Improving Web Security by
Automated Extraction of
Web Application Intent

BY
PRITHVI PAL SINGH BISHT
M.Tech., Indian Institute of Technology, Kanpur, India, 2002
B.E., Govind Ballabh Pant Engineering College, Pauri, India, 2000

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2011

Chicago, Illinois

Defense Committee:
Arvind Prasad Sistla
Jon Solworth
Mark Grechanik, Accenture Labs Chicago
Ugo Buy
Venkat Venkatakrishnan, Chair and Advisor
Copyright by

Prithvi Pal Singh Bisht

2011
For my Eja and Babu
ACKNOWLEDGMENTS

Completion of this dissertation has been made possible by support, advice, encouragement and love of many people. I take this opportunity to express my sincere gratitude to all of them.

My babu (father), Shri Nandan Singh Bisht, and my late eja (mother), Smt Malati Devi, instilled love for engineering from early days of my childhood. They nurtured and facilitated different aspects of my life and career through thick and thin. To them I dedicate this work.

I am indebted to my wife, Bhavana, who provided unrelenting support, guidance and unconditional love during the last leg of this thesis. My brothers, sisters and their spouses always provided encouragement that helped me in staying focused and grounded. All my friends, Ayeshwarya Mahajan in specific, guided me through tough times.

Professor V.N. Venkatakrishnan, my Ph.D. advisor, played a pivotal role throughout my stint as a doctoral candidate. Starting from early e-mail exchanges when I was exploring graduate studies at UIC from India, to the toast after my defense, he has been a true mentor, keen researcher, resourceful facilitator and a friend. He has helped me hone my skills not only in the research front but also in various practical aspects of life.

Working with Professor Arvind Prasad Sistla has been a real treat for me. He not only helped shape some key aspects of this work but, in doing so, also taught ways to reason about difficult problems in a systematic and rigorous fashion. The other members of my dissertation committee, Professor Ugo Buy, Professor Mark Grechanik and Professor Jon Solworth, provided extremely valuable and timely feedback that helped shape contents and style of this thesis.

I also had the good fortune to collaborate with the following intelligent and resourceful researchers during this thesis: Tim Hinrichs, Professor Madhusudan Parthasarthy, Mike Ter Louw, Sruthi Bandhakavi, Kalpana Gondi, Nazari Skrupsky, Radoslaw Bobrowicz, Michelle Zhou, Megha
ACKNOWLEDGMENTS (Continued)

Chauhan, Tanushree Varshaneya. Through insightful feedback and discussions these researchers have left a long lasting impression on various aspects of my research skills.

During the first year of my Ph.D. studies, I worked with Professor Ugo Buy and Professor Tanya Berger-Wolf as a teaching assistant. I have constantly been in touch with them since then and it greatly helped me in settling down in the graduate life.

Phillip Porras and Vinod Yegneswaran mentored my internship at SRI International Lab and enabled expansion of my research skills as well as helped me gain a broader perspective of security research.

Last but not the least, the UIC computer science department provided support in various ways that I am grateful for.

My research experience has been enriched and sweetened by involvement of all the above individuals. Thank you, everyone!

The author gratefully acknowledges financial support received from the following funding agencies during various phases of this dissertation research: National Science Foundation grants CNS-0845894, CNS-1065537, CNS-0917229, CNS-0716584, and University of Illinois at Chicago Campus Research Board grant UIC-CRB-2-200250-699000-699012. The equipment used for this research was funded in part by National Science Foundation grant CRI-0551660.

PPSB
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Security Problems in Legacy Applications</td>
<td>8</td>
</tr>
<tr>
<td>1.2 Thesis Goals</td>
<td>11</td>
</tr>
<tr>
<td>1.2.1 Summary of Implementation and Results</td>
<td>14</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>17</td>
</tr>
<tr>
<td>2.1 Running Example: Shopping Cart</td>
<td>17</td>
</tr>
<tr>
<td>2.2 SQL Injection Attacks</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1 SQL Injection Attack Example</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2 SQL Injection Attack Vulnerability</td>
<td>22</td>
</tr>
<tr>
<td>2.2.3 In Scope</td>
<td>23</td>
</tr>
<tr>
<td>2.3 Parameter Tampering Attacks</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Parameter Tampering Attack Example</td>
<td>24</td>
</tr>
<tr>
<td>2.3.2 Parameter Tampering Attack Vulnerability</td>
<td>25</td>
</tr>
<tr>
<td>2.3.3 In Scope</td>
<td>25</td>
</tr>
<tr>
<td>2.4 Common Attributes of Vulnerable Applications</td>
<td>26</td>
</tr>
<tr>
<td>2.4.1 Revisiting SQL Injection Attack</td>
<td>26</td>
</tr>
<tr>
<td>2.4.2 Revisiting Parameter Tampering Attack</td>
<td>27</td>
</tr>
<tr>
<td>3 APPROACH OVERVIEW</td>
<td>29</td>
</tr>
<tr>
<td>3.1 Intentions</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1 Observations</td>
<td>30</td>
</tr>
<tr>
<td>3.2 High Level Approach</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Challenges</td>
<td>35</td>
</tr>
<tr>
<td>4 DETECTION OF PARAMETER TAMPERING VULNERABILITIES</td>
<td>40</td>
</tr>
<tr>
<td>4.1 Motivation</td>
<td>40</td>
</tr>
<tr>
<td>4.2 Contributions</td>
<td>43</td>
</tr>
<tr>
<td>4.3 Problem Formulation</td>
<td>44</td>
</tr>
<tr>
<td>4.3.1 Problem Description</td>
<td>44</td>
</tr>
<tr>
<td>4.4 NoTamper</td>
<td>46</td>
</tr>
<tr>
<td>4.4.1 Overview</td>
<td>46</td>
</tr>
<tr>
<td>4.4.2 Architecture and Challenges</td>
<td>48</td>
</tr>
<tr>
<td>4.4.3 Algorithms and Implementation</td>
<td>55</td>
</tr>
<tr>
<td>4.5 WAPTEC</td>
<td>68</td>
</tr>
<tr>
<td>4.5.1 Overview</td>
<td>68</td>
</tr>
<tr>
<td>4.5.2 Architecture and Challenges</td>
<td>76</td>
</tr>
<tr>
<td>4.5.3 Algorithms and Implementation</td>
<td>84</td>
</tr>
<tr>
<td>4.6 Evaluation</td>
<td>91</td>
</tr>
<tr>
<td>4.6.1 NOTAMPER</td>
<td>91</td>
</tr>
<tr>
<td>4.6.2 WAPTEC</td>
<td>99</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>Summary</td>
</tr>
<tr>
<td>5</td>
<td>PREVENTION OF SQL INJECTION VULNERABILITIES</td>
</tr>
<tr>
<td>5.1</td>
<td>Motivation</td>
</tr>
<tr>
<td>5.2</td>
<td>Contributions</td>
</tr>
<tr>
<td>5.3</td>
<td>Running Example</td>
</tr>
<tr>
<td>5.4</td>
<td>Approach Overview</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The Transformation Problem</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Main Assumption</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Intuitions Behind Our Approach</td>
</tr>
<tr>
<td>5.5</td>
<td>Algorithms and Implementation</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Handling Straight Line Programs</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Handling of Conditionals and Procedures</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Loop Handling</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Implementation</td>
</tr>
<tr>
<td>5.6</td>
<td>Evaluation</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Effectiveness</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Performance Experiment</td>
</tr>
<tr>
<td>5.7</td>
<td>Summary</td>
</tr>
<tr>
<td>6</td>
<td>DISCUSSION</td>
</tr>
<tr>
<td>6.1</td>
<td>Goal</td>
</tr>
<tr>
<td>6.2</td>
<td>Intuition for Patching</td>
</tr>
<tr>
<td>6.3</td>
<td>Challenges</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Patching the Vulnerable Path</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Incorporating Server-side Changes to Inputs</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Factoring in Dynamism in Form Generation</td>
</tr>
<tr>
<td>7</td>
<td>RELATED WORK</td>
</tr>
<tr>
<td>7.1</td>
<td>Vulnerability Detection Techniques</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Static Analysis Based Techniques</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Dynamic Analysis Based Techniques</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Test Input Generation</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Discussion</td>
</tr>
<tr>
<td>7.2</td>
<td>Vulnerability Prevention Techniques</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Learning Based Whitebox Defenses</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Learning Based Blackbox Defenses</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Discussion</td>
</tr>
<tr>
<td>7.3</td>
<td>Specification Extraction for Security</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Discussion</td>
</tr>
<tr>
<td>7.4</td>
<td>New Paradigms</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.1 Developing Secure Applications</td>
<td>159</td>
</tr>
<tr>
<td>7.4.2 New Browser Platforms</td>
<td>160</td>
</tr>
<tr>
<td>7.4.3 Prevention Architectures</td>
<td>161</td>
</tr>
<tr>
<td>7.4.4 Discussion</td>
<td>161</td>
</tr>
<tr>
<td>8 CONCLUSION</td>
<td>164</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>167</td>
</tr>
<tr>
<td>VITA</td>
<td>181</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Evolution of “points of attacks” in the last 30 years (Source: Trustwave SpiderLabs)</td>
<td>2</td>
</tr>
<tr>
<td>II</td>
<td>Underground economy: Top five goods and services available for sale and requested (Source: Symantec Corporation)</td>
<td>6</td>
</tr>
<tr>
<td>III</td>
<td>NoTamper: Language of generated formulas</td>
<td>56</td>
</tr>
<tr>
<td>IV</td>
<td>NoTamper: Constraints imposed by HTML form controls</td>
<td>57</td>
</tr>
<tr>
<td>V</td>
<td>NoTamper: The reduced constraint language: ( \land ) and ( \lor ) over the above atoms. ( \otimes ) is one of (&lt;,), (\leq), (\geq). (\oplus) is either (\in) or (\notin).</td>
<td>63</td>
</tr>
<tr>
<td>VI</td>
<td>WAPTEC constraint language</td>
<td>77</td>
</tr>
<tr>
<td>VII</td>
<td>WAPTEC: Trace generation transformation</td>
<td>87</td>
</tr>
<tr>
<td>VIII</td>
<td>NoTamper analyzed 8 open source applications and 5 live websites</td>
<td>92</td>
</tr>
<tr>
<td>IX</td>
<td>Summary of NoTamper results (Opportunities:169, Examined: 50, Confirmed exploits: 9, False Positives: 43)</td>
<td>93</td>
</tr>
<tr>
<td>X</td>
<td>Details of NoTamper results</td>
<td>97</td>
</tr>
<tr>
<td>XI</td>
<td>Summary of WAPTEC results</td>
<td>100</td>
</tr>
<tr>
<td>XII</td>
<td>Comparing WAPTEC and NoTamper results</td>
<td>104</td>
</tr>
<tr>
<td>XIII</td>
<td>Additional WAPTEC results</td>
<td>106</td>
</tr>
<tr>
<td>XIV</td>
<td>TAPS: Effectiveness suite applications</td>
<td>133</td>
</tr>
<tr>
<td>XV</td>
<td>Effectiveness suite applications, transformed SQL sinks and control flows: TAPS transformed over 93% and 99% of the analyzed control flows for the two largest applications</td>
<td>134</td>
</tr>
<tr>
<td>XVI</td>
<td>TAPS: Transformation altered less than 5% lines for large applications</td>
<td>135</td>
</tr>
<tr>
<td>XVII</td>
<td>TAPS: The largest application was transformed in approximately two hours and the rest in less than an hour</td>
<td>137</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running example of a shopping application.</td>
</tr>
<tr>
<td>2</td>
<td>High level approach: step 1) extract intentions encoded in the source code, step 2a) use intentions to construct concrete exploits or step 2b) enforce them to prevent vulnerabilities.</td>
</tr>
<tr>
<td>3</td>
<td>NoTamper end-to-end architecture and application.</td>
</tr>
<tr>
<td>4</td>
<td>WAPTEC end-to-end architecture and application.</td>
</tr>
<tr>
<td>5</td>
<td>NoTamper: Graph illustrating the importance of hostile input ranking, with bold triangles denoting thresholds used.</td>
</tr>
<tr>
<td>6</td>
<td>TAPS: step (1) generates symbolic queries, steps (2-3) separate data reaching the queries, step (4) removes data from symbolic queries, and step (5) generates the transformed program. ($n$ in this figure represents $__POST[_name]$).</td>
</tr>
<tr>
<td>7</td>
<td>TAPS: Labeled derivation tree for symbolic values of $q$ after execution of statement 6.</td>
</tr>
</tbody>
</table>
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>Web</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SQLIA</td>
<td>SQL Injection Attacks</td>
</tr>
<tr>
<td>CVE</td>
<td>Common Vulnerabilities and Exposures</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>WASC</td>
<td>Web Application Security Consortium</td>
</tr>
<tr>
<td>SDLC</td>
<td>Software Development Life Cycle</td>
</tr>
<tr>
<td>TAPS</td>
<td>Tool for Automatically Preparing SQL queries</td>
</tr>
<tr>
<td>WAPTEC</td>
<td>Whitebox Analysis for Parameter Tampering Exploit Construction</td>
</tr>
<tr>
<td>SDGs</td>
<td>System Dependency Graphs</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines Of Code</td>
</tr>
<tr>
<td>JS</td>
<td>JavaScript</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>AJAX</td>
<td>Asynchronous JavaScript and XML</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DNF</td>
<td>Disjunctive Normal Form</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus Naur Form</td>
</tr>
<tr>
<td>APIs</td>
<td>Application Programming Interfaces</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP Hypertext Preprocessor</td>
</tr>
<tr>
<td>PHC</td>
<td>PHP Compiler</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>UIC</td>
<td>University of Illinois at Chicago</td>
</tr>
<tr>
<td>CAPTCHA</td>
<td>Completely Automated Public Turing test to tell Computers and Humans Apart</td>
</tr>
<tr>
<td>XSS</td>
<td>Cross-Site Scripting</td>
</tr>
<tr>
<td>OWASP</td>
<td>Open Web Application Security Project</td>
</tr>
<tr>
<td>IIS</td>
<td>Internet Information Services</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>WAF</td>
<td>Web Application Firewall</td>
</tr>
</tbody>
</table>
SUMMARY

Over the past decade, the Web has been transformed from a collection of static HTML pages to a complex, distributed computing platform, as evidenced by the success of sites such as Facebook and YouTube. This transformation has been enabled primarily by web applications. The goal of this thesis is to investigate fundamental ways to improve the security of existing (legacy) web applications. To do so, we pursue research efforts in two complementary directions: a) techniques to uncover security flaws and b) techniques to automatically fix security flaws.

Finding and fixing security flaws in a legacy web application typically requires detailed knowledge of its behavior. This knowledge is a result of understanding high-level design artifacts combined with an analysis of the source code of the web application. However, it is well known that manual effort spent towards analysis of the source code is labor and cost-intensive and is often error-prone. Additionally, design level artifacts are often unavailable for legacy web applications and the only available resource is the source code. While source code is the most accurate description of the behavior of a web application, this description is expressed in low-level program statements. Due to its inherent low-level nature, source code does not readily offer a high-level understanding of an application’s intended behavior which is necessary to identify and fix security flaws.

This thesis develops techniques to compute the high-level intended behavior of a legacy web application directly from its low-level source code description. The philosophy of discovering intent in order to detect vulnerabilities and prevent attacks rests on two simple observations: (a) web applications are written implicitly assuming benign inputs, and encode programmer intentions to achieve a certain behavior on these inputs, and (b) maliciously crafted inputs subvert the program into straying away from intended behaviors, leading to successful attacks. Leveraging on these
SUMMARY (Continued)

observations we develop techniques for inferring intentions in the realms of uncovering security flaws as well as fixing them. Through two practical results, we demonstrate that this philosophy of inferring intent is a powerful one, and is broadly applicable to addressing challenges in web application security.

The first result in this thesis presents a systematic approach for detection of parameter tampering vulnerabilities. These vulnerabilities arise in form processing code when the server-side fails to re-validate inputs that were rejected by the corresponding client-side validation. To detect vulnerabilities, our approach systematically explores the space of inputs that violate intended restrictions to find those that the server-side code fails to enforce. Evaluation of several open source and commercial web applications reveals serious security problems such as unauthorized monetary transactions at a bank and unauthorized discounts added in a shopping session. These results provide a strong evidence that extracting and checking intended behaviors, offers an effective mechanism for reasoning about vulnerabilities in web applications.

The second result in this thesis offers a sound approach to prevent SQL injection vulnerabilities. These vulnerabilities arise when an application fails to restrict the influence of untrusted inputs on SQL queries. This approach first extracts web application intended SQL queries by analyzing its source code. Our strategy for fixing vulnerable web applications involves rewriting the source code to employ \texttt{PREPARE} statements, one of the well known robust defenses for SQL injection attacks. Experimental evaluation demonstrates effectiveness and scalability of our approach by successfully transforming large open source applications. Our approach presents a robust solution to the long standing problem of incorporating \texttt{PREPARE} statements in legacy web applications.
SUMMARY (Continued)

The philosophy of extracting and using intentions offers a systematic and scalable way to combat security problems in legacy web applications. By presenting extensive results on both detection and prevention fronts, this thesis offers convincing evidence that reasoning of application intent enables development of principled approaches for improving security of web applications.
CHAPTER 1

INTRODUCTION

The World Wide Web (WWW) has become an important day-to-day resource for the people of this planet. According to the Internet usage statistics trends, as of Jun 2010, the number of Web users has raised to 1.9 billion, roughly a fourth of the world population (1). In addition to being the primary source of information for a majority of these users, the Web is being increasingly used for providing everyday services such as health care, education, banking and electronic commerce. Due to the importance of these services, the Web has attracted participation from a diverse populace, regardless of barriers of age, gender, or geography.

