Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution

Giovanni Agosta, Alessandro Barenghi, Antonio Parata, Gerardo Pelosi
Dipartimento di Elettronica e Informazione (DEI) – Politecnico di Milano
Via Giuseppe Ponzio 34/5, 20133-Milano, Italy
Email: {agosta, barenghi, pelosi}@elet.polimi.it

***Abstract*—The automatic identification of security vulnerabilities is a critical issue in the development of web-based applications. We present a methodology and tool for vulnerability identification based on symbolic code execution exploiting Static Taint Analysis to improve the efficiency of the analysis. The tool targets PHP web applications, and demonstrates the effectiveness of our approach in identifying cross-site scripting and SQL injection vulnerabilities on both NIST synthetic benchmarks and real world applications. It proves to be faster and more effective than its main competitors, both open source and commercial.**

***Keywords*-Cross-Site Scripting; SQL Injection; Static Taint Analysis; Symbolic Execution.**

I. Introduction

The modern scenario of software and web development shows a convergence of the two areas: on one hand, web sites rely more and more on dynamically generated content, allowing complex interaction with the users through collecting and storing information provided by them; on the other hand, many software applications require remote user access and collaboration, which are easily obtained through using rich web browsers within client-server applications. This convergence has led to the widespread adoption of Web Applications, which are largely employed in e-government, e-banking, e-commerce, social networking, collaborative software development and text editing. Most of these applications collect sensitive data about their users, and are therefore potential targets for a wide range of malicious activities including stealing data, hijacking user sessions, or phishing. Thus, it is of the utmost importance for Web Application software to be secured against the whole range of possible attacks. Traditionally, this is done through manual processes of code review or code inspection, where experienced analysts look for specific vulnerable patterns or even perform simulations of program execution by hand. The time required for this analysis is huge – indeed, many modern Web Applications fall in the 100,000+ lines of code range – leading to higher chance that vulnerabilities remain undetected. Several automated software analysis tools and methodologies have been developed to identify security issues or mitigate their threats. We propose a methodology and software framework for fast static vulnerability identification, and showcase it on the two most dangerous classes of vulnerabilities, *Cross-Site Scripting* (XSS) and *SQL Injection* (SQLI). Our methodology builds over existing ones, combining Static Taint Analysis with Symbolic Code Execution to identify whether malicious user inputs can be used to subvert the semantics of the application. The rest of this paper is organized as follows. Section II outlines the vulnerability identification problem, and describes the two most common vulnerabilities. Section III describes our methodology, while Section IV provides an experimental evaluation of our approach. Finally, Section V discusses the related work and Section VI draws some conclusions and outlines future directions.

II. Problem Statement

Web Applications are in general deployed in a two tier architecture. The client receives dynamic web page composed by the application server in response to HTTP requests. The application server contains the application logic, which can be expressed in any programming language supporting the CGI interface or an RPC-like remote calling convention. While in the past binary applications written in C were common, nowadays interpreted or dynamically compiled languages, such as PHP, Java, Ruby, and Perl are increasingly popular. The second tier usually provides support to the Web Application for data intensive operations through a dedicated database management system. PHP is one of the most popular scripting languages, and specifically it is widely used in Web Applications: the PHP language support module was reported to be installed on about 40% instances of the Apache web server[1], which itself is by far the most popular web server, reaching a 70% market share[2]. It is used in popular Web Applications such as phpBB, PHP-Nuke, SquirrelMail and MediaWiki. The two most common, high risk vulnerabilities in PHP based applications are SQL Injection attacks (SQLI) and cross site scripting (XSS) according to OWASP[3] and SANS[4].

