Fatal injection: a survey of modern code injection attack countermeasures

Dimitris Mitropoulos and Diomidis Spinellis

Department of Management Science and Technology, Athens University of Economics and Business, Greece

ABSTRACT

With a code injection attack (CIA) an attacker can introduce malicious code into a computer program or system that fails to properly encode data that comes from an untrusted source. A CIA can have different forms depending on the execution context of the application and the location of the programming flaw that leads to the attack. Currently, CIAs are considered one of the most damaging classes of application attacks since they can severely affect an organisation's infrastructure and cause financial and reputational damage to it. In this paper we examine and categorize the countermeasures developed to detect the various attack forms. In particular, we identify two distinct categories. The first incorporates static program analysis tools used to eliminate flaws that can lead to such attacks during the development of the system. The second involves the use of dynamic detection safeguards that prevent code injection attacks while the system is in production mode. Our analysis is based on nonfunctional characteristics that are considered critical when creating security mechanisms. Such characteristics involve usability, overhead, implementation dependencies, false positives and false negatives. Our categorization and analysis can help both researchers and practitioners either to develop novel approaches, or use the appropriate mechanisms according to their needs.

Subjects Security and Privacy

Keywords Application security, Code injection attacks, Countermeasures, Static analysis, Dynamic prevention, Software vulnerabilities, Cross-site scripting

Submitted 31 January 2017
Accepted 3 October 2017
Published 27 November 2017

Corresponding author
Dimitris Mitropoulos,
dimitro@aueb.gr

Academic editor
Cynthia Irvine

Additional Information and Declarations can be found on page 26

DOI 10.7717/peerj-cs.136

INTRODUCTION AND COVERED AREA

Security vulnerabilities derive from a small number of programming flaws that lead to security breaches (*Wurster & Van Oorschot, 2008*; *Viega & McGraw, 2001*). One common mistake that developers make concerns user input, assuming, for example, that only word characters will be entered by the user, or that the user input will never exceed a certain length (*Mitropoulos et al., 2011*). Developers may assume, correctly, that a high-level language in an application will protect them against threats like buffer overflows (*Keromytis, 2011*). Developers may also assume, incorrectly, that user input is not a security issue any more. Such an assumpion can lead to the processing of invalid data that an attacker can introduce into a program and cause it to execute malicious code. This kind of exploit is known as a *code injection attack* (CIA) (*Ray & Ligatti, 2012*; *Mitropoulos et al., 2011*).

Code injection attacks have been topping the vulnerability lists of numerous bulletin providers for several years. (http://www.sans.org/top-cyber-security-risks/, http://cwe.mitre.org/top25/) Consider the Open Web Application Security Project (https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project) (OWASP) Top Ten project which represents a broad consensus about what the most critical web application security flaws are, and is referenced by Payment Card Industry Security Standards Council (https://www.pcisecuritystandards.org/security_standards/) (PCI DSS), Defense Information Systems Agency (www.disa.mil/) (DISA) and numerous researchers. In its three consecutive Top Ten lists (2007, 2010, 2013), different classes of code injection are included in the top five positions.

Over several years of efforts, a large body of knowledge has been assembled regarding code injection attacks consisting of countermeasures, novel ways of attacking, and others. In this paper we first identify the basic categories of CIAs ('Code Injection Attacks'). Then, we analyze the basic approaches used to counter such attacks, and the mechanisms that implement them ('Countermeasures'). Specifically, there are two categories of countermeasures that can be used by developers to: (a) identify and eliminate the vulnerabilities that the system contains during the development process, (b) guard the system against code injection attacks while it is in production mode. Then, we highlight the positive and negative aspects of each countermeasure and finally we evaluate them based on the following requirements (see 'Analysis and Discussion'):

- **Flexibility**: We check if an approach can be adjusted in order to detect different code injection attack categories.
- **Effectiveness Tests**: As long as we examine security mechanisms that detect either attacks or defects, we want to see if researchers have measured the effectiveness of their proposed mechanisms in terms of false positive and negative rates.
- **Implementation independence**: We check if the mechanisms depend either on the characteristics of the programming language that was used to develop them or on the implementation details of the protecting entity.
- **Computational Overhead**: Finally, we examine if a mechanisms imposes a cost due to its use, as it may introduce an amount of extra computation on an application.

All the aforementioned requirements are considered critical when building security mechanisms (*Anderson, 2001*; *Romero-Mariona et al., 2009*; *Mellado, Fernández-Medina & Piattini, 2010*; *Halfond, Viegas & Orso, 2006*). Finally, we discuss some emerging challenges for future research on the field ('Emerging Challenges'), and provide some general observations together with some concluding remarks ('Conclusions').

There is already a survey on mitigating software vulnerabilities in general (*Shahriar & Zulkernine, 2012*). The scope of that research is very broad and leaves out many approaches and mechanisms that we report here. Also, countermeasures that prevent two subcategories, namely: binary code injection (*Younan, Joosen & Piessens, 2012*) and SQL injection (*Halfond, Viegas & Orso, 2006*) in particular, have already been surveyed. The body of work done on the field though, exceeds the boundaries of this research too. Furthermore, in the latter case (*Halfond, Viegas & Orso, 2006*), the survey is quite old and since then the number of countermeasures that detect SQL injection attacks alone, has doubled. Finally, the authors of this survey do not take false positives and false negatives into account in their research.

CODE INJECTION ATTACKS

Code injection is a technique to introduce malicious code into a computer program by taking advantage of unchecked or wrong assumptions the program makes about its inputs (*Mitropoulos et al., 2011*). *Bratus et al. (2011)* portray the issue in a generic fashion: "*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")*". An example of the above definition is the following: The code fragment below, defines the operation of addition in the Scheme programming language (*Abelson & Sussman, 1996*; *Dybvig, 2009*):

```
(define (add x y) (+ x y))
```

Consider the case where instead of a number, a function that leads to an endless loop is passed as an argument by the user. This will cause the interpreter to enter an endless loop and lead to a denial of service. The intuition here is that "*every application that copies untrusted input verbatim into an output program is vulnerable to code injection attacks*". *Ray & Ligatti (2012)* actually proved the above claim based on formal language theory.

Code injection attacks are one of the most critical class of attacks (*Francillon & Castelluccia, 2008*; *Su & Wassermann, 2006*; *Baca, Carlsson & Lundberg, 2008*) due to the following reasons:

- They can occur in different layers, such as databases, libraries, native code and the browser.
- They span a wide range of security issues, such as viewing sensitive information, editing of personal data, or even stopping the execution of a system.

Figure 1 presents a categorization of CIAs divided into two categories. The first involves binary code and the second source code. We illustrate the attack categories and subcategories that have been analyzed in other research papers, in grey color. JavaScript injection has lately become a prominent subcategory, hence we provide some basic examples in Appendix.

Binary Code Injection Attacks: Such attacks involve the insertion of binary code into an application to alter its execution flow and execute malicious compiled code. This category involves buffer-overflow attacks (*Cowan et al., 1998*; *Keromytis, 2011*; *Szekeres et al., 2013*), a staple of security problems. These attacks may occur when the bounds of memory areas are not checked, and access beyond these bounds is possible by the program. Based on this, malicious users can inject additional data overwriting the existing data of adjacent memory. From there they can take control over a program or crash it. Another attack vector involves format string vulnerabilities. The basis of this defect is the unexpected behaviour of functions with variable arguments. Typically, a function that handles a number of arguments has to read them from the stack. If we specify a format string that will make `printf` expect two integers on the stack, and we provide only one parameter, the second one will have to be something else on the stack. If attackers have control over the format string, then they could eiter read from or write to arbitrary memory addresses. C and C++

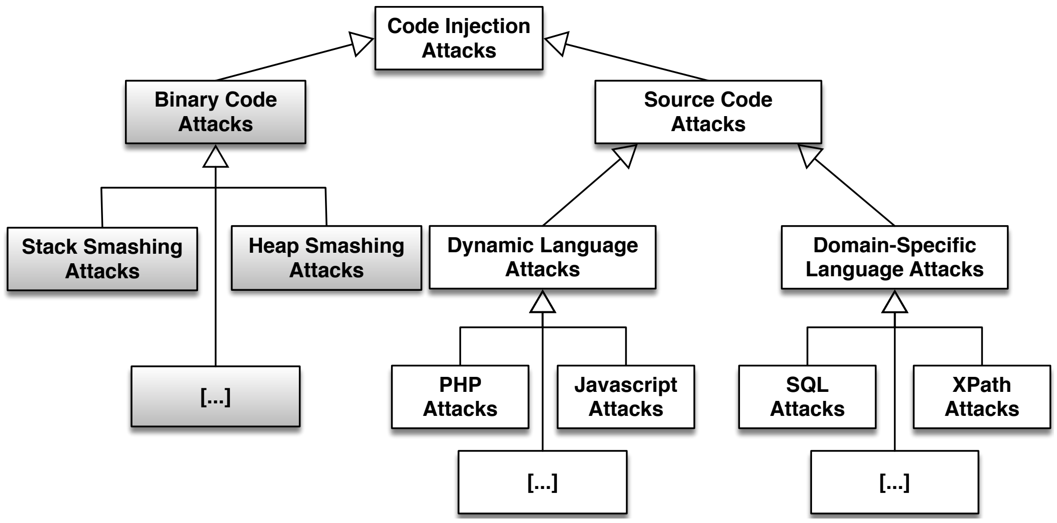

Figure 1 **A categorization of code injection attacks.** The subcategories that have been extensively analyzed in other research papers (*Lhee & Chapin, 2003*; *Pincus & Baker, 2004*) can be seen in grey colour.
Full-size DOI: 10.7717/peerjcs.136/fig-1

are two programming languages vulnerable to this kind of attacks since corresponding implementations lack a protection scheme against overwriting data in any part of the memory (*Mitropoulos et al., 2011*).

There are two research papers that present various techniques that belong to this category. In particular, an extensive survey on binary code injection attacks can be found in reference (*Lhee & Chapin, 2003*). Furthermore, specific advances in exploiting such vulnerabilities (i.e., *heap smashing*, *arc injection* and others) have been presented in reference (*Pincus & Baker, 2004*). Finally, the countermeasures used to detect such defects have already been surveyed (*Younan, Joosen & Piessens, 2012*) (many of them are also included in a book: *Das, Kant & Zhang, 2012*—Section 13.8). Nevertheless, we include some of them in this survey because they prompted the development of some sophisticated countermeasures.

Source Code Injection Attacks: Code injection also includes the use of source code, either of a *Domain Specific Language* (DSL) or a *Dynamic Language*. Note that binary code injection attacks can only occur when the target system is implemented in languages lacking array bounds checking, like C and C++. Contrariwise, source code-driven injection attacks can target applications written in various programming languages with different features and characteristics.

Code injection attacks that involve DSLs are critical, as DSLs like SQL and XML play an important role in the development of web applications. For instance, many web applications have interfaces through which web users enter input to interact with the application's data. In this way, they interact with the underlying RDBMS (Relational Database Management System). Typically, this input can become part of an SQL statement and then gets executed on the corresponding RDBMS. An attack that exploits the defects of these interfaces by taking advantage of input validation issues (e.g., inefficient type handling), is called an "SQL injection attack" (*CERT, 2002*; *Mitropoulos & Spinellis, 2009*). The various techniques used

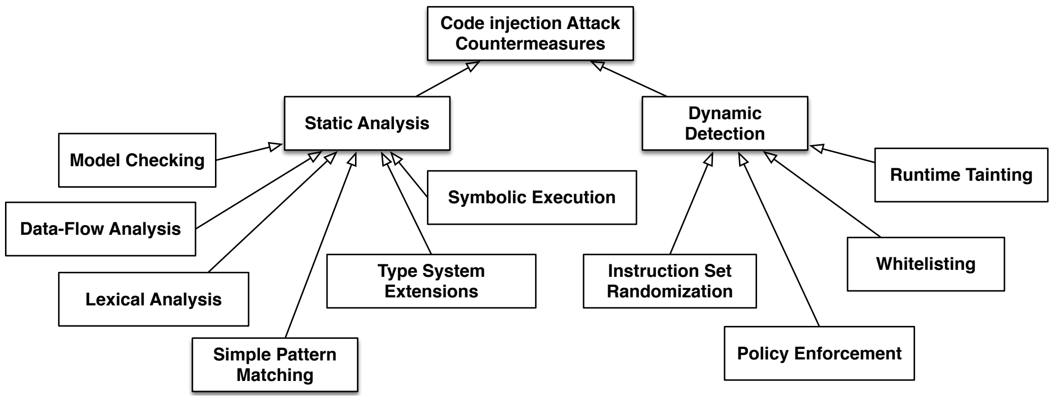

Figure 2 **The basic categories of code injection attack countermeasures.**
Full-size DOI: 10.7717/peerjcs.136/fig-2

to perform such attacks have can be found in reference (*Halfond, Viegas & Orso, 2006*). Instructive examples can be also found in reference (*Su & Wassermann, 2006*). By using very similar techniques to the ones presented in the aforementioned references, attackers can perform other exploits based on DSLs, like XML (*Mattos, Santin & Malucelli, 2013*) and XPath (*Su & Wassermann, 2006*; *Cannings, Dwivedi & Lackey, 2007*; *Mitropoulos, Karakoidas & Spinellis, 2009*).

A critical class of code injection attacks involve dynamic languages such as Python, Perl, JavaScript, and PHP (*Seixas et al., 2009*; *Egele et al., 2009*; *Son, McKinley & Shmatikov, 2013*). In particular, JavaScript injection attacks comprise a wide subset of dynamic language-driven attacks. Such attacks are manifested when an application accepts and redisplays data of unknown origin without appropriate validation and filtering. Based on this vulnerability, a malicious user can manage to inject a script in the JavaScript engine of a browser and alter its execution flow (*Erlingsson, Livshits & Xie, 2007*). JavaScript injection attacks are considered as a crucial issue in application security because they are associated with major vulnerabilities such as: XSS attacks (*Sivakumar & Garg, 2007*) and XCS (Cross-Channel Scripting) attacks (*Wang, 2010*; *Bojinov, Bursztein & Boneh, 2009*). As we mentioned earlier, typical examples of JavaScript injection attacks can be found on the Appendix of this paper—A.