This growth that has fueled the Web has already enabled a trend of using customized web applications to offer these services. Driven by customized web applications, highly popular social networking sites such as MySpace and Facebook, with exciting and interactive user driven content (such as blog, wiki entries and YouTube videos), are becoming the norm for web content rather than the exception. The growth of these sites has been fueled by highly attractive revenue models and business opportunities from advertising. As a result, we are moving away from a Web that mainly had static HTML pages to a Web which contains customized and feature rich pages laden with end-user supplied content. Web application platforms such as PHP and JSP have enabled this transition by facilitating application developers with rapid prototyping facilities that have eased the creation of new web applications, thus leading to the trend of widespread deployment of web applications.
Several recent studies (2, 3) indicate that this popularity of web applications has unfortunately also attracted attention of attackers. The web security firm Trustwave reviewed (2) vulnerabilities of the last 30 years to illustrate how “points of attack” have evolved, and the scope of the security problems we face today. We summarize findings of this report in Table I. For each decade shown in column 1, column 2 lists the key points of attacks. For each point of attack, the last column shows the top vulnerability. As suggested by this table, SQL injection attacks (SQLIA) – one of the top threats to web applications – was ranked as one of the most potent vulnerabilities for the last 10 years. Similarly, the web security firm Symantec reported (3) a 93% increase in the volume of Web-based attacks in 2010 over the volume observed in 2009. In summary, these reports suggest that web applications are one of the most attractive targets for attackers today.

The above studies are strongly supported by several widely publicized attacks on web applications that include: the 2011 LulzSec SQLIA (4) on Sony website that compromised over 1 million user accounts, the 2011 Lizamoon SQLIA (5) that has infected over 1 million websites and is still active, the 2010 mass SQLIA (6) that compromised hundreds of thousands of web applications

<table>
<thead>
<tr>
<th>Decade</th>
<th>Points of Attacks</th>
<th>Top Vulnerability (Potential for Damage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980s</td>
<td>Physical Access</td>
<td>Sensitive data left in plain view</td>
</tr>
<tr>
<td>1990s</td>
<td>Network Access</td>
<td>Weak passwords</td>
</tr>
<tr>
<td>2000s</td>
<td>E-mail, Applications, Wireless</td>
<td>SQL injection attacks</td>
</tr>
</tbody>
</table>

TABLE I

Evolution of “points of attacks” in the last 30 years (Source: Trustwave SpiderLabs).
running on Microsoft Internet Information Services (IIS) servers, the SQLIA based 2009 Heartland (7) and 2006 CardSystems (8) data breaches that compromised hundreds of millions of debit / credit card records and the 2005 cross-site scripting based MySpace Samy worm (9) that infected more than a million accounts within the first 20 hours of infecting the MySpace social networking website. In summary, these well known attacks firmly establish the risks posed by vulnerable web applications.

The rise in attacks on web applications can be attributed to the following key reasons.

- **Users as Exploit Propagators.** Several reports suggest that the lack of security awareness among web application users is being exploited to propagate attacks on popular websites (3,10, 11). As web applications are attracting a diverse populace, not all users of these applications are security-savvy. As a result, attackers are inventing social engineering based attacks that trick such users into divulging sensitive information or propagating malicious code (10, 11). Symantec reported (3) that such social engineering attacks are on the rise and often utilize information available online to make such attempts resemble legitimate requests. In summary, security unawareness of users is contributing towards faster propagation of attacks.

- **Stepping Stones.** Attacks on web applications are being used as “stepping stones” i.e., they are being used to enable further attacks (12, 13, 14). A common property of vastly successful attacks is that they use SQL injection attack as an injection mechanism in multi-staged attacks (12,13). Specifically SQLIA are used to inject malicious JavaScript or HTML payloads in databases of legitimate websites. As compromised databases are used in generating HTML pages, these malicious payloads are rendered in a user’s browser. Subsequently, these pages redirect the user’s browser to malicious servers that serve attacks relevant to
the user’s browser. The two well studied mass SQLIA, namely the robint.us (12) and the Lizamoon mass SQLIA (13) plant malicious JavaScript URLs in legitimate databases of popular websites. These malicious JavaScript URLs are subsequently embedded in legitimate web pages generated by these websites. When rendered in browsers, the robint.us JavaScript URL launched a zero-day exploit to execute shell code in the flash plugin whereas the Lizamoon JavaScript URL tricks the user into installing a scareware that disables legitimate Windows defender software on PCs. In both these cases, SQL injection attacks are being primarily used to facilitate further exploitation. Similarly, attacks on web applications have been used as the starting points to setup phishing web sites or to launch and control a botnet (14). By serving as potent starting points for multi-staged attacks on software installed on users’ machines, attacks on web applications provide convenient stepping stones for attackers.

- **Evolution of Attack Toolkits.** Attack toolkits are becoming the primary means of attacking web applications (3, 15, 16, 17, 18). Some of the most popular attack toolkits often feature easy-to-use interfaces and do not require any technical abilities to create an actual exploit (15). Further, these toolkits provide capabilities such as conducting Denial-of-Service attacks, spamming and phishing campaigns, and finding exploitable web applications (16). Thus these toolkits are reducing efforts and skills required on the part of attackers. Symantec reported (16) that toolkits are being sold on certain forums and recently researchers at the web security firm Seculert found that such toolkits are now being offered in the form of Software as a Service (SaaS) (17). Furthermore, these toolkits are evolving and offer features such as upgrades, patches, online support and warranties (18). Symantec attributes 93% increase in volume of attacks on web applications in 2010 over the volume in 2009 to the use of attack
toolkits (3). In summary, the attack toolkits are automating the entire hacking process thus providing a potent means of attacking web applications.

• **Inadequate Web Application Development Process.**

The web application development process fails to incorporate security measures at various levels (19, 20). The security firm Veracode studied (19) security practices involved in developing more than 5000 applications over the last 18 months and the security firm Ponemon surveyed (20) more than 600 developers with an average of 11 years of experience in their profession. These two surveys report widespread problems in the web application development process. First, insignificant budgets are allotted to incorporate security measures into applications thus showing a lack of consideration at the highest level e.g., 88% of respondents suggested that their web application security budget was less than the organization’s coffee budget (20). The lack of considerations in the initial phases such as budgeting then trickles down to all subsequent phases of software development e.g., the same study reported that 20% of surveyed developers do not test and 40% test only 5% of their web applications. The extrapolated average for all web applications that are being tested by organizations in that study was estimated to be 13%. One of the main reasons sighted for not testing their web applications is the lack of budget (20). Second, the provisions for security training seem inadequate e.g., the Veracode study (19) found 80% of applications to be suffering from well documented top vulnerabilities in the OWASP top 10 list (21). The OWASP top 10 list not only describes these vulnerabilities but also suggests preventive measures that developers can take to avoid them. Failing to prevent widely publicized known vulnerabilities shows a lack of awareness. Further, developers lack proper background to develop secure web applications.
<table>
<thead>
<tr>
<th>Goods and Services</th>
<th>Range of Prices</th>
<th>Rank for Sale</th>
<th>Rank Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank account credentials</td>
<td>$10 - $1,000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Credit cards with CVV2 numbers</td>
<td>$0.50 - $12</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Credit cards</td>
<td>$0.10 - $25</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Email addresses</td>
<td>$0.30/MB - $40/MB</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Email passwords</td>
<td>$4 - $30</td>
<td>5</td>
<td>14</td>
</tr>
</tbody>
</table>

**TABLE II**

Underground economy: Top five goods and services available for sale and requested (Source: Symantec Corporation).

Over 50% of developers taking an application security fundamentals exam received a grade of C or lower (less than 79 marks) whereas over 30% received a failing grade of D (60 to 69 marks) or F (less than 60 marks) (19). The exam covers knowledge of broad security concepts, including common threats. In summary, widespread failure to incorporate security measures into web application development process leads to web applications with “ripe and low hanging” vulnerabilities.

- **Underground Economy of Attacks.** Several recent studies suggest that attacks on web applications are fueling a profit driven system (16, 22, 23, 24, 25, 26). This system, termed as underground economy, offers commodities such as attack toolkits, exploits and vulnerabilities, stolen information, and bandwidth in bots. Each of these commodities offers variety of ways for attackers to make profit. Most popular attack toolkits such as the Neon, the Luckysploit Pack, the Phoenix exploit kit and the Sploit 25 Pro are being sold for prices ranging from
hundreds of dollars to a few thousand dollars (22). Attack toolkits in turn enable buyers to find exploits in web applications. An attacker can resort to selling exploits or could actually exercise these exploits to steal information from web applications. Selling exploits has been found to fetch anywhere from hundreds of dollars to thousands of dollars, legally (25) or illegally (26) whereas selling information can fetch anywhere from tens of dollars to thousands of dollars (16). Symantec analyzed (16) prices offered for stolen information in certain forums and Table II provides the summary of their findings. The first column shows available information which includes bank account details and credit cards with CVV numbers, the second column shows offered prices and the last two columns show ranks of supply and demand for the stolen information. Bots offer several avenues for making profit: computers on a bot can be rented out for under 6 pounds per hour (23), or can be used in conducting large scale Denial-of-Service attacks to extort money from online businesses. The latter has been reported to be specially profitable as reports suggest that it has been used to extort tens of thousands of dollars from online gambling and other websites (24). In summary, these reports suggest that exploiting web applications is highly profitable.

The above stated reasons are inter-related and help explain the rise in attacks on web applications. First, the underground economy offers a variety of ways for making profit from attacking web applications. Further, as attacks on web applications are providing potent starting points for multi-staged attacks, they are helping attackers in finding new ways for making profit. These profit making opportunities offer attackers strong incentives to exploit web applications. Unfortunately, the existing web application development process employs inadequate means to incorporate security measures and thus produces applications which offer little resistance to attackers. Second, attack
toolkits offer automated ways to exploit such insecure applications thus lowering the technical bar for attackers. Finally, the lack of security awareness on the part of end-users is further facilitating exploitation of web applications.

1.1 Security Problems in Legacy Applications

The already deployed applications, we call legacy applications, have serious security problems. The security firm Whitehat analyzed (27) over 3000 websites that included reputed financial institutions, banks, healthcare services, etc. This study concluded that each one of these applications contained one or more serious vulnerabilities. Similarly the Web Application Security Consortium (WASC) analyzed (28, 29) tens of thousands of web applications and found that over 95% of the studied web applications were vulnerable to one or more serious attacks. These applications are often written with insufficient knowledge of the possible security threats or employ insufficient means to prevent them. In summary, these studies suggest that a significant majority of legacy web applications are exploitable.

The web statistics firm Netcraft suggests (30) that the Web contains over 100 million active web applications. As a significant majority of legacy web applications are exploitable and the Web contains millions of such applications, securing these applications is essential to improving security of the Web.

Securing legacy web applications poses several challenges. First, any manual analysis / fixing attempt requires extensive effort. Specifically, such an analysis needs to identify vulnerable portions of an application and then rewrite them to make the application secure. This entails extensive analysis of the source code which requires specialized knowledge of the application’s source code and efforts in terms of time. Such efforts must ensure that fixes do not alter the desired functionality
of the application. The difficulty in fixing legacy applications manually is reflected in the fact that most reported vulnerabilities remain unresolved for an average of 116 days in 50% of vulnerable applications (27). Given that a significant majority of legacy web applications are exploitable and there are a large number of such applications, manual analysis may not provide a scalable solution. In summary, manual analysis is error-prone, time consuming and expensive.

Second, for legacy web applications, source code is the only available resource for automated security analysis. Software development life cycle (SDLC) artifacts such as design and requirement documents could assist a manual analysis but are often unavailable. Also, source code of an application is often the most updated artifact in a product and may not be in sync with other SDLC artifacts. As no SDLC artifacts other than the source code assist automated analysis techniques, source code is the only relevant resource for automated analysis.

The research community has responded to this crisis with mainly the following three types of solutions: a) new paradigms for developing secure web applications and b) vulnerability detection approaches and c) vulnerability prevention approaches.

New paradigms focus on developing fundamentally secure web applications. In general, these paradigms offer certain features that if used consistently during the development process, ensure security by construction. Unfortunately, these paradigms do not provide any assistance in safeguarding legacy applications thus do not address the problem posed by insecure legacy web applications. In summary, new paradigms facilitate secure web application development but do not offer assistance in mitigating security issues in legacy web applications.

Detection based approaches employ source code analysis to find potential vulnerabilities in web applications. Specifically, these techniques try to identify vulnerable flows in the source code. These
efforts are essential to build a better understanding of the current state of security in web applications. However, these efforts lack in the following two aspects: a) most detection techniques focus on well publicized vulnerabilities thus failing to provide a comprehensive picture of vulnerabilities in web applications and b) each detection technique proposes an ad-hoc approach that may not be applicable to a wider range of vulnerabilities. In summary, detection based approaches provide the necessary first step towards securing web applications but more research is needed to improve our understanding of vulnerabilities in web applications through development of principled detection approaches.

Automated prevention solutions are the most desired as they can safeguard legacy and newly developed web applications alike. Unfortunately, the existing prevention techniques have several limitations. A widely deployed solution is filtering that analyzes user inputs to identify malicious patterns in them. Although effective as a first layer of defense, it is shown to be ineffective against subtle attacks (31, 32). Another prevalent type of solution in this category employs a reference monitor to analyze execution of a web application and forbid exploits from succeeding. In order to identify and forbid malicious behavior, these solutions typically need guidance in the form of a specification. Typically this specification enables monitors to forbid runs of a web application that may lead to attacks while allowing the application to perform benign computations. Given the large number of legacy web applications, an approach to manually construct such specifications is not scalable and is error-prone. Automated techniques to infer and enforce such specifications have been proposed but have several precision issues (refer to chapter 7 for detailed treatment of related work).
In summary, the Web contains a large number of legacy web applications and studies suggest that a significant majority of such applications is exploitable. Further, the growing number of attacks suggest that security features implemented in these applications are inadequate. Only automated techniques to find and fix problems in legacy applications provide a scalable recourse to make the large number of legacy web application safe. Finally, the current state-of-the-art work in this area does not address some of the most pressing issues, thus warranting further research.

1.2 Thesis Goals

As legacy web applications constitute majority of the Web and are often exploitable, our key goal is to invent ways to secure them. Given the large volume of legacy web applications, automated techniques that require minimal human intervention offer the only scalable solution to improving security of these applications. Further, such techniques provide “stop gap” solutions to safeguard newly developed vulnerable web applications as well. This may be necessary until the web application development community embraces fundamentally secure practices and applies them consistently to develop secure web applications.

In this thesis we explore techniques that make progress on the front of improving security of legacy web applications. To this end we set out to answer the following two questions:

- Do less studied vulnerabilities pose threat to legacy web applications? Are they pervasive?
  If so, can we find them automatically?

- Is it possible to safeguard legacy web applications fundamentally by leveraging recent robust defenses? If so, can we devise systematic and automated techniques to do so?

To answer the first question, we chose to study parameter tampering vulnerabilities which have received less attention from the research community. Our goal was to quantify existence of
these vulnerabilities by studying web applications. To this end, we developed techniques to uncover these vulnerabilities systematically. These techniques build understanding of the intended behavior with respect to input validation and then find vulnerabilities by identifying successful violations of intended behavior. By evaluating these approaches on a large number of open source and commercial web applications, we offer strong evidences that parameter tampering vulnerabilities are widespread in contemporary web applications.

To answer the second question, we investigate automated ways to prevent SQL injection vulnerabilities which are one of the top threats to legacy web applications. As PREPARE statements offer a robust defense for SQLIA for freshly written applications, we investigated a possibility to automatically extend this robust defense to legacy web applications. To this end, we offer a novel source code analysis technique that builds an understanding of the SQL query generation process encoded by the source code. This provides an insight into the application intent with respect to SQL queries. Our technique then enforces this intent by altering the SQL query generation process to employ PREPARE statements. By evaluating this solution on several open source applications, we offer compelling results on effectiveness of this technique. By retrofitting legacy web applications with a “text-book” defense provided by PREPARE statements, we offer a strong evidence that legacy web applications can be retrofitted to reap benefits provided by new robust defenses.

The common link across the above efforts is the notion of program intent. The program intent broadly refers to the benign behavior as intended by developers of the program. To answer the above two questions, we essentially developed techniques to extract and use these intentions. To answer the first question, we chose to find instances that successfully violated program intent whereas we chose to enforce program intent through robust new defenses in the latter. The ability to extract
This thesis makes the following contributions:

- We provide strong evidences that by analyzing source code of an application, a precise specification of its intent can be inferred. For a legacy web application, most often, its source code is the only available resource for inferring its behavior.

