 3, pp. 186–205, Aug. 2000.
[9] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 48–62.
[10] "Code share," *Nature*, vol. 514, pp. 536–537, 2014.
[11] S. Bratus, M. E. Locasto, L. S. M. L. Patterson, and A. Shubina, "Exploit programming: From buffer overflows to 'Weird Machines' and theory of computation," *Login*, vol. 36, no. 6, pp. 13–21, Dec. 2011.

[12] K.-S. Lhee and S. J. Chapin, “Buffer overflow and format string overflow vulnerabilities,” *Softw. Practice Experience*, vol. 33, no. 5, pp. 423–460, 2003.

[13] Y. Younan, W. Joosen, and F. Piessens, “Runtime countermeasures for code injection attacks against C and C++ programs,” *ACM Comput. Surveys*, vol. 44, no. 3, pp. 17:1–17:28, Jun. 2012.

[14] S. Son, K. S. McKinley, and V. Shmatikov, “Diglossia: Detecting code injection attacks with precision and efficiency,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 1181–1192.

[15] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, “Precise client-side protection against DOM-based cross-site scripting,” in *Proc. 23rd USENIX Secur.*, 2014, pp. 655–670.

[16] X. Lin, P. Zavarsky, R. Ruhl, and D. Lindskog, “Threat modeling for CSRF attacks,” in *Proc. Int. Conf. Comput. Sci. Eng.*, 2009, pp. 486–491.

[17] H. Bojinov, E. Bursztein, and D. Boneh, “XCS: Cross channel scripting and its impact on web applications,” in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 420–431.

[18] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna, “Client-side cross-site scripting protection,” *Comput. Secur.*, vol. 28, no. 7, pp. 592–604, 2009.

[19] M. T. Louw and V. N. Venkatakrishnan, “Blueprint: Robust prevention of cross-site scripting attacks for existing browsers,” in *Proc. 30th IEEE Symp. Secur. Privacy*, 2009, pp. 331–346.

[20] T. Jim, N. Swamy, and M. Hicks, “Defeating script injection attacks with browser-enforced embedded policies,” in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 601–610.

[21] Y. Nadji, P. Saxena, and D. Song, “Document structure integrity: A robust basis for cross-site scripting defense,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2006, pp. 463–472.

[22] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E. P. Markatos, and T. Karagiannis, “xJS: Practical XSS prevention for web application development,” in *Proc. USENIX Conf. Web Appl. Develop.*, 2010, pp. 13–13.

[23] L. A. Meyerovich and B. Livshits, “ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser,” in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 481–496.

[24] D. Yu, A. Chander, N. Islam, and I. Serikov, “JavaScript instrumentation for browser security,” in *Proc. 34th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2007, pp. 237–249.

[25] P. H. Phung, D. Sands, and A. Chudnov, “Lightweight self-protecting JavaScript,” in *Proc. 4th Int. Symp. Inf. Comput. Commun. Secur.*, 2009, pp. 47–60.

[26] S. Van Acker , P. De Ryck , L. Desmet, F. Piessens, and W. Joosen, “WebJail: Least-privilege integration of third-party components in web mashups,” in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 307–316.

[27] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, “SOMA: Mutual approval for included content in web pages,” in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 89–98.

[28] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen, “CsFire: Transparent client-side mitigation of malicious cross-domain requests,” in *Proc. 2nd Int. Conf. Eng. Secure Softw. Syst.*, 2010, pp. 18–34.

[29] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Cross-site scripting prevention with dynamic data tainting and static analysis,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2007, pp. 12–24.

[30] P. Bisht and V. N. Venkatakrishnan, “XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks,” in *Proc. 5th Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2008, pp. 23–43.

[31] S. Stamm, B. Sterne, and G. Markham, “Reining in the web with content security policy,” in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 921–930.

[32] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “JSFlow: Tracking information flow in JavaScript and its APIs,” in *Proc. 29th Annu. ACM Symp. Appl. Comput.*, 2014, pp. 1663–1671.