COUNTERMEASURES

Two different basic methods are used to deal with the code injection problem (see Fig. 2):

- **Static Analysis** involves the inspection of either source or binary code to find software bugs that could lead to a code injection attack without actually executing the program.
- **Dynamic Detection** observes the behavior of a running system in order to detect and prevent a code injection attack.

In the first case, programmers try to eliminate software vulnerabilities while applications are created (also known as the *build-in security* (*McGraw, 2006*) concept). The second concept involves the development of methods and tools that secure systems after their deployment. There are numerous approaches that belong to each of these two basic methods and for each approach Fig. 2, provides the corresponding bibliography.

Static analysis

The main concept behind static analysis is to identify security vulnerabilities during the development process. Currently, there are many software development processes that include static analysis tools for security as their integral parts (*Brown & Paller, 2008*; *Gregoire et al., 2007*; *Fagan, 1999*). From the usage of utilities like `grep` to complex methods, static analysis has been an evolving approach to detect software vulnerabilities (*Chess & West, 2007*).

Initially, the most straightforward approach is the adoption of *secure coding practices* (*Howard & LeBlanc, 2003*; *Viega & McGraw, 2001*; *McGraw, 2006*). For example, to prevent an SQL injection attack, developers can use specific features provided by the language they use (e.g., Java's `PreparedStatement` object). Nevertheless, this does not usually happen, as developers may not be aware of them, or time schedules may be tight, encouraging sloppy practices instead.

Simple pattern matching

During a manual code review it is easy to look for functions associated with code injection defects. Using existing tools available in almost every operating system, security auditors can search through a set of files for an arbitrary text pattern. These patterns are commonly specified through a regular expression. Accompanied by a well organized list of patterns, the auditor can quickly identify locations at which a program might face security problems.

If auditors choose to use utilities like `grep` and `qgrep` though, they must check for every vulnerability manually. Apart from this, they must have an expert knowledge because there are many different kinds of such defects. Furthermore, the analysis these utilities perform is naive. For example there is no distinction between a vulnerable function call, a comment, and an unrelated identifier. Hence, a higher prevalence of false positives should be expected (*Chess & McGraw, 2004*). Finally the output can be disorganized and overwhelming. The distinct disadvantages of pattern scanning and the continuous emergence of new defects were some of the main reasons that led to more sophisticated approaches.

Lexical analysis

Lexical analysis is one of the first approaches used for detecting security defects. This is because it is simple and easy to use. Lexical analysis is based upon formal language theory and finite state automata (*Aho et al., 2006*). As a term, it is mostly used to describe the first phase of the compilation process. However, there is no difference between this phase and the method that we describe here (*McGraw, 2008*). The two differ only in the manipulation of their outcome.

There are three phases that can be distinguished here, namely: scanning, tokenizing and matching (*Kantorovitz, 2004*). In the first two phases possible character sequences are recognized, classified and transformed into various tokens. Then the resulting sequences are associated with security vulnerabilities. Specifically, there are lists that contain entries of vulnerable constructs used during the matching phase. After a successful match an alert message warns auditors, describing the vulnerability and providing them with alternative usages.

The lexical analysis approach is implemented by security utilities such as BOON (*Wagner et al., 2000*), PS*can* (*Johnson, 2006*; *Heffley & Meunier, 2004*; *Chen & Wagner, 2007*), ITS4 (*Viega et al., 2002*; *Viega et al., 2000*; *Wilander & Kamkar, 2002*), *Flawfinder* (http://www.dwheeler.com/flawfinder/) (*Wilander & Kamkar, 2002*) and RATS (http://www.security-database.com/toolswatch/RATS-v2-3-Rough-Auditing-Tool-for.html) (*Kong et al., 2007*; *Chess & McGraw, 2004*; *Wilander & Kamkar, 2002*). For the most part, these tools scan source code pointing out unsafe calls of string-handling functions that could lead to a CIA. Then, they provide a list of possible threats ranked by risk level. This level is usually specified by checking the arguments of such functions. The vulnerability lists are simply constructed making the addition, removal, and modification of an entry quite easy. All the aforementioned tools scan C and C++. This exclusiveness lies on the fact that C and its standard libraries are very susceptible to binary code injection attacks as we mentioned in 'Code Injection Attacks'.

Lexical analysis can be flexible, straightforward and extremely fast. With one or more non-processed files as input, and simple descriptions as output, developers can quickly check their code for vulnerabilities. Also they can easily update and edit their vulnerability library with new possible threats due to its simplistic nature. Although superior to manual pattern matching, this approach has no knowledge of the code's semantics or how data circulates throughout a program. As a result there are several false positive and negative reports (*Chess & West, 2007*; *Cowan, 2003*). Note though, that lexical analysis utilities helped the gathering and depiction of a tentative set of security rules in one place for the first time (*McGraw, 2008*).

Data-flow analysis

Data-flow analysis is another compiler-associated approach used to discover software defects. It is more sophisticated and more appropriate for a comprehensive code review than lexical analysis.

Data-flow analysis can be described as a process that gathers details that concern the definition and dependencies of data within a program without executing it (*Moonen, 1997*; *Fosdick & Osterweil, 1976*). In addition, data-flow analysis algorithms can document all sequences of specific types of events which might occur in a program execution. The key insight of this approach is a Control-Flow Graph (CFG). Based on the program's CFG, this method examines how data moves throughout a program by representing all its possible execution paths (*Chess & West, 2007*; *Cahoon & McKinley, 2001*). By traversing the CFG of a program, data-flow analysis can determine where values are generated and where they are used. Hence this approach can be used to describe safety properties that are based on the flow of information (*Abi-Antoun, Wang & Torr, 2007*).

As an example: it is very likely that the CFG of a program with an SQL injection defect, will include a data-flow path from an input function to a vulnerable operation. For instance, in the following code fragment, user input reaches a method that interacts with a database without any prior validation:

```
uName = request.getParameter("username");
String query = null;
if (uName != null) {
    query = "SELECT *"+
    "FROM table WHERE uname = ' "+uName+" '";
    rs = stmt.executeQuery(query);
} else {
    ...
}
```

As a result, the danger of an SQL injection attack is prominent. As it is already clear from the aforementioned example, data-flow analysis is tailored to localize code injection vulnerabilities since it can be applied to associate the unchecked input with the execution of the query and issue a notification. This is why there are numerous adaptations that detect SQL injection defects, cross-site scripting vulnerabilities, buffer overflows and others. In addition, most of the creators of such frameworks, claim that with minor changes, their prototypes can be equally applied to also detect other kinds of such defects.

Contrary to lexical analysis, to counter such anomalies, a data-flow analysis mechanism needs more than a vulnerability library that connects coding constructs with software defects. Furthermore, a rule-pack containing specific control flow rules and ad-hoc checkers that run upon the CFG are required. The most common rules used in this method, are the *source*, the *pass-through* and the *sink* rules (*Chess & West, 2007*). A source rule denotes the starting point of a possible hazard while a sink rule depicts the coding construct where the hazard takes place. In the aforementioned example, a source rule will apply for the first line where input can come from a malicious user. The sink rule on the other hand will refer to the fifth line where attacked-controlled data can reach the database. The pass rule indicates the code that exists between the above two and carries the possibly corrupted data. For the most part, these rules are maintained in external files that use a specific format to describe them.

Livshits & Lam (2005) based their work in the functionality presented above to detect possible SQL and JavaScript injection defects in Java applications. *Nagy & Mancoridis (2009)* have proposed a number of checkers that locate buffer overflow and format string defects. The idea behind their proposal is to mark all the user-input-related parts of the source code. These checkers are implemented as plug-ins to the *CodeSurfer* (http://www.grammatech.com/research/technologies/codesurfer) tool (*Anderson & Zarins, 2005*), a commercial tool that performs data-flow analysis on C/C++ programs. Another two indicative tools used to detect injection anomalies are *Pixy* (*Jovanovic, Kruegel & Kirda, 2006*) and XSS*detect* (https://blogs.msdn.microsoft.com/ace_team/2007/10/22/xssdetect-public-beta-now-available/). Both of these tools detect cross-site scripting vulnerabilities in web applications. The latter, released by Microsoft, runs as a Visual Studio plug-in and analyzes .NET IL (Intermediate Language) which is read directly from the compiled binaries. Pixy on the other hand, is a standalone open source tool, that examines PHP scripts. In many cases, rules can appear directly in the code of the program in the form of annotations.

A tool that considers control flow graphs and uses annotations at the same time to find buffer overflows and memory leaks is *Splint* (*Evans & Larochelle, 2002*). *FindBugs* (*Ayewah & Pugh, 2010*; *Hovemeyer & Pugh, 2007*; *Spacco, Hovemeyer & Pugh, 2006*) is also a static analyzer based on data-flow analysis. *Dahse & Holz (2014)* have proposed a refined type of data-flow analysis to detect second-order vulnerabilities. Notably, such vulnerabilities occur when an attack payload is first stored by the application on the web server and then later on used in a security-critical operation.

As a more sophisticated approach than lexical analysis, data-flow analysis exhibits fewer false positives and negatives than the former. For example, many buffer overflows are not exploitable because the attacker cannot handle the data that overflows the buffer. By using this method, an auditor can in fact distinguish exploitable from non-exploitable buffer overflows. The advantage of data flow static analysis is that it can identify vulnerabilities that could actually occur when real application paths are exercised and not just dangerous coding constructs.

Model checking

Model checking is a formal verification approach developed based on graph theory and finite state automata (*Clarke, Emerson & Sifakis, 2009*; *Merz, 2001*). A software model checking framework accepts a system's source or binary code as input and checks automatically if it satisfies specific properties. First, the framework analyzes statically the code to extract a high-level representation of the system, namely a model. This model usually corresponds to a control-flow graph or a pushdown automaton (*Beyer et al., 2007*; *Chen & Wagner, 2002*). The properties are often expressed either as assertions, as formulas of temporal logic, or as finite state automata (*Pnueli, 1977*; *Miller, Donaldson & Calder, 2006*). By traversing every execution path of the model, the framework determines whether certain states represent a violation of the provided properties.

There is a great number of dangerous programming practices that can be accurately modeled with equivalent security properties. For example, the `chroot` system call should be followed by a call to `chdir ("/")`. Otherwise, the current working directory could be outside the isolated hierarchy and provide access to a malicious user via relative paths. With a high-level representation of the system at hand and by using ad-hoc algorithms (*Reps, Horwitz & Sagiv, 1995*), properties like the above can be easily checked. Likewise, it is possible to state the detection of code injection defects as a reachability problem (*Tsitovich, 2008*).

There are many tools based on model checking to detect software vulnerabilities. Classic tools include SPIN (*Holzmann, 1997*), SMV (*McMillan, 1992*) and MOPS (*Chen & Wagner, 2002*; *Schwarz et al., 2005*). These tools are representative of the approach but they do not support the detection of CIA defects. QED is a model checking system that accepts as input web application written in the standard Java servlet specification (https://jcp.org/aboutJava/communityprocess/final/jsr315/) and examine them for various code injection vulnerabilities (*Martin & Lam, 2008*). Also, *Fehnker, Huuck & Rödiger (2011)* have proposed a model checking approach to detect binary code injection defects in embedded systems.

The users of a model checking tool do not need to construct a correctness proof. Instead, they just need to enter a description of the circuit or program to be verified and the specification to be checked. Still, writing specifications is hard and code reviewers with experience are needed. One of the key features of model checking is that it can either reassure developers that the system is correct or provide them with a counterexample. As a result, together with the discover of a security issue, auditors are provided with a possible solution. A major problem in model checking is the state explosion issue (*Clarke, Emerson & Sifakis, 2009*; *Merz, 2001*). The number of all states of a system with many processes or complicated data structures can be enormous.

Symbolic execution

Symbolic execution generalizes testing by using unknown symbolic variables during evaluation (*King, 1976*; *Cadar et al., 2011*). In essence, it provides the means to analyze a program to determine which inputs cause each part of a program to execute. This concept can be easily adapted to detect vulnerabilities that may lead to code injection attacks.

To counter SQL injection attacks, *Fu & Qian (2008)* have proposed SAFELI. First, SAFELI analyzes the code to detect code constructs used by the application, to interact with a database. At each location that submits an SQL query, an equation is constructed to find out the initial values that could lead to a security breach. The equation is then solved by a hybrid string solver where the solution obtained is used to construct test cases. If a defect is detected, an attack is replayed by the tool to developers. *Ruse, Sarkar & Basu (2010)* detect SQL injection vulnerabilities in a similar manner. In addition, *Rubyx* (*Chaudhuri & Foster, 2010*) follows a similar approach to counter JavaScript injection attacks in applications written in Ruby. S3 (*Trinh, Chu & Jaffar, 2014*) is a symbolic string solver that can be used to detect vulnerabilities that may lead to SQL injection and XSS attacks. To do so, it makes use of a symbolic representation so that membership in a set defined by a regular expression can be expressed as a string equation. Then, there is a constraint-based generation of instances from these expressions so that the number of instances can be limited. *Saxena et al. (2010)* have proposed a framework called *Kudzu*, to detect JavaScript injection attacks. To achieve this, Kudzu explores the application's execution space by creating test cases. Then, like SAFELI, Kudzu uses a solver which is implemented by the authors in order to overcome the complexity of JavaScript's string operations. Finally, by using data-flow analysis, it identifies possible defects based on specific *sink rules* (see 'Data-flow analysis').

KLEE (*Cadar, Dunbar & Engler, 2008*) was the first symbolic execution engine introduced to detect software bugs in an efficient manner. Such bugs include defects that may lead to binary code injection attacks. In addition, *MergePoint* (*Avgerinos et al., 2014*) is a binary-only symbolic execution system for large-scale testing of commodity software. Notably, it is based on *veritesting*, an approach that employs the merging of execution paths during static symbolic execution, to reinforce the effects of dynamic symbolic execution.

