# Patcher: An Online Service for Detecting, Viewing and Patching Web Application Vulnerabilities

Fang Yu and Yi-Yang Tung
Department of Management Information Systems
National Chengchi University
yuf@nccu.edu.tw

## Abstract

*Web application security becomes a critical issue as more and more web applications appear and serve common life and business routines in recent years. It is known that web applications are vulnerable due to software defects. Open to public users, vulnerable websites may encounter lots of malicious attacks from the Internet. We present a new web service platform where system developers can detect, view and patch potential vulnerabilities of their web applications online. Taking advantage of static string analysis techniques, our analysis ensures that the patched programs are free from vulnerabilities with respect to given attack patterns. Specifically, we integrate the service front end with program visualization techniques, developing a 3D interface/presentation for users to access and view the analysis result under visualization environment with the aim of improving users' comprehension on programs, especially how vulnerabilities get exploited and patched. We report our analysis result on several open source applications, finding and patching various unknown/known vulnerabilities.*

***Keywords: visualization, web security, string analysis, program comprehension***

## 1. Introduction

Web applications have become a crucial part of commerce, entertainment and social interaction and they are rapidly replacing desktop applications. In the near future, they are expected to play critical roles in national infrastructures such as healthcare, national security, and the power grid. However, there is a large stumbling block to the ever-increasing reliance on web applications in almost every aspect of society: they are notorious for security vulnerabilities. Global accessibility of web applications makes this a very serious problem. Malicious users all around the world can exploit a vulnerable web application and cause serious damages.

According to the Open Web Application Security Project (OWASP)'s top ten list that identifies the most serious web application vulnerabilities, the top three vulnerabilities in 2007 were: (1) Cross Site Scripting (XSS), (2) Injection Flaws (such as SQL Injection) and (3) Malicious File Execution (MFE). Even after it has been widely reported that web applications suffer from these vulnerabilities, the top two of the vulnerabilities were still listed in the top three of the OWASP's top ten list in 2010 and 2013. That is to say, in the past decade, even with the increased awareness about their importance due to OWAPS reports, these vulnerabilities continued to be widely spread in modern web applications, causing great damages.

An XSS vulnerability results from the application inserting part of the user's input in the next HTML page that it renders. Once the attacker convinces a victim to click on a URL that contains malicious HTML/JavaScript code, the user's browser will then display HTML and execute JavaScript that can result in stealing of browser cookies and other sensitive data. An SQL Injection vulnerability, on the other hand, results from the application's use of user input in constructing database statements. The attacker can invoke the application with a malicious input that is part of an SQL command that the application executes. This permits the attacker to damage or get unauthorized access to data stored in a database. Finally, MFE vulnerabilities occur if developers directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files.

One important observation is, all these vulnerabilities are caused by improper string manipulation. Programs that propagate and use malicious user inputs without sanitization or with improper sanitization are vulnerable to these well-known attacks.

In this work, we present *Patcher*, a new service platform for system developers patching and viewing vulnerabilities related to string manipulation in their web applications. Particularly, we incorporate the service with novel string analysis techniques that not

978-1-4799-2504-9/14 $31.00 © 2014 IEEE
DOI 10.1109/HICSS.2014.598

only check whether a web application is vulnerable to the types of attacks we discussed above, but also generate the corresponding patches that ensure the applications free from malicious exploits of identified vulnerabilities.

Patcher is a new online service that is open to public users. Users can access and upload their code to check potential vulnerabilities. Users can also insert patches that are automatically generated to prevent malicious exploits of their programs. While deploying new web services, it is essential to build the confidence on their security mechanisms. To the best of our knowledge, this is the first public online service that secures web applications using formal verification techniques.

Another advantage of our service is a user-friendly interface that aims at improving users' comprehension on where the program vulnerabilities are and how they get exploited and patched. Particularly, we develop an interactive 3D interface/presentation for users to access and view the risk status of their applications and vulnerabilities. The service provides users a clear view of vulnerabilities of target applications and a quick fix to reduce their risks. To sum up, we provide a new service platform for patching and viewing web application vulnerabilities, combining advance static string analysis techniques as the back end and visualization techniques as the front end. We believe this service would certainly reduce the risks of Web applications and improve their security quality.