A. Cross Site Scripting Attack

XSS vulnerabilities[5] allow an attacker to inject malicious code in a dynamic web page, embedding it into the generated web page, thus effectively subverting the layout of the target form. In a reflected XSS attack, the user is tricked into clicking on a link to the vulnerable dynamic page, where the link contains a malicious script crafted by the attacker within a CGI parameter, as shown in Figure 1. The forged link points to a website (`example.com`, which may be a

[1]www.securityspace.com/s_survey/data/man.200704/apachemods.html
[2]http://www.securityspace.com/s_survey/data/200904/index.html
[3]http://www.owasp.org/index.php/Top_10_2007
[4]http://www.sans.org/top20/s1
[5]http://www.cert.org/advisories/CA-2000-02.html

website known or trusted by the user) where a vulnerable application form (`comment.cgi`) will produce an output page containing the injected script (`mycomment`). This grants the attacker the ability to disguise malicious code as part of a trusted web page, thus cheating the user on the real origin of the page. This ability may be exploited to harvest

```
<A HREF="http://example.com/comment.cgi?
 mycomment=<SCRIPT>malicious code</SCRIPT>">
 Click here </A>
```

Figure 1. Example of reflected XSS attack

sensitive data such as user account or credit card numbers, input by the user, or directly retrieving sensitive data present in the client machine such as session cookies or password files. The stored XSS attack works in a similar fashion, except that the script is stored into the server (e.g., as a message board post) and then sent to other clients without proper processing. When the poisoned page is rendered, the malicious script is executed on the client machine.

B. SQL Injection Attack

SQLI attacks exploit SQL queries that include user provided strings without proper checking. The attacker can alter the form of the query according to the field he is in control of. The simplest example is shown in Figure 2, where the query checks into a table the presence of tuples with a key matching with the user input string. By passing a suitably crafted parameter (e.g. `foo" OR 1=1 ; --`) the attacker can circumvent the check performed by the `WHERE` clause, and obtain the entire table in `$result`. Assuming that the login form is checking for a non-null value of `$result` to grant access to the user, the attacker is able to gain access without even knowing a valid username or password. The

```
$result=mysql_query(
  'SELECT * FROM table WHERE key="'.
  $_GET['key'].'"'   );
```

Figure 2. Example of PHP statement vulnerable to SQLI

same mechanism can be exploited for more complex actions, up to gaining remote execution on the DB server, if the DBMS allows native execution primitives via hooks [1].

III. Automated Code Auditing

A relevant part of life cycle of a reliable software is represented by periodical code auditing phases. Our approach targets security oriented auditing, aiming at a sound re-engineering of Web Applications lacking a reliable and secure structure. Under a formal perspective, a program affected by security flaws is a syntactically correct and semantically sound codebase, where the introduction of specially crafted input values may subvert either the execution flow or lead to unwanted information leakage. To cope with the need for precise and correct code review process, a number of computer-aided code auditing suites have been developed. The achieving of a fully automated vulnerability detection infrastructure is the holy grail of security-oriented code auditing. State-of-the-art tools try to either mimic human code review techniques through finding code patterns usually bound to security flaws or use a formal framework to obtain provable checking of the application structure. The first technique boasts extremely high efficiency but comparatively low effectiveness, since it lacks a proper understanding of complex vulnerability patterns. The second approach, on the other hand, has a very high detection accuracy, thanks to the more structured methodology, but often suffers from the employment of computationally hard algorithms and may find vulnerabilities which are not practically exploitable. In the following, we introduce our modular, extensible, multi-platform code analysis suite named Codeminer, which aims at being a highly effective checker while retaining the efficiency required to process large real-world codebases. The proposed approach exploits a formal description of the program control flow and data flow to spot inter-procedural vulnerability patterns which are ignored by pure expression matching approaches. This enables us to identify complex vulnerability patterns which are beyond the capability of tools like RATS [2]. We rely on Symbolic Code Execution to escape the weight of computationally hard algorithm which are commonly used in purely formal methods, while retaining a high precision in modeling the code patterns. We also exploit Taint Analysis techniques to further enhance the efficiency, thus making it possible to perform inter-procedural analysis which would otherwise be unfeasible on large codebases due to its high computational weight. In the following subsections we will describe the metodology adopted to design our tool: Codeminer. Figure 3 shows the architecture of the tool, highlighting its main components.