Symbolic execution has also been used together either with genetic algorithms to detect JavaScript injection attacks (*Avancini & Ceccato, 2013*) or with runtime tainting (see 'Runtime Tainting') to detect SQL injection attacks (*Corin & Manzano, 2012*). Symbolic execution shares a similar problem with model checking (see 'Model checking'). Symbolically executing all program paths does not scale with large programs since the number of feasible paths grows exponentially.

Type system extensions

A type system is a collection of rules that assign a property called a type to the various constructs of a program (*Pierce, 2002*). One of the most typical advantages of static type checking is the discovery of programming errors at compile time. As a result, numerous errors can then be detected immediately, rather than discovered later upon execution.

For the most part, type extensions aim to overcome the problems of integrating different programming languages. For instance, the integration of SQL with the Java programming language is typically realised with the JDBC application library (*Fisher, Ellis & Bruce, 2003*). By using it, the programmer has to pass the SQL query to the database as a string. Thought this process, the Java compiler is completely unaware of the SQL language contained within the Java code paving the way for an SQL injection attack (recollect the example of 'Data-flow analysis'). Type-safe programming interfaces like SQL DOM (Domain Object Model) (*McClure & Krüger, 2005*) and the *Safe Query Objects* (*Cook & Rai, 2005*) were two of the first attempts to detect SQL injection attacks via type extension. Both of the above mechanisms act as preprocessors and translate an SQL database schema into the host general purpose language. The generated collection of objects is used as an application library for the main application, thus ensuring type safety and syntax checking at compile-time. SQLJ (*Eisenberg & Melton, 1999*) is a language extension of Java that supports SQL. It offers type and syntax checking for both languages at compile-time. *SugarJ* (*Erdweg, 2013*) provides a method through which languages can be extended with specific syntax, in order to embed DSL's. The major contribution of this framework is that can be easily applied on many languages as host languages. Currently it supports Java, Haskell and Prolog. All the aforementioned mechanisms wipe out the relationship between untyped Java strings and SQL queries, but do not address legacy code. In addition, developers require to learn a new API to use them.

*Web*SSARI is used to verify web applications (*Xie & Aiken, 2006*) written in PHP. It is based on Denning's lattice model which analyzes the information flow of a program (*Denning & Denning, 1977*) and uses type qualifiers to associate security classes with variables and functions that can lead to SQL injection defects. *Wassermann & Su (2007)* have proposed an approach that deals with static analysis and coding practices together (*Wassermann & Su, 2004*) to detect SQL injection attacks. Specifically, they analyze the application's code to locate queries that are considered unsafe. To achive this, they use context free grammars and language transducers (*Minamide, 2005*).

Type extensions is a formal way to wipe out code injection defects, but they have a distinct disadvantage: programmers need to learn new constructs and modify their code in multiple places.

Dynamic detection

Dynamic detection involves the development of methods and tools to fortify such applications without actually removing the defects from the application's code. A great number of methods that belong to this category involves some kind of *dynamic program analysis* (*Boujarwah, Saleh & Al-Dallal, 2000*). Dynamic analysis requires a running system and involves sufficient test inputs to examine the behavior of a system.

Runtime tainting

Runtime tainting is based on data-flow analysis (see 'Data-flow analysis'). In practice, it enforces security policies by marking untrusted ("tainted") data and tracing its flow through the program. Runtime tainting may be viewed as an approximation of the verification of non-interference (*Von Oheimb, 2004*) or the more general concept of secure information flow. Since information flow in a system cannot be verified by examining a single execution trace of the system, the results of taint analysis will necessarily reflect approximate information regarding the information flow characteristics of the system to which it is applied.

Runtime tainting is a feature in some programming languages, such as Perl (http://search.cpan.org/~rhandom/Taint-Runtime-0.03/lib/Taint/Runtime.pm) and Ruby. The following Perl code is vulnerable to SQL injection since it does not check the value of the `$foo` variable, which is instantiated by user input:

```
#!/usr/bin/perl
my $name = $cgi->param("foo");
...
$dbh->TaintIn = 1;
$dbh->execute("SELECT *
                FROM users
                WHERE name = '$foo';");
```

If taint mode is turned on, Perl would refuse to run the command and exit with an error message, because a tainted variable is being used in a query.

SigFree (*Wang et al., 2010*) is a mechanism that follows this method to counter buffer overflow attacks by detecting the presence of malicious binary code. This is based on the fact that such attacks typically contain executable code while legitimate requests never contain executable code. However, this is not always the case and therefore the mechanism suffers from false alarms. LIFT (*Qin et al., 2006*) also counters binary code injection attacks in a similar manner. The system by *Haldar, Chandra & Franz (2005)* provides runtime tainting for applications written in Java, while the work by *Xu, Bhatkar & Sekar (2006)* covers applications written in C. *SecuriFly* (*Martin, Livshits & Lam, 2005*) is a similar mechanism based on PQL (http://pql.sourceforge.net/) (Program Query Language), which is a language for expressing patterns of events on objects.

A dynamic checking compiler called WASC (*Nanda, Lam & Chiueh, 2007*) includes runtime tainting to prevent JavaScript injection attacks. To counter similar attacks, PHP *Aspis* (*Papagiannis, Migliavacca & Pietzuch, 2011*) applies partial taint tracking at the language level to augment values with taint meta-data in order to track their origin. Vogt et al. (2007), use runtime tainting to prevent JavaScript injection attacks. This is done by inspecting the information flow within the browser. When critical information is about to be sent to a third party, the web user decides if this should be allowed or not. Stock et al. (2014) have proposed a method that operates on the client-side too. This method uses a taint-enhanced JavaScript engine that tracks the flow of data controlled by the attacker. To detect an attack, the method uses HTML and JavaScript parsers that can identify the generation of malicious code coming from tainted data. Runtime tainting has been partially or fully used in other similar approaches (Nadji, Saxena & Song, 2009; Sekar, 2009; Nguyen-Tuong et al., 2006). Notably, such approaches may require numerous changes to the compiler or the runtime system.

In positive data flow tracking, tagged data is considered to be legitimate. Information Flow Control (IFC) mechanisms employ positive taint tracking to prevent JavaScript-driven XSS attacks on the browser. Representative implementations such as JSF*low* (Hedin et al., 2014), COWL (Stefan et al., 2014) and the framework by Bauer et al. (2015) allow programmers to express information flow policies by extending the type system of JavaScript. Then, the policies are checked at runtime by the JavaScript interpreter through dynamic checks.

Instruction set randomization

Another approach that has been previously proposed as a generic methodology to counter code injection attacks is Instruction Set Randomization (ISR) (Keromytis, 2009; Kc, Keromytis & Prevelakis, 2003). The concept behind ISR is to create an execution environment that is unique to the running process. This environment is created by using a randomization algorithm. Hence, an attack against this system will fail as the attacker cannot guess the key of this algorithm. The main issue with this approach is that it uses a cryptographic key in order to match the execution environment. As a result, security depends on the fact that malicious users cannot discover the secret key. Note that, randomization algorithms are also employed in another popular technique that has been extensively used to prevent binary code injection attacks, address space layout randomization (ASLR) (Shacham et al., 2004). To do so, ASLR randomly arranges the address space positions of critical data areas of a process such as the base of the executable and the positions of the stack and heap.

SQL*rand* (Boyd & Keromytis, 2004) is based on ISR to detect SQL injections in the following manner: initially, it allows developers to create queries using randomized instructions instead of standard SQL keywords. The modified SQL statements are either reconstructed at runtime using the same key that is inaccessible to the attacker, or the user input is tagged with delimiters that allow an augmented SQL grammar to detect the attack. Even if SQLrand imposes a low computational overhead, it imposes an infrastructure overhead since it requires the integration of a proxy.

In the case of JavaScript, consider a XOR function that encodes all JavaScript source of a web page on the server-side and then, on the client-side, the web browser decodes the source by applying the same function again. Implementations of this approach include:

Noncespaces (*Gundy & Chen, 2009*) and *x*JS (*Athanasopoulos et al., 2010*). SM*ask* (*Johns & Beyerlein, 2007*) identifies malicious code by automatically separating user input from legitimate code by using JavaScript and HTML keyword masking (in a way similar to SQLrand *Boyd & Keromytis, 2004*). Even if ISR is theoretically a sound approach for countering code injection, these implementations have flaws. For example, Noncespaces does not protect from persistent data injection. XJS does not have such problems and covers a wide variety of JavaScript injection attacks.

Policy enforcement

Policy enforcement is mainly associated with database security (*Thuraisingham & Ford, 1995*; *Null & Wong, 1992*; *Chlipala, 2010*; *Son, Chaney & Thomlinson, 1998*) and operating system strict access controls (*Winsor, 2000*; *Hicks et al., 2010*). In such contexts, policies expressed in specific languages (*Anderson, 2005*), usually limit information dissemination to authorized entities only. Currently, policy enforcement is one of the most common approaches to detect JavaScript injection attacks. In this approach, web developers define security policies on the server-side. Then, these policies are enforced either in the user's browser or on a server-side proxy that intercepts all HTML responses.

All modern browsers include a JavaScript (JS) engine to support the execution of JavaScript. Most JS engines employ restrictions like the *same origin policy* (*Takesue, 2008*) and a *sandbox mechanism* (*Dhawan & Ganapathy, 2009*). In particular, scripts run in a sandbox where they can only perform web-related actions and not general-purpose programming tasks (e.g., creating files) (*Dhawan & Ganapathy, 2009*). Also, scripts are constrained by the same origin policy. This policy permits scripts running on pages originating from the same site to access each other's methods and properties with no specific restrictions, but prevents access to most methods and properties across pages on different sites (*Takesue, 2008*). Still, such schemes cannot stop malicious users from injecting scripts into the user's browser. Consider a legitimate web page that does not validate the input posted by its users. By exploiting this vulnerability, an attacker can post data that will inject JavaScript into a dynamically generated page. Thus the attacker can trick a legitimate user into downloading a well-hidden script from this host in order to steal the user's cookies. This injected script is confined by a sandboxing mechanism and conforms to the same origin policy, but it still violates the security of the browser (*De Groef et al., 2012*; *Saiedian & Broyle, 2011*).

Implementations of this approach include mechanisms such as *BrowserShield* (*Reis et al., 2006*) and *CoreScript* (*Yu et al., 2007*). Both mechanisms intercept JavaScript code on a page as it executes and rewrite it in order to check if it is subject to server-provided, vulnerability descriptions. Such implementations impose a significant overhead due to the JavaScript rewriting. DSI (*Nadji, Saxena & Song, 2006*), MET (*Erlingsson, Livshits & Xie, 2007*), and BEEP (*Jim, Swamy & Hicks, 2007*) require source modifications by the web developers in order to introduce their policies. Specifically, in MET the security policies are specified as JavaScript functions and they are included at the top of every web page while in BEEP web developers need to write security hooks for every embedded script of the application. *Blueprint* (*Louw & Venkatakrishnan, 2009*) is a policy enforcement framework that uses parsed trees to detect JavaScript injection attacks. However, to use it developers need to learn and use a new API in order to correctly escape dynamic content.

Google Caja (http://code.google.com/p/google-caja/) is another policy enforcement approach provided by Google. It is based on the object-capability security model (*McGraw, 2006*) and it aims to control what embedded third party code can do with user data. A security layer called "Content Security Policy" (CSP) (*Stamm, Sterne & Markham, 2010*) was first introduced into Firefox to detect various types of attacks, including cross-site scripting https://developer.mozilla.org/en/Introducing_Content_Security_Policy. Currently, it is supported by almost all the available browsers. To eliminate such attacks, web site administrators must specify which domains the browser should treat as valid sources of script and which not. Then, the browser will only execute scripts that exist in source files from white-listed domains. Notably, *Auto*CSP (*Fazzini, Saxena & Orso, 2015*) and *deDacota* (*Doupé et al., 2013*) are two schemes that are based on CSP.

Apart from JavaScript injection attacks, policy enforcement has been also used to detect binary code injection attacks. Specifically, *Kiriansky, Bruening & Amarasinghe (2002)* have proposed *program shepherding* which monitors control flow transfers in order to restrict execution privileges based on code origins and ensure that program sandboxing will not be breached. In a similar manner, *Control-flow Integrity* (CFI) (*Abadi et al., 2005*), follows a predetermined flow graph that serves as a specification of control transfers allowed in the program. Then, at runtime, specific checks enforce this specification. Adaptations of CFI include *Control-Pointer Integrity* (CPI) (*Kuznetsov et al., 2014*) and *Cryptographically Enforced Control Flow Integrity* (CCFI) (*Mashtizadeh et al., 2015*). The former ensures the integrity of all pointers in a program (e.g., function pointers) and as a result prevents different attacks. The latter employs Message Authentication Codes (MACs) to protect elements such as return addresses and function pointers. In general, CFI implementations track control edges separately, without taking into account the context of preceding edges. *Context-sensitive* CFI (*Van der Veen et al., 2015*) provides enhanced security by considering the backward and forward edges of the graph too. Notably, there is a number of attempts to overcome CFI. For instance, *Göktas et al. (2014)* have indicated that CFI can be bypassed by using return oriented programming (ROP) (*Buchanan et al., 2008*). Through ROP, attackers can gain control of the call stack to hijack program control flow. To do so, they execute specific machine instruction sequences that are already presented in machine's memory.

Whitelisting

Whitelisting approaches are based on the features of Denning's original intrusion detection framework (*Denning, 1987*). In the code injection context, a whitelisting mechanism registers all valid benign code statements during a learning phase. This can be done in various ways according to the implementation. Then, only those will be accepted, approved or recognized during production.

JavaScript injection whitelisting approaches generate and store valid JavaScript code in various forms, and detect attacks as outliers from the set of valid code statements. SWAP (*Wurzinger et al., 2009*) registers all the benign scripts that exist in the original application and stores an identifier for every benign script. Then, a JavaScript detection component placed in a proxy searches for malicious scripts in the server's responses. If no malicious scripts are detected, the proxy forwards the response to the client-side. Note that, this approach is inflexible since it does not support dynamic scripts. Similar limitations exist in XSS-GUARD (*Bisht & Venkatakrishnan, 2008*), which maps benign scripts to HTTP responses. To support dynamic scripts during the creation of the legitimate identifiers the authors of XSSDS (*Johns, Engelmann & Posegga, 2008*) substitute string-tokens with specified identifiers.