The rest of this paper is organized as follows: We summarize previous work in string analysis and web application security detection and visualization in Section 2. We briefly introduce our string analysis techniques and the service architecture (as the back-end analysis engine) in Section 3. We discuss our design and implementation of visualization (as the front-end viewer) in Section 4. We report some analysis results against open-source Web applications in Section 5 and draw our conclusion in Section 6.

## 2. Literature Review

**String Analysis and Vulnerability Detection:** Due to its importance in security, string analysis has been widely studied. One influential approach has been grammar-based string analysis that statically computes an over-approximation of the values of string expressions in Java programs (Christensen et al., 2003), which has also been used to check for various types of errors in Web applications (Gould et al., 2004; Minamide 2005; Wassermann and Su, 2007 and 2008).

There are also several recent string analysis tools that use symbolic string analysis based on deterministic finite automata (DFA) encodings (Shannon et al., 2007; Fu et al., 2007; Yu et al., 2008). Some of them are based on symbolic execution and use a DFA representation to model and verify the string manipulation operations in Java programs (Shannon et al., 2007; Fu et al., 2007). HAMPI (Kiezun et al. 2009) is a bounded string constraint solver. It outputs a string that satisfies all the constraints, or reports that the constraints are unsatisfiable. Note that this type of bounded analysis cannot be used for sound string analysis whereas the string analysis techniques we adopt in this paper are sound.

Yu et al. (2008, 2010) have used single-track DFA based symbolic reachability analysis to verify the correctness of string sanitization operations in PHP programs. Their preliminary results on generating (non-relational) vulnerability signatures using single-track DFA were reported in a short paper (Yu et al., 2009).

All of the above results use single-track DFA and encode the reachable configurations of each string variable separately, i.e., they use a non-relational string analysis. Yu et al. reported the results on foundations of string analysis using multi-track automata (Yu et al., 2010). Multi-track automata read tuples of characters as input instead of only single characters. Each string variable corresponds to a particular track (i.e., a particular position in the tuple), thereby allowing a relational analysis. As demonstrated in (Yu et al., 2010), a relational analysis enables verification of properties that cannot be verified with these earlier approaches. However, relational string analysis can generate automata that are exponentially larger than the automata generated during non-relational string analysis. To tackle this problem, we also incorporate the abstraction techniques with the tool including alphabet abstractions and relation abstractions (Yu et al., 2011), which enable us to improve the performance of the relational string analysis by adjusting its precision. The earlier results on relational string analysis presented in (Yu et al., 2010) do not use any abstraction techniques.

It is critical that vulnerabilities are not only discovered fast, but they are also repaired fast. There has been previous work on automatically generating filters for blocking bad input (Costa et al., 2007). The work focuses on buffer-overflow vulnerabilities that are different than the string vulnerabilities we investigate here. In (Costa et al., 2007) the generation of filters is done starting with an existing exploit whereas we plan to start with an attack pattern instead. In (Yu et al., 2009), Yu et al. use single-track automata to generate the vulnerability signatures of the detected vulnerabilities in the Web applications. In addition to generate vulnerability signatures, Yu et al. use the vulnerability signatures to generate effective

sanitization statements. By applying their techniques, we are able to prove the absence of vulnerabilities in the applications that are patched with these statements. On the other hand, single-track automata are limited to model relations among variables. As shown in (Yu et al., 2009), this limitation makes the analysis to generate rather coarse vulnerability signatures, e.g., $\Sigma^*$ (any arbitrary string), for a vulnerability that can be exploited from multiple inputs. To tackle this problem, Yu et al. proposed a new approach to apply relational string analysis (Yu et al., 2010). They are able to generate more precise vulnerability signatures by catching the relations among inputs (Yu et al., 2011). We realize these approaches in Patcher, providing a new service to public users for patching and viewing their web application vulnerabilities. Compared to black-box security tools such as Wapiti (2008) and Netsparker (2013) that detects vulnerabilities via runtime testing, Patcher adopts static white-box analysis on source codes of applications and is able to ensure the correctness of patched programs.

**Program Visualization and Comprehension:** Visualization can help improve the comprehension of abstract program logics. Gammatella (Orso et al.,2004) uses the analogy with a traffic light to convey the concepts of danger, caution and safety with colors: red, yellow and green. Red represents the maximum value and green represents the minimum value. The way how colors are assigned to status depends on the view of targeting dimension. In Patcher, we also use colors to decorate vulnerable files and deliver the concept of risks of the whole applications.