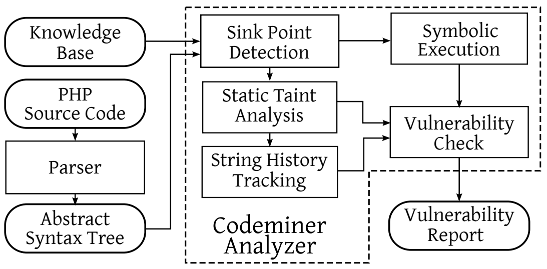

Figure 3. Architecture of the Codeminer Tool

First of all the PHP code is transformed into a language independent representation of the source code, namely an *abstract syntax tree*, through the use of a parser. This allows an easier extraction of the control and data flow information needed for our analysis. Moreover, using a language independent representation enables us to further extend our tool to analyze code in other programming languages with minimal effort. Among all the well tested PHP parsers freely available, we chose to reuse the one implemented by Minamide [3] for its high efficiency. The crucial point in secure programming is dealing appropriately with *user controlled input*, that is a portion of data in the program which is directly supplied by a (possibly malicious) user. From now on we will be referring to all the values influenced by user input as *tainted*, to emphasize the potential threat. To track the effects of *tainted* inputs, our tool performs a data flow analysis and ascertains the data paths affected by the tainting. This allows us to discard all the data untouched by user supplied values, thus reducing the complexity of the vulnerability identification without affecting its effectiveness. Module *Static Taint Analysis* in Figure 3 is the one implementing the data flow analysis technique. A *sink point* is a location in the program where the user controlled input is employed disrupting the expected behavior or output. The module *Sink Point Detection* in Figure 3 is in charge of recognizing the sink points in the application and is responsible for triggering the analysis of the inputs. The information collected during data flow analysis and sink point detection is subsequently used to decide whether the tainted values are passing through sink points without any checking for the presence of possibly malicious inputs. The *Symbolic Code Execution* module evaluates the taint status of a variable, depending on the point of the code in which is employed, through collecting information about the phases of execution it has undergone. The *Vulnerability Check* module is called by the Symbolic Code Execution module every time a sink point should be executed and checks for the actual presence of a vulnerability against a *knowledge base* of dangerous patterns according to the sink point type.

A. Sink Point Detection

The first module run by Codeminer is the sink point detection. This module scans the codebase and checks for the presence of sensitive functions. Their presence is ascertained through the comparison with a language-dependent, extensible *knowledge base* that is provided as input to the tool. It is thus possible to extend the tool to detect sink points also in programming languages different from PHP. All the sink points are characterized by different vulnerability patterns which require different checkers: the sink point detection module contains also a reference to the correct checker to be invoked for each kind of vulnerability.

B. Static Taint Analysis

Once the detection of the sink points is complete, Codeminer starts examining the abstract syntax tree provided by the parser to perform Static Taint Analysis and extract information on the chain of modifications which every variable undergoes. Static Taint Analysis has been first used on Web Applications in [4], though it was first introduced for the identification of format string vulnerabilities in traditional applications [5]. This technique has been implemented in Codeminer as a forward dataflow analysis, a well-known technique in optimizing compilation [6], based on the solution of fixed point equations. Our goal is to compute the *reaching definitions* for the tainted variables to check if any part of them is used in a sink point. To this end, the dataflow analysis is enriched with information regarding the taint status of each variable. In particular, the *transfer function* of the *reaching definitions* algorithm will set the taint status for any assigned variable to *tainted* if at least one of the variables that are used in computing its value is tainted, or if it is assigned a value from one of the builtin functions of the source language that read input values. In case a conditional construct is met, the branch taken may also influence the taint status of a variable. Taking a conservative approach, the *join function* of the dataflow analysis preserves the taint property for a variable if it is tainted in at least one of the control flow branches. After performing the forward dataflow analysis, we are able to pinpoint exactly which tainted variables are fed as parameters to any sink point of the program. Any untainted variables from now on will be irrelevant as far as security analysis purposes go, therefore we may safely avoid to involve them from now on.