In the case of DSL-driven injection attacks the various countermeasures follow a similar pattern. DIDAFIT (*Lee, Low & Wong, 2002*) detects SQL injection attacks by registering all benign database transactions. Subsequent improvements by *Valeur, Mutz & Vigna (2005)* tagged each benign transaction with the corresponding web application. To do so, they have extended their anomaly detection framework called *libAnomaly* (http://seclab.cs.ucsb.edu/academic/projects/projects/libanomaly/). Furthermore, AMNESIA (*Halfond & Orso, 2005b*; *Halfond & Orso, 2006*) is a tool that detects SQL injection attacks by associating a query model with the location of every query in the web application. Then in production mode, monitors the execution of the application to examine when queries diverge from the expected model.

SQLGuard (*Buehrer, Weide & Sivilotti, 2005*) is another mechanism that detects SQL injection attacks based on parse tree validation. In particular, the mechanism compares the parse tree of the query before the inclusion of user input with the one resulting after the inclusion of user input. If the trees diverge, the application is probably under attack. *Diglossia* (*Son, McKinley & Shmatikov, 2013*) also uses parse trees to detect code injection attacks. The main idea behind Diglossia is based on the theory introduced by Ray and Ligatti in reference (*Ray & Ligatti, 2012*). Apart from SQL injection attacks, it can also be used to detect another emerging type of attacks: NoSQL (*Chodorow & Dirolf, 2010*) injection attacks (see also 'Emerging Challenges'). SD*river* (*Mitropoulos & Spinellis, 2009*; *Mitropoulos, Karakoidas & Spinellis, 2009*; *Mitropoulos et al., 2011*) is a mechanism that prevents SQL and XPath injection attacks against web applications by using location-specific signatures. The signatures are generated during a learning phase, and are based on elements that can depend either on the query or on its execution environment (for example the stack trace). Then, during production, the mechanism checks all queries for compliance and can block queries containing injected elements. By associating a stack trace with the origin of a query, the mechanism can correlate SQL statements with their call sites. This increases the specificity of the stored signatures and avoids false alarms. *nSign* (*Mitropoulos et al., 2016*) and SICILIAN (*Soni, Budianto & Saxena, 2015*) follow the same approach as SDriver to prevent XSS attacks on the client-side. To do so, nSign includes script origins and the type of a script as environment elements, and JavaScript keywords and their number of appearance as elements coming from the code that is about to be executed. SICILIAN on the other side, includes more elements from the script (class names, variavle names and more) and less from the environment.

Laranjeiro et al. (*Laranjeiro, Vieira & Madeira, 2009*; *Antunes et al., 2009*; *Laranjeiro, Vieira & Madeira, 2010*) have proposed a similar mechanism to detect both SQL and

Xpath injection attacks in Web services. When it is not possible to run a complete learning phase, a set of heuristics is used by the mechanism to accept or discard doubtful cases. Finally, Mattos, Santin & Malucelli (2013) developed an signature-based attack detection engine that utilizes ontologies to counter XML and Xpath injection attacks. By using ontologies to model data provides explicit and formal semantic relationships between data and possible attacks.

ANALYSIS AND DISCUSSION

We have analyzed the mechanisms described earlier based on the requirements mentioned in 'Introduction and Covered Area'. Tables 1 and 2 illustrate the comparison summaries of the static and dynamic countermeasures.

Flexibility

Flexibility indicates if an approach can be adjusted in order to detect different attacks categories. Typically, all approaches, except for lexical analysis have been used to detect various code defects. As we described earlier, lexical analysis is a simplistic approach that cannot be used to identify source code-driven injection attacks. Even if a corresponding tool existed, the false alarms would be far too many. This is because source code-driven injection attacks are language independent (see 'Introduction and Covered Area') and lexical analysis can only search for specific keywords or sequences of keywords. As a result, it is only used to detect code constructs that can lead to binary code injection attacks.

In all other cases, the approaches are flexible and they can be used to deal with different kinds of attacks. For instance, policy enforcement is a method that seems to be tailored to prevent JavaScript injection attacks since it involves the interaction of two entities: the client's browser and the server-side application (policies are set to the browser and are enforced on the client-side). Notably, it can be also successfully employed to detect binary code injection via CFI mechanisms.

Effectiveness tests

The effectiveness of security mechanisms can be judged by the existence of incorrect data,[1] namely: false positives (FP) and false negatives (FN). Specifically, a FP is a result that indicates that an attack is taking place, when it has not. A FN occurs when an attack actually takes place, and the mechanism fails to detect it. In Tables 1 and 2, we show if the researchers have performed any tests to evaluate the effectiveness of their proposed mechanisms in terms of FPs and FNs. If this is the case we put a tick mark (✔). If no tests were performed we put an X mark (✗). We see that there are many cases where no such tests were performed: 4 out of 17 in the case of static analysis and 12 our of 41 in the case of dynamic detection. A reasonable argument for the latter case, would be that defenses like JSFlow (Hedin et al., 2014), COWL (Stefan et al., 2014) do not need to be fully validated through testing, because they provide systematic arguments as to why their design is secure. In order for this to stand though, their implementation should closely follow its specification, which may not be the case in practical terms.

[1] Also known in statistics as *type I* and *type Ii* errors (*Peck & Devore, 2010*).

Going one step further, we observed that there were cases such as XSS-GUARD (*Bisht & Venkatakrishnan, 2008*) and the system by *Valeur, Mutz & Vigna (2005)*, where researchers

Table 1 Static Analysis: Comparison summary of tools designed to detect vulnerabilities that can lead to a code injection attack.

Approach	Flexibility[a]	Mechanism	Requirements			Attack vector
			Effectiveness tests[b]	Implementation independence[c]	Computational overhead[d]	
Lexical analysis	✗	ITS4 (Viega et al., 2002; Viega et al., 2000)	✔	✗(C)	¬	binary code
		PScan (Heffley & Meunier, 2004)	✔	✗(C)	¬	binary code
		Flawfinder (Wilander & Kamkar, 2002)	✔	✗(C)	¬	binary code
		RATS (Chess & McGraw, 2004)	✔	✔	¬	binary code
		BOON (Wagner et al., 2000)	✔	✗(C)	¬	binary code
		CodeSurfer (Anderson & Zarins, 2005; Nagy & Mancoridis, 2009)	✔	✔	¬	binary code
		Splint (Evans & Larochelle, 2002)	✔	✗(C)	¬	binary code
Data-Flow Analysis	✔	Livshits & Lam (2005)	✔	✗(Java)	¬	SQL, JavaScript
		FindBugs (Ayewah & Pugh, 2010; Hovemeyer & Pugh, 2007; Spacco, Hovemeyer & Pugh, 2006)	✔	✗(Java)	¬	SQL, JavaScript
		Pixy (Jovanovic, Kruegel & Kirda, 2006)	✔	✔	¬	SQL, JavaScript
		XSS*detect*	✔	✔	¬	JavaScript
		Dahse & Holz (2014)	✔	✔	¬	SQL, JavaScript
Model Checking	✔	QED (Martin & Lam, 2008)	✔	✗(Java)	¬	SQL, JavaScript
		Fehnker, Huuck & Rödiger (2011)	✔	✗(C)	¬	binary code
Symbolic Execution	✔	SAFELI (Fu & Qian, 2008)	✔	✔	¬	SQL
		KLEE (Cadar, Dunbar & Engler, 2008)	✔	✗(C)	¬	binary code
		Kudzu [139]	✔	✔	¬	JavaScript
		Rubyx (Chaudhuri & Foster, 2010)	✔	✗(Ruby)	¬	JavaScript
		MergePoint (Avgerinos et al., 2014)	✗	✗(C)	¬	binary code
		S3 (Trinh, Chu & Jaffar, 2014)	✔	✔	¬	SQL, JavaScript
Type system extensions	✔	SQL DOM (McClure & Krüger, 2005)	✗	✔	20%	SQL
		Safe Query Objects (Cook & Rai, 2005)	✗	✗(Java)	?	SQL
		SQLJ (Eisenberg & Melton, 1999)	✔	✗(Java)	?	SQL
		SugarJ (Erdweg, 2013; Erdweg et al., 2011)	✗	✔	?	SQL, XML
		Wassermann & Su (2007)	✔	✔	✗	SQL
		WebSSARI (Xie & Aiken, 2006)	✔	✗(PHP)	98.4%	SQL

Notes.

[a] *Flexibility* indicates if the approach can be adjusted in order to detect different categories.

[b] *Effectiveness Tests*. This column shows if the researchers performed any tests regarding the effectiveness of their mechanism in terms of false positive and negative results.

[c] *Implementation Independence* indicates if the static analysis mechanism is tailored to a specific programming language.

[d] *Computational Overhead*. This column shows the runtime overhead that the mechanism may add to the application. In the context of static analysis this can be measured in the case of the type system extension approach. Note that 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.

Table 2 **Dynamic Detection: Comparison summary of mechanisms developed to counter code injection attacks.**

Approach	Flexibility[a]	Mechanism	Requirements			Attack vector
			Effectiveness tests[b]	Implementation independence[c]	Computational overhead[d]	
Runtime tainting	✔	*SigFree* (*Wang et al., 2010*)	✔	✗(C)	10%	binary code
		LIFT (*Qin et al., 2006*)	✔	✔	6.2%	binary code
		Haldar, Chandra & Franz (2005)	✗	✗(Java)	✗	SQL
		SecuriFly (*Martin, Livshits & Lam, 2005*)	✔	✗(Java)	9–125%	SQL, JavaScript
		Xu, Bhatkar & Sekar (2006)	✔	✔	76%	SQL, JavaScript
		WASC (*Nanda, Lam & Chiueh, 2007*)	✔	✔	30%	JavaScript
		PHP*Aspis* (*Papagiannis, Migliavacca & Pietzuch, 2011*)	✔	✗(PHP)	2.2 ×	SQL, JavaScript, PHP
		Stock et al. (2014)	✔	✔		JavaScript
		Vogt et al. (2007)	✔	✔	?	JavaScript
		JS*Flow* (*Hedin et al., 2014*)	?	✔	2 ×	JavaScript
		COWL (*Stefan et al., 2014*)	✗	✔	16%	SQL, JavaScript
		Bauer et al. (2015)	?	✔	55%	JavaScript
ISR	✔	SQL*rand* (*Boyd & Keromytis, 2004*)	✗	✔	6.5*ms*	SQL
		S*Mask* (*Johns & Beyerlein, 2007*)	✔	✔	?	SQL, JavaScript
		Noncespaces (*Gundy & Chen, 2009*)	✔	✔	10.3%	JavaScript
		*x*JS (*Athanasopoulos et al., 2010*)	✔	✔	1.6–40*ms*	JavaScript
Policy enforcement	✔	DSI (*Nadji, Saxena & Song, 2006*)	✔	✔	1.85%	JavaScript
		BrowserShield (*Reis et al., 2006*)	✔	✔	8%	JavaScript
		Blueprint (*Louw & Venkatakrishnan, 2009*)	✔	✔	13.6%	JavaScript
		CoreScript (*Yu et al., 2007*)	✗	✔	?	JavaScript
		MET (*Erlingsson, Livshits & Xie, 2007*)	✗	✔	?	JavaScript
		BEEP (*Jim, Swamy & Hicks, 2007*)	✔	✔	14.4%	JavaScript
		CSP (*Stamm, Sterne & Markham, 2010*)	✗	✔	?	JavaScript
		Google Caja	✗	✔	?	JavaScript
		Kiriansky, Bruening & Amarasinghe, (2002)	?	✗	∼1%	binary code
		CFI (*Abadi et al., 2005*; *Van der Veen et al., 2015*)	✗	✗(C)	0.09–26.78%	binary code
		CPI (*Kuznetsov et al., 2014*)	✔	✗(C)	2.9–8.4%	binary code
		CCFI (*Mashtizadeh et al., 2015*)	✔	✗(C)	3–18%	binary code

(*continued on next page*)

Table 2 (*continued*)

Approach	Flexibility[a]	Mechanism	Requirements			Attack vector
			Effectiveness tests[b]	Implementation independence[c]	Computational overhead[d]	
Whitelisting	✔	AMNESIA (*Halfond & Orso, 2005b*; *Halfond & Orso, 2006*; *Halfond & Orso, 2005a*)	✔	✔	?	SQL
		DIDAFIT (*Lee, Low & Wong, 2002*)	✔	✔	?	SQL
		(*Valeur, Mutz & Vigna (2005)*	✔	✔	1*ms*	SQL
		SQLGuard (*Buehrer, Weide & Sivilotti, 2005*)	✗	✔	3%	SQL
		Diglossia (*Son, McKinley & Shmatikov, 2013*)	✔	✔	13%	SQL, NoSQL
		SD*river* (*Mitropoulos & Spinellis, 2009*; *Mitropoulos, Karakoidas & Spinellis, 2009*; *Mitropoulos et al., 2011*)	✔	✔	39%	SQL, Xpath
		nSign (*Mitropoulos et al., 2016*)	✔	✔	11.1%	JavaScript
		SICILIAN (*Soni, Budianto & Saxena, 2015*)	✔	✔	7.02%	JavaScript
		Laranjeiro et al. (*Laranjeiro, Vieira & Madeira, 2009*; *Antunes et al., 2009*; *Laranjeiro, Vieira & Madeira, 2010*)	✔	✔	✗	SQL, Xpath
		Mattos, Santin & Malucelli (2013)	✔	✔	?	XML, Xpath
		SWAP (*Wurzinger et al., 2009*)	✔	✔	~180%	JavaScript
		XSSDS (*Johns, Engelmann & Posegga, 2008*)	✔	✔	?	JavaScript
		XSS-GUARD (*Bisht & Venkatakrishnan, 2008*)	✔	✔	5–24%	JavaScript

Notes.
[a] *Flexibility* indicates if the approach can be adjusted in order to detect different categories.
[b] *Effectiveness Tests.* This column shows if the researchers performed any tests regarding the effectiveness of their mechanism in terms of false positive and negative results.
[c] *Implementation Independence* shows if the mechanism depends either on the characteristics of the programming language that was used to develop it or on the implementation details of the protecting entity.
[d] *Computational Overhead.* This column shows the runtime overhead that the mechanism may add to the application. Note that the different results do 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.

performed tests to measure false alarms but they did not look for false negatives. Notably, we observed that there are mechanisms that even if they seem effective, their testing might be really poor contrary to other schemes that may have false alarms, but have been tested thoroughly. For example, Blueprint appears to be an effective solution to detect JavaScript injection attacks, but the corresponding publication includes only two test cases. On the other hand, Dsi appears to have false positives and negatives, but it was evaluated on a set of 5,328 vulnerable web sites.