Bohnet and Döllner (2006) developed the technique that provides the extraction of system architectures and dependencies between code components. The model includes the class-level model and the architectural-level model, and users can choose a scenario (a sequence of interactions between users and the system) that triggers specific executions. The visual layout separates nodes by the components such as functions and directories. It also provides the facility to quick access source codes by synchronizing the textual source code view with its graph exploration view. If users click the function shape or call relation, the corresponding source code will be loaded into the textual source code view area along with the selected code line highlighted. Vice versa, the corresponded shape in the graph exploration view will be highlighted when the source code line is selected. Later we will show its facilities cover program codes (level 0), data flows (level 1), dependency graphs (level 2) and architectural-level views (level 4) specified in Table 1.

Some recent works focus on how to present contents in mobile devices. SmartFoxServer (2013) supports the multi-platform technology to integrate the web servers with mobile applications and devices including techniques on Adobe Flash, Unity, iOS, Android, HTML5. It enables developers building an integrated multi-user platform for servers and clients. Ahmadi and Kong (2012) introduce an adaptive layout on the mobile screen based on advance visual analysis and structure analysis. It provides users a tool to customize their screen layout of mobile devices. Virpi et al. (2006) investigate the principles of content presentation in mobile devices. Gateway (Mackay, 2003) reduces the page scale for having users doing much less vertical and horizontal scrolling. Minimap (Virpi et al., 2006) adopts a novel visualization method that provides users suitable layouts of contents by listing the requirement of a good content presentation for mobile devices. It is better to fit more content in to the screen and eliminate the manipulation of horizontal scrolling. We also develop an app to present information in mobile devices.

Our implementation for visualization is on Unity. Unity is a game engine and IDE cross-platform that not only supports PC, Mac, Xbox 360 and Web servers but also mobile operation systems such as iOS and Android. The graphic engine of Unity can be incorporated with Direct3D (Windows), OpenGL (Mac, Windows), OpenGL ES (iOS, Android), and proprietary APIs (Wii). Using Unity, our visualization tool can be deployed to multiple platforms, as the remark from the Unity official "Author Once, Deploy Everywhere". Unity also supports integration with 3ds Max, Maya, Softimage, Blender, Cinema 4D, Photoshop, Adobe Fireworks. Changes that are made to the listed assets can be automatically updated in the Unity environment.

Compared to other security tools, we provide a new service for visualization of web application vulnerabilities. The interactive 3D environment provides users a unique experience on exploring web application vulnerabilities that traditional chart- or text-based summary presentation cannot offer.

## 3. Automata-based String Analysis and the Web Service

In this section, we present the back-end techniques that facilitate Patcher the ability to detect and patch vulnerabilities of string manipulating programs with respect to attack patterns. We conduct the static source code analysis according to the stages shown in Figure 1. The analysis takes two inputs: the source code of the web applications and attack patterns that characterize malicious strings for specific vulnerabilities. Users have to upload their programs to the service, which can

be a script or a whole application. We provide several attack patterns by default for detecting different vulnerabilities. We then perform taint analysis (Pixy, 2003) to identify sensitive functions in the (PHP)