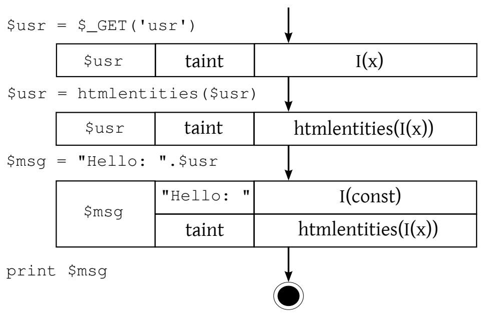

Figure 4. Construction of the sequences of functions associated with strings $usr and $msg. I is the identity function, corresponding to a direct read of a user-input value. C is the constant function, corresponding to the assignment of a known value

C. String History Tracking

The crucial step towards detecting vulnerabilities caused by unchecked user inputs is to enrich the reaching definitions through building a precise history of all the modifications undergone by a string during its lifetime. This allows us to consider the status of a variable at a finer grain through distinguishing how it became tainted and its internal structure, when its value is generated by concatenation of several strings. We choose to store this information in the form of a set of function stacks bound to each tainted variable. We will now describe how these sets are built starting from a single basic block. When analyzing the code from a security auditing point of view, the possible actions on the values of variables (containing strings) may be split into four kinds: (i) initialization, either from user input or a constant value; (ii) string concatenation; (iii) application of a builtin function of the source language; (iv) use in a sink point.

Figure 4 shows an example of the construction of the data structures associated with the string variables $usr and $msg from the following PHP code fragment:

```
$usr = $_GET['usr'];
$usr = htmlentities($usr);
$msg = "Hello: ".$usr;
print $msg;
```

The first line of code contains a statement that initializes a variable directly from user input: this implies that a new function stack is assigned to the variable $usr, where the content of the variable is the *taint* value and the associated function is the identity function $I(\cdot)$ representing the absence of manipulations of the variable value so far. The second line of the code is a sample application of the built-in PHP function `htmlentities`, whose task is to remove all the character involved in HTML tag termination, from the variable `$usr`. To represent this action, we add to the function stack of the involved variable the function `htmlentities` through function composition with the previous function stack. In the third line, we show an example of the effects of variable concatenation on the variable representation. Remember that, when two or more strings are concatenated, the variable that is assigned the generated value will be considered tainted if at least one of its components is tainted. Its sequence of functions is represented by the ordered packing of the function sets belonging to its components. Figure 4 shows the result of the concatenation operation between the tainted variable `$usr` and the constant valued variable `Hello:` whose function sets are composed in concatenation order to obtain the set associated to `$msg`. The last line of code is a typical sink point for XSS vulnerabilities in PHP language, and is employing the tainted variable `$msg`. Upon reaching any sink point, the vulnerability checking procedure is triggered and the Codeminer tool evaluates the possible security issues, returning to the variable status analysis after reporting its findings. The execution of the Codeminer tool ends when all the variable statuses have been fully analyzed throughout the whole application and therefore all the uses of the variables in the sink points have been checked with the Symbolic Code Execution based routine.

D. Symbolic Code Execution

To obtain a precise model of which functions are applied to each(*exec* variable during the execution of each basic block, Codeminer employs the Symbolic Code Execution technique, which simulates the actual execution of the program through running it *in the aggregate* [7], i.e. considering the inputs as belonging to a finite set. In our case, the set of possible inputs is defined by all the constant values known at static time, plus the value *taint*. During the first phase of Symbolic Code Execution each variable is initialized according to the status of the taint attribute, which may be a known value (e.g., 42 or “foo”) or the *taint* value. After variable initialization has been completed, each statement of the basic block is simulated, and the effects on the variables are recorded. As far as numeric variables go, we consider the combination of two known values as a known value, while any combination involving a tainted variable is considered tainted as a whole, with the exception of the absorbing element of an operation (e.g., $taint \otimes 0 \rightarrow 0$). This is sufficient to address any possible security concerns with numeric variables, since operations on numeric types uniformly propagate the tainting, regardless of the order of the operands. Variables composed by multiple substrings behave differently, since the interleaving of user input with different constant values yields strings with different security properties depending on the nature and ordering of the constants. Symbolic code execution evaluates them through