An interesting observation involves the mechanisms that detect JavaScript injection attacks. Unfortunately, most countermeasures, even if the corresponding publications state that they are accurate, are actually vulnerable to attacks that involve non-HTML elements (except for *Athanasopoulos et al., 2010*). For instance, there are browsers that treat PostScript files as HTML. A malicious user can embed a script within a PostScript file, upload it as a valid document and then use it to trigger the attack (*Barth, Caballero & Song, 2009*) (also see the Appendix). Since most mechanisms require the presence of a Document Object Model Dom tree to detect an attack, in this case they will fail.

Notably, there are cases where the effectiveness of some mechanisms have been questioned. For example, *Sovarel, Evans & Paul (2005)* have examined the effectiveness of ISR and showed that an attacker may be able to circumvent the approach by determining the randomization key. Furthermore, their results indicate that doing ISR in a way that provides a certain degree of security against a motivated malicious user is not as easy as previously thought. In the same manner, *Zitser, Lippmann & Leek (2004)* and *Wilander & Kamkar (2002)*, have extensively tested and questioned some of the aforementioned tools that detect binary code injection attacks.

Implementation independence

In the case of static analysis mechanisms, implementation independence indicates if a mechanism is developed based upon a specific programming language. For instance, all lexical analysis tools except for Rats, only analyze applications written in the C programming language. Still, Rats, which can be used on other languages, does not find code injection vulnerabilities in any other language except for C. In the same manner, SQLJ can only be used by Java developers. In every case, we list the corresponding language.

In the dynamic detection context, implementation independence shows if the mechanism depends either on the characteristics of the programming language that was used to develop it or on the implementation details of the protecting entity. For instance, PHP Aspis can detect various forms of CIAs that target applications written in PHP only. In the same manner, CFI mechanisms can only protect programs written in C.

Computational overhead

The user's experience is affected if a mechanism suffers from runtime overhead. Take for example a mechanism from the dynamic detection category. If this mechanism adds significant overhead to the applications functionality, the application's owner would consider it useless. In the static analysis context, this can be measured in the case of the type system extension approach since their use affects the application overall. In the table we list the overhead for every mechanism as stated in the original publication. If the publication mentions that the mechanism suffers from a runtime overheard but does not explicitly state the occurring overhead we use the X mark (✗). If the authors did not measure the overhead we use a question mark (?). Note though that each number is an indication that has been computed under different assumptions and settings and it cannot be used to compare mechanisms directly (especially the ones coming from different categories). However, in cases like nSign and SICILIAN, this could be meaningful because both are mechanisms that wrap up the JavaScript engine of a browser. Hence, both overheads are imposed on the execution time of the engine.

Note that the overhead is displayed in different manners (e.g., percentages, absolute numbers and more). In every case this indicates the cost due to the use of each mechanism.

For example, it may be due to some form of run-time checks. Furthermore, depending on the approach, the cost may be incurred on different places: it may affect a server (e.g., CPU usage, response latency and more), it may affect the client, or both.

A note on usability

The value of a security mechanism as a practical tool depends on how easy it is to deploy it in a production setting. In the static analysis context, a mechanism should require minimum effort from the security auditor. Observe that lexical analysis and data-flow analysis mechanisms are easy to use since the only thing that is needed to perform their analysis is the source code. Note though that, based on simple assumptions and without considering context in any way lexical analysis could report every possible dangerous function call as a problem, no matter how carefully it is used in the program. Hence, auditors must be experienced programmers in order to interpret the results of lexical analysis tools and they must regard them as as an aid in the code review process and not as a firm solution to find software vulnerabilities (*Cowan, 2003*; *Zitser, Group & Leek, 2004*). In addition, model checking and type system extensions require too much effort from the side of the auditor, either to write specifications, modify source code or learn new constructs (see 'Model checking' and 'Type system extensions').

In the case of dynamic detection, usability involves the deployment of the mechanism. To determine the effort required to use the mechanism, we examined the mechanism's description, its deployment, and its implementation details. One of our basic criteria was if developers are required to modify their code and if they do, to what extent. As an example, consider the mechanisms coming from the policy enforcement category. In most cases programmers should modify multiple software components to enable a mechanism. Note also that mechanisms such as MET and BEEP require modifications both on the server and the client-side. Thus, it would not be easy for them to be adopted by browser vendors. In the same manner SQLrand imposes a major infrastructure overhead because it requires the integration of a proxy for the RDBMS to detect SQL injection attacks. In addition, the whitelisting mechanisms that detect DSL-driven injection attacks, require multiple source code modifications. In particular, to use AMNESIA, developers should modify every code fragment that involves the execution of a query. Nevertheless, there are tools like Sdriver, which minimize such modifications down to one line of code.

EMERGING CHALLENGES

There are several challenges which indicate that code injection attacks will continue to be an issue in the field of cyber security.

First, attackers seem to find new ways to introduce malicious code into programs by using a variety of techniques. For instance, an attack called PHP Object Injection (POI) (*Dahse, Krein & Holz, 2014*) 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 (for example as part of cookies). When deserialized by the application, these objects result in arbitrary code execution. In a similar way, XCS (Cross-channel scripting) (*Bojinov, Bursztein & Boneh, 2009*; *Bojinov, Bursztein & Boneh,* *2010*) attacks are a prominent XSS variation. In an XCS attack, an attacker utilizes a non-web channel to inject code. For example, there are several NAS (Network-Attached Storage) devices which allow unauthorized users to upload files via the SMB protocol (Server Message Block). A malicious user could upload a file with a filename that contains a well-crafted script. When a legitimate user connects over a web channel to the device to browse its contents, the device will send through an HTTP response the list of all filenames, ncluding the malicious one which is going to be interpreted as a valid script by the browser.

Architectures that include modern technologies such as MongoDB (http://www.mongodb.org/) could be vulnerable to complex attacks that may involve more than one subcategories as *Son, McKinley & Shmatikov (2013)* have pointed out. In particular, a JavaScript injection attack could be performed to change an SQL-like MongoDB query that is built dynamically based on user input. Specifically, when using JavaScript, developers have to make sure that any variables that cross the PHP-to-JavaScript boundary are passed in the scope field of the `MongoCode` class, (http://www.php.net/manual/en/class.mongocode.php) that is not interpolated into the JavaScript string. This can come up when using the `MongoDB::execute()` method and clauses like `$where` and `group-by`. For example, suppose that JavaScript is used to greet a user in the database logs:

```
<?php
$username = $_POST['username'];
$db->execute("print('Hello, $username!');");
?>
```

If attackers pass `'); db.users.drop(); print('` as a username, they could actually delete the entire database.

Recent work indicates that code injection can also be used as an attack vector to exploit mobile applications (*Bao et al., 2017*; *Jin et al., 2014*). This is not surprising because even though there are slightly different components that interact in the context of mobile applications, programming vulnerabilities thay may lead to code injection can still show up. Specifically, as *Jin et al. (2014)* point out, vulnerable HTML5-based mobile applications can be vulnerable to XSS variations. Such attacks could involve different channels to send malicous scripts to the user's browser including 2D barcodes and Wi-Fi access points.

CONCLUSIONS

Code injection attacks can be divided into two classes: those that target binary executable code and those that target the source code of domain specific and dynamic languages. Approaches that defend against source code injection attacks can be grouped into two major categories: static analysis mechanisms that detect code injection vulnerabilities, and dynamic detection approaches that prevent code injection attacks the moment they take place. Tools coming from the static analysis category are mainly used during application development, while dynamic detection mechanisms are employed during production.

We examined the defenses based on their flexibility, implementation independence, computational overhead, and effectiveness tests.

We observed that researchers do not extensively test their mechanisms in terms of effectiveness. A reasonable explanation for this would be that some defenses do not need to be fully validated through testing, because researchers provide formal arguments as to why they are secure. However, this is true only if implementations closely follow the specification, which may not be the case in practical terms. Notably, there are cases where the effectiveness of some mechanisms has been questioned (*Sovarel, Evans & Paul, 2005*; *Zitser, Lippmann & Leek, 2004*; *Wilander & Kamkar, 2002*).

Moreover, we saw that computational overheads are mostly computed and reported under different assumptions and settings, hence they cannot be used to compare mechanisms directly. Overheads also depend on context: in interactive applications latency is important, whereas in a batch setting the important measure is throughput or the corresponding slowdown. By taking account the above it would be fair to compare the SICILIAN to nSign in terms of computational overhead, because they both wrap up the JavaScript engine of a browser to defend against XSS attacks. Nevertheless, it would be spurious to compare both of them to a mechanism that acts as a proxy on the server side to defend against the same threat (e.g., BrowserShield).

We also found that most approaches are flexible, meaning that they can be used to counter different forms of code injection. For example, ISR and whitelisting have been applied to counter all kinds of code injection attacks. This is not the case though with lexical analysis, which is used only to detect binary code injection vulnerabilities.

Approaches can be interdependent and they can borrow heavily from others. Consider for instance runtime tainting and data-flow analysis. Both examine the flow of data but in different ways: the former does so dynamically and the latter statically. For this reason though, methods can also share the same disadvantages. For example, the state explosion issue appears in both model checking and symbolic execution.

Currently, most defenses target a small number of attacks, but this will probably change in the future. Specifically, a large amount of work has been done to prevent either SQL and JavaScript-driven injection attacks. This makes sense, because these attacks are very common and can have a large impact. Less effort has been put to develop approaches that can defend against XPath or XML injection attacks. As a result, there are few corresponding defenses. Similarly, there is only one documented mechanism designed to detect PHP injection attacks, PHP Aspis, and only one that prevents NoSQL injection attacks, Diglossia.

Attacks and defenses are likely to evolve in the coming years. The driver will be new threats, such as the PHP Object Injection (POI) (*Dahse, Krein & Holz, 2014*) attack, and Cross-Channel Scripting (XCS), both discussed in 'Emerging Challenges' Apart from the above, attackers seem to continuously find new ways to introduce malicious code to applications by using a variety of languages and techniques as we observed earlier (see 'Code Injection Attacks' and 'Emerging Challenges'. Besides, there are several recent attempts to perform code injection on mobile applications (*Bao et al., 2017*; *Jin et al., 2014*) which will potentially lead to the development of context-aware defenses. We hope that our categorization, our analysis, and the findings of our research, will aid researchers and practitioners to further study the code injection problem and develop more robust and effective defenses.

ACKNOWLEDGEMENTS

We want to thank the reviewers for providing us with valuable suggestions and insightful comments.

APPENDIX. JAVASCRIPT INJECTION ATTACKS

A JavaScript injection vulnerability is manifested when a web application accepts and redisplays data of uncertain origin without appropriate validation and filtering. Such content can compromise the security of these applications and the privacy of their corresponding users. Many web sites allow registered users to post data which are stored on the server-side (i.e., a third-party comment on a blog page). If attackers hide a script in such data, they could manipulate the browser of another user. For example consider the following code snippet:

```
<script type="text/javascript">
document.location='http://host.example/cgi-
bin/cookiestealing.cgi?'+document.cookie
</script>
```

If a malicious user could post data containing the above script, web users visiting the page that contains this data could have their cookies stolen. Through this script the attacker calls an external Common Gateway Interface (CGI) script and passes all the cookies associated with the current document to it as an argument via the `document.cookie` property.

A common but rough way to stop malicious behaviors like this is server-side code filtering (i.e., the server strips out the word "javascript" from any external source) (*Jim, Swamy & Hicks, 2007*). Still, there are many ways to bypass such defense mechanisms. For example, one could escape special characters to bypass simple filtering operations, or take advantage of issues in the implementation of Cascading Style Sheets (Css) rendering engines of browsers like Microsoft Internet Explorer (versions prior to 7).

Consider the case where an attacker manages to hide the following listing in the Css of a web page:

```
<div id=code style="background:url('java
script:eval(document.all.code.foo)')"
foo="alert('xss')"></div>
```

The attacker utilizes the `eval` function and a newline character ("java–*newline*–script") to bypass the security checks and manoeuvre browser to execute the code contained in the `foo` variable. This is done by using the `document.all` array that contains all of the elements within a document.

Attacks like the above take advantage of the fact that `eval` executes the code passed to it in the same environment as the function's caller. Malicious users can also use `eval` to assemble innocuous-looking parts into harmful strings that the protecting mechanisms of a web page would normally consider dangerous and remove (*Richards et al., 2011*). Furthermore, a JavaScript injection attack does not necessarily have to involve HTML elements. A malicious user can embed a script within a PostScript file, upload it as a valid document and then use it to trigger the attack (*Barth, Caballero & Song, 2009*).

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

This work was funded under Action 2 of the Athens University of Economics and Business Research Center Program for Excellence and Extroversion of the academic year 2016/2017 (EP-2606-01: The "Meta-Life" of JavaScript). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Grant Disclosures

The following grant information was disclosed by the authors:
Athens University of Economics and Business Research Center Program: EP-2606-01.

Competing Interests

The authors declare there are no competing interests.

Author Contributions

- Dimitris Mitropoulos analyzed the data, wrote the paper, prepared figures and/or tables, reviewed drafts of the paper.
- Diomidis Spinellis wrote the paper, reviewed drafts of the paper.

Data Availability

The following information was supplied regarding data availability:

The research in this article did not generate, collect or analyse any raw data or code.