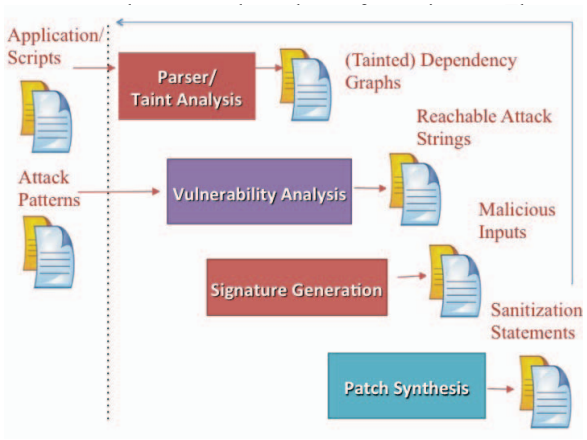


Figure 1. String Analysis Stage

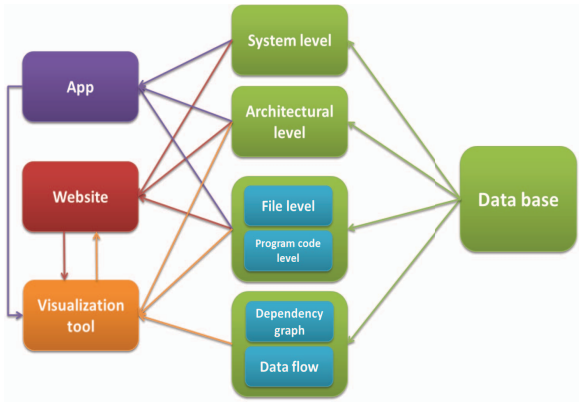


Figure 2. Architecture of the system

attack patterns. The intersection result identifies all reachable attack strings. The next stage is characterizing user inputs that can exploit the vulnerabilities, called vulnerability signatures. Depending on the number of input nodes, we conduct back ward reachability analysis on single-track automata to generate atomic signatures or forward reachability analysis on multi-track automata (Yu et al. 2010) to generate relational signatures. The later specifies the relations among multiple inputs, and presents a more precise characterization of malicious values of multiple user inputs. The final stage is to synthesize effective patches (Yu et al. 2011) that can be inserted in the right position of the programs so that user inputs that match vulnerability signatures (the characterized malicious inputs) can be identified and modified to avoid exploits during the program executions.

We have implemented the tool Patcher to realize the above static source code analysis. Patcher can automatically analyze PHP programs end-to-end without user interventions. We have also provided the web version of Patcher so that users can directly upload their code and view the results through the web pages. One can first write the script or upload a local file to Patcher. Patcher analyzes the script, detects potential vulnerability in the program, and generates its patches as the sanitization statement(s) with proper positions to insert. Developers can also upload a compressed file of the whole application as a package. Patcher will check all the execution entries of PHP scripts in the application automatically, and report and synthesize all vulnerabilities and their sanitization statements.

Figure 2 shows the architecture of Patcher that includes an app for mobile devices, a web site for the web service, and a display device for 3D graphic visualization. Each device deals with different level tasks and information described as below:

- **System level**: Users first login through the app or the web site. After passing the authentication, they can upload their applications for investigation or access previous analyzed analysis. The server keeps the list that includes all the applications that a user has uploaded with application names and features, performance data and status, and paths to access analysis results.
- **Architectural level:** Users can select an application in the list to view more details on how many files are vulnerable within the application. This can be done through the app or the website, and data will be represented based on the interface. The level information includes all the file names, index, and vulnerability counts within the application. Users can select a file through both the website and the app to drill down to the details of a file. The vulnerable files in the visualization tool will be display as a bee comb, where each file is represented a honey cell and colored in red, yellow, or green. The color indicates how vulnerable a file is (dangerous, moderate, or safe).
- **File and Program code level:** This level provides the information of single file and its source code and users can check the vulnerabilities in the file. In this level, users can exam each vulnerability in detail such as the sensitive function and where it is

(line of code), and how to patch it (where to insert the patch statement). The information can be presented in different formats. In the website, we create a source code editor so that users can directly modify the vulnerability or add the patch code in the source code. Patches are automatically generated. System developers can fix vulnerabilities without knowing much about the vulnerabilities or the codes. It is particularly useful to patch legacy parts of Web applications. On the other hand, users may want to know what vulnerabilities are and how they are raised and exploited. We generate an interactive dependency graph for each vulnerability to serve this purpose.

- **Data flow level:** Every dependency graph corresponds to one vulnerability in the file. Both app and web site can access it by the index of vulnerabilities and trigger the displace device to show the graph in 3D. Our server provides an interactive environment for engineers to trace program execution step by step or to run a simulation of a sample path to exploit the vulnerability. We also synchronize the dependency graph and source code, and highlight the corresponding line of the selected execution (node in the dependency graph) in the source code. That is, users can trace the source code when explore executions on the dependency graph.

Figure 3 shows a sample sequence of our web service in which we integrate our mobile device with the web service and visualization. First users can login and upload the application through the website, then view the analysis result from the mobile app or the website, and launch the visualization tool. When users choose an application from the list, the visualization tool shows the status graph (bee comb) of the application. Both the website and the app provide functions to view details of vulnerable files for users drilling down to exam vulnerabilities and their patches. By clicking a listed file, users can check its vulnerabilities and source code. For each vulnerability, users can click its Dependency Graph to generate the interactive dependency graphs to trace how values of input nodes flow to the sink node. To patch the vulnerability, users can insert the corresponding patch code at the identified line to sanitize user inputs.