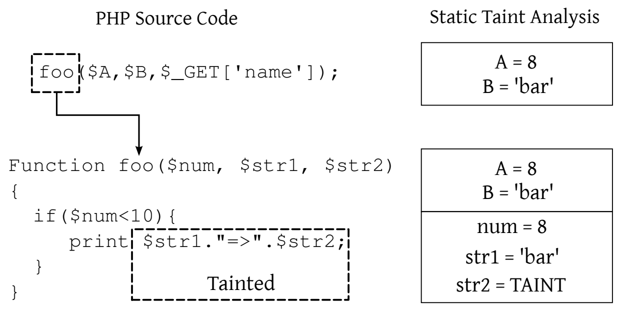

Figure 5. Taint analysis & symbolic execution across a function call. The variables are updated by the analyzer. The `print` function is a sink point

resolving all the constant values known at static time and preserving the structure of each string variable in terms of elementary (known or tainted) substrings. Keeping track of the full structure of the variables enables us to avoid many false positives which are often encountered in tools which do not take this information into account. In the example reported in Figure 5, the Symbolic Code Execution performs also inter-procedural analysis: the techniques adopted to achieve this capability are reported in Section III-F. At first the constant values `8` and `'bar'` are respectively assigned to the variables `$A` and `$B` since they are both known at static time. After this, the symbolic execution engine analyzes the function `foo` and, recognizing that it is a user defined one then, it jumps to the point where the implementation is contained. Following the execution flow, the actual parameters of the function are mapped to the formal parameters. This allows the analyzer to correctly resolve the conditional statement present in the code block, thus taking the branch and locating the sink point represented by the `print` function.

E. Vulnerability Checking

The vulnerability checking stage exploits a database of sink point definitions: every definition includes a vulnerability evaluation function which is in charge to discern whether the sink point argument may represent a threat. To provide an accurate prediction of the sink point argument, the Symbolic Code Execution stage examines which manipulations have been applied to the user input. Of particular interest is a class of functions, commonly defined as *sanitizing functions*, which are able to filter dangerous patterns from the user input. In case a sanitizing function is applied to a tainted input, the vulnerability checking stage correctly detects no threat for the program security The vulnerability checks are expressed through a set of rules describing the structure of the input strings, i.e. to each input is associated a set of forbidden characters that must have been filtered or escaped by the sanitizing functions. Moreover, structural patterns of the input are evaluated to match the ones which practically permit to exploit a vulnerability. The use of a program-wide history of the variables and the information on the control flow extracted through Symbolic Code Execution allows Codeminer to locate complex vulnerability patterns which are ignored by tools performing only local analysis of the code. The solution proposed is able to provide a thorough checking procedure without constructing the language of strings generated by the program at each sink point, as done in [8] – an extremely time consuming process, where analysis times are often hard to predict, since they depend more on the structure of the language of string values passed to the sink points than on the size of the processed codebase.

F. Interprocedural analysis

To analyze real world Web Applications, Codeminer needs to track string manipulation history and taint propagation through a number of user defined function invocation. Since a function can perform arbitrarily complex manipulations of its inputs to produce an output, dataflow analysis must be performed at inter procedural level. The PHP programming language allows the definition of functions either in the same file where the caller is present or in a different one which may be included dynamically at runtime. In the first case, it is possible to follow the control flow of the program through the function calls and to perform the symbolic execution of its statements . In the second case, the program needs to perform a simulated execution to construct the correct *inclusion tree* of the files within the application codebase. After the inclusion tree has been built, the analyzer evaluates the functions with the same method employed for the ones present in the same file. This technique, employed in both Symbolic Code Execution and Static Taint Analysis, enables a thorough exam of the whole codebase, thus raising the effectiveness of the tool over those that consider each source file separately.