- We study parameter tampering vulnerabilities, a relatively less explored class of vulnerabilities, and demonstrate that they are pervasive in today’s web applications. By developing systematic techniques to discover these vulnerabilities we offer an evidence that the specification of program intent can help uncover vulnerabilities.

- We provide a strong evidence that it is possible to fundamentally secure legacy web applications by retrofitting them with newly developed robust defenses. Often legacy web applications are manually retrofitted with such defenses resulting in effort-intensive and error-prone exercises.

- Based on the above techniques this thesis also contributes tools to discover parameter tampering attacks on web applications and to eliminate serious threats such as SQL injection.

To demonstrate effectiveness of the proposed techniques, we provide extensive evaluation of these tools on several commercial as well as open source web applications.

\textit{This thesis demonstrates that legacy web applications can be fundamentally secured by systematically inferring and enforcing specifications of safe behavior from their source code and that such specifications can also be used in a systematic fashion to find vulnerabilities.}
1.2.1 **Summary of Implementation and Results**

In this section we provide highlights of techniques developed in this thesis and summarize key results. The following two sections summarize efforts made to answer the two questions posed above. Section 1.2.1.1 summarizes our approach for automated discovery of parameter tampering vulnerabilities followed by Section 1.2.1.2 that summarizes our approach for SQL injection prevention.

1.2.1.1 **Detection of Parameter Tampering Vulnerabilities**

We present the first systematic approach for detecting parameter tampering attacks on legacy web applications as well as the first approach to generate parameter tampering attacks by construction.

We implemented these approaches in two tools NoTamPER and WAPTEC (Whitebox Analysis for Parameter Tampering Exploit Construction). These tools systematically analyze the client-side code of a web application to learn restrictions imposed on user supplied inputs. These restrictions act as a specification of program intent and must be enforced at the server-side code. NoTamPER employs a blackbox approach to infer missing checks in the server-side code whereas WAPTEC employs a more precise whitebox analysis of the server-side code to generate parameter tampering attacks by construction.

We empirically demonstrate that parameter tampering vulnerabilities are widespread in legacy web applications and these vulnerabilities could be exploited through severe attacks. By evaluating these tools on 17 open source and commercial web applications we identify a total of 52 confirmed parameter tampering vulnerabilities. Further, starting with the vulnerabilities identified by these tools we construct exploits that demonstrate serious security problems: unauthorized monetary
transactions at a bank, unauthorized discounts added to a shopping cart, and so on. These experiments strongly suggest that parameter tampering vulnerabilities are widespread in open source as well as commercial web applications and can be exploited through severe attacks.

The supplementary websites for WAPTEC (33) and NoTamper (34) provide detailed exploit reports and demos of the prototype implementation.

In summary, results obtained by NoTamper and WAPTEC strongly highlight a significant gap between the server-side parameter validation that should occur and the server-side validation that does occur in today’s web applications. Further, these techniques provide an evidence that it is possible to extract specifications of intended behavior to automatically find these gaps.

1.2.1.2 Prevention of SQL Injection Vulnerabilities

We develop an automated technique for preventing SQL injection vulnerabilities in legacy web applications. In specific, we propose the first sound program transformation approach for automatically transforming the code of a legacy web application to employ PREPARE statements in place of unsafe SQL queries. By making use of PREPARE statements our technique extends “textbook defense” offered by PREPARE statements to legacy web applications.

We implemented this approach in a tool called TAPS (Tool for Automatically Preparing SQL queries). TAPS first employs program analysis techniques to determine the web application intended SQL queries which act as a specification of the desired behavior. TAPS then employs a novel program transformation technique to enforce this specification by introducing PREPARE statements.

TAPS’s effectiveness was evaluated by transforming several real world applications, including one with over 22,000 lines of code. It automatically transforms over 90% of SQL queries in large web applications. As TAPS automates a majority of tasks involved in transitioning legacy web
applications to make use of PREPARE statements, it offers an effective tool to combat serious threat posed by legacy web applications.

The supplementary website (35) for TAPS tool provides a demo of our prototype implementation.

In summary, TAPS provides a strong evidence that it is possible to successfully design retrofitting techniques that guarantee security (by construction) in legacy applications, and eliminate well known attacks.

The rest of this thesis is organized as follows. Chapter 2 discusses SQL injection and parameter tampering attacks on web applications and presents insights on vulnerable applications that showcase common attributes. Chapter 3 then proposes a high level approach to detect and prevent vulnerabilities in web applications and discusses challenges in realizing it. Chapter 4 and Chapter 5 present applications of the high level approach to automatically discover parameter tampering vulnerabilities and to prevent SQL injection attacks in web applications, respectively. Chapter 6 provides guidance to future efforts in preventing parameter tampering attacks by identifying open research challenges. Chapter 7 compares contemporary work with the contributions of this thesis. Chapter 8 concludes this thesis.
CHAPTER 2

BACKGROUND

In this chapter, we first introduce a shopping cart application that showcases typical features of a web application (Section 2.1). This application is used as a running example in the rest of this thesis. We then provide details of SQL injection and parameter tampering vulnerabilities and discuss concrete attacks in the running example (Section 2.2 and Section 2.3, respectively) and finally offer a deeper analysis of SQL injection and parameter tampering vulnerabilities in the context of running example (Section 2.4).

2.1 Running Example: Shopping Cart

Typically web applications are comprised of a client-side codebase that executes in a browser and a server-side codebase that executes in a web server. The client-side code executing in browsers interacts with users and supplies user inputs to the server-side code. The server-side code performs computations based on these inputs to provide services.

Figure 1 shows the checkout form of a shopping cart application in which a user has already selected two products for purchase. The checkout form asks the user to specify the quantity of each product, the credit-card to be charged (displayed in a drop-down list of previously used cards), the recipient’s name and the shipping address along with any special delivery instructions. We outline the end to end processing of this form below.

The first step in processing this form is validation of user supplied inputs. As user inputs are untrusted, web applications are required to validate them. Traditionally, this validation took place
at the server-side. However, a popular contemporary trend is to use JavaScript to validate inputs at the client. The ability to validate inputs at the client allows an application to avoid round-trips to the server to identify invalid inputs thus resulting in faster and more responsive applications. Further, localized validation at the client reduces overall network traffic and load on the server. Listing 2.1 shows the client-side JavaScript code for the running example. This code checks if user inputs possess certain properties and if so, submits them to the server-side code for further processing; otherwise it forces users to correct invalid inputs. In specific, the JavaScript code forces submission of positive quantities for the selected products (lines 8 to 11) and use of 30 or less characters in specifying the recipient’s name (lines 13 to 16).
Listing 2.1. client.js

```javascript
function validateForm() {

    var q1 = document.getElementById("quantity1");
    var q2 = document.getElementById("quantity2");
    var n = document.getElementById("name");

    // disallow submission if negative quantities are specified.
    if (q1 < 0 || q2 < 0) {
        alert("Please specify positive quantities for items.");
        return false;
    }

    if (n.length() > 30) {
        alert("Please use 30 or less characters to specify name.");
        return false; // show error, don’t submit
    }

    return true; // submit form
}
```

Once the client-side validation is successful, user inputs are sent to the server-side for further processing. Listing 2.2 shows the server-side code for our running example. The server-side code first checks value of the card parameter and generates HTML to highlight the selected credit card (lines 2 to 6). It then sanitizes value of the name parameter by truncating it to the first 30 characters (line 8). It then checks if the op parameter is set to “purchase”, and if so, computes the total cost of requested products (line 12). If a special parameter discount is set, the server-side code accordingly applies a discount (lines 15 to 16), computes a SQL INSERT query to store the order details and finally executes this query (lines 18 to 22).
The following section describes SQL injection attacks and presents a concrete attack on the
above running example.

### 2.2 SQL Injection Attacks

SQL injection attacks are extremely prevalent and ranked as one of the most potent vulnera-
bilities of the last decade (2). Further, the mass injection attacks discussed in the previous chapter
exemplify how these attacks can victimize an organization and members of the general public. By
using Google code search, analysts have found several applications whose sources exhibit these
vulnerabilities (36). Several reports suggest that a large number of applications on the Web are

---

**Listing 2.2. server.php**

```
$ca = $_POST['card'];
if ($ca matches '1234-5678-9012-3456' ||
   $ca matches '7890-1234-5678-9012'){
  // generate HTML to show a
  // selected card in the form
}
$n = substr($_POST['name'], 30);
if ($_GET['op'] == "purchase"){
  $cost = $_POST['quantity1'] * $price1 +
           $_POST['quantity2'] * $price2 ;
  if (isset($_POST['discount']))
    $cost = $cost - $_POST['discount'] * $cost / 100;
  $q = "INSERT INTO orders ('na', 'co', 'ca', 'ad', 'de') ";
  $q .= " VALUES ('$n', $cost, '{$_POST['card']}', ";
  $q .= " '$_POST['address']}', '$_POST['delivery']')');";
  mysql_query($q);
```
indeed vulnerable to SQL injection attacks (37), that the number of attacks are on the increase, and is on the list of most prevalent forms of attack (37).

2.2.1 SQL Injection Attack Example

Web applications that employ backend databases for storage, often parameterize backend database queries with user inputs. Malicious inputs could influence these queries thus causing SQL injection attacks (SQLIA). The running example web application constructs an INSERT query by appending user provided inputs with literal strings. Consider the run of this application with the following user inputs:

```
quantity_1 = 1, quantity_2 = 1, card = "1234-5678-9012-3456",
name = "Alice", address = "wonderland", delivery = "none",
op="purchase"
```

Let us also assume that prices of the selected products are $400 and $2000, respectively. For these inputs the following SQL query is issued.

```
INSERT INTO ORDERS('na', 'co', 'ca', 'ad', 'de')
VALUES('Alice', 2400, '1234-5678-9012-3456', 'wonderland', 'none')
```

When the above query is executed a record for this purchase is created in the ORDERS table. Now suppose a malicious user supplies crafted delivery instructions as `delivery=" '); DROP TABLES; -- "`. Further, assuming that all other parameters have the same values as above, the following query is issued.

```
DROP TABLES; --
```
Notice that value of the `delivery` parameter (shown above in bold fonts) balances the string delimiter (single quote) opened by the application, completes the `INSERT` query and then adds a `DROP` query after a query separator (semi-colon). Further, it makes use of the SQL comment operator (two hyphens - -) to comment out the application generated part of the `INSERT` query to ensure that the resulting queries are well formed. When this query is executed, it deletes all database tables instead of adding a new entry in the `ORDERS` table. This is a typical example of SQL injection attack, wherein malicious user inputs alter semantics of the query application intended to generate. These attacks enable attackers to manipulate queries and compromise confidentiality and integrity of backend databases.

### 2.2.2 SQL Injection Attack Vulnerability

The running example code embodies the key problem of web applications that enables SQLIA. Specifically those applications that combine trusted code (literal strings present in the web application codebase) with untrusted data (user supplied inputs) through low level string operations and fail to confine untrusted data are vulnerable to SQL injection attack. In our running example, the application concatenates literal SQL keywords with inputs `name`, `card`, `address` and `delivery`. By enclosing these inputs within string delimiters (single quotes), it expects these inputs to remain confined within these string delimiters. However, as it fails to enforce this restriction, it is exploitable through SQL injection in any one of these parameters.

Researchers have categorized these attacks based on injection mechanisms as well as based on different types of attack strings. These attack strings are embedded in user inputs but can be injected through cookies, Hypertext Transfer Protocol (HTTP) headers or body of an HTTP request. Further, the time when malicious inputs are actually used is also used as a categorizing
factor. For example, in a first-order attack (also called reflected attack) malicious inputs are used immediately in constructing SQL queries whereas in a second-order attack malicious payloads are stored in the database and used subsequently. We refer the interested reader to (31) for a detailed classification of SQLIA.

2.2.3 In Scope

As the number of first-order attacks dominate the total number of reported SQL injection attacks (38), the rest of this thesis focuses on reflected SQLIA. Second-order injection is an important class of attacks and is not explored further in this thesis. Further, those applications that expect users to supply partial SQL queries (e.g., BugZilla user interface allows users to combine partial queries) are out of the scope of this thesis as SQLIA is not well defined in such cases.

2.3 Parameter Tampering Attacks

In this section we describe another important class of attacks that involves user inputs supplied through forms. Such forms are a widely deployed means of capturing user inputs and thus interactive form processing is pervasive in today’s web applications. Interactive form processing is crucial for electronic commerce and banking sites, which rely heavily on web forms for billing and account management. Originally, form processing took place only on the server, but in recent years the task has migrated to the web browser and been implemented in JavaScript. Processing user-supplied inputs to a web form using browser-side JavaScript eliminates the latency of communicating with the server, and therefore results in a more interactive and responsive experience for the end user. Furthermore, browser-side form processing reduces network traffic and server loads.

The form processing performed by the browser mostly involves checking user-provided inputs for errors. For instance, an e-commerce application accepting credit card payments requires the credit
card expiry date to be valid (e.g., be a date in future and be a valid month / day combination). Once user inputs have been validated, they are sent to the server as part of an HTTP request, with inputs appearing as parameters to the request.

A server accepting such a request may be vulnerable to attack if it assumes that the supplied parameters are valid (e.g., the credit card has not yet expired). This assumption is indeed enforced by the browser-side JavaScript; however, malicious users can circumvent client-side validation by disabling JavaScript, changing the code itself, or simply crafting an HTTP request by hand with any parameter values of the user’s choice. Servers with parameter tampering vulnerabilities are open to a variety of attacks (e.g., enabling unauthorized access, SQL injection, cross-site scripting).

2.3.1 Parameter Tampering Attack Example

In the running example a malicious user can bypass the validation check on the quantity of each product (parameters $quantity_1$ and $quantity_2$) and submit a negative number for one or both products. It is possible that submitting a negative number for both products would result in the user’s account being credited; however, that attack will likely be thwarted because of differences in credit card transactions on the server involving debit and credit. However, if a negative quantity is submitted for one product and a positive quantity is submitted for the other product so that the resulting total is positive, the negative quantity acts as a rebate on the total price. Suppose the malicious user supplies the following inputs.

\[
\begin{align*}
\text{quantity}_1 &= -4, \quad \text{quantity}_2 = 1, \quad \text{card} = "1234-5678-9012-3456", \\
\text{name} &= "Alice", \quad \text{address} = "wonderland", \quad \text{delivery} = "none"
\end{align*}
\]

For these inputs, the running example computes a total cost of $400 and is tricked into offering a ‘discount’ of $1600. We note that unlike SQLIA, this attack did not attempt to inject any code
but altered the logic to compute the total cost of the shopping cart. Such attacks are termed as logic attacks and can be hard to detect (39, 40).

2.3.2 Parameter Tampering Attack Vulnerability

This vulnerability arises from differences in implementation of input validation done by the client- and the server-side code. Specifically the server-side code must replicate validation done at the client. If the server-side code performs weaker validation than the client-side code, it contains a parameter tampering vulnerability. The server-side code may fail to replicate checks for several reasons. First, the client and the server codebases are implemented in different languages and may be developed independently. Second, any changes to one codebase must be reflected in another and mistakes in doing so may result in differences. Intuitively, the client-side validation imposes certain restrictions on values of user inputs that the server-side code is willing to process. However, in the event of tampering, these restrictions can be violated and must be enforced by the server-side code. Parameter tampering vulnerability arises if the server-side code fails to do so. In summary, an application whose client-side validation is more restrictive than the corresponding server-side validation, is vulnerable to parameter tampering.

In our running example, the client-side JavaScript code ensures that only positive values are submitted for the parameters \( \text{quantity}_1 \) and \( \text{quantity}_2 \). However, the server-side code fails to check the same and hence performs weaker validation than the client. As a result, it is exploitable through tampering of the parameters \( \text{quantity}_1 \) and \( \text{quantity}_2 \).

2.3.3 In Scope

Parameter tampering attacks can be specialized to yield code injection attacks such as SQLIA or logic attacks (39) as exemplified in the running example. Intuitively, a tampering vulnerability of
an application represents failure to reject undesired inputs and depending on how the application uses tampered parameter, tampering attempt can be specialized to conduct specific attacks. In this thesis, we are concerned with finding tamperability of parameters and leave specialization of attacks as a future work.

The previous two sections discussed SQL injection and parameter tampering vulnerabilities in detail. We also presented concrete exploits of these vulnerabilities in the context of our running example. The following section takes a closer look at these attacks and identifies attributes of the running example that allowed these attacks to succeed.

2.4 Common Attributes of Vulnerable Applications

Interestingly, the vulnerable running example shows common attributes that led to exploitation through the above two attacks. In both the attacks, vulnerabilities were exploited by malicious user inputs. These inputs are processed by the application and then used at sensitive operations (e.g., SQL query execution). Our key observation is that the way in which an application processes inputs encodes certain restrictions on inputs, and violation of these restrictions leads to successful attacks. We call these restrictions, intentions of the original application. Intuitively, attacks violate an application’s intentions to perform unintended operations.