## 4. Information Representation

In this section, we discuss our design of Patcher on how the information is represented in different abstraction levels. By retrieving and manipulating analysis results from the backend, we can visualize those data to users in a way for better program comprehension than data in traditional text formats.

| Abstraction level | Structure | Presentation component |
|---|---|---|
| 5(macroscopic) | Performance analysis | Website<br>mobile app |
| 4 | Architectural-level view of application | Visualizing tool<br>mobile app<br>Website |
| 3 | File-level representation | Visualizing tool<br>mobile app<br>Website |
| 2 | Dependency graph | Visualizing tool |
| 1(microscopic) | Data flow | Visualizing tool |
| 0 | Program code | Website<br>mobile app |

Table 1.Abstraction levels of the system

Pacione (2004) suggests that tools addressing software comprehension are supposed to support abstraction, structural and behavioral information, as well as the integration of statically and dynamically extracted data, separating the subject as six abstraction levels (shown in Table 1). Each level is a view with a name, a description and a set of diagrams that illustrate software at that level of that facet.

Our main purpose is to enhance the comprehension about the vulnerability and the program execution of applications or files. We represent the result analysis in six levels as proposed in (Pacione, 2004); each level provides analysis result of applications in different aspects. Table 1 lists the corresponding information that we provided at each level. Level 0, 1, and 2 provide detailed information regarding one vulnerability.

- **Level 0**: In this code level representation, the source code and lines related to a specific vulnerability is presented. Users can insert the generated patches to prevent malicious exploits of vulnerabilities.
- **Level 1:** In this exploit execution level representation, the simulation of a potential exploit on the dependency graph is provided. Users can

traverse how tainted data flows from user inputs to sensitive functions.

- **Level 2:** In the vulnerability structure level representation, an interactive dependency graph is presented with sink nodes, function nodes, and input nodes in different shapes. Users can check the dependency graph as a whole to reason the structure of the vulnerability.
- **Level 3:** In the file level representation, a list of vulnerabilities in a file is represented along with a brief summary on how many vulnerabilities exist for each kind. Each vulnerability also has a brief description on the variable, the sink location, the input location(s), etc. System developers can figure out what vulnerabilties exist in this file.
- **Level 4:** In the application level representation, a bee comb is used to show the risk status from the entire view of an application. Users can realize the security status overall by viewing the bee comb and choose most dangerous files to patch. A list of files with a brief summary on total number of vulnerabilities is also presented.
- **Level 5:** In the user level presentation, a list of applications that have been uploaded by the same user is presented along with a brief summary of status of each application.

Information in a large-scale view (higher level) gives users a clear view about high level components and the architecture. Information in a small-scale view that provides users detailed information from specific aspects. Our visualization tool provides browsing facilities to answer users' questions that are generally broad at beginning and then narrow down to specific issues. That is to say, users check the vulnerable file in applications when vulnerability is reported and then drill down to examine the source code to reveal how the vulnerability is raised. Program visualization also requests data from various sources to display different views as the dimension changes. It improves the comprehension to provide information from different platforms.

## 5. Evaluation

To evaluate our platform, we have uploaded various open source Web applications to Patcher. Table 2 summarizes the analysis results. There are ten applications that contain 3055 files in total. 2895 out of 3055 files have been analyzed successfully under 4000 seconds (1.3 second per file on average). The success rate is about 95%. Within these applications, Patcher reveals 2823 vulnerabilities. Most of them are previously unknown. The complete list can be found at http://soslab.nccu.edu.tw/patcher.

As for the performance issue, Patcher takes 1343 seconds to analyze *php-fusion-6-01-18* application and 993 seconds to analyze *moodle1_6* that has more files and vulnerabilities. We have taken a close look of vulnerabilities and found that the dependency graphs of *php-fusion-6-01-18* has larger number of nodes and may increase the analysis time of string analysis to find vulnerabilities and generate patches.

As for some graphic results, in the application *benchmarks*, we can find three vulnerable files (cells colored with yellow) that has one vulnerability in each of them. The dependency graph of the second vulnerability is complicated due to various method calls and string operations. As for the bee comb of *schoolmate*, we can see that the application has high security risk with half of its cells are colored in red. After taking a close look of the files, we found that most of its sensitive functions directly use user's inputs without any sanitization.