IV. Experimental Results

To demonstrate the effectiveness of our approach, we have implemented the framework described in Section III in the Codeminer tool. We compare the performance of our tool with two well-known competitors, the "Pixy" [9] open-source tool and the commercial "Static Code Analyzer v5.1" (SCA) tool, provided by Fortify [10]. [6] Pixy is one of the very few open-source tools in the field, while SCA is one of the leading proprietary solutions, and represents an industry-strength term of comparison. The experiments have been performed on an Intel Core2 Quad clocked at 2.33 GHz with 4 GB of main memory and a Windows-XP OS. The tool we developed, Codeminer, employs the PHP parser developed by Y. Minamide [3]. It is implemented in OCaml, and supports the extension of its *knowledge base* by means of runtime-loaded OCaml modules. External OCaml modules are used to implement the extensibility of the following features: (i) modelling the semantics of PHP builtin functions that cannot be directly simulated by the symbolic execution; (ii) checking whether a tainted input value has been sanitized, by checking whether the inputs to a sink point trigger a specific vulnerability; (iii) monitor input values to potentially dangerous functions (e.g., a XSS detection module would monitor `print` and `echo`). While the tool implements a full-fledged plugin framework to allow module development, for the purpose of the reported experimental campaign we have developed a set of plugins focusing on the identification of XSS and SQLI vulnerabilities. To provide a complete evaluation of our approach and tool, we employ two different sets of benchmark programs. The first set is composed of synthetic benchmarks provided by the NIST as part of the "Software Assurance Metrics And Tool Evaluation" (SAMATE) project [11]. Specifically, we employed test cases 1769, 1937, 1938 and 1939, i.e. all the PHP XSS test cases. Synthetic benchmarks are needed to check the accuracy of the tools, especially in terms of false negatives, which are otherwise difficult to verify on large applications. The second set, characterized in Table I, is composed of large real world applications including Mediawiki, which is the foremost wiki software (employed e.g. in Wikipedia) and Communitycms, which is a PHP/MySQL-based content management system. Testing a vulnerability identification suite on real world Web Applications is obviously important to gauge its ability to process codebases that represent its target application. However, it is also a time consuming process, since each reported vulnerability must be manually checked to verify whether it is a false positive or an actual vulnerability.

Table I
Real-world benchmarks characterization.

Benchmark	Version	# Files	LoC
Mediawiki	v1.6.12	537	176,801
Communitycms	v0.5	87	6,565

Synthetic NIST Benchmarks: In the case of the SAMATE benchmarks, the goal is to verify that no false negatives are produced. Table II reports the aggregated results of the test. Codeminer, correctly identifies all vulnerabilities, while producing no false negatives. The Fortify SCA tool does produce two false negatives, but these are pointed out in the vulnerability report as lacking input validation (rather then as actual vulnerabilities). On the other hand, Pixy produces two actual false negatives.

Table II
Tools comparison using the SAMATE NIST benchmark.
* marks vulnerabilities not detected as such, but reported as having a "low input validation" issue.

Tool	Vulnerabilities identified	False Positives	False Negatives
Codeminer	8	0	0
Pixy	6	0	2
SCA 5.1	6	0	2*

Real-World Applications: Table III reports the comparison among the three selected tools on the set of benchmarks, not taking into account *indirect* vulnerabilities (i.e., those depending on tainted query responses rather than tainted input) in the case of SCA, as our tool does not currently implement indirect vulnerability identification. It can be easily seen that Codeminer correctly identifies the same vulnerabilities as the other tools, but produces a lower

[6]We used the free evaluation version of this tool

amount of false positives. Note that, since verifying false positives is time expensive (and therefore costly in the applicative scenario), this reduction has a major impact on the usability of the tool. Moreover, Codeminer is much faster than its competitors, allowing it to scale better to both very small and very large target codebases.