2.4.1 Revisiting SQL Injection Attack

Let’s revisit construction of the INSERT query in our running example. Notice that the application enclosed value of the user supplied input delivery in string delimiters (single quotes). By doing so, it encoded a restriction that this value should remain confined within the string delimiters. In the SQL grammar, these delimiters are used to enclose data items and define the string literal data context. Hence, the application wants value of the delivery parameter to remain confined to a
string literal data context that represents data in the \texttt{INSERT} query. Intuitively, most applications expect user inputs to supply data that is then used to parameterize queries and they encode this expectation by enclosing user inputs with string delimiters while generating SQL queries. The malicious value `'\); DROP TABLES; --` of the \texttt{delivery} parameter, contained a string delimiter (single quote) that prematurely closed the application initiated string literal data context thus allowing it to specify malicious \texttt{DROP} query. In doing so, this attack violated a restriction encoded by the application that value of the \texttt{delivery} parameter should remain confined to a string literal data context.

\subsection*{2.4.2 Revisiting Parameter Tampering Attack}

Lets revisit the form processing in our running example. When a user submits values to this form, the JavaScript client-side code checks values of the parameters \texttt{quantity}_1, \texttt{quantity}_2 and \texttt{name}. Specifically it rejects user inputs that specify negative values for the parameters \texttt{quantity}_1 and \texttt{quantity}_2 and strings longer than 30 characters for the \texttt{name} parameter and forces the user to correct these values. As the JavaScript client-side code is part of the web application codebase, these rejections encode application’s intent to accept a limited set of values for these parameters. Specifically the conditions evaluated by the client-side code encode certain restrictions on user inputs. The JavaScript client-side code only submits inputs to the server when these restrictions are satisfied. Intuitively, the server-side code \textit{expects} to only encounter user inputs that satisfy these restrictions. However, the attack payload violated this restriction by specifying a negative value for the \texttt{quantity}_1 parameter. As the server-side code did not re-enforce the above restriction, it yielded the attacker controlled pricing of the order.
In this chapter we provided in-depth details of two vulnerabilities: parameter tampering and SQL injection, and identified common attributes of vulnerable programs that led to exploitation of these vulnerabilities. The next chapter provides a high level overview of our approach to detect and prevent vulnerabilities in web applications.
CHAPTER 3

APPROACH OVERVIEW

This chapter first develops a notion of intention which, as discussed before, are restrictions on user inputs encoded by an application’s source code (Section 3.1). It then presents a high level approach to detect and prevent vulnerabilities by systematically extracting and using intentions (Section 3.2) and finally discusses challenges in realizing the high level approach (Section 3.3).

3.1 Intentions

Web applications can be viewed as functions that take user inputs and produce outputs. Let function $F$ represent a web application that takes a set of inputs $I_1, I_2, \ldots, I_n$ and produces outputs $O_1, O_2, \ldots, O_n$, respectively. Each input set, $I_i$ consists of several name-value pairs where name represents a parameter and associated value represents user provided data. Each output $O_j$ is an operation that either queries or alters some sensitive web application state. Some of the most interesting outputs are executable code fragments that are executed subsequently e.g., SQL queries, HTML / JavaScript. We note that the function $F$, is purely defined by source code of the web application i.e., source code dictates how outputs are produced from inputs.

For our running example, $F$ is a function that computes and executes an `INSERT` query by concatenating fixed strings with user provided data. Let $I_1$ and $O_1$ represent the input set and computed output for the benign run of the running example as discussed in Section 2.2.1.
\( I_1 = \{(\text{name}, "Alice"), (\text{card}, "1234-5678-9012-3456"), (\text{address}, "wonderland"),
\text{(quantity}_1, 1), (\text{quantity}_2, 1), (\text{op}, "purchase"),
\text{(delivery}, "none"\} \}

\( O_1 = \text{mysql_query(" INSERT INTO ORDERS('na', 'co', 'ca', 'ad', 'de')}
VALUES('Alice', 2400, '1234-5678-9123-4567', 'wonderland',
'none') ")

3.1.1 Observations

We first notice that the generated output \( O_1 \) is a SQL query that is subsequently executed by the database engine. The output \( O_1 \) is computed with user supplied name-value pairs e.g., name-value pair (\text{delivery}, "none") is used to compute the output.

We are interested in understanding how the application wanted to utilize value of the delivery parameter. In the generated output \( O_1 \), value of the delivery parameter appears enclosed within two single quotes. The generated output \( O_1 \) is a SQL query and in the SQL grammar single quotes are used as string delimiters and enclose literal data strings. As the application enclosed value of the delivery parameter within single quotes, it expected user to supply a value that remained confined within the string delimiters it added. Intuitively, the application has encoded its expectation that the user would supply a data value for the INSERT query.

Having gained this insight, we revisit the SQL injection example discussed in Section 2.2.1. Let \( I_2 \) and \( O_2 \), shown below, represent the input set and computed output for the SQL injection attack through the delivery parameter. Note that the input sets \( I_1 \) and \( I_2 \) only differ in the value of parameter delivery as \( I_2 \) contains a malicious value for this parameter (shown in the bold font).
Notice that in the output $O_2$, the application initiated a string literal data context for the `delivery` parameter by adding a string delimiter to the query (single quote). However, the malicious `delivery` value closed this prematurely by specifying a string delimiter as the first character. Having closed the data context, the malicious `delivery` value is able to specify code to be executed thus resulting in a SQL injection attack. Clearly, in this instance the malicious `delivery` instructions violated expectation of the application that it will be used only as data in the generated SQL query.

Intuitively, the application code was written implicitly assuming `delivery` value to be a data string (no unescaped SQL meta characters e.g., single quotes). The application encoded this assumption by enclosing user supplied `delivery` value within single quotes without factoring in the possibility of `delivery` instructions pre-maturely closing these quotes. Further, the application then used single quotes enclosed `delivery` instruction to generate an `INSERT` query i.e., achieve specific behavior. However, the malicious `delivery` value contained single quotes that violated the application’s assumption and tricked the application into also issuing a `DROP` query thus causing it to stray away from the specific behavior it intended to achieve. In summary, the application implicitly assumed benign value for `delivery` instruction and was subsequently tricked into straying away from its intended behavior.
The above discussion is consistent with key insights reported by several researchers that successful SQL injection attacks alter SQL parse tree structures of the program generated queries (41,42). Our running example intended to generate an \texttt{INSERT} query. To trick it to generate a different query e.g., \texttt{DROP} query, malicious inputs must break out of the application specified string data contexts. The main requirement of SQL injection attacks i.e., injection of malicious SQL code, necessitates altering structure of the program generated queries.

We summarize the above discussion in the following key observation.

**Observation 1**: The application source code encodes syntactic restrictions on user inputs and successful attacks manifest as deviations from these restrictions.

Parameter tampering attack presented in Section 2.3.1 subverts a different form of intended behavior. In this attack, a negative value for the parameter \texttt{quantity}$_1$ = -4 led to a discount of $1600. However, the resulting query did not violate any syntactic restrictions encoded by the application. In specific, all user supplied data remained confined in the application specified data contexts. We take a closer look at the shopping cart cost computation below.

\[
\begin{align*}
\text{For quantity}_1 = 1, \text{quantity}_2 = 1 \\
\text{cost} &= 1 \times 400 + 1 \times 2000 = 2400 \\
\text{For quantity}_1 = -4, \text{quantity}_2 = 1 \\
\text{cost} &= -4 \times 400 + 1 \times 2000 = 400 \\
\end{align*}
\]

The above attack, changed an “addition” operation in cost computation into a “negation” thus affecting the sensitive SQL query operation. The client JavaScript code (Listing 2.1), forbids negative values for \texttt{quantity}$_1$ and \texttt{quantity}$_2$ and hence encodes restrictions on values (semantics) of user inputs. However, as the attack bypassed client side validation, it could violate these restrictions thus conducting a successful parameter tampering attack.
Intuitively, the application code was written implicitly assuming positive values of quantity parameters received by the server-side. The client-side validation encoded checks that ensured that this assumption was satisfied in normal runs of the application. These quantities were then used by the server-side code to increase cost of the shopping cart i.e., achieve specific behavior. However, negatives values of quantities that bypassed client-side checks, violated the implicit assumption of the server-side code. Subsequently, these values tricked the application to reduce cost of the shopping cart thus causing it to stray away from the specific behavior it wanted to achieve.

This discussion yields the following key observation:

Observation\textsubscript{2}: The application source code encodes semantic restrictions on user inputs and successful attacks manifest as deviations from these restrictions.

These observations lead us to the following definition of intentions.

Intentions: The source code of a web application specifies syntactic as well as semantic restrictions on user inputs. These restrictions define intended use of user inputs and we call them intentions.

3.2 High Level Approach

The key intuition for our high level approach is derived from intentions. Recall that intentions are restrictions on user inputs as encoded by the source code and hostile inputs that violate these restrictions succeed in controlling web application’s output. This notion enables us to reason about: a) finding vulnerabilities by generating inputs that violate restrictions and b) preventing vulnerabilities by enforcing restrictions.

Figure 2 provides a blueprint of our high level approach. The two key components are a) intention generator and b) intention utilizer. In the first step, intention generator extracts program
Figure 2. High level approach: step 1) extract intentions encoded in the source code, step 2a) use intentions to construct concrete exploits or step 2b) enforce them to prevent vulnerabilities.

intent. These intentions are used in the second step to either a) detect vulnerabilities by generating concrete exploits or b) prevent vulnerabilities by transforming vulnerable web applications. The intention generator builds on well known techniques such as symbolic execution (43) to extract and correlate intentions from different components of a web application. The intention utilizer proposes novel techniques and performs the following two tasks: a) to detect vulnerabilities it explores the input space to find inputs that violate intentions and uses them to identify missing enforcement of intentions and b) to prevent vulnerabilities it identifies and re-writes parts of the web application that allow an attacker to violate intentions.

The blueprint presented here is specialized in Chapter 4 to find parameter tampering vulnerabilities and Chapter 5 to prevent SQL injection vulnerabilities. In Chapter 4 we present an approach that analyzes the client-side code of a web application to extract application intended semantic restrictions on user inputs. The client-side code expresses constraints on data values through HTML
and JavaScript checks. We present a scheme that systematically evaluates this code to extract the application’s intentions. To ensure correctness of this analysis, we analyze the server-side code and propose a technique to find zero-day vulnerabilities by construction.

Chapter 5 presents a systematic approach that employs symbolic execution to learn application intended data contexts for each user input. To ensure that user inputs do not break out of these data contexts, the enforcer makes use of \texttt{PREPARE} statements.

We highlight challenges that must be addressed to realize the intention extractor and the intention utilizer components of our high level approach in the following section.

3.3 Challenges

The high level challenge is in precisely extracting intentions encoded by the source code and then in devising ways to find violations of intentions or to enforce them such that only hostile attempts are forbidden. We elaborate specific challenges below and revisit them while discussing related work in Chapter 7 to show how most contemporary works fall short of addressing one or more of these challenges.

3.3.0.1 Inter Component Analysis

Typical web applications are quite complex and comprise of components that are written in multiple languages. The key challenge then lies in precisely learning intentions through sound program analysis of desperate components of a web application. An incomplete or unsound specification of intentions, when enforced would either leave the application vulnerable or interfere with its functionality. Similarly, imprecise specification when used in vulnerability detection, would yield both false positives and false negatives. As intentions may be embedded in multiple components, a precise analysis must reason about intentions embedded in each component but more importantly
about how intentions map across components. In the running example, the client-side JavaScript code checks value of the parameter \( n \) which the client-side HTML code denotes as \texttt{name}. This value propagates to a server-side variable \( \$n \) and then to a database column \texttt{na}. Note that each of these four components may encode certain restrictions that when combined together, provide a precise specification of intentions with respect to the \texttt{name} parameter. In the absence of such reasoning, resulting intentions would either be incomplete or incorrect.

### 3.3.0.2 Flow Sensitivity

Most web applications use functions and conditional code to implement a set of functions. Each control flow leading to a sensitive operation may encode an entirely different set of intentions. For example, suppose an application has only two control flows that lead to a sensitive operation. In one control flow, the application correctly validates inputs while it fails to do so in the other. For this application, two runs of the application through these control flows encode different sets of intentions. This mandates a \textit{flow-sensitive} analysis i.e., reasoning about intentions on a per run basis. Further complications arise as multiple runs of an application may encode multiple intentions for a sensitive operation common to all these runs. This requires the analysis to be able to reason about intentions for each run of the application.

Further, the intention enforcement technique must also be flow sensitive i.e., the challenge here is in devising ways to enforce intentions for a specific run of the application. Consider an application that encodes multiple intentions for a single sensitive operation in multiple runs. A flow-insensitive approach may collectively enforce intentions encoded in all possible runs leading to the sensitive operation. Unfortunately, such an approach may yield both: \textit{overly-restrictive} and \textit{insufficiently-restrictive} enforcement.
3.3.0.3 **Contextual Sensitivity**

A typical web application’s source code performs several tasks in tandem e.g., code for processing parameters submitted via a form and code for generating HTML code annotated with erroneous fields, may be interleaved. In such applications, analysis must be *contextual* i.e., it must be able to identify and process operations relevant to a sensitive sink. In the absence of such analysis, intentions may be incorrectly inferred. The server-side code in the running example checks value of the `card` parameter to highlight user selected card in the form HTML (lines 2 to 6). However, this check is not relevant to the `INSERT` SQL query executed at line 22 as an attacker can still use an out-of-range value for the `card` parameter.

3.3.0.4 **Factoring in Input Validation / Sanitization**

The server-side code may process user inputs before using them in sensitive operations. In the wake of attacks, a large number of applications employ validation routines to reject malicious inputs or sanitization routines to rewrite malicious inputs. The challenge here is in developing techniques that leverage such information in refining specification of intentions and offer defenses that do not interfere with existing defenses. In our running example, the server-side code sanitizes value of the `name` parameter by truncating it to the first 30 characters. Such sanitization should guide the process of extracting intentions such that false alarms are not reported in vulnerability detection and enforcement of intentions does not alter its outcome.

3.3.0.5 **Understanding Program Logic through Code Analysis**

We wish to devise intention enforcement techniques that *correct* vulnerable statements in applications. However, designing a sound method to identify and correct vulnerable statements is extremely challenging. In specific, a completely automated method needs to replicate the human
understanding of the program logic that constructs arguments for a sensitive operation. Quite often, this understanding of program logic is guided by additional documentation such as high-level system designs, flow charts and low level program comments. An automated method that aims to eliminate / minimize human effort cannot depend on the availability or use of any such additional specifications. The web application code is, therefore, the only specification available to such a method, from which an understanding of the program logic needs to be automatically extracted to guide the transformation.

3.3.0.6 Preserving Semantics of Original Application

A challenge for intention enforcement techniques is that they must not alter semantics of the original application i.e., an application employing such techniques must continue to process benign inputs as it did before employing these techniques. Specifically functionality of the application should remain unaltered for benign inputs whereas malicious inputs should not yield attacker controlled behavior. The challenge here is in programmatically reasoning about the affect of intention enforcement technique on processing of benign inputs and avoiding it if the original functionality of the application may be altered. In this regard, the intention enforcement technique must not only reason about malicious but also benign inputs.

3.3.0.7 Scalability

Given the large number of deployed web applications, detection and prevention schemes must minimize human intervention. In specific, it should require minimal human intervention on a per application basis. Such efforts are often time consuming, error prone and limit scalability of a defense. The challenge is in automating significant portions of analysis to mimic human understanding of the code.
This chapter presented our high level approach to detect and prevent vulnerabilities in web applications and provided an in-depth discussion of challenges involved in realizing this approach. The next chapter presents our approach to detect parameter tampering vulnerabilities that addresses these issues.
CHAPTER 4

DETECTION OF PARAMETER TAMPERING VULNERABILITIES

This chapter presents two approaches NoTamper and WAPTEC that automatically detect parameter tampering vulnerabilities in web applications. This chapter is organized as follows. Section 4.1 presents motivations behind the key design decisions made in these approaches. Section 4.2 summarizes contributions of these approaches. Section 4.3 formulates parameter tampering problem. The next two sections, Section 4.4 and Section 4.5, present high level overview, architecture, and algorithms used to implement NoTamper and WAPTEC tools. Section 4.6 provides a comprehensive evaluation based on these two tools. Section 4.7 summarizes this chapter.

4.1 Motivation

We presented details of parameter tampering attacks in Chapter 2 and restate the key observations below. Parameter tampering attacks manifest by violating intentions of web applications. These intentions are encoded by the client-side code that solicits inputs from users and validates them. As the client-side validation can be bypassed, the server-side code must re-enforce these intentions by re-validating inputs. If the server-side code fails to replicate the client-side validation it would fail to re-enforce these intentions. Subsequently, an attacker can supply inputs that violate restrictions imposed by the client-side code and result in parameter tampering attacks.

We first note that parameter tampering attacks have received insignificant attention of the research community. There has been extensive research to address specific server-side input validation problems such as SQL injection and cross-site scripting. However, the contemporary research liter-
nature does not provide any means to find or fix parameter tampering vulnerabilities in legacy web applications. To the best of our knowledge the only two relevant research works are SWIFT (44) and Ripley (45). However, they focus on the broader issue of ensuring data integrity in web application development frameworks. The goal of these approaches is to realize new web applications that are effectively immune to parameter tampering attacks.