In sum, the preliminary result shows that Patcher is capable of analyzing large size Web applications and revealing previous unknown/known vulnerabilities, and generating effective patches to prevent these vulnerabilities been exploited. The visualization tool also enhances our understanding on program risks and structures.

## 6. Conclusion

We present a new web service and platform for patching web application vulnerabilities online. We adopt automata-based symbolic string analysis to perform static analysis on web applications, detecting potential severe string related vulnerabilities as well as generating effective patches. We also incorporate advance visualization and mobile techniques to enhance the service usability and program comprehension.

It is a new service for displaying web application vulnerabilities by incorporating advance visualization and static analysis techniques to enhance the service usability and program comprehension. Along with vivid graphics. analysis results can be viewed in different abstraction levels, representing information on needs. Managers of IT department can check the system level to evaluate the security status of whole application and make quick decisions. Engineers can modify and patch source codes by viewing vulnerabilities in more detailed file and source code levels.

| Application name | Total files | Success files | vulnerabilities | Analysis time |
|---|---|---|---|---|
| sendcard_3-4-1 | 72 | 72 | 21 | 25 sec |
| sanitized_schoolmate | 63 | 45 | 94 | 341 sec |
| servoo | 27 | 27 | 6 | 17 sec |
| schoolmate | 63 | 45 | 139 | 1078 sec |
| e107 | 218 | 212 | 0 | 0 sec |
| php-fusion-6-01-18 | 1156 | 1094 | 476 | 1343 sec |
| market | 22 | 22 | 0 | 20 sec |
| Nucleus3.64 | 71 | 68 | 24 | 52 sec |
| moodle1_6 | 1353 | 1300 | 2057 | 993 sec |
| benchmarks | 10 | 10 | 4 | 68 sec |

Table 2. The analysis and performance results against some open source web applications

Our web service is open to public. One of our ongoing works is to dig out more analysis data from back end servers, collecting more vulnerabilities in real-life Web applications with the aim of deriving more efficient detection and patching mechanisms. The front-end service can be further improved using visualization tools such as dashboard with bar charts, as well as enhancing program comprehension by extending graphic visualizations to more complicated structures such as flow graphs with conditions and relations among files and vulnerabilities.

## 6. Acknowledgement

This work is funded by the NSC grants: 99-2218-E-004-002-MY3 and 102-2221-E-004-002-.

## References

[1] Hamed Ahmadi and Jun Kong. User-centric adaptation of web information for small screens. Journal of Visual Languages and Computing ,Vol.23, No.1,pages 13-28, 2012.

[2] Johannes Bohnet, Stefan Voigt and Jürgen Döllner. Locating and understanding features of complex software systems by synchronizing time-, collaboration- and code-focused views on execution traces. In Proc. of the 16th IEEE International Conference on Program Comprehension, ICPC '08, pages 268-271, Amsterdam, The Netherlands, June 10-13, 2008.

[3] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In Proc. of the 21st ACM Symposium on Operating Systems Principles, SOSP '07, pages 117-130, Stevenson, Washington, USA, October 14-17, 2007.

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In Proc. of the 10th International Static Analysis Symposium, SAS '03, pages 1-18, San Diego, CA, USA, June 11-13, 2003.

[5] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. Associating the visual representation of user interfaces with their internal structures and metadata. In Proc. of the 24th Annual ACM Symposium on User Interface Software and

Technology, UIST '11, pages 245-256, Santa Barbara, CA, USA, October 16-19, 2011.

[6] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. Gliimpse:Animating from markup code to rendered documents and vice versa. In Proc. of the 24th annual ACM symposium on User interface software and technology, UIST '11, pages 245-256, Santa Barbara, CA, USA, October 16-19, 2011.

[7] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting sql injection vulnerabilities. In Proc. of the 31st Annual International Computer Software and Applications Conference, COMPSAC '07, pages 87-96, Beijing, China, , July 24-2, 2007.

[8] gotoAndPlay(). Smartfoxserver @ONLINE, http://www.smartfoxserver.com/. Jan. 2013.

[9] Paul A. Gross and Caitlin Kelleher. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. Journal of Visual Languages and Computing, Vol. 21 No. 5, pages263-276, December 2010.

[10]Carl Gould, Zhendong Su and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In Proc. of the 26th International Conference on Software Engineering, ICSE '04, pages 645-654, Edinburgh, United Kingdom, May 23-28, 2004.