REFERENCES

Abadi M, Budiu M, Erlingsson U, Ligatti J. 2005. Control-flow Integrity. In: Jaeger T, ed. *Proceedings of the 12th ACM conference on computer and communications security, CCS'05*. New York: ACM, 340–353.

Abelson H, Sussman GJ. 1996. Structure and interpretation of computer programs. Second Edition. Cambridge: MIT Press.

Abi-Antoun M, Wang D, Torr P. 2007. Checking threat modeling data flow diagrams for implementation conformance and security. In: Stirewalt K, ed. *Proceedings of the 22nd IEEE/ACM international conference on automated software engineering, ASE'07*. New York: ACM, 393–396.

Aho AV, Lam MS, Sethi R, Ullman JD. 2006. *Compilers: principles, techniques, and tools*. Second Edition. Harlow, Essex: Addison-Wesley Longman Publishing Co., Inc.

Anderson A. 2005. A comparison of two privacy policy languages: EPAL and XACML. Technical report. Sun Microsystems, Inc., Mountain View, CA, USA.

Anderson P, Zarins M. 2005. The codesurfer software understanding platform. In: Cordy JR, Gall H, Maletic JI, eds. *Proceedings of the 13th international workshop on program comprehension, IWPC'05*. Washington, D.C.: IEEE Computer Society, 147–148.

Anderson RJ. 2001. *Security engineering: a guide to building dependable distributed systems*. First Edition. New York: John Wiley & Sons, Inc.

Antunes N, Laranjeiro N, Vieira M, Madeira H. 2009. Effective detection of SQL/XPath injection vulnerabilities in web services. In: Sarkar S, Vin HM, Zhao JL, eds. *Proceedings of the 2009 IEEE international conference on services computing, SCC'09*. Washington, D.C.: IEEE Computer Society, 260–267.

Athanasopoulos E, Pappas V, Krithinakis A, Ligouras S, Markatos EP, Karagiannis T. 2010. xJs: practical XSS prevention for web application development. In: Ousterhout J, ed. *Proceedings of the 2010 USENIX conference on Web application development, WebApps'10*. Berkeley: USENIX Association, 13–13.

Avancini A, Ceccato M. 2013. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. *Information and Software Technology* **55(12)**:2209–2222 DOI 10.1016/j.infsof.2013.08.001.

Avgerinos T, Rebert A, Cha SK, Brumley D. 2014. Enhancing symbolic execution with veritesting. In: Jalote P, ed. *Proceedings of the 36th international conference on software engineering, ICSE 2014*. New York: ACM, 1083–1094.

Ayewah N, Pugh W. 2010. The Google FindBugs fixit. In: Tonella P, ed. *Proceedings of the 19th international symposium on Software testing and analysis, ISSTA'10*. New York: ACM, 241–252.

Baca D, Carlsson B, Lundberg L. 2008. Evaluating the cost reduction of static code analysis for software security. In: Erlingsson Ú, Pistoia M, eds. *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security, PLAS'08*. New York: ACM, 79–88.

Bao W, Yao W, Zong M, Wang D. 2017. Cross-site Scripting attacks on android hybrid applications. In: *Proceedings of the 2017 international conference on cryptography, security and privacy, ICCSP'17*. New York: ACM, 56–61.

Barth A, Caballero J, Song D. 2009. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In: Sterritt R, ed. *Proceedings of the 30th IEEE symposium on security and privacy*. Washington, D.C.: IEEE Computer Society, 360–371.

Bauer L, Cai S, Jia L, Timothy P, Michael S, Yuan T. 2015. Run-time monitoring and formal analysis of information flows in Chromium. In: Tsudik G, Perrig A, eds. *Network and distributed system security (NDSS)'15, 8–11 February 2015, San Diego, CA, USA*. Reston: Internet Society.

Beyer D, Henzinger TA, Jhala R, Majumdar R. 2007. The software model checker blast: applications to software engineering. *International Journal on Software Tools for Technology Transfer* **9(5)**:505–525 DOI 10.1007/s10009-007-0044-z.

Bisht P, Venkatakrishnan VN. 2008. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In: Zamboni D, ed. *Proceedings of the 5th international conference on detection of intrusions and malware, and vulnerability assessment, DIMVA'08*. Berlin: Springer-Verlag, 23–43.

Bojinov H, Bursztein E, Boneh D. 2009. XCS: cross channel scripting and its impact on web applications. In: Al-Shaer E, ed. *Proceedings of the 16th ACM conference on computer and communications security*. New York: ACM, 420–431.

Bojinov H, Bursztein E, Boneh D. 2010. The emergence of cross channel scripting. *Communications of the ACM* **53(8)**:105–113 DOI 10.1145/1787234.1787257.

Boujarwah AS, Saleh K, Al-Dallal J. 2000. Testing Java programs using dynamic data flow analysis. In: Carroll J, Daminani E, Haddad H, Oppenheim D, eds. *Proceedings of the 2000 ACM symposium on applied computing—volume 2, SAC'00*. New York: ACM, 725–727.

Boyd S, Keromytis A. 2004. SQLrand: preventing SQL injection attacks. In: Jakobsson M, Yung M, Zhou J, eds. *Proceedings of the 2nd applied cryptography and network security conference, ACNS'04*. Springer-Verlag, 292–304.

Bratus S, Locasto ME, Patterson LSML, Shubina A. 2011. Exploit programming: from buffer overflows to "Weird Machines" and theory of computation. *j-LOGIN* **36(6)**:13–21.

Brown M, Paller A. 2008. Secure software development: why the development world awoke to the challenge. *Information Security Technical Report* **13(1)**:40–43 DOI 10.1016/j.istr.2008.03.001.

Buchanan E, Roemer R, Shacham H, Savage S. 2008. When good instructions go bad: generalizing return-oriented programming to RISC. In: Ning P, ed. *Proceedings of the 15th ACM conference on computer and communications security, CCS'08*. New York: ACM, 27–38.

Buehrer G, Weide BW, Sivilotti PAG. 2005. Using parse tree validation to prevent SQL injection attacks. In: Di Nitto E, Murphy AL, eds. *Proceedings of the 5th international workshop on software engineering and middleware, SEM'05*. New York: ACM, 106–113.

Cadar C, Dunbar D, Engler D. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves R, Van Renesse R, eds. *Proceedings of the 8th USENIX conference on operating systems design and implementation, OSDI'08*. Berkeley: USENIX Association, 209–224.

Cadar C, Godefroid P, Khurshid S, Păsăreanu CS, Sen K, Tillmann N, Visser W. 2011. Symbolic execution for software testing in practice: preliminary assessment. In: Taylor RN, ed. *Proceedings of the 33rd international conference on software engineering, ICSE'11*. New York: ACM, 1066–1071.

Cahoon B, McKinley KS. 2001. Data flow analysis for software prefetching linked data structures in java. In: Valero M, ed. *Proceedings of the 2001 international conference on parallel architectures and compilation techniques, PACT'01*. Washington, D.C.: IEEE Computer Society, 280–291.

Cannings R, Dwivedi H, Lackey Z. 2007. *Hacking exposed web 2.0: web 2.0 security secrets and solutions*. New York: McGraw-Hill Osborne Media.

CERT. 2002. CERT vulnerability note VU282403 Online. *Available at http://www.kb.cert.org/vuls/id/282403* (accessed on 7 January 2007).

Chaudhuri A, Foster JS. 2010. Symbolic security analysis of ruby-on-rails web applications. In: Al-Shaer E, ed. *Proceedings of the 17th ACM conference on computer and communications security, CCS'10*. New York: ACM, 585–594 DOI 10.1145/1866307.1866373.

Chen H, Wagner D. 2002. MOPS: an infrastructure for examining security properties of software. In: Atluri V, ed. *Proceedings of the 9th ACM conference on computer and communications security, CCS'02*. New York: ACM, 235–244 DOI 10.1145/586110.586142.

Chen K, Wagner D. 2007. Large-scale analysis of format string vulnerabilities in debian linux. In: Hicks M, ed. *Proceedings of the 2007 workshop on programming languages and analysis for security, PLAS'07*. New York: ACM, 75–84 DOI 10.1145/1255329.1255344.

Chess B, McGraw G. 2004. Static analysis for security. *IEEE Security and Privacy* **2(6)**:76–79 DOI 10.1109/MSP.2004.111.

Chess B, West J. 2007. *Secure programming with static analysis*. Upper Saddle River: Addison-Wesley Professional.

Chlipala A. 2010. Static checking of dynamically-varying security policies in database-backed applications. In: Arpaci-Dusseau R, Chen B, eds. *Proceedings of the 9th USENIX conference on operating systems design and implementation, OSDI'10*. Berkeley: USENIX Association, 1.

Chodorow K, Dirolf M. 2010. *MongoDB: the definitive guide*. Sebastopol: O'Reilly Media.

Clarke EM, Emerson EA, Sifakis J. 2009. Model checking: algorithmic verification and debugging. *Communications of the ACM* **52(11)**:74–84 DOI 10.1145/1592761.1592781.

Cook WR, Rai S. 2005. Safe query objects: statically typed objects as remotely executable queries. In: Roman G-C, ed. *Proceedings of the 27th international conference on software engineering, ICSE'05*. New York: ACM, 97–106 DOI 10.1109/ICSE.2005.1553552.

Corin R, Manzano FA. 2012. Taint analysis of security code in the KLEE symbolic execution engine. In: Chim TW, Yuen TH, eds. *Proceedings of the 14th international conference on information and communications security, ICICS'12*. Berlin: Springer-Verlag, 264–275 DOI 10.1007/978-3-642-34129-8_23.

Cowan C. 2003. Software security for open-source systems. *IEEE Security and Privacy* **1(1)**:38–45 DOI 10.1109/MSECP.2003.1176994.

Cowan C, Pu C, Maier D, Hintony H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q. 1998. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: *Proceedings of the 7th USENIX security symposium*. Berkeley: USENIX Association, 5–5.

Dahse J, Holz T. 2014. Static detection of second-order vulnerabilities in web applications. In: Fu K, ed. *Proceedings of the 23rd USENIX security symposium*. Berkeley: USENIX Association, 989–1003.

Dahse J, Krein N, Holz T. 2014. Code reuse attacks in PHP: automated POP chain generation. In: Ahn G-J, ed. *Proceedings of the 21st ACM conference on computer and communications security*. 42–53 DOI 10.1145/2660267.2660363.

Das SK, Kant K, Zhang N. 2012. *Handbook on securing cyber-physical critical infrastructure*. First Edition. San Francisco: Morgan Kaufmann Publishers Inc.

De Groef W, Devriese D, Nikiforakis N, Piessens F. 2012. FlowFox: a web browser with flexible and precise information flow control. In: Yu T, ed. *Proceedings of the 2012 ACM conference on computer and communications security, CCS'12*. New York: ACM, 748–759 DOI 10.1145/2382196.2382275.

Denning DE, Denning PJ. 1977. Certification of programs for secure information flow. *Communications of the ACM* **20**(**7**):504–513 DOI 10.1145/359636.359712.

Denning DER. 1987. An intrusion detection model. *IEEE Transactions on Software Engineering* **13**(**2**):222–232 DOI 10.1109/TSE.1987.232894.

Dhawan M, Ganapathy V. 2009. Analyzing information flow in JavaScript-based browser extensions. In: *Proceedings of the 2009 annual computer security applications conference, ACSAC'09*. Washington, D.C.: IEEE Computer Society, 382–391 DOI 10.1109/ACSAC.2009.43.

Doupé A, Cui W, Jakubowski MH, Peinado M, Kruegel C, Vigna G. 2013. deDacota: toward preventing server-side XSS via automatic code and data separation. In: *Proceedings of the 2013 ACM conference on computer and communications security, CCS'13*. New York: ACM, 1205–1216.

Dybvig RK. 2009. *The Scheme programming language*. Fourth Edition. Cambridge: MIT Press.

Egele M, Wurzinger P, Kruegel C, Kirda E. 2009. Defending Browsers against drive-by downloads: mitigating heap-spraying code injection attacks. In: Flegel U, Bruschi D, eds. *Proceedings of the 6th international conference on detection of intrusions and malware, and vulnerability assessment, DIMVA'09*. Berlin: Springer-Verlag, 88–106 DOI 10.1007/978-3-642-02918-9_6.

Eisenberg A, Melton J. 1999. SQLJ Part 1: SQL routines using the Java programming language. *Newsletter, ACM SIGMOD Record* **28**(**4**):58–63 DOI 10.1145/344816.344864.

Erdweg S. 2013. Extensible languages for flexible and principled domain abstraction. PhD thesis, Philipps-Universitat Marburg.

Erdweg S, Kats LCL, Rendel T, Kästner C, Ostermann K, Visser E. 2011. Library-based model-driven software development with SugarJ. In: *Proceedings of the ACM international conference companion on object oriented programming systems languages and applications companion, SPLASH'11*. New York: ACM, 17–18 DOI 10.1145/2048147.2048156.

Erlingsson Ú, Livshits B, Xie Y. 2007. End-to-end web application security. In: Hunt G, ed. *Proceedings of the 11th USENIX workshop on hot topics in operating systems*. Berkeley: USENIX Association, 18:1–18:6.

Evans D, Larochelle D. 2002. Improving security using extensible lightweight static analysis. *IEEE Software* **19(1)**:42–51 DOI 10.1109/52.976940.

Fagan ME. 1999. Design and code inspections to reduce errors in program development. *IBM Systems Journal* **38(2–3)**:258–287 DOI 10.1147/sj.382.0258.

Fazzini M, Saxena P, Orso A. 2015. AutoCSP: automatically retrofitting CSP to web applications. In: Bertolino A, ed. *Proceedings of the 37th international conference on software engineering, ICSE'15*. New York: ACM.

Fehnker A, Huuck R, Rödiger W. 2011. Model checking dataflow for malicious input. In: *Proceedings of the workshop on embedded systems security, WESS'11*. New York: ACM, 4:1–4:10.

Fisher M, Ellis J, Bruce J. 2003. *JDBC API tutorial and reference*. Third Edition. Boston: Addison Wesley.