This lack of systematic work on parameter tampering attacks raises the following interesting research question: Do parameter tampering vulnerabilities actually exist in web applications and if so, can we devise automated techniques to find them?.

To answer this question we want to realize the high level approach outlined in Section 3.2 and develop tools that can establish the existence of parameter tampering vulnerabilities in web applications. Specifically we aim to determine if a given web site (i.e., a deployed web application) is vulnerable to parameter tampering attacks, and produce a report of potential vulnerabilities and the associated HTTP parameters that triggered these vulnerabilities. To realize the high level approach, we identify the following two main tasks: a) extract the application intended restrictions on user inputs and b) prove the existence of parameter tampering vulnerabilities by finding successful violations of these intentions.

The first task entails realizing the intention generator component mentioned before (Figure 2). We aim to analyze the form processing code to learn intentions. We first note that the client-side form processing code encodes restrictions on user inputs. These restrictions are encoded in both: HTML and JavaScript. The HTML provides various controls to solicit user inputs e.g., drop down lists, textboxes, radio buttons, etc. Each such control embodies restrictions on user inputs e.g., a drop down list requires the user to select a value from the list. The JavaScript code checks if user
inputs possess certain properties and disallows submission to the server if these checks fail. These checks encode JavaScript code intended restrictions on user inputs. To extract these intentions we need to analyze HTML and JavaScript code relevant to forms.

The second task requires completion of the following two steps: i) generate hostile inputs by violating the application encoded restrictions and ii) determine if the application successfully processed hostile inputs. To complete the first step we want to leverage strategies from the literature on test input generation (46,47,48,49). Specifically, several researchers have modeled the test input generation problem as generating solutions to logical formulas that express some testing criteria. If we can extract restrictions imposed by the client-side code as logical formulas, an off-the-shelf constraint solver could facilitate generation of hostile inputs.

The second step can be realized through a blackbox or a whitebox approach. A blackbox approach would only require the client-side code and rely on responses generated for hostile inputs to assess success of hostile attempts whereas a whitebox approach would additionally analyze the server-side code to establish success of hostile attempts.

We aim to realize the second step using both a blackbox and a whitebox approach as they have complementary advantages. The blackbox approach does not rely on availability of the server-side code and hence can allow us to evaluate commercial applications for which the server-side source code is unavailable. Further, the blackbox analysis is agnostic about the server’s implementation (e.g., PHP, JSP, ASP) and is therefore broadly applicable, even including antiquated and proprietary server technology. However, the blackbox approach does not utilize the server-side code and hence may be imprecise e.g., server-side code may not utilize a hostile input in a sensitive operation thus causing the blackbox approach to raise a false alarm. The whitebox approach may eliminate
imprecision of the blackbox approach by factoring in the processing done by the server-side code
e.g., if the hostile input is not used in a sensitive operation, the whitebox approach can flag the
hostile attempt as unsuccessful. We therefore want to develop both, a blackbox approach and a
whitebox approach to detect parameter tampering vulnerabilities and compare their effectiveness
through experiments.

4.2 Contributions

• We develop the first systematic blackbox approach for detecting parameter tampering oppor-
tunities in web applications. We implement our approach in a tool that we call NoTamper,
which makes the following technical advances.
  – Client-side JavaScript code analysis techniques specialized to form validation code.
  – Input-generation techniques that cope with the many challenges of black-box vulnera-
    bility analysis.
  – Novel heuristics to generate and prioritize inputs that are likely to result in vulnerabili-
    ties.

• We develop the first systematic whitebox approach that generates parameter tampering ex-
  ploits by construction. We implement this approach in a tool that we call WAPTEC, which
  makes the following technical advances.
  – Server-side PHP code transformation techniques to precisely capture a run of the web
    application.
  – Techniques to incorporate affect of database operations.
• We empirically demonstrate that parameter tampering vulnerabilities are widespread and pose serious threat to vulnerable applications.

  – Using NoTamper we evaluated eight open source applications and five online commercial web sites and found 169 parameter tampering opportunities. Furthermore, starting from these opportunities, we successfully developed concrete exploits for the majority of these applications / web sites. Our exploits demonstrate serious security problems: unauthorized monetary transactions at a bank, unauthorized discounts added to a shopping cart, and so on.

  – Using WAPTEC we evaluated six open source web applications. WAPTEC produced parameter tampering exploits in each one of them and reported a total of 43 unique vulnerabilities. These vulnerabilities have serious real world consequences including privilege escalation to an administrator account, overwriting files on the web server and denial of service.

4.3 Problem Formulation

In this section, we formally describe the parameter tampering problem.

4.3.1 Problem Description

In a form submission, the client side of a web application solicits \( n \) string inputs from the user and sends them to the server for processing. Formally, each string input is a finite sequence of characters from some alphabet \( \Sigma \). We will denote an \( n \)-tuple of such inputs as \( I \), and the set of all such \( I \) as \( \mathcal{I} \).

\[
\mathcal{I} = \Sigma^* \times \Sigma^* \times \cdots \times \Sigma^*
\]
Conceptually, both the client and the server perform two tasks: checking that user-supplied inputs satisfy certain constraints, and either communicating errors to the user or processing those inputs. For the problem at hand, we ignore the second task on both the client and server and focus entirely on the constraint-checking task. Formally, constraint-checking code can be formulated as a function $I \rightarrow \{true, false\}$, where false indicates an error. We use $p_{client}$ to denote the constraint-checking function on the client and $p_{server}$ to denote the constraint-checking function on the server.

**Problem formulation.** Our approach is based on the observation that for many typical form processing web applications there is a specific relationship between $p_{server}$ and $p_{client}$: that $p_{server}$ is more restrictive than $p_{client}$. Because the server often has access to more information than the client, $p_{server}$ sometimes rejects inputs accepted by $p_{client}$. For example, when registering a new user for a website, the server will guarantee that the user ID is unique, but the client will not. In contrast, if $p_{server}$ accepts an input, then we expect $p_{client}$ to accept it as well; otherwise, the client would be hiding server-side functionality from legitimate users. Thus, we expect that for all inputs $I$

\[
p_{server}(I) = true \Rightarrow p_{client}(I) = true. \quad (4.1)
\]

The server-side constraint checking is inadequate for those inputs $I$ when the negation of this implication holds:

\[
p_{server}(I) = true \land p_{client}(I) = false. \quad (4.2)
\]

We call each input satisfying (Equation 4.2) a potential parameter tampering attack vector.
4.4 NoTamper

This section describes NoTamper, a blackbox approach to find parameter tampering opportunities in legacy web applications. Section 4.4.1 presents the high level overview of the blackbox approach. Section 4.4.2 discusses architecture of the NoTamper tool along with challenges faced by each component. Section 4.4.3 presents algorithms employed by each component.

4.4.1 Overview

Our goal is to automatically construct inputs that exercise parameter tampering vulnerabilities using a black-box analysis of the server. As mentioned before, the drawback of black-box server analysis is that we may not have sufficient information to eliminate false positives and false negatives. In particular, we may not be able to reasonably generate all of the inputs the server should be tested on, and even for those inputs that we do generate, there is no reliable way to know if the server accepts them. Our goal is therefore to identify opportunities for parameter tampering while requiring as little manual guidance as possible. In particular, we ask two things of human developers / testers: to provide hints about vital information not present on the client and to check whether or not the parameter tampering opportunities we identify are true vulnerabilities (perhaps by generating actual exploits).

Our high level approach is as follows: On the client, whose source is in HTML and JavaScript, we extract $f_{\text{client}}$: a logical representation of $p_{\text{client}}$ using techniques from program analysis. Subsequently, using logical tools, we generate inputs $h_1, \ldots, h_n$ such that $f_{\text{client}}(h_i) = false$ for each $i$. We call each such input hostile because it is designed to illustrate a possible parameter tampering attack. In addition, with the help of human developers/testers we generate inputs $b_1, \ldots, b_m$ such
that \( f_{client}(b_j) = true \) for each \( j \). We call each such input \textit{benign} because it is an input the server will process normally.

The benign inputs help assess which hostile inputs represent actual opportunities. We submit each hostile and benign input to the server, producing responses \( H_1, \ldots, H_n \) and \( B_1, \ldots, B_m \), respectively. We then compare each hostile response \( H_i \) to the benign responses \( B_1, \ldots, B_m \) to produce a score that represents the likelihood that the server accepted \( h_i \). Intuitively, each of the benign responses represent success messages from the server, and the more similar a hostile response is to the benign responses, the more likely the hostile input was successful and therefore a parameter tampering opportunity.

Finally, the hostile inputs and responses are presented to the human tester ranked by similarity to benign responses. The tester is then free to verify hostile inputs as bona fide parameter tampering vulnerabilities and explore the severity of each vulnerability by sending modified hostile inputs to the server.

\textbf{Discussion.} While we believe observation (Equation 4.1) holds for many interactive form processing applications, sometimes it does not, e.g., when the server is a generic web service (such as Google maps), and the client is an application using a portion of that service (such as a map of Illinois). While this falls outside our intended scope, \textsc{NoTamper} can be used in such settings by replacing the automatic extraction of \( f_{client} \) from HTML / JavaScript with a manually constructed \( f_{client} \). The construction of benign / hostile inputs and their evaluation then proceeds as described above. In other words, \textsc{NoTamper} treats \( f_{client} \), however it is generated, as an approximate specification for the intended behavior of the server and then attempts to find inputs that fail to
satisfy that specification. NoTamper can therefore be viewed as a formal verification tool with a program analysis front-end for extracting a specification of intended behavior.

Finally, due to the inherent limitations of black-box analysis, we cannot offer guarantees of NoTamper’s completeness; rather, we justify the utility of our approach by the severity of the real vulnerabilities we have discovered.

4.4.2 Architecture and Challenges

In this section, we discuss the architecture of NoTamper and the high level challenges addressed by each of its components. In Section 4.4.3, we discuss our implementation, focusing on our constraint language and algorithms.

Figure 3 shows the high-level architecture: the three components comprising NoTamper and how they interact. First, given a web page, the HTML / JavaScript (JS) Analyzer constructs logical formulas representing the constraint-checking function for each form on that web page. For our running example, the HTML / JavaScript Analyzer constructs the following formula \( f_{\text{client}} \) that says the parameters \( \text{quantity}_1 \) and \( \text{quantity}_2 \) must be greater than or equal to 0; length of
the parameter \texttt{name} must be 30 or less, and the parameter \texttt{card} must be one of the values in the drop-down list.

\[
\begin{align*}
\text{quantity}_1 & \geq 0 \\
\text{quantity}_2 & \geq 0 \\
\text{len}(\text{name}) & \leq 30 \\
\text{card} & \in \{1234-5678-9012-3456 \mid 7890-1234-5678-9012\}
\end{align*}
\]

The Input Generator takes the resulting formulas and any hints provided by the user and constructs two sets of inputs for the server: (i) those the server should accept (benign inputs \(b_1, \ldots, b_m\)) and (ii) those the server should reject (hostile inputs \(h_1, \ldots, h_n\)). In our example, the Input Generator constructs one benign input (variable assignment that satisfies the above formula):

\[
\{\text{quantity}_1 \rightarrow 0, \text{quantity}_2 \rightarrow 0, \text{name} \rightarrow "", \text{card} \rightarrow 1234-5678-9012-3456\}.
\]

The Input Generator also constructs a number of hostile inputs (variable assignments that falsify the formula above). Below are two such inputs that are the same as above except in (1) \texttt{quantity}_1 is less than 0 and in (2) \texttt{name} is longer than 30 characters.

1. \[
\{\text{quantity}_1 \rightarrow -1, \text{quantity}_2 \rightarrow 0, \text{name} \rightarrow "", \text{card} \rightarrow 1234-5678-9012-3456\}
\]

2. \[
\{\text{quantity}_1 \rightarrow 0, \text{quantity}_2 \rightarrow 0, \text{name} \rightarrow "aa(\ldots 27 \ a's \ldots)aa"", \text{card} \rightarrow 1234-5678-9012-3456\}
\]

The third component, the Opportunity Detector takes the hostile and benign inputs, generates server responses for each one, ranks the hostile inputs by how likely they are parameter tampering opportunities, and presents the results to an external tester for further analysis.

Below we discuss the challenges each of the three components addresses in more detail.
4.4.2.1 HTML/JavaScript Analyzer

Web page initialization. The JavaScript analysis of NoTamper specifically focuses on features / properties that concern form validation and submission. In order to analyze the JavaScript code pertaining to form processing, NoTamper simulates an environment similar to a JavaScript interpreter in a browser, including the Document Object Model (DOM). In such an environment, user interactions cause JavaScript code to be executed, resulting in changes to the JavaScript environment and the DOM. (User interactions may trigger asynchronous server requests via AJAX, but our implementation currently does not support AJAX).

To analyze the JavaScript code that actually performs validation, it is often important to understand the global JavaScript state as it exists when the browser first loads the form. To compute this global state, NoTamper executes all the initialization code for the web form concretely. It downloads external JavaScript, executes inlined JavaScript snippets, and keeps track of changes to global variables.

Identifying JavaScript validation code. To construct $f_{client}$, the HTML / JavaScript Analyzer must identify the code snippets relevant to parameter validation and understand how those snippets interact. This can be difficult because validation routines can be run in two different ways: (1) when a form is submitted and (2) in event handlers each time the user enters or changes data on the form.

A state machine naturally models the event-driven execution of JavaScript. Each state represents the data the user has entered and flags indicating which data contains an error. As the user supplies or edits data, JavaScript code validates the data and updates the error flags accordingly, resulting in a state transition. The constraints imposed by the client on some particular data set
could in theory be dependent on the path the user took through the state machine to enter that data, and hence the formula $f_{\text{client}}$ could depend upon the structure of that state machine.

NoTamper addresses this challenge by analyzing the JavaScript event handlers as if they were all executed when the form was submitted. The benefit of doing so is computational: it obviates the need to manually simulate events or consider the order in which events occur. But it also reflects a reasonable assumption users often make about data entry—that the order in which data was entered does not affect the validity of that data. For those cases where the order of data entry matters, our analysis may be overly restrictive, e.g., considering all event handlers may simulate the occurrence of mutually exclusive events.

Analyzing JavaScript validation code. Once the validation routines contributing to $f_{\text{client}}$ are identified, they must be analyzed. Such code may span several functions each of which may consist of multiple control paths. Each such control path may enforce a unique set of constraints on inputs, requiring an all-path inter-procedural analysis. Further, JavaScript may enforce constraints that are not dependent on user inputs e.g., disallow repeated submissions of a form through a global variable. The challenge is to extract only the constraints imposed on inputs by a given piece of JavaScript validation code.

NoTamper addresses this challenge by employing a mixed concrete-symbolic execution approach (48) to analyze JavaScript and identify the constraints enforced on user supplied data. Symbolic execution provides coverage of all control paths in the validation code and simulates validation of user supplied data. Concrete execution enables NoTamper to ignore code snippets not dependent on symbolic inputs and to provide a suitably initialized environment for symbolic execution.
Resolving document object model (DOM) references. JavaScript validation routines typically use the DOM to access the form input controls. In our simulation of the JavaScript environment, associating DOM references in JavaScript to HTML input controls is non-trivial but necessary for constructing $f_{\text{client}}$. Further, the DOM may be dynamically modified by JavaScript by adding / deleting additional input controls or disabling / enabling existing input controls.

NoTamper addresses this challenge by constructing the pertinent portion of the DOM from the given HTML in such a way that it is available to the JavaScript concrete - symbolic evaluation engine during execution. Additionally, this DOM is maintained during the JavaScript evaluation by simulating DOM functions that are used to modify the DOM structure.

4.4.2.2 Input Generator

The logical formulas given to the Input Generator are written in the language of string constraints (described in Section 4.4.3). The Input Generator encompasses two independent tasks: (i) constructing new logical formulas whose solutions correspond to hostile and benign inputs and (ii) solving those formulas to build concrete inputs. Here we focus on the first task, leaving the second to Section 4.4.3.

Avoiding spurious rejections. Two superficial but common forms of server-side parameter validation hide server vulnerabilities from a naïve analysis: checking that all “required” variables have values and checking that all variables have values of the right type. Without accounting for such simple parameter validation, NoTamper would have discovered only a few parameter tampering opportunities.

To address this challenge, the Input Generator constructs hostile and benign inputs where all required variables have values and all values are of the right type. NoTamper employs heuristics
Generating orthogonal hostile inputs. Each hostile input would ideally probe for a unique weakness on the server. Two hostile inputs rejected by the server for the same reason (by the same code path on the server) are redundant. In our running example, the client requires one variable \( \text{quantity}_1 \) to be greater than or equal to zero and another variable \( \text{name} \) to be assigned a value containing 30 or less characters. To avoid redundancy, NoTamper should generate one hostile input where \( \text{quantity}_1 \) violates the constraints (is less than zero) but \( \text{name} \) satisfies the constraints (is less than 30 characters), and another input where \( \text{quantity}_1 \) satisfies the constraints but \( \text{name} \) does not.