[33] D. Stefan, et al., “Protecting users by confining JavaScript with COWL,” in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 131–146.

[34] L. Bauer, S. Cai, L. Jia, P. Timothy, S. Michael, and T. Yuan, “Runtime monitoring and formal analysis of information flows in Chromium,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[35] D. B. Giffin, et al., “Hails: Protecting data privacy in untrusted web applications,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 47–60.

[36] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, “BrowserShield: Vulnerability-driven filtering of dynamic HTML,” *ACM Trans. Web*, vol. 1, Sep. 2007, Art. no. 11.

[37] N. Jovanovic, E. Kirda, and C. Kruegel, “Preventing cross site request forgery attacks,” in *Proc. 2nd Int. Conf. Secur. Privacy Commun. Netw.*, 2006, pp. 1–10.

[38] M. V. Gundy and H. Chen, “Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks,” in *Proc. 16th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2009.

[39] M. Johns and C. Beyerlein, “SMASK: Preventing injection attacks in web applications by approximating automatic data/code separation,” in *Proc. ACM Symp. Appl. Comput.*, 2007, pp. 284–291.

[40] S. Nanda, L.-C. Lam, and T.-C. Chiueh, “Dynamic multi-process information flow tracking for web application security,” in *Proc. Int. Conf. Middleware Companion*, 2007, pp. 19:1–19:20.

[41] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, “SWAP: Mitigating XSS attacks using a reverse proxy,” in *Proc. ICSE Workshop Softw. Eng. Secure Syst.*, 2009, pp. 33–39.

[42] M. Johns, B. Engelmann, and J. Posegga, “XSSDS: Server-side detection of cross-site scripting attacks,” in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2008, pp. 335–344.

[43] R. Pelizzi and R. Sekar, “A server- and browser-transparent CSRF defense for web 2.0 applications,” in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 257–266.

[44] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using parse tree validation to prevent SQL injection attacks,” in *Proc. 5th Int. Workshop Softw. Eng. Middleware*, 2005, pp. 106–113.

[45] V. Haldar, D. Chandra, and M. Franz, “Dynamic taint propagation for Java,” in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2005, pp. 303–311.

[46] W. Xu, S. Bhatkar, and R. Sekar, “Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks,” in *Proc. 15th USENIX Secur. Symp.*, Aug. 2006, pp. 121–136.

[47] T. Pietraszek and C. V. Berghe, “Defending against injection attacks through context-sensitive string evaluation,” in *Proc. 8th Int. Conf. Recent Advances Intrusion Detection*, 2006, pp. 124–145.

[48] I. Papagiannis, M. Migliavacca, and P. Pietzuch, “PHP Aspis: Using partial taint tracking to protect against injection attacks,” in *Proc. 2nd USENIX Conf. Web Appl. Develop.*, 2011, pp. 2–2.

[49] D. Mitropoulos and D. Spinellis, “SDriver: Location-specific signatures prevent SQL injection attacks,” *Comput. Secur.*, vol. 28, pp. 121–129, May/Jun. 2009.

[50] W. G. Halfond and A. Orso, “AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks,” in *Proc. 20th Int. Conf. Automated Softw. Eng.*, Nov. 2005, pp. 174–183.

[51] S. Boyd and A. Keromytis, “SQLrand: Preventing SQL injection attacks,” in *Proc. 2nd Appl. Cryptography Netw. Secur. Conf.*, 2004, pp. 292–304.

[52] S. Y. Lee, W. L. Low, and P. Y. Wong, “Learning fingerprints for a database intrusion detection system,” in *Proc. 7th Eur. Symp. Res. Comput. Secur.*, 2002, pp. 264–280.

[53] F. Valeur, D. Mutz, and G. Vigna, “A learning-based approach to the detection of SQL attacks,” in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2005, pp. 123–140.

[54] S. Chong, K. Vikram, and A. C. Myers, “SIF: Enforcing confidentiality and integrity in web applications,” in *Proc. 16th USENIX Secur. Symp.*, 2007, pp. 1:1–1:16.