Table III
TOOLS COMPARISON USING REAL WORLD WEB APPLICATIONS

Benchmark	Tool	Time [s]	Vuln.s (False Pos.) XSS	SQLI
Mediawiki	Codeminer	34	3 (0)	0 (0)
	Pixy	1093	3 (55)	0 (0)
	SCA 5.1	374	3 (0)	0 (0)
Communitycms	Codeminer	0.88	3 (0)	2 (0)
	Pixy	33	3 (6)	2 (0)
	SCA 5.1	111	3 (2)	2 (0)

V. RELATED WORK

Su and Wassermann developed security analysis tools focusing on scenarios similar to those considered in this paper [8], [12], and starting from the same PHP code analysis infrastructure by Minamide [3]. Their work is the closest to our own, especially the approach to SQLI detection presented in [8]. With respect to [8], we do not rely on the computationally costly query grammar generation, but on a much faster symbolic execution, which allows us to analyze large applications with significant database interaction in few minutes, which favorably compares to the analysis times of more than 3 hours reported for the same applications in [8]. Our approach, being based on checking that illegal strings cannot be passed through the sink points rather than ensuring that only legal strings reach them, cannot provide soundness guarantee. However, when comparing the experimental results on the same application we see that there is no loss of precision in practice – i.e., the Codeminer tool is always able to detect the same vulnerabilities as the tool proposed in [8]. Pixy [9] is one of the very few open source static analysis tools, and focuses on detecting XSS vulnerabilities. We compare directly to it, showing that we manage to obtain higher precision (both in terms of false positives and false negatives) with analysis times lower by almost one order of magnitude. The MiMoSA [13] tool (based on Pixy) covers both SQLI and XSS, focusing on attacks that leverage the interactions among several modules in the Web Application and considering as tainted the output of database queries. Our Codeminer tool addresses the first issue, but currently does not consider any saved value – though the issue is orthogonal and our tool could be extended to cover it. Xie and Aiken [14] use an approach to taint analysis that is, like ours, based on symbolic execution and dataflow analysis. However, they compute, for each basic block, a taint/untaint function that only takes into account known sanitization functions, which have the effect of cleaning the values returned from the input tainted values, thus missing an important degree of precision in identifying different kinds of vulnerabilities.

VI. CONCLUSION AND FUTURE WORK

We presented a methodology and tool, Codeminer, for the static analysis of code to identify vulnerabilities. Its key feature is the ability to accurately track the manipulation history of each string parameter passed to a sink point, allowing the vulnerability checkers to exploit accurate information on the structure of it. Codeminer can be used to identify most major vulnerabilities, and plugins for the identification of SQLI and XSS vulnerabilities have been developed and evaluated, proving that Codeminer is faster and more effective than its main competitors, both open-source and commercial. The work can be extended with new plugins to provide a coverage of more vulnerability classes.

REFERENCES

[1] D. Stuttard and M. Pinto, *The web application hacker's handbook: discovering and exploiting security flaws*. Wiley, 2007.

[2] Fortify, "Rough auditing tool for security," http://www.fortify.com/security-resources/rats.jsp.

[3] Y. Minamide, "Static approximation of dynamically generated web pages," in *WWW '05: Proc. 14th international conference on World Wide Web*. 2005, pp. 432–441.

[4] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW '04: Proc. 13th international conference on World Wide Web*. 2004, pp. 40–52.

[5] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *SSYM'01: Proc. 10th conference on USENIX Security Symposium*. 2001, pp. 16–16.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[7] T. W. Reps, A. Lal, and N. Kidd, "Program analysis using weighted pushdown systems," in *FSTTCS*, 2007, pp. 23–51.

[8] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *PLDI '07: Proc. 2007 ACM SIGPLAN conference on Programming language design and implementation*. 2007, pp. 32–41.

[9] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *PLAS '06: Proc. 2006 workshop on Programming languages and analysis for security*. 2006, pp. 27–36.

[10] B. Chess, "Fortify 360 whitepaper," http://www.fortify.com.

[11] NIST, "SAMATE Reference Dataset," http://samate.nist.gov/.

[12] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *ICSE '08: Proc. 30th int'l conference on Software engineering*. 2008, pp. 171–180.

[13] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *CCS '07: Proc. 14th ACM conference on Computer and communications security*. 2007, pp. 25–35.

[14] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX-SS'06: Proc. 15th conference on USENIX Security Symposium*. 2006.