Fosdick LD, Osterweil LJ. 1976. Data flow analysis in software reliability. *ACM Computing Surveys* **8(3)**:305–330 DOI 10.1145/356674.356676.

Francillon A, Castelluccia C. 2008. Code injection attacks on harvard-architecture devices. In: Ning P, ed. *Proceedings of the 15th ACM conference on computer and communications security, CCS'08*. New York: ACM, 15–26 DOI 10.1145/1455770.1455775.

Fu X, Qian K. 2008. SAFELI: SQL injection scanner using symbolic execution. In: *Proceedings of the 2008 workshop on testing, analysis, and verification of web services and applications, TAV-WEB'08*. New York: ACM, 34–39 DOI 10.1145/1390832.1390838.

Göktas E, Athanasopoulos E, Bos H, Portokalidis G. 2014. Out of control: overcoming control-flow integrity. In: *Proceedings of the 2014 IEEE symposium on security and privacy*. Washington, D.C.: IEEE Computer Society, 575–589 DOI 10.1109/SP.2014.43.

Gregoire J, Buyens K, Win BD, Scandariato R, Joosen W. 2007. On the secure software development process: CLASP and SDL Compared. In: *Proceedings of the third international workshop on software engineering for secure systems, SESS'07*. Washington, D.C.: IEEE Computer Society, 1 DOI 10.1016/j.infsof.2008.01.010.

Gundy MV, Chen H. 2009. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: *Proceedings of the 16th annual network and distributed system security symposium (NDSS)*. San Diego.

Haldar V, Chandra D, Franz M. 2005. Dynamic taint propagation for Java. In: *Proceedings of the 21st annual computer security applications conference, ACSAC'05*. Washington, D.C.: IEEE Computer Society, 303–311 DOI 10.1109/CSAC.2005.21.

Halfond WG, Viegas J, Orso A. 2006. A classification of SQL-injection attacks and countermeasures. In: *Proceedings of the international symposium on secure software engineering*.

Halfond W. GJ, Orso A. 2005a. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In: *Proceedings of the 20th IEEE/ACM international conference on automated software engineering, ASE'05*. New York: ACM, 174–183.

Halfond WGJ, Orso A. 2005b. Combining static analysis and runtime monitoring to counter SQL-injection attacks. In: *Proceedings of the third international workshop on dynamic analysis, WODA'05*. New York: ACM Press, 1–7 DOI 10.1145/1083246.1083250.

Halfond W. GJ, Orso A. 2006. Preventing SQL injection attacks using AMNESIA. In: Osterweil LJ, ed. *Proceedings of the 28th international conference on software engineering, ICSE'06*. New York: ACM, 795–798 DOI 10.1145/1134285.1134416.

Hedin D, Birgisson A, Bello L, Sabelfeld A. 2014. JSFlow: tracking information flow in JavaScript and Its APIs. In: Cho Y, Shin SY, eds. *Proceedings of the 29th annual ACM symposium on applied computing, SAC'14*. New York: ACM, 1663–1671 DOI 10.1145/2554850.2554909.

Heffley J, Meunier P. 2004. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In: *Proceedings of the 37th annual Hawaii international conference on system sciences, HICSS'04*. Washington, D.C.: IEEE Computer Society DOI 10.1109/HICSS.2004.1265654.

Hicks B, Rueda S, Clair st.L, Jaeger T, McDaniel P. 2010. A logical specification and analysis for SELinux MLS Policy. *ACM Transactions on Information and System Security* **13(3)**:26:1–26:31 DOI 10.1145/1805874.1805982.

Holzmann GJ. 1997. The model checker SPIN. *IEEE Transactions of Software Engineering* **23(5)**:279–295 DOI 10.1109/32.588521.

Hovemeyer D, Pugh W. 2007. Finding more null pointer bugs, but not too many. In: *Proceedings of the 7th ACM workshop on program analysis for software tools and engineering, PASTE'07*. New York: ACM, 9–14 DOI 10.1145/1251535.1251537.

Howard M, LeBlanc D. 2003. *Writing secure code*. Second Edition. Redmond: Microsoft Press.

Jim T, Swamy N, Hicks M. 2007. Defeating script injection attacks with browser-enforced embedded policies. In: Williamson C, Zurko ME, eds. *Proceedings of the 16th international conference on World Wide Web, WWW'07*. New York: ACM, 601–610 DOI 10.1145/1242572.1242654.

Jin X, Hu X, Ying K, Du W, Yin H, Peri GN. 2014. Code injection attacks on HTML5-based mobile apps: characterization, detection and mitigation. In: Ahn G-J, ed. *Proceedings of the 2014 ACM conference on computer and communications security, CCS'14*. New York: ACM, 66–77 DOI 10.1145/2660267.2660275.

Johns M, Beyerlein C. 2007. SMask: preventing injection attacks in web applications by approximating automatic data/code separation. In: Cho Y, Wainwright RL, Haddad HM, eds. *Proceedings of the 2007 ACM symposium on applied computing, SAC'07*. New York: ACM, 284–291 DOI 10.1145/1244002.1244071.

Johns M, Engelmann B, Posegga J. 2008. XSSDS: server-side detection of cross-site scripting attacks. In: *Proceedings of the 2008 annual computer security applications conference, ACSAC'08*. Washington, D.C.: IEEE Computer Society, 335–344 DOI 10.1109/ACSAC.2008.36.

Johnson RT. 2006. Verifying security properties using type-qualifier inference. PhD thesis, Berkeley, CA, USA. AAI3253911.

Jovanovic N, Kruegel C, Kirda E. 2006. Pixy: a static analysis tool for detecting web application vulnerabilities (Short Paper). In: *Proceedings of the 2006 IEEE symposium on security and privacy*. Washington, D.C.: IEEE Computer Society, 258–263 DOI 10.1109/SP.2006.29.

Kantorovitz IP. 2004. Lexical analysis tool. *ACM SIGPLAN Notices* **39(5)**:66–74 DOI 10.1145/997140.997147.

Kc GS, Keromytis AD, Prevelakis V. 2003. Countering code-injection attacks with instruction-set randomization. In: Jajodia S, ed. *CCS'03: proceedings of the 10th ACM conference on computer and communications security*. New York: ACM, 272–280 DOI 10.1145/948109.948146.

Keromytis AD. 2009. Randomized instruction sets and runtime environments past research and future directions. *IEEE Security and Privacy* **7(1)**:18–25 DOI 10.1109/MSP.2009.15.

Keromytis AD. 2011. Buffer overflow attacks. In: *Encyclopedia of cryptography and security*. Second Edition. 174–177.

King JC. 1976. Symbolic execution and program testing. *Communications of the ACM* **19(7)**:385–394 DOI 10.1145/360248.360252.

Kiriansky V, Bruening D, Amarasinghe SP. 2002. Secure execution via program shepherding. In: Boneh D, ed. *Proceedings of the 11th USENIX security symposium*. Berkeley: USENIX Association, 191–206.

Kong D, Zheng Q, Chen C, Shuai J, Zhu M. 2007. ISA: a source code static vulnerability detection system based on data fusion. In: Li J, Lee W-C, Silvestri F, eds. *Proceedings of the 2nd international conference on scalable information systems, InfoScale'07*. Brussels, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 55:1–55:7.

Kuznetsov V, Szekeres L, Payer M, Candea G, Sekar R, Song D. 2014. Code-pointer integrity. In: Flinn J, Levy H, eds. *Proceedings of the 11th USENIX conference on operating systems design and implementation, OSDI'14*. Berkeley: USENIX Association, 147–163.

Laranjeiro N, Vieira M, Madeira H. 2009. Protecting database centric web services against SQL/XPath injection attacks. In: Bhowmick SS, Küng J, Wagner R, eds. *Proceedings of the 20th international conference on database and expert systems applications, DEXA'09*. Berlin: Springer-Verlag, 271–278 DOI 10.1007/978-3-642-03573-9_22.

Laranjeiro N, Vieira M, Madeira H. 2010. A Learning-based approach to secure web services from SQL/XPath injection attacks. In: *Proceedings of the 2010 IEEE 16th Pacific rim international symposium on dependable computing, PRDC'10*. Washington, D.C.: IEEE Computer Society, 191–198 DOI 10.1109/PRDC.2010.24.

Lee SY, Low WL, Wong PY. 2002. Learning fingerprints for a database intrusion detection system. In: Gollmann D, Karjoth G, Waidner M, eds. *Proceedings of the 7th European symposium on research in computer security, ESORICS'02*. London, UK: Springer-Verlag, 264–280 DOI 10.1007/3-540-45853-0_16.

Lhee K-S, Chapin SJ. 2003. Buffer overflow and format string overflow vulnerabilities. *Software: practice and experience* **33(5)**:423–460 DOI 10.1002/spe.515.

Livshits VB, Lam MS. 2005. Finding security vulnerabilities in Java applications with static analysis. In: *Proceedings of the 14th USENIX security symposium*. Berkeley: USENIX Association, 18–18.

Louw MT, Venkatakrishnan VN. 2009. Blueprint: robust prevention of cross-site scripting attacks for existing browsers. In: *Proceedings of the 2009 30th IEEE symposium on security and privacy*. Washington, D.C.: IEEE Computer Society, 331–346 DOI 10.1109/SP.2009.33.

Martin M, Lam MS. 2008. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In: Van Oorschot P, ed. *Proceedings of the 17th USENIX security symposium*. Berkeley: USENIX Association, 31–43.

Martin M, Livshits B, Lam MS. 2005. Finding application errors and security flaws using PQL: a program query language. In: Johnson R, ed. *Proceedings of the 20th ACM conference on object oriented programming, systems, languages, and applications, OOPSLA'05*. New York: ACM Press, 365–383 DOI 10.1145/1094811.1094840.

Mashtizadeh AJ, Bittau A, Boneh D, Mazières D. 2015. CCFI: cryptographically enforced control flow integrity. In: Ray I, ed. *Proceedings of the 22Nd ACM SIGSAC conference on computer and communications security, CCS'15*. New York: ACM, 941–951 DOI 10.1145/2810103.2813676.

Mattos T, Santin A, Malucelli A. 2013. Mitigating XML injection 0-day attacks through strategy-based detection systems. *IEEE Security and Privacy* **11(4)**:46–53 DOI 10.1109/MSP.2012.83.

McClure RA, Krüger IH. 2005. SQL DOM: compile time checking of dynamic SQL statements. In: Roman G-C, ed. *Proceedings of the 27th international conference on software engineering, ICSE'05*. 88–96 DOI 10.1145/1062455.1062487.

McGraw G. 2006. *Software security: building security in*. Boston: Addison-Wesley Professional.

McGraw G. 2008. Automated code review tools for security. *IEEE Computer* **41(12)**:108–111 DOI 10.1109/MC.2008.514.

McMillan KL. 1992. Symbolic model checking: an approach to the state explosion problem. PhD thesis, Carnegie Mellon University.

Mellado D, Fernández-Medina E, Piattini M. 2010. Security requirements engineering framework for software product lines. *Information and Software Technology* **52(10)**:1094–1117 DOI 10.1016/j.infsof.2010.05.007.

Merz S. 2001. Model checking: a tutorial overview. In: Cassez F, Jard C, Rozoy B, Ryan MD, eds. *Proceedings of the 4th summer school on modeling and verification of parallel processes*. London: Springer-Verlag, 3–38 DOI 10.1007/3-540-45510-8_1.

Miller A, Donaldson A, Calder M. 2006. Symmetry in temporal logic model checking. *ACM Computing Surveys* **38(3)**:432–441.

Minamide Y. 2005. Static approximation of dynamically generated web pages. In: Ellis A, Hagino T, eds. *Proceedings of the 14th international conference on World Wide Web, WWW'05*. New York: ACM, 432–441 DOI 10.1145/1060745.1060809.

Mitropoulos D, Karakoidas V, Louridas P, Spinellis D. 2011. Countering code injection attacks: a unified approach. *Information Management and Computer Security* **19(3)**:177–194 DOI 10.1108/09685221111153555.

Mitropoulos D, Karakoidas V, Spinellis D. 2009. Fortifying applications against XPath injection attacks. In: *4th mediterranean conference on information systems, MCIS 2009*. 1169–1179.

Mitropoulos D, Spinellis D. 2009. SDriver: location-specific signatures prevent SQL injection attacks. *Computers and Security* **28**:121–129 DOI 10.1016/j.cose.2008.09.005.

Mitropoulos D, Stroggylos K, Spinellis D, Keromytis AD. 2016. How to train your browser: preventing XSS attacks using contextual script fingerprints. *ACM Transactions on Privacy and Security* **19**(**1**):2:1–2:31 DOI 10.1145/2939374.

Moonen L. 1997. A generic architecture for data flow analysis to support reverse engineering. In: *Proceedings of the 2nd international conference on theory and practice of algebraic specifications, Algebraic'97*. Swinton: British Computer Society, 10–10.

Nadji Y, Saxena P, Song D. 2006. Document structure integrity: a robust basis for cross-site scripting defense. In: *Proceedings of the 22nd annual computer security applications conference, ACSAC'06*. Washington, D.C.: IEEE Computer Society, 463–472.

Nadji Y, Saxena P, Song D. 2009. Document structure integrity: a robust basis for cross-site scripting defense. In: *Proceeding of the network and distributed system security symposium (NDSS)*.

Nagy C, Mancoridis S. 2009. Static security analysis based on input-related software faults. In: *Proceedings of the 2009 European conference on software maintenance and reengineering, CSMR '09*. Washington, D.C.: IEEE Computer Society, 37–46 DOI 10.1109/CSMR.2009.51.

Nanda S, Lam L-C, Chiueh T-C. 2007. Dynamic multi-process information flow tracking for web application security. In: *Proceedings of the 2007 international conference on middleware companion, MC'07*. New York: ACM, 1–19 DOI 10.1145/1377943.1377956.

Nguyen-Tuong A, Guarnieri S, Greene D, Shirley J, Evans D. 2006. Automatically hardening web applications using precise tainting. In: Sasaki R, Qing S, Okamoto E, Yoshiura H, eds. *IFIP international information security conference*. 295–308 DOI 10.1007/0-387-25660-1_20.