To generate such orthogonal inputs, the Input Generator converts \( f_{\text{client}} \) to disjunctive normal form (DNF) \(^1\) and constructs a hostile input for each disjunct. Generally, each disjunct represents inputs that violate \( f_{\text{client}} \) for a different reason than the other disjuncts.

Coping with incomplete information. Sometimes the formula \( f_{\text{client}} \) fails to contain sufficient information to generate a true benign or a hostile input that exposes a real vulnerability, yet a human tester is willing to provide that information. For example, many web forms only accept inputs that include a valid login ID and password, but the client-side code does not itself provide a list of valid IDs and passwords; in this case, \( f_{\text{client}} \) does not contain sufficient information for generating inputs that will be accepted by the server.

\(^1\)In our experience DNF conversion was inexpensive (despite its worst-case exponential character) because of \( f_{\text{client}} \)’s structural simplicity.
To address this issue, the Input Generator accepts hints that guide the search for hostile and benign inputs. Those hints take the form of logical constraints (in the same language as $f_{client}$) and are denoted $\sigma$. For example, to force the login variable user to the value “alice” and the password variable pass to the value “alicepwd”, the user would supply the logical statement $\text{user} = \text{“alice“} \land \text{pass} = \text{“alicepwd“}$.

**Addressing state changes.** Web applications often store information at the server, and web form submissions change that state. This can cause the set of valid inputs to change over time. For example, a user registration web form will ask for a login ID that has not already been chosen. Submitting the form twice with the same login ID will result in a rejection on the second attempt. This is problematic because NOTAMPER submits many different inputs to check for different classes of potential vulnerabilities, yet the login ID is both required and must be unique across inputs.

To address this issue, the Input Generator takes as an optional argument a list of variables required to have unique values and ensures that the values assigned to those variables are distinct across submissions. In our evaluation, generating inputs where certain variables all have unique values has been sufficient to address server-side state changes, though in general more sophisticated graybox mechanisms will be necessary (e.g., the ability to roll-back the server-side databases between test cases).

**Summary.** In total, the Input Generator expects the following arguments (1) the formula logical $f_{client}$ (representing the set of inputs accepted by the client), (2) a list of required variables, (3) types for variables, (4) a manually supplied set of constraints (hints), and (5) a list of unique variables ((4) and (5) are optional). It generates hostile inputs (a set of $I$ such that $f_{client}(I) = false$) and benign inputs (a set of $I$ such that $f_{client}(I) = true$) such that all required variables
have values, all values are of the right type, all manual constraints are satisfied, and each unique variable has a different value across all inputs. All arguments to the Input Generator are computed by the HTML / JavaScript Analyzer (as described in Section 4.4.3).

### 4.4.2.3 Opportunity Detector

The Input Generator produces a set of hostile inputs $h_1, \ldots, h_n$ and a set of benign inputs $b_1, \ldots, b_m$. The goal of the opportunity detector is to determine which hostile inputs are actually parameter tampering opportunities. The main challenge is that NOtamper must ascertain whether or not a given hostile input is accepted by the server while treating the server as a black box.

NOtamper addresses this challenge by ordering hostile inputs by how structurally similar their server responses are to the server responses of benign inputs. The more similar a hostile response is to the benign responses, the more likely the hostile input is a parameter tampering opportunity.

In our running example, consider a hostile input where the parameter $\text{quantity}_1$ is assigned a negative number. If the server fails to verify that $\text{quantity}_1$ is greater than or equal to zero, both the hostile and benign responses will present a confirmation screen, the only difference being the number of copies and total price. On the other hand, if the server checks for a negative number of $\text{quantity}_1$, the hostile response will be an error page, which likely differs significantly from the confirmation screen.

### 4.4.3 Algorithms and Implementation

This section details the core algorithms employed by NOtamper. All but one of them manipulate a logical language for representing restrictions on user-data enforced by the client. Currently, the language employed by NOtamper is built on arithmetic and string constraints. It includes the usual boolean connectives: conjunction ($\land$), disjunction ($\lor$), and negation ($\neg$). The atomic
\begin{align*}
\text{<sent>} & ::= \text{<atom>} \mid \text{<conj>} \mid \text{<disj>} \mid \text{<neg>} \\
\text{<conj>} & ::= (\text{<sent>} \land \text{<sent>}) \\
\text{<disj>} & ::= (\text{<sent>} \lor \text{<sent>}) \\
\text{<neg>} & ::= (\neg \text{<sent>}) \\
\text{<atom>} & ::= (\text{<term>} \text{<op>} \text{<term>}) \\
\text{<op>} & ::= < \mid \leq \mid > \mid \geq \mid = \mid \neq \mid \in \mid \notin \\
\text{<term>} & ::= \text{<var>} \mid \text{<num>} \mid \text{<str>} \mid \text{<len>} \mid \text{<reg>}
\end{align*}

\text{TABLE III}

\text{NOTAMPER: Language of generated formulas}

Constraints restrict variable lengths using $<$, $\leq$, $>$, $\geq$, $=$, $\neq$ and variable values using $\in$, $\notin$ in addition to the above operators. The semantics for the only non-obvious operators, $\in$ and $\notin$, express membership constraints on regular languages. For example, the following constraint requires $x$ to be a non-negative integer: $x \in [0-9]^*$. Table III shows a Backus-Naur Form (BNF) grammar defining the constraint language.

Below we describe algorithms in the order they are executed by \text{NOTAMPER}: (1) extracting client constraints from HTML and JavaScript, (2) generating the additional inputs accepted by the Input Generator component, (3) constructing logical formulas whose solutions are hostile and benign inputs, (4) solving such logical formulas, and (5) identifying similarity between hostile and benign server responses.
### Table IV

NoTAMPER: Constraints imposed by HTML form controls.

<table>
<thead>
<tr>
<th>Control</th>
<th>Example</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td><code>&lt;select name=x&gt;</code> &lt;option value=&quot;1&quot;&gt; &lt;option value=&quot;2&quot;&gt; &lt;option value=&quot;3&quot;&gt;</td>
<td>x ∈ {1</td>
</tr>
<tr>
<td>RADIO / CHECKBOX</td>
<td>`&lt;input type=radio name=x value=&quot;10&quot;&gt; &lt;input type=radio name=x value=&quot;20&quot;&gt;</td>
<td>x ∈ {10</td>
</tr>
<tr>
<td>HIDDEN</td>
<td><code>&lt;input name=x type=hidden value=&quot;20&quot;&gt;</code></td>
<td>x = 20</td>
</tr>
<tr>
<td>maxlength</td>
<td><code>&lt;input name=x maxlength=10 type=text/password&gt;</code></td>
<td>len(x) ≤ 10</td>
</tr>
<tr>
<td>readonly</td>
<td><code>&lt;input name=x readonly value=&quot;20&quot;&gt;</code></td>
<td>x = 20</td>
</tr>
</tbody>
</table>

#### 4.4.3.1 Client Constraint Extraction

Extracting the constraints enforced by the client on user-supplied data and representing them logically as $f_{client}$, is done in two steps. First, an HTML analyzer extracts three items from a given web page: (1) constraints on individual form fields, (2) a code snippet representing JavaScript executed on loading the web page as well as JavaScript executed for parameter validation performed by the client, and (3) a DOM representation of the form. Second, a concrete / symbolic JavaScript evaluator uses (3) during the symbolic evaluation of (2) to extract additional constraints that it then combines with (1). The result is the formula $f_{client}$.

**Step 1: HTML analyzer.** Table IV summarizes the constraints imposed by each HTML input control through examples. In our running example, there is a drop-down list for the card control that includes two credit card values. The resulting constraint requires card to be assigned one of the values in that list: \( card \in (1234-5678-9012-3456 \mid 7890-1234-5678-9012). \)
The construction of a JavaScript snippet representing the parameter validation performed by the client is accomplished by collecting all the event handlers (and associated scripts) and generating a single function that invokes all those event handlers, returning \texttt{true} exactly when all the event handlers return true. All the inlined JavaScript in the web page is then added as a preamble to the above script to initialize environment for the form validation JavaScript. The DOM representation for the form is constructed by recursively building the \texttt{document} object in the above JavaScript snippet i.e., the form being analyzed is initialized as a property of the \texttt{document} object which captures input controls as properties. Further, the \texttt{document} object simulates a small set of core methods that were necessary for processing forms e.g., \texttt{getElementsByld}. Currently, we do not support \texttt{document.write} or \texttt{document.innerHTML} and we are working towards adding support for these.

**Step 2: JavaScript Symbolic evaluator.** The key observation for extracting parameter validation constraints from a given JavaScript snippet is that form submission only occurs if that code returns \texttt{true}. In the simplest case, the code includes the statement \texttt{return true} or \texttt{return <boolexp>}, where \texttt{<boolexp>} is a boolean expression. In theory, the code could return any value that JavaScript casts to \texttt{true}, but in our experience the first two cases are far more common. This observation leads to the key insight for extracting constraints: determine all the program conditions that lead to \texttt{true} return values from all event handler functions.

To extract validation constraints, the symbolic analyzer begins by executing the validation code concretely. When a boolean expression with symbolic variables is encountered, the execution forks: one assuming the boolean expression is \texttt{true} and the other assuming it is \texttt{false}. Both executions replicate the existing variable values (program state) except for those affected by assuming the
boolean expression is true or false. Concrete execution then resumes. Supported DOM modification APIs act on the DOM specific to a fork.

For a given program location, the program condition is the set of conditions that must be satisfied for control to reach that point. If a fork returns false, it is stopped and discarded. If a fork returns true, it is stopped and the program conditions to reach that point are noted. Further, the DOM representation at this point reflects state of the HTML input controls while submitting the form including any modifications done by the JavaScript as well. The constraints checked on this fork are then computed by combining constraints of enabled controls in the DOM representation and program conditions using a conjunction (\( \land \)).

Once all forks have been stopped, \( f_{\text{client}} \) is computed by combining formulas for each path that returned true with disjunction (\( \lor \)).

For the running example one control path succeeds in returning true, resulting in the following formula.

\[-(\text{quantity}_1 < 0 \lor \text{quantity}_2 < 0) \land \text{len(name)} \leq 30\]

The above is then combined with constraint on variable card mentioned before to generate \( f_{\text{client}} \).

### 4.4.3.2 Hostile Input Guidance

NoTamper’s overall success depends crucially on generating interesting hostile inputs. Below we discuss the heuristics the HTML / JavaScript component uses to compute these values from a given web page. These heuristics were tested and refined by manually examining two of our test applications (SMF and LegalCase) but were left unchanged for the remainder of our experiments.

**Initial values.** While generating \( f_{\text{client}} \), NoTamper uses a heuristic to determine the intentions of default values for form fields. Some form fields are initialized with values that are simply
illustrative of the kind of input expected, e.g., the value 1 for the number of product copies. Other form fields are initialized with a value that cannot be changed if submission is to be successful, e.g., a hidden field initialized to a session identifier. Currently, NoTamper uses the default value for a hidden field as a constraint included in $f_{client}$ and considers the default value for all other fields as illustrative of the expected value. In either case, the list of initial values is provided to the input generator and used for other heuristics as described below.

**Types.** The type for each variable controls the set of possible values occurring in both the hostile and benign inputs. Choosing appropriate types can greatly improve the odds of success. In our running example, if the type of $quantity_1$ were the positive integers, the input generator would never find the vulnerability that appears when $quantity_1$ is less than zero. Similarly, if the type of $quantity_1$ were all strings, the likelihood that the generator randomly chooses a string that represents a negative integer is unlikely. Currently, NoTamper chooses a type for each variable based on (i) its occurrence in arithmetic constraints, (ii) the HTML widget associated with that variable, and (iii) its initial value. Occurrence in an arithmetic constraint implies a numeric type. An HTML widget that enumerates a set of possible values implies a value drawn from the set of all characters in the enumerated values. An initial value that is numeric also implies a numeric type. Integers are assumed unless there is evidence that real values are required.

**Required variables.** The list of required variables ensures that every hostile input includes a value for every variable in the list. Choosing too small a list risks hostile inputs being rejected because they did not pass the server’s requirements for required values, and choosing too large a list can cause the server to reject hostile inputs because unnecessary variables are given invalid values. NoTamper employs two techniques for estimating the required variables. One is analyzing
the HTML for indications that a variable is required, e.g., asterisks next to field labels. The other is extracting the variables from $f_{client}$ that are required to be non-empty, e.g., the variable cannot be the empty string or the variable must be assigned one of several values (from a drop-down list).

**Unique variables.** When a variable appears in the unique variable list, every pair of hostile inputs differs on that variable’s value. This is useful, for example, when testing user registration pages, where submitting the same user ID twice will result in rejection because the ID already exists. Choosing too large a list, however, can result in fewer hostile inputs being generated and therefore fewer vulnerabilities being found. For example, if a field can only take on one of three values and is required to be unique across all hostile inputs, at most three inputs will be generated. Currently, NoTamper is conservative in the variables it guesses should be unique. If there is any indication that a variable can only take on a small number of values, it is not included in the unique list.

### 4.4.3.3 Input Generation

The Input Generator constructs a series of formulas in the constraint language whose solutions correspond to hostile and benign inputs. Here we detail how the construction of formulas for benign and hostile inputs differ.

**Benign inputs.** To generate benign inputs satisfying $f_{client}$, NoTamper converts $f_{client}$ to DNF\(^1\), augments each disjunct with the user-provided constraints $\sigma$ and required-variable and type constraints, and finds one solution per disjunct.

In the running example, suppose $f_{client}$ is the formula $(\text{quantity}_1 > 0 \lor \text{quantity}_1 = 0) \land (\text{len(name)} \leq 30)$. NoTamper finds one solution for $\text{quantity}_1 > 0 \land \text{len(name)} \leq 30$ and another for $\text{quantity}_1 = 0 \land \text{len(name)} \leq 30$. If the type of $\text{quantity}_1$ is $[0-9]^+$ and the type of
name is [a-zA-Z]•, NoTamper includes the constraints \( \text{quantity}_1 \in [0-9]^+ \) and name \( \in [\text{a-zA-Z}]^* \).

If \( \sigma \) is nonempty, NoTamper includes it as well.

Satisfying the unique variable constraint is accomplished by keeping track of the values assigned to each variable for each generated input and adding constraints that ensure the next value generated for each unique variable is distinct from those previously generated.

**Hostile inputs.** To generate hostile inputs, NoTamper starts with \( \neg f_{\text{client}} \) instead of \( f_{\text{client}} \) and then proceeds as for the benign case with one exception: filling in values for required variables. Consider any disjunct \( \delta \) in the DNF of \( \neg f_{\text{client}} \). If all the required variables occur within \( \delta \), NoTamper simply finds a variable assignment satisfying \( \delta \) and returns the result; otherwise, NoTamper augments that assignment with values for the required variables not appearing in \( \delta \). To do so, it finds values that satisfy \( f_{\text{client}} \). The hope is that if the server rejects the input it is because of the variables appearing in \( \delta \), not the remaining variables; otherwise, it is unclear whether or not the server performs sufficient validation to avoid the potential vulnerability \( \delta \).

In the example above, the disjunctive normal form of \( \neg f_{\text{client}} \) produces a formula with two disjuncts.

\[
\bigvee \neg(\text{quantity}_1 > 0) \land \neg(\text{quantity}_1 = 0) \\
\quad \neg(\text{len(name)} \leq 30)
\]

Suppose that both \( \text{quantity}_1 \) and name are required. The first disjunct does not include name, and the second does not include \( \text{quantity}_1 \). After solving the first disjunct with, for example, \( \text{quantity}_1 = -1 \), NoTamper assigns name a value that satisfies the original formula, i.e., that
\begin{table}
\begin{tabular}{|l|l|}
\hline
\textsf{len(<var>) = len(<var>)} & <\var> \otimes <\var> \\
\textsf{<var> \neq <var>} & <\var> \otimes \textsf{len(<var>)} \\
\textsf{<var> \neq len(<var>)} & \textsf{len(<var>) \otimes len(<var>)} \\
\textsf{len(<var>) \neq len(<var>)} & <\var> \oplus <\textsf{reg}> \\
\hline
\end{tabular}
\caption{TABLE V}
\end{table}

NoTamper: The reduced constraint language: \(\land\) and \(\lor\) over the above atoms. \(\otimes\) is one of \(<,>,\leq,\geq\). \(\oplus\) is either \(\in\) or \(\not\in\).

satisfies \(\text{len(name) \leq 30}\). Likewise, after solving the second disjunct producing a value for \textsf{name}, NoTamper assigns \textsf{quantity}_1 a value that satisfies the original formula, e.g., \(\text{quantity}_1 = 1\).

4.4.3.4 Constraint Solving

To solve formulas in the constraint language, NoTamper uses a custom-written constraint solver built on top of HAMPI (50), a solver that handles a conjunction of regular language constraints on a single variable of a fixed length. Our formula involves multiple variables, and therefore we developed our own procedure that uses HAMPI as described below.

NoTamper handles disjunction by converting a given formula to DNF \(^1\) and solving each disjunct independently. For a given disjunct (which is a conjunction), NoTamper performs type inference to determine which variables are numeric and which are strings, extracts bounds on the size of all variables, and simplifies the disjunct to produce a conjunction of atoms from Table Table V. Then applies Algorithm 1 to search for a variable assignment satisfying the resulting conjunction.