[11] Paul A. Gross, Jennifer Yang, and Caitlin Kelleher. Dinah: an interface to assist non-programmers with selecting program code causing graphical output. In Proc. of the International Conference on Human Factors in Computing

Systems, CHI '11, pages 3397-3400, Vancouver, BC, Canada, May 7-12, 2011.
[12] Liviu Iftode, Cristian Borcea, Nishkam Ravi, Porlin Kang, and Peng Zhou. Smart phone: An embedded system for universal interactions. In Proc. of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems FTDCS '04, pages 88-94, Suzhou, China, May 26-28, 2004.
[13] James A. Jones, Mary Jean Harrold and John Stasko. Visualization of test information to assist fault localization. In Proc. of the 24th International Conference on Software Engineering, ICSE '02, pages 467-477, New York, NY, USA, May 19-25, 2002.
[14] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer and Michael D. Ernst. Hampi: a solver for string constraints. In Proc. of the 18th International Symposium on Software Testing and Analysis ,ISSTA '09, pages 105-116, Chicago, IL, USA, July 19-23, 2009
[15] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann and Jan Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In Proc. of the 24th annual ACM symposium on User interface software and technology, UIST '11, pages 217-224, New York, NY, USA, October 16-19, 2011.
[16] Bonnie MacKay. The gateway: a navigation technique for migrating to small screens. In the Proc. of Extended abstracts of the 2003 Conference on Human Factors in Computing Systems, CHI '03, pages 684-685, Ft. Lauderdale, Florida, USA, April 5-10, 2003.
[17] Mavitunasecurity. Netsparker @ONLINE,May. 2013.
[18] Alessandro Orso, James A. Jones, Mary Jean Harrold, and John T. Stasko. Gammatella: Visualization of program-execution data for deployed software. In the Proc. of 26th International Conference on Software Engineering, ICSE '04, pages 699-700, Edinburgh, United Kingdom, May 23-28, 2004.
[19] Michael J. Pacione. Software visualization for object-oriented program comprehension. In Proc. of the 26th International Conference on Software Engineering, ICSE '04, pages 63-65, Edinburgh, United Kingdom, May 23-28, 2004.
[20] Virpi Roto, Andrei Popescu, Antti Koivisto, and Elina Vartiainen.Minimap: a web page visualization method for mobile phones. In the Proc. of the 2006 Conference on Human Factors in Computing Systems, CHI '06, pages 35-44, Montreal, Quebec, Canada, April 22-27, 2006.
[21] Michael Risi and Giuseppe Scanniello. Metricattitude: a visualization tool for the reverse engineering of object oriented software. In the Proc. of International Working Conference on Advanced Visual Interfaces, AVI '12, pages 449-456, Capri Island, Naples, Italy, May 22-25 2012.
[22] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In the Proc. of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, TAICPART-MUTATION '07, pages 13-22, Washington, DC, USA, September10-14 2007.
[23] Tarja Systä, Kai Koskimies and Hausi A. Müller. Shimba -an environment for reverse engineering java software systems. Journal of Software: Practice and Experience, Vol.31 No.4, pages 371-394, 2001.
[24] Unity Technologies. Unity documentation @ONLINE,Jan. 2013.
[25] Standford University. iPhone application development @ONLINE, Jan. 2013.
[26] Nicolas Surribas. Wapiti @ONLINE,May. 2013.
[27] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In Proc. of the 24th IEEE/ACM International Conference on Automated Software Engineering ASE '09, pages 605-609, Auckland, New Zealand, November 16-20, 2009.
[28] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In Proc. of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'10, pages 154-15, Paphos, Cyprus, March 20-28, 2010.
[29] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching vulnerabilities with sanitization synthesis. In Proc. of the 33rd International Conference on Software Engineering, ICSE '11, pages 251-260, Waikiki, Honolulu, HI, USA, May 21-28, 2011.
[30] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In Proc. of the 15th International SPIN Workshop on Model Checking Software, SPIN '08, pages 306-324 Los Angeles, CA, USA, August 10-12, 2008.
[31] Fang Yu, Tevfik Bultan, and Ben Hardekopf. String abstractions for string verification. In Proc. of the 15th International SPIN Workshop on Model Checking Software, SPIN '11, pages 20-37, Snowbird, UT, USA, July 14-15, 2011.
[32] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Relational string verification using multi-track automata. In Proc. of the 15th International Conference on Implementation and Application of Automata, CIAA '10, pages 290-299, Winnipeg, MB, Canada, August 12-15, 2010. [1] A.B. Smith, C.D. Jones, and E.F. Roberts, “Article Title”, *Journal*, Publisher, Location, Date, pp. 1-10.