Null LM, Wong J. 1992. The diamond security policy for object-oriented databases. In: Agrawal JP, Kumar J, Wallentine V, eds. *Proceedings of the 1992 ACM annual conference on communications, CSC'92*. New York: ACM, 49–56 DOI 10.1145/131214.131221.

Papagiannis I, Migliavacca M, Pietzuch P. 2011. PHP aspis: using partial taint tracking to protect against injection attacks. In: Fox A, ed. *Proceedings of the 2nd USENIX conference on web application development, WebApps'11*. Berkeley: USENIX Association, 2–2.

Peck R, Devore J. 2010. *Statistics: the exploration & analysis of data*. Boston: Brooks/Cole, Cengage Learning.

Pierce BC. 2002. *Types and programming languages*. Cambridge: MIT Press.

Pincus J, Baker B. 2004. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security and Privacy* **2**(**4**):20–27 DOI 10.1109/MSP.2004.36.

Pnueli A. 1977. The temporal logic of programs. In: *Proceedings of the 18th annual symposium on foundations of computer science, SFCS '77*. Washington, D.C.: IEEE Computer Society, 46–57 DOI 10.1109/SFCS.1977.32.

Qin F, Wang C, Li Z, Kim H-S, Zhou Y, Wu Y. 2006. LIFT: a low-overhead practical information flow tracking system for detecting security attacks. In: *Proceedings of the 39th annual IEEE/ACM international symposium on microarchitecture, MICRO 39*. Washington, D.C.: IEEE Computer Society, 135–148 DOI 10.1109/MICRO.2006.29.

Ray D, Ligatti J. 2012. Defining code-injection attacks. In: Field J, ed. *Proceedings of the 39th annual ACM symposium on principles of programming languages, POPL'12*. New York: ACM, 179–190 DOI 10.1145/2103621.2103678.

Reis C, Dunagan J, Wang HJ, Dubrovsky O, Esmeir S. 2006. BrowserShield: vulnerability-driven filtering of dynamic HTML. In: Bershad B, Mogul J, eds. *Proceedings of the 7th symposium on operating systems design and implementation, OSDI'06*. Berkeley, CA, USA: USENIX Association, 61–74.

Reps T, Horwitz S, Sagiv M. 1995. Precise interprocedural dataflow analysis via graph reachability. In: Cytron RK, Lee P, eds. *Proceedings of the 22nd ACM symposium on principles of programming languages, POPL'95*. New York: ACM, 49–61 DOI 10.1145/199448.199462.

Richards G, Hammer C, Burg B, Vitek J. 2011. The eval that men do: a large-scale study of the use of eval in javascript applications. In: Mezini M, ed. *Proceedings of the 25th European conference on object-oriented programming, ECOOP'11*. Berlin: Springer-Verlag, 52–78 DOI 10.1007/978-3-642-22655-7_4.

Romero-Mariona J, Ziv H, Richardson DJ, Bystritsky D. 2009. Towards usable cyber security requirements. In: *Proceedings of the 5th annual workshop on cyber security and information intelligence research: cyber security and information intelligence challenges and strategies, CSIIRW'09*. New York: ACM, 64:1–64:4 DOI 10.1145/1558607.1558681.

Ruse M, Sarkar T, Basu S. 2010. Analysis & detection of SQL injection vulnerabilities via automatic test case generation of programs. In: *Proceedings of the 2010 10th IEEE/IPSJ international symposium on applications and the internet, SAINT'10*. Washington, D.C.: IEEE Computer Society, 31–37 DOI 10.1109/SAINT.2010.60.

Saiedian H, Broyle D. 2011. Security vulnerabilities in the same-origin policy: implications and alternatives. *Computer* **44(9)**:29–36 DOI 10.1109/MC.2011.226.

Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D. 2010. A symbolic execution framework for JavaScript. In: *Proceedings of the 2010 IEEE symposium on security and privacy*. Washington, D.C.: IEEE Computer Society, 513–528 DOI 10.1109/SP.2010.38.

Schwarz B, Chen H, Wagner D, Lin J, Tu W, Morrison G, West J. 2005. Model checking an entire linux distribution for security violations. In: *Proceedings of the 21st annual computer security applications conference, ACSAC'05*. Washington, D.C.: IEEE Computer Society, 13–22 DOI 10.1109/CSAC.2005.39.

Seixas N, Fonseca J, Vieira M, Madeira H. 2009. Looking at web security vulnerabilities from the programming language perspective: a field study. In: *Proceedings of the*

2009 20th international symposium on software reliability engineering, ISSRE'09. Washington, D.C.: IEEE Computer Society, 129–135 DOI 10.1109/ISSRE.2009.30.

Sekar R. 2009. An efficient black-box technique for defeating web application attacks. In: *Proceeding of the network and distributed system security symposium (NDSS)*. Reston: Internet Society.

Shacham H, Page M, Pfaff B, Goh E-J, Modadugu N, Boneh D. 2004. On the effectiveness of address-space randomization. In: Atluri V, ed. *Proceedings of the 11th ACM conference on computer and communications security, CCS'04*. New York: 298–307 DOI 10.1145/1030083.1030124.

Shahriar H, Zulkernine M. 2012. Mitigating program security vulnerabilities: approaches and challenges. *ACM Computing Surveys* **44(3)**:11:1–11:46 DOI 10.1145/2187671.2187673.

Sivakumar K, Garg K. 2007. Constructing a "Common cross site scripting vulnerabilities enumeration (CSE)" using CWE and CVE. In: McDaniel P, Gupta SK, eds. *Proceedings of the 3rd international conference on information systems security*. Berlin: Springer-Verlag, 277–291 DOI 10.1007/978-3-540-77086-2_25.

Son S, McKinley KS, Shmatikov V. 2013. Diglossia: detecting code injection attacks with precision and efficiency. In: Sadeghi A-R, ed. *Proceedings of the 2013 ACM conference on computer and communications security, CCS'13*. New York: ACM, 1181–1192 DOI 10.1145/2508859.2516696.

Son SH, Chaney C, Thomlinson NP. 1998. Partial security policies to support timeliness in secure real-time databases. In: *Proceedings of the IEEE symposium on security and privacy*. Charlottesville: IEEE DOI 10.1109/SECPRI.1998.674830.

Soni P, Budianto E, Saxena P. 2015. The SICILIAN defense: signature-based whitelisting of web JavaScript. In: Ray I, ed. *Proceedings of the 22nd ACM conference on computer and communications security, CCS'15*. New York: ACM, 1542–1557 DOI 10.1145/2810103.2813710.

Sovarel AN, Evans D, Paul N. 2005. Where's the FEEB? the effectiveness of instruction set randomization. In: *Proceedings of the 14th USENIX security symposium*. Berkeley: USENIX Association, 10–10.

Spacco J, Hovemeyer D, Pugh W. 2006. Tracking defect warnings across versions. In: Diehl S, Gall H, Hassan AE, eds. *Proceedings of the 2006 international workshop on mining software repositories, MSR'06*. New York: ACM, 133–136 DOI 10.1145/1137983.1138014.

Stamm S, Sterne B, Markham G. 2010. Reining in the web with content security policy. In: Rappa M, Jones P, eds. *Proceedings of the 19th international conference on world wide web, WWW'10*. New York: ACM, 921–930 DOI 10.1145/1772690.1772784.

Stefan D, Yang EZ, Marchenko P, Russo A, Herman D, Karp B, Mazières D. 2014. Protecting users by confining JavaScript with COWL. In: *Proceedings of the 11th USENIX conference on operating systems design and implementation, OSDI'14*. Berkeley: USENIX Association, 131–146.

Stock B, Lekies S, Mueller T, Spiegel P, Johns M. 2014. Precise Client-side protection against Dom-based cross-site scripting. In: *Proceedings of the 23rd USENIX security symposium*. 655–670.

Su Z, Wassermann G. 2006. The essence of command injection attacks in web applications. In: Morrisett G, ed. *Conference record of the 33rd ACM symposium on principles of programming languages, POPL'06*. New York: ACM, 372–382 DOI 10.1145/1111037.1111070.

Szekeres L, Payer M, Wei T, Song D. 2013. SoK: eternal war in memory. In: Sommer R, ed. *Proceedings of the 2013 IEEE symposium on security and privacy*. 48–62 DOI 10.1109/SP.2013.13.

Takesue M. 2008. A protection scheme against the attacks deployed by hiding the violation of the same origin policy. In: *Proceedings of the 2008 second international conference on emerging security information, systems and technologies*. Washington, D.C.: IEEE Computer Society, 133–138 DOI 10.1109/SECURWARE.2008.24.

Thuraisingham B, Ford W. 1995. Security constraint processing in a multilevel secure distributed database management system. *IEEE Transactions on Knowledge and Data Engineering* **7(2)**:274–293 DOI 10.1109/69.382297.

Trinh M-T, Chu D-H, Jaffar J. 2014. S3: a symbolic string solver for vulnerability detection in web applications. In: Ahn G-J, ed. *Proceedings of the 2014 ACM conference on computer and communications security, CCS'14*. New York: ACM, 1232–1243 DOI 10.1145/2660267.2660372.

Tsitovich A. 2008. Detection of security vulnerabilities using guided model checking. In: Garcia de la Banda M, Pontelli E, eds. *Proceedings of the 24th international conference on logic programming, ICLP'08*. Berlin: Springer-Verlag, 822–823 DOI 10.1007/978-3-540-89982-2_90.

Valeur F, Mutz D, Vigna G. 2005. A Learning-based Approach to the Detection of SQL Attacks. In: Julisch K, Kruegel C, eds. *Proceedings of the second international conference on detection of intrusions and malware, and vulnerability assessment, DIMVA'05*. Berlin: Springer-Verlag, 123–140 DOI 10.1007/11506881_8.

Van der Veen V, Andriesse D, Göktaş E, Gras B, Sambuc L, Slowinska A, Bos H, Giuffrida C. 2015. Practical context-sensitive CFI. In: Ray I, ed. *Proceedings of the 22nd ACM conference on computer and communications security, CCS'15*. New York: ACM, 927–940 DOI 10.1145/2810103.2813673.

Viega J, Bloch JT, Kohno T, McGraw G. 2002. Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security* **5(3)**:238–261 DOI 10.1145/545186.545188.

Viega J, Bloch JT, Kohno Y, McGraw G. 2000. ITS4: a static vulnerability scanner for C and C++ code. In: *Proceedings of the 16th annual computer security applications conference, ACSAC'00*. Washington, D.C.: IEEE Computer Society, 257.

Viega J, McGraw G. 2001. *Building secure software: how to avoid security problems the right way*. Boston: Addison-Wesley.

Vogt P, Nentwich F, Jovanovic N, Kirda E, Kruegel C, Vigna G. 2007. Cross-site scripting prevention with dynamic data tainting and static analysis. In: *Proceeding* *of the network and distributed system security symposium (NDSS)*. Reston: Internet Society.

Von Oheimb D. 2004. Information flow control revisited: noninfluence = noninterference +nonleakage. In: Samarati P, Ryan P, Gollmann D, Molva R, eds. *Proceedings of the 9th European symposium on research in computer security, ESORICS'04*. Springer, 225–243 DOI 10.1007/978-3-540-30108-0_14.

Wagner D, Foster JS, Brewer EA, Aiken A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In: *Proceeding of the network and distributed system security symposium (NDSS)*. Reston: Internet Society, 3–17.

Wang H. 2010. Attacks target Web server logic and prey on XCS weaknesses: technical persepctive. *Communications of the ACM* **53(8)**:104–104 DOI 10.1145/1787234.1787256.

Wang X, Pan C-C, Liu P, Zhu S. 2010. SigFree: a signature-free buffer overflow attack blocker. *IEEE Transactions on Dependable and Secure Computing* **7(1)**:65–79 DOI 10.1109/TDSC.2008.30.

Wassermann G, Su Z. 2004. An analysis framework for security in web applications. In: Taylor RN, ed. *SAVCBS 2004: proceedings of the FSE workshop on specification and verification of component-based systems SAVCBS*, 70–78.

Wassermann G, Su Z. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In: Ferrante J, ed. *PLDI '07: proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation*. New York: ACM Press, 32–41 DOI 10.1145/1273442.1250739.

Wilander J, Kamkar M. 2002. A comparison of publicly available tools for static intrusion prevention. In: *Proceedings of the 7th nordic workshop on secure IT systems*. Karlstad, Sweden: Karlstad University, 68–84.

Winsor J. 2000. *Solaris system administrator's guide*. Third Edition. Upper Saddle River: Prentice Hall PTR.

Wurster G, Van Oorschot PC. 2008. The developer is the enemy. In: Bishop M, Probst CW, eds. *NSPW'08: proceedings of the 2008 workshop on new security paradigms*. New York: ACM, 89–97 DOI 10.1145/1595676.1595691.

Wurzinger P, Platzer C, Ludl C, Kirda E, Kruegel C. 2009. SWAP: mitigating XSS attacks using a reverse proxy. Washington, D.C.: IEEE Computer Society, 33–39.

Xie Y, Aiken A. 2006. Static detection of security vulnerabilities in scripting languages. In: *Proceedings of the 15th conference on USENIX security symposium—Volume 15, USENIX-SS'06*. Berkeley: USENIX Association.

Xu W, Bhatkar S, Sekar R. 2006. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: *Security'06: proceedings of the 15th USENIX security symposium*. Berkeley: USENIX Association, 121–136.

Younan Y, Joosen W, Piessens F. 2012. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys* **44(3)**:17:1–17:28 DOI 10.1145/2187671.2187679.

Yu D, Chander A, Islam N, Serikov I. 2007. JavaScript instrumentation for browser security. In: Hofmann M, ed. *Proceedings of the 34th ACM symposium on principles of programming languages*. New York: ACM, 237–249 DOI 10.1145/1190215.1190252.

Zitser M, Group DES, Leek T. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Software Engineering Notes* **29(6)**:97–106 DOI 10.1145/1041685.1029911.

Zitser M, Lippmann R, Leek T. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Software Engineering Notes* **29(6)**:97–106 DOI 10.1145/1041685.1029911.