[55] W. Cheng, et al., “Abstractions for usable information flow control in Aeolus,” in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 12–12.

[56] G. Gu, P. Fogla, D. Dagon, W. Lee, and B. Skorić, “Measuring intrusion detection capability: An information-theoretic approach,” in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, pp. 90–101.

[57] S. Linn, “A new conceptual approach to teaching the interpretation of clinical tests,” *J. Statist. Educ.*, vol. 12, no. 3, pp. 1–9, 2004.

[58] C. Pfleeger and S. Pfleeger, *Analyzing Computer Security: A Threat/Vulnerability/Countermeasure Approach*. Englewood Cliffs, NJ, USA: Prentice Hall, 2012.

[59] S. M. Easterbrook, “Open code for open science,” *Nature Geoscience*, vol. 7, pp. 779–781, 2014.

[60] R. Jain, *The Art of Computer Systems Performance Analysis*. Hoboken, NJ, USA: Wiley, 1991.

[61] G. Brendan, *Systems Performance: Enterprise and the Cloud*. Englewood Cliffs, NJ, USA: Prentice Hall, 2014.

[62] D. Mitropoulos, V. Karakoidas, P. Louridas, and D. Spinellis, "Countering code injection attacks: A unified approach," *Inf. Manage. Comput. Secur.*, vol. 19, no. 3, pp. 177–194, 2011.
[63] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976.
[64] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2002, pp. 255–264.
[65] CSP, XSS Jigsaw, 2015. [Online]. Available: http://blog.innerht.ml/csp-2015/
[66] A. D. Keromytis, "Randomized instruction sets and runtime environments: Past research and future directions," *IEEE Secur. Privacy*, vol. 7, no. 1, pp. 18–25, Jan. 2009.
[67] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2003, pp. 272–280.
[68] A. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? The effectiveness of instruction set randomization," in *Proc. 14th USENIX Secur.*, 2005, pp. 10–10.
[69] A. Naderi, M. Bagheri, and S. Ramezany, "Taintless: Defeating taint-powered protection tachniques," presented at the Black Hat, USA, Aug. 2014.
[70] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011.
[71] S. McCamant and M. D. Ernst, "A simulation-based proof technique for dynamic information flow," in *Proc. Workshop Program. Languages Anal. Secur.*, 2007, pp. 41–46.
[72] D. E. R. Denning, "An intrusion detection model," *IEEE Trans. Softw. Eng.*, vol. 13, no. 2, pp. 222–232, Feb. 1987.
[73] L. D. Brown, T. T. Cai, and A. DasGupta , "Interval estimation for a binomial proportion," *Statist. Sci.*, vol. 16, no. 2, pp. 101–133, 2001.
[74] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of DOM-based XSS," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2013, pp. 1193–1204.
[75] S. Vance, *Quality Code: Software Testing Principles, Practices, and Patterns*. Reading, MA, USA: Addison-Wesley, 2014.

Dimitris Mitropoulos received the PhD degree in cyber security from Athens University of Economics and Business, in 2014. He is a post-doctoral researcher in the Computer Science Department, Columbia University. His research interests include application security, systems security, and software engineering.

Panos Louridas is an associate professor with Athens University of Economics and Business. He has published in many areas of software engineering and is actively involved in the application of security research for the development of high-stakes production systems, such as e-voting.

Michalis Polychronakis received the PhD degree in computer science from the University of Crete, Greece, in 2009. He is an assistant professor with Stony Brook University. Before joining Stony Brook, he was an associate research scientist with Columbia University. His research interests include network and system security and network monitoring and measurement.

Angelos Dennis Keromytis is an associate professor of computer science with Columbia University, and the director of the Network Security Lab. He is currently serving as program manager in the Information Innovation Office (I2O), Defense Advanced Research Projects Agency (DARPA). His research interests include systems and network security, and cryptography.