Algorithm 1 takes as input a list of variables that require values, a logical formula, a partial variable assignment, and a function that maps each variable to that variable’s bounds. It either
returns unsat (denoting that no satisfiable assignment is possible) or an extension of the given variable assignment that satisfies the logical formula.

The first step of the algorithm is choosing a variable to assign. Currently, NOTAMPER chooses the variable with the smallest range of possible lengths. Then search commences. String variables and numeric variables are treated differently. For numeric variables, NOTAMPER loops over possible values and for each one checks that assigning the variable the current loop value satisfies the constraints. If satisfaction holds, the variable is assigned the loop value.

Algorithm 1 \textsc{solve}(vars, \phi, asgn, BOUNDS)

\begin{algorithmic}
  \STATE if vars = \emptyset then return asgn
  \STATE values := \emptyset
  \STATE var := \textsc{choose}(vars, \phi, asgn, BOUNDS)
  \FORALL i in \textsc{low}(BOUNDS(var)) .. \textsc{high}(BOUNDS(var))
    \STATE if \textsc{numeric-var}(var) then
      \STATE if \textsc{sat}(\phi, asgn \cup \{var \rightarrow i\}) then
        \STATE newasgn := \textsc{solve}(vars\setminus\{var\}, \phi, asgn \cup \{var \rightarrow i\}, BOUNDS)
        \STATE if newasgn \neq unsat then return newasgn
      \STATE else
        \STATE if not \textsc{sat}(\phi \land \textsc{len}(var)=i, asgn) then goto next i
        \STATE loop
        \STATE val := \textsc{hampi}(\phi|_{var} \land var \not\in values, i)
        \STATE if val = unsat then goto next i
        \STATE values := values \cup \{val\}
        \STATE if \textsc{sat}(\phi, asgn \cup \{var \rightarrow val\}) then
          \STATE newasgn := \textsc{solve}(vars\setminus\{var\}, \phi, asgn \cup \{var \rightarrow val\}, BOUNDS)
          \STATE if newasgn \neq unsat then return newasgn
        \STATE return unsat
  \STATE return unsat
\end{algorithmic}
For strings, NoTAMPER loops over possible lengths (as opposed to possible values), and for each one satisfying the length constraints invokes HAMPI to generate a variable assignment. HAMPI takes as input a logical formula with one variable and a length for that variable. It either returns unsat or a value satisfying the formula. Reducing the given formula $\phi$ with multiple-variables to a formula with just the chosen variable, denoted $\phi|_{\text{var}}$, is performed by selecting the subset of constraints where only the chosen variable occurs. If HAMPI finds a satisfying value, the algorithm checks that the value satisfies the relevant constraints HAMPI does not check: those constraining multiple variables. Additionally, the algorithm keeps a list of values HAMPI returns so that if the search fails at a later point in the search, and another value needs to be generated for the current variable, we can augment the logical formula given to HAMPI to require a value not already chosen.

Once a variable has been assigned a value, Algorithm 1 recurses on the original variable list after having removed the chosen variable, the original logical formula, the original variable assignments augmented with the chosen variable’s assignment, and the original variable bounds. When the variable list becomes empty, the algorithm returns the given variable assignment, indicating that all constraints are satisfied by that assignment. If no such assignment can be found, the algorithm returns unsat.

4.4.3.5 HTML Response Comparison

In order to determine whether hostile inputs were accepted by the server, our approach compares the server’s response against a response that is known to have been generated by benign (valid) inputs. Since the server’s responses are in HTML, we have to employ HTML similarity detection. There are many similarity detection algorithms for HTML responses in the literature, the most notable being algorithms for computing tree edit distance (ref. (51)). These are especially useful
in case of documents derived from a variety of sources that may contain similar content (e.g., news articles from various newspapers). In our case, since the HTML documents are produced by a single web application, it is very likely that these responses are structurally more aligned than documents from different sources, and therefore we use a home-brewed document comparison strategy based on the Ratcliff and Obershelp algorithm (52) on approximate string matching.

**Approximate matching.** An important issue to be addressed in response comparison is that the contents of a HTML response will frequently include a number of variable elements that are not dependent on the server inputs, e.g., time stamps, user names, number of people logged in. A large number of such elements introduce differences in benign responses, even when the inputs are identical; therefore, we resort to an approximate matching strategy that filters out such noise from benign responses before comparing to hostile responses.

Suppose we have just two benign responses $B_1$ and $B_2$. Analyzing these responses and extracting their differences will often isolate the noisy elements in the page. These noisy elements can then be removed. For this purpose, we developed a utility that analyzes these two responses and returns the following: (1) the common sequences in $B_1$ and $B_2$ (2) content in $B_1$ that is not in $B_2$, and (3) content in $B_2$ that is not in $B_1$. Elements (2) and (3) comprise the noise, and once eliminated from $B_1$ and $B_2$ respectively, we arrive at the same HTML document $C_1$.

To analyze hostile response $h_i$, we repeat the noise elimination procedure, only this time with files $B_1$ and $H_i$. The resulting HTML, $C_2$, produces two possibilities, depending on whether the input $h_i$ was accepted or not. If the input was accepted, based on our observation above, the server response $H_i$ is likely to be similar (modulo noise) to $B_1$, and therefore the result $C_2$ is likely to be
structurally the same as $C_1$. In case the input was rejected, the server returns a response that is likely to be structurally dissimilar, and therefore $C_2$ will be less similar to $C_1$.

The final step is the comparison between $C_1$ and $C_2$. Again, a naive comparison will not work because of the possibility that not all noise causing elements were removed during the earlier step. For example, page generation times are often embedded in the page itself, if the times were the same for $B_1$ and $B_2$, but different for $H_1$, then $C_1$ and $C_2$ will not be strictly structurally the same. Instead, we again use our approximate matching strategy on $C_1$ and $C_2$ as inputs. Only this time, we compute the edit distance between the two structures, resulting in a numeric value (that we call difference rank) for each hostile input. The higher the rank for a given hostile input, the less likely it is that the input points to a potential vulnerability.

**Complexity.** Our comparison strategy for HTML files is based on the gestalt pattern matching procedure (52), which itself finds the longest common subsequence between HTML files, and then recursively finds the common elements to the left and right of the common sequence. Our procedure has linear complexity in its best case and has quadratic worst-case complexity.

**4.4.3.6 Implementation**

The HTML analysis was implemented on top of the APIs provided by the HTML Parser (53), specifically using visitors for `<form>` and `<script>` tags. The JavaScript analysis was performed using a modified Narcissus JavaScript engine-based symbolic evaluator. Narcissus is a meta-circular JavaScript interpreter that uses SpiderMonkey JavaScript engine’s interfaces.

The Input Generator was built as a wrapper around the solver HAMPI (50) using the subroutine library Epilog (54) for manipulating logical expressions written in of 1700 lines of Lisp code.
The Opportunity Detector was primarily implemented in Java. Based on inputs generated by the constraint solver, a Java-based module relayed HTTP requests to the test server, saved the responses for processing, and implemented algorithm to compute the difference rank.

This section provided details of NoTamper, a blackbox approach to automatically identify parameter tampering opportunities in web applications. The next section presents a whitebox approach to generate parameter tampering attacks.

4.5 **WAPTEC**

This section presents WAPTEC, a whitebox approach to automatically generate parameter tampering exploits by construction. Section 4.5.1 presents the high level overview of a whitebox approach. Section 4.5.2 discusses architecture of the WAPTEC tool along with challenges faced by each component. Section 4.5.3 presents algorithms employed by each component.

4.5.1 **Overview**

**Running example.** We add three additional lines to the server-side code of our running example (Listing 2.2). This additional code is shown in Listing 4.1 and checks the status of query execution and reports an error if it fails (lines 23 to 25). Intuitively, this code captures a database integrity constraint that the `address` parameter must be initialized.

```php
Listing 4.1. server.php
22 mysql_query($q);
23 if (mysql_error()){
24     $html .= " Please specify an address";
25 }
```
Basic idea. The high level idea behind our analysis of a server to find parameter tampering vulnerabilities is illustrated in Figure 4 and can be described as follows. We start by modeling the client side constraints as a formula $f_{\text{client}}$ (steps 1-2 in Figure 4). The client formula for our running example will include $\text{quantity}_1 > 0$. By solving this formula using a constraint solver, we generate (step 3) an input $i_b$ that is considered “benign”, i.e., satisfies $f_{\text{client}}$ and upon submission to the server, leads to execution of a sensitive operation (such as the \texttt{INSERT} query in line 22 of Listing 2.2). We then analyze the trace of the server’s response to that input and model the operations that were performed as another formula $f_{\text{server}}$ (steps 4-5). Using constraint solving, we look for solutions to the formula $(\neg f_{\text{client}} \land f_{\text{server}})$ in steps 6-7. The solution to this formula signifies an input $i_h$ that does not respect the client validation, i.e., $(\neg f_{\text{client}})$, but goes uncaught by the server-side validation represented by the conjunction $f_{\text{server}}$, and is therefore an exploit.
In our running example, $f_{server}$ imposes no conditions on $\text{quantity}_1$; thus, $-(\text{quantity}_1 > 0)$ represents a set of vulnerabilities. A solution to this formula, e.g., $\text{quantity}_1 = -1$, is an exploit leading to an attack with a negative total cost for a product. Since our approach generates inputs that do not respect client validation but do reach a sensitive operation at the server, we generate exploits by construction.

The basic idea sketched above is in fact a two step process:

- find inputs leading to sensitive operations (§4.5.1.1).
- check if these operations are vulnerable to attack (§4.5.1.2).

### 4.5.1.1 Finding inputs leading to sinks

Intuitively, the client spec (or formula) $f_{client}$ gives a way to generate inputs that lead to sensitive operations. That is, if the form submission is successful at the client, we generally expect the server to accept it. For example, submitting the form in our running example with a positive integer value for quantity will result in execution of the (sensitive) INSERT query in line 22 (Listing 2.2). Thus, by finding an input that satisfies the client formula, we can often generate an input that leads to a sensitive sink.

Sometimes, however, the server has more information than the client and enforces constraints that the client cannot. In our example, the server checks if the user supplied address is non-null (i.e., address is a required field in the transaction). Since the client does not check this to be the case in our example, merely generating an input satisfying the client spec may not lead to a sensitive sink. This also highlights the nature of the client spec as (inherently) incomplete. However, if the client spec can be appropriately augmented, then generating any input satisfying that augmented spec will induce execution along a control path leading to a sensitive sink.
To appropriately augment the client spec, we begin by submitting an input satisfying the client spec to the server and observe the code trace the server executes in response (see Section 4.5.2.3). If the server performs a sensitive operation, we call that operation a \textit{success sink} and proceed towards checking the server for vulnerabilities that exploit this success sink. On the other hand, if the server fails to perform a sensitive operation, we analyze the code trace that was executed, extract a logical formula representing that trace (called $f_{server}$, discussed in Section 4.5.2.2), and use it to augment the client spec (steps 5 followed by step 9). We then repeat the process of constraint solving, this time starting with the augmented spec, finding an input satisfying the augmented spec (see Section 4.5.2.1), and iterating until we find a “good” input that forces the server to execute a sensitive operation.

The way we use the $f_{server}$ formula to augment $f_{client}$ is in fact a form of constraint-guided search for a sensitive operation. It turns out that $f_{server}$ always takes the form of a conjunction $C_1 \land \cdots \land C_n$ (see Section 4.5.3.1), and any input satisfying that conjunction fails to reach a sensitive operation; thus, to augment $f_{client}$, we add (conjoin) the negation: $\neg(C_1 \land \cdots \land C_n)$. More generally, each time an input satisfying (the augmented) $f_{client}$ fails, we augment $f_{client}$ with the negation of $f_{server}$.

Once we find an augmentation of $f_{client}$ and a satisfying input leading to a sensitive operation, we perform a depth-limited version of the procedure above to find additional, nearby control paths that lead to sensitive operations, essentially “perturbing” the successful benign input to search around the “neighborhood” of the success sink. If $f_{server}$ is $C_1 \land \cdots \land C_n$, for each $C_i$, we augment $f_{client}$ with $\neg C_i$, find a satisfying input, and check if that input leads to a sensitive operation. Each $C_i$ can potentially produce a distinct control path leading to a sensitive operation; thus, after this
depth-limited search we have between 1 and \( n + 1 \) control paths leading to sensitive operations, where \( n \) is the number of conjuncts in \( f_{server} \).

### 4.5.1.2 Checking server paths for vulnerabilities

For each control path we find that leads to a sensitive operation, we attempt to generate inputs that the server should not accept but that invoke that same sensitive operation. Generating inputs the server should not accept is straightforward: find solutions to the negation of the client spec (i.e., \( \neg f_{client} \)), for if the client rejects a given input, the server code intends to reject that as well, since the client code is indeed a specification of the server’s intentions. Generating inputs that cause the server to follow the same control path is likewise straightforward: find solutions to the logical formulae representing that control path, which in this case is \( f_{server} \). Thus, generating inputs that lead to the sensitive sink we identified but that the server should not accept amounts to finding a solution to \( \neg f_{client} \land f_{server} \). Conceptually, every such solution would amount to a parameter tampering exploit, but to ensure that the input is in fact an exploit, we submit it to the server to ensure it reaches a sensitive operation (step 9 in Figure 4), and verify the results.

An algorithmic summary of the discussion in §4.5.1.1 to generate benign inputs is found in Algorithm 2 and the discussion in §4.5.1.2 for generating exploits is summarized in Algorithm 3.

**Negative parameter tampering.** Sometimes a server side file, such as `server.php` is written to handle multiple forms. In the running example (Listing 2.2), the server-side code additionally checks for parameter discount. While this code was intended for processing a totally different form that contains discounts for the user, it is not uncommon for applications to reuse the code that has some shared processing of content. An exploit that introduces this field discount can result in
Algorithm 2 WAPTEC(url)
1: \( f_{\text{client}} := \text{clientAnalyzer}(url) \)
2: \( Q := \{\text{true}\} \)
3: loop
4: \( \alpha := \text{pop}(Q) \)
5: \( \nu := \text{solve}(f_{\text{client}} \land \alpha) \)
6: \( (\text{success}, f_{\text{server}}) := \text{server}(url, \nu) \)
7: if success then
8: \( \text{genHostiles}(url, f_{\text{client}}, f_{\text{server}}) \)
9: for all \( C_i \mid f_{\text{server}} = C_1 \land \cdots \land C_m \) do
10: \( \nu := \text{solve}(f_{\text{client}} \land \alpha \land \neg C_i) \)
11: \( (\text{success}, f_{\text{server}}) := \text{server}(url, \nu) \)
12: if success then \( \text{genHostiles}(url, f_{\text{client}}, f_{\text{server}}) \)
13: else
14: \( Q := Q \cup \{\alpha \land \neg C_i \mid \neg f_{\text{server}} = \neg C_1 \lor \cdots \lor \neg C_m\} \)
15: \( Q := \text{simplify}(Q) \)
16: if \( \text{empty}(Q) \) then return

Algorithm 3 genHostiles(url, f\text{client}, f\text{server})
1: for all \( \delta \in \text{DNF}(\neg f_{\text{client}}) \) do
2: \( \nu := \text{solve}(\delta \land f_{\text{server}}) \)
3: \( \text{success} := \text{server}(url, \nu) \)
4: if success then print Exploit found: \( \nu \)
providing unlimited discounts to the total price. We call this negative tampering, as it is performed by an input field that is not present in the original form.

By whitebox analysis of server side code, we are able to identify such vulnerabilities. We first identify a parameter $p$ such that $f_{server}$ contains one or more conjuncts on $p$ but $f_{client}$ does not i.e., the server-side code checks values of $p$ but the client-side code does not. Intuitively, such parameters represent values that the server did not expect to receive with the form that was used in generating $f_{client}$. To find negative tampering we check if the server processes the request when we set such parameters. For each conjunct $c$ on parameter $p$, we generate inputs for $f_{client} \land (f_{server} - c) \land -c$ i.e., remove the conjunct from $c$ from $f_{server}$ and instead satisfy its negation. We confirm negative tampering if the server-side code uses the above value of parameter $p$ in a sensitive operation. For the running example, $f_{server}$ contains a conjunct $\neg \text{isset($\_POST['discount']$)}$ on the discount parameter which the $f_{client}$ does not. We negate this conjunct an set the discount parameter to 1. When submitted to the server, it results in the discount parameter being used in the sink thus confirming a negative tampering vulnerability.

We note that not all negative tampering attempts may reveal vulnerabilities. Specifically such attempts may also yield runs that exercise additional functionality of an application. However, hidden parameters such as discount have been used in implementing backdoors in applications (55). Further, using this strategy we found an actual vulnerability in an open source application that is exploitable with a privilege escalation attack (Section 4.6.2). This discussion provides the impetus for WAPTEC to report all successful negative tampering attempts as vulnerabilities even though some of them may lead to false positives.
4.5.1.3 Discussion

It is important to describe at a high level the mechanisms that we use for generating the client formula \( f_{\text{client}} \) and the server formula \( f_{\text{server}} \), and their implications for the correctness and precision of our approach.