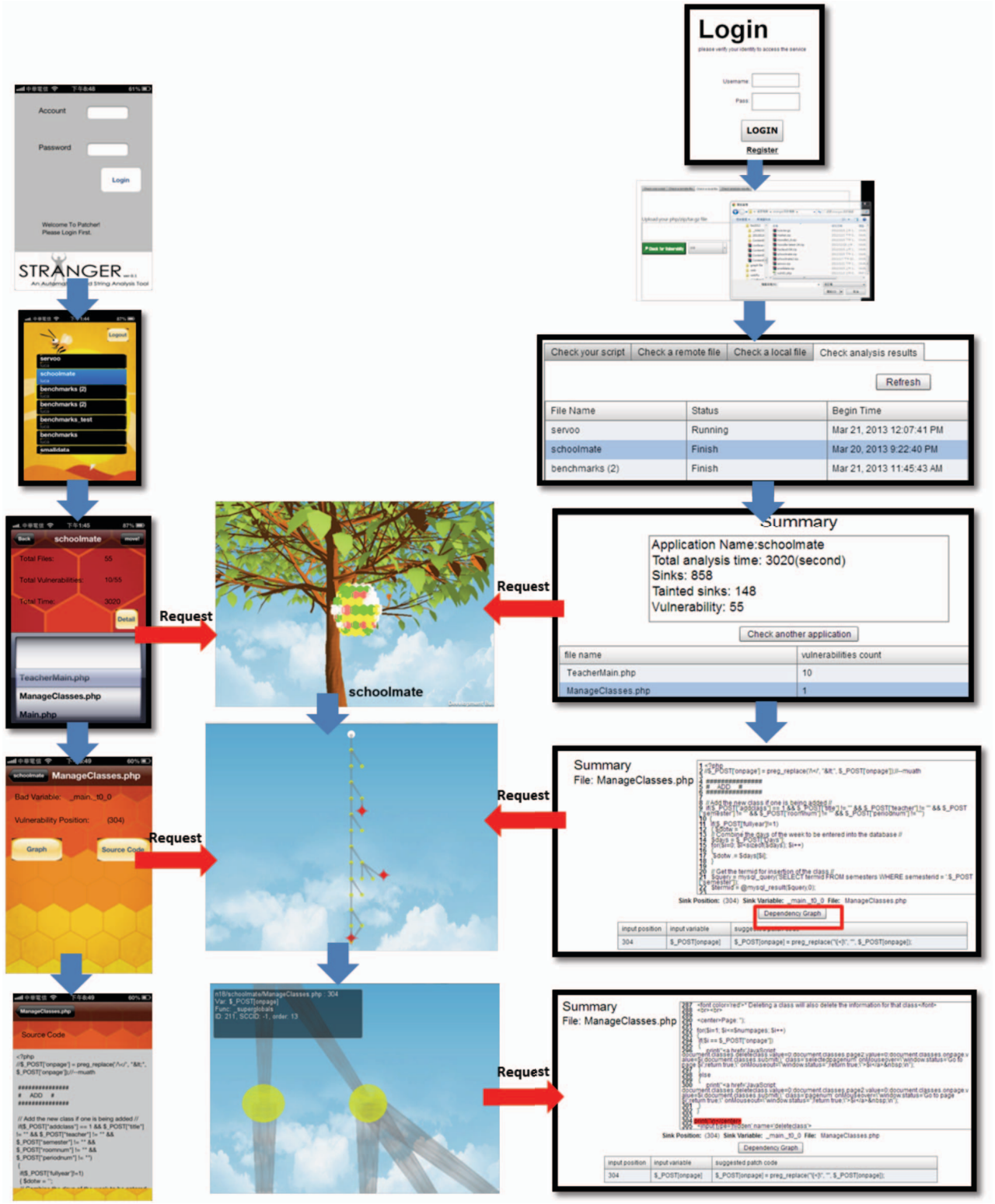


Figure 3. A sample sequence of execution and view of Patcher