The client formula \( f_{\text{client}} \) is generated by the HTML / JavaScript Analyzer (shown in Figure 4), and is computed by previously discussed algorithm (Section 4.4.3.1). Since the formula is statically computed from the source, the generated formula is in fact an approximation. Specifically, due to the nature of the approximations made in Section 4.4.3.1 \( f_{\text{client}} \) is an under-approximation of the client behavior, which means that every time a benign input \( i_b \) is generated that satisfies \( f_{\text{client}} \), it is indeed the case that this input will lead to a successful form submission from the client. Similarly, \( \neg f_{\text{client}} \), represents an over-approximation of input instances that are rejected by the client (e.g., line 15 of client code listing 2.1 in our running example). While solving \( \neg f_{\text{client}} \) to generate an exploit, we cannot therefore assume that each one of these solutions will actually be rejected, so we execute those exploits and check if they are indeed rejected by the client.

In our approach, the server side behavior is obtained by dynamic analysis of server side code. This means that the server side formula \( f_{\text{server}} \) will be specifically tied to each run, and is generated from the program trace induced by the run. By its very nature, dynamic analysis only considers the operations done by code that is executed, \( f_{\text{server}} \) precisely captures the server behavior for the run without any approximations.

Since \( f_{\text{server}} \) is precise, then any solutions to \( \neg f_{\text{client}} \land f_{\text{server}} \) that actually are rejected by the client represent concrete parameter tampering exploits based on the above discussion. Our implementation seeks to find such exploits.
4.5.2 Architecture and Challenges

Our approach to automatically detecting parameter tampering exploits relies on four main components shown in Figure 4. The Client Formulae Extractor (Section 4.4.3.1) analyzes HTML and JavaScript of the client to produce the specification for correct server behavior: \( f_{\text{client}} \). The String Solver (Section 4.5.2.1) generates inputs satisfying logical formula (such as \( f_{\text{client}} \) and \( \neg f_{\text{client}} \)) that are sent to the server to test its behavior. The Trace Analyzer (Section 4.5.2.2) examines a trace of the server's code execution to determine if the server invoked a sensitive operation in response to one of the inputs constructed by the String Solver. The Trace Generation Transformer (Section 4.5.2.3) instruments the server to produce the traces required by the Trace Analyzer.

4.5.2.1 String Solver

The string solver component analyzes logical formulae to construct inputs that are fed to the server; some of those inputs the system was designed to accept, while other inputs are intended to expose server-side vulnerabilities. The string solver component of WAPTEC was built on top of Kaluza (56), a state-of-art solver that finds variable assignments satisfying string and numeric constraints. The main challenge in building the string solver component was translating the WAPTEC constraint language into the language supported by Kaluza.

**Constraint language.** WAPTEC allows all boolean combinations of the atomic constraints shown in Table VI. The equality and numeric constraints are standard; regular expression constraints require a variable to belong to a given regular expression; \texttt{PHP} constraints include functions from \texttt{PHP} and JavaScript such as \texttt{trim} (found in e.g., the MyBloggie application) for removing whitespace from the ends of a string and \texttt{strpos} for computing the index at which one string appears inside another string. Kaluza roughly supports those categories of constraints marked with
<table>
<thead>
<tr>
<th>Class</th>
<th>Examples</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality*</td>
<td>=, ≠</td>
<td>x ≠ y</td>
</tr>
<tr>
<td>Numeric*</td>
<td>+, *, −, /, &lt;, &gt;</td>
<td>x &lt; 7</td>
</tr>
<tr>
<td>Modal</td>
<td>required</td>
<td>required(x)</td>
</tr>
<tr>
<td>Regex*</td>
<td>∈, ∉</td>
<td>x ∈ [abc]*</td>
</tr>
<tr>
<td>PHP</td>
<td>trim, len, concat</td>
<td>len(x) &lt; len(concat(y, z))</td>
</tr>
</tbody>
</table>

TABLE VI

WAPTEC constraint language

an asterisk, plus functions for computing the length of a string and concatenating two strings. Thus, translating WAPTEC’s constraint language to Kaluza’s language requires handling modals and PHP functions.

**Static versus dynamic typing.** Besides the difference in atomic constraints, there is a more fundamental difference between the constraint languages of Kaluza and WAPTEC. Kaluza requires every variable to have a single type and does not provide functions to cast from one type to another\(^1\), whereas PHP allows variables to take on arbitrary values. This mismatch makes the translation difficult because a constraint such as \(x ≠ 0 ∧ x ≠ "0"\) causes a type error in Kaluza but appears frequently in the semantics of PHP, e.g., when defining whether a variable evaluates to true or false.

Our approach approximates the semantics of PHP functions with a combination of type inference to detect type mismatches, type resolution to choose one type for mismatched arguments, static casting to convert problematic arguments to the chosen types, and type-based simplification to

---
\(^1\)Type casting functions, while included in the documentation, were unavailable at the time of evaluation.
eliminate constraints that do not actually affect the satisfiability of the constraints but cause Kaluza to throw type errors.

**Untranslatable constraints.** Some of WAPTEC’s constraints cannot faithfully be translated into Kaluza’s constraint language. For example, PHP employs a number of built-in data structures not handled by Kaluza, and PHP functions often accept and return such data structures. For example, MyBloggie employs the `preg_replace` function, which is a regular-expression version of a string replacement operation. `preg_replace` can both accept and return arrays as arguments. Arrays are difficult to translate to Kaluza because they correspond to an unknown number of variables, and Kaluza expects a fixed number of variables in the constraints. Another example of a function we did not translate is found in DCPPortal application: the `md5` function computes the MD5 hash of its argument.

For constraints that cannot be translated to Kaluza’s language, WAPTEC simply drops those constraints, producing a constraint set that is weaker than it ought to be, potentially leading to unsoundness and incompleteness in the search for parameter tampering exploits. However, because WAPTEC always checks if the variable assignment produced by the solver satisfies the original constraints, unsound results are never reported.

**Disjunction.** As mentioned above, disjunction is employed heavily by WAPTEC, and while Kaluza handles disjunction natively, the search for parameter tampering exploits sometimes requires finding different solutions for different disjuncts in a set of constraints—functionality Kaluza does
not support. Thus WAPTEC manages disjunctions itself, sometimes converting to disjunctive normal form (DNF)\(^1\) explicitly.

### 4.5.2.2 Trace analyzer

After WAPTEC generates inputs to the server using the String Solver, it analyzes a trace of the server’s execution to decide how to proceed. The main task of the analyzer is constructing \(f_{server}\), which is obtained by combining the conditionals along the execution path leading to SQL query or file I/O operations, and then representing this condition in terms of user inputs. We briefly discuss the challenges faced in this process below.

**Eliminating redundant conditions and variables.** Traces contain conditions evaluated on tainted data during a run of the web application. A naive approach may consider all such conditions as relevant to a sink which may lead to imprecise analysis. For example, consider a trace of the running example (Listing 2.2). The lines 2 through 6 check the value of parameter `card` and set the HTML form to show the selected entry. Although the trace contains check on `card`, it does not prevent the query computed at line 22 from using malicious values of `card`. Similarly, a form may contain several parameters but a server side sink may only use some of them. Analysis must factor this in as tampered parameter may not be used at a sensitive operation.

WAPTEC identifies conditionals relevant to a given sink. To do so, we employ standard data-and control-dependency analysis which identifies conditionals that either contributed data to a sink or dictated control flow to a sink. For the running example, the query generated at line 22 is not

---

\(^1\)In our experience, converting to DNF was usually inexpensive (despite its worst-case exponential behavior) because of the structural simplicity of the constraint sets.
data- or control-dependent on conditional statement at line 2 and hence this conditional is ignored while analyzing sink at line 22.

**Extracting database constraints.** Database query operations present interesting consequences for approaches that analyze server-side code. With respect to such operations, many security analysis approaches limit their reasoning to reachability e.g., most tainting approaches aim to find if a tainted data item can reach a database query. Without analyzing outcome of the query execution such approaches will result in imprecision as database engine may either sanitize hostile inputs to comply with its schema or reject them. For blackbox approaches database triggered sanitization may result in false alarms. Whereas whitebox approaches that ignore these constraints may never generate a benign set of inputs that will be accepted at the sink. For our running example, without considering database constraint (NOT NULL) on `address` field it is not possible to generate acceptable benign inputs. This also forbids discovery of legitimately exploitable parameters for such sinks thus resulting in false negatives e.g., `quantity_1` exploit cannot be constructed without providing a non-null address value.

WAPTEC offers a *deep* analysis of server-side code by incorporating constraints implied by database schema of applications. The main challenge in computing these constraints is to bridge the namespace differences between application code and database schema i.e., application code and database tables may refer to same data with different names. WAPTEC computes and analyzes symbolic queries from traces which enables it to map database names to program variables.

### 4.5.2.3 Trace Generation Transformer

In order for the Trace Analyzer to examine a trace of the code the server executes, the server must generate such a trace whenever it processes inputs. The Trace Generation Transformer is the
component of WAPTEC that is run on server’s code offline to instrument it to generate that trace. It is a PHP source-to-source transformer that causes the instrumented server code to behave exactly the same as the original server code, except it produces a trace of the code that was executed at the end of its output. The resulting trace is a straight-line program that is comprised only of assignments, calls to inbuilt functions and IF-THEN statements. The key challenges and how WAPTEC addresses them are discussed below.

**Trace simplification.** The primary challenge faced in this transformation was to represent computations done by generic PHP programs with a restricted instruction set. To simplify further analysis, traces must not contain functions and therefore all executed functions (except PHP inbuilt) need to be in-lined. Such in-lining requires care as merging variables from several lexical scopes could result in *name collisions* and could generate traces that misrepresent run of the web application e.g., name-collisions could result in traces that incorrectly capture use / reachability of an important variable.

Intuitively, assignments, calls to inbuilt functions and IF-THEN statements suffice to capture a run of the web application, and constitute the required instruction set. All control decisions made in a run can be represented by IF-THEN statements with conditions which always evaluate to true. Each iteration of a loop can be represented by statements executed in that iteration which are enclosed in an IF-THEN statement that captures guard condition of the loop. Therefore, the offline instrumentation performs a systematic transformation of each program construct to capture its execution in the target instruction set. To avoid name-collisions, the transformation ensures renaming of variables appearing in traces by appending unique prefixes e.g., for variables
appearing in a function, signature of the function is used as a prefix. This scheme does not work for object-oriented features which are discussed next.

**Capturing object-oriented operations in traces.** Object-oriented features are often used in PHP programs (2 of the 6 applications we evaluated were object-oriented and used inheritance). As multiple instantiations of a class yield objects with same methods, method signatures are same for all such objects. Thus appending signatures to variable names may still lead to name collisions in object-oriented programs. Further, a member variable can be accessed using multiple namespaces e.g., by using the *this* operator (inside methods) or by using names assigned to objects. Although, all such instances are accessing the same memory region, a naive renaming scheme may lose precision by failing to identify these accesses with a single variable name.

The key idea used in the instrumentation is to augment object instantiation process to uniquely identify each object and use this additional information in avoiding name-collisions. To do so, constructor methods are transformed to assign a unique identifier to each object. All accesses to member variables are uniformly renamed with the help of this identifier (*this* or object names are eliminated as a result). Similarly, local variables in a method are prefixed with the identifier as well as the signature of the method.

**Generating well-formed traces.** Intuitively, a trace is a program induced by a run of the web application. For the convenience of being able to employ program analysis tools/techniques, traces must be well formed PHP programs. To do this correctly, even simple tasks such as outputting equal number of scope delimiters with each IF-THEN statement is complicated by the presence of certain language features e.g., *exit, die, break, continue*, etc. These construct alter normal flow of
execution and may transfer control to immediately nesting scope (e.g., break), any nesting scope (e.g., break <level>) or terminate the execution (e.g., exit).

WAPTEC adopts several strategies to ensure that it generates well-formed traces e.g., for outputting matching scope delimiters, WAPTEC keeps track of enclosing lexical scopes through a counter. Before original program executes a flow altering statement, the correct number of scope delimiters are written to the trace to retain the well-formedness of the trace.

**Identifying conditional statements evaluated on user supplied data.** Identification of such conditionals requires two capabilities: a) uniformly identify conditions evaluated in various program constructs and b) identify conditions evaluated on user supplied data.

Several PHP constructs evaluate code conditionally e.g., IF-ELSE, FOR, etc. The challenge here was to uniformly capture conditions evaluated in all such constructs. The WAPTEC transformer simplifies programs by using the open source PHP compiler (PHC) (57) and then re-writes program constructs to make evaluated conditions explicit through IF-ELSE statements. This enables transformer to uniformly report conditions evaluated in all PHP constructs.

To identify conditionals evaluated on user inputs, we used standard information flow (taint) for tracking user supplied data (58, 59). Here we limit the discussion to challenges specific to this problem setup: a) taint initialization and b) inbuilt functions.

In PHP, user supplied data in an HTTP request is accessible through global arrays namely: 
- GET, POST, REQUEST, FILE and COOKIES. We initialized taint arrays for each one of these. Such an initialization requires careful consideration as PHP recursively defines some of these arrays e.g., super global array GLOBALS contains itself as a member.
In-built functions may generate complex data types (e.g., arrays). The standard taint propagation techniques use a taint variable to track taintedness of each program variable. To track taint precisely, the transformed program requires a taint variable that has identical data type as the corresponding program variable e.g., if an in-built function returns an array, corresponding taint variable must be initialized with an array of the same size. An approach to summarize each PHP in-built function may help resolve this problem. However, given the large number of such functions it may be an effort-intensive and error-prone exercise. We address this challenge by delaying the generation of taint variables until runtime. At runtime program variables contain concrete values that enables the transformed program to correctly initialize a taint variable such that it mirrors the data type of the corresponding program variable.

4.5.3 Algorithms and Implementation

This section first discusses algorithms used in analyzing traces to generate $f_{server}$ (Section 4.5.3.1). Details of program transformation that enables generation of these traces is discussed next (Section 4.5.3.2).

4.5.3.1 Trace analysis

$f_{server}$ is obtained by taking all conditions that were satisfied in the execution and rewriting them in terms of user inputs. However, all conditions appearing in a trace may not be relevant to each sink appearing in the trace. For the running example, Listing 4.2 shows the generated trace for inputs $\text{card}='1234-5678-9012-3456'$, $\text{name}='alice'$, $\text{address}='wonderland'$, $\text{op}='purchase'$ and $\text{quantity}=1$. The $f_{server}$ computed for this trace is:

\[
\text{AND} \left( = \text{op} \text{"purchase"} \right) \left( \leq \text{len( name ) 30} \right) \left( \text{!isset discount} \right) \left( \text{! NULL address} \right)
\]
In this formula the first three constraints are contributed by the application code and the last one is contributed by the database schema. Note that the \texttt{len} constraint on the \texttt{name} parameter is generated by the solver to capture inherent restrictions imposed by inbuilt functions (Section 4.5.2.1).

**Data- and control-dependency analysis.** The trace generated by a run of the web application may contain several sinks. This analysis identifies statements that are relevant to a sink. To do so it first locates all sinks and performs standard data-dependency analysis for each one of them. For each data-dependent program location, our analysis then locates conditional statements that guard it, if any. All variables appearing in these conditionals are then added as control-dependencies for the sink. We then repeat the data- and control-dependency analysis for the identified variables. The above computation is repeated until a fixed-point is reached. Note that the implicit dependencies may not be captured by the above procedure. To address this, special meta data is added in traces to force inclusion of certain conditionals (refer to \texttt{force-analyze} in Section 4.5.3.2).

Additionally, we use heuristics to select functions to be analyzed in traces. The presence of SQL query execution or file I/O operations typically indicate successful run of the web application. Whereas absence of these and presence of certain functions e.g., \texttt{exit}, \texttt{die} in traces may indicate erroneous runs. We analyze both types of sinks as the analysis of the first enables us to reason about exploitability of important operations in the application and the latter guides input generation to reach the former.

**Database constraint extraction.** In the first step, this analysis parses the schema of an application’s database. For each table creation statement we analyze the column definitions that typically specify constraints on values that can be stored e.g., "\texttt{NOT NULL}" clause enforces non-null
values whereas \texttt{enum} specifies domain of accepted values. We handle MySQL formatted schemas and extract such conditions in the solver language.

In the second step, we generate a symbolic query for SQL sinks found in traces and parse them. This parsing enables us to map table column names to program variables. For example, on parsing a symbolic SQL query "select * from \texttt{T} where \texttt{uid} = \texttt{'$_GET[u]'"}, we can associate column \texttt{uid} of table \texttt{T} to program variable \texttt{$_GET[u]}. Once this mapping is available, we generate constraints by replacing column names with program variables in constraints generated by the first step e.g., if \texttt{uid} column had a NOT NULL constraint, this analysis will yield a constraint (NOT NULL \texttt{u}).

4.5.3.2 Trace generation transformation

The key goal of this transformation is to enable generation of traces that faithfully capture processing of user supplied data in web applications. We first simplify PHP applications using PHC. In specific, we convert applications into HIR intermediate representation that represents PHP programs using a smaller subset of PHP language features. More detailed treatment of key language features is given below.

\textbf{Assignments}. Table VII shows the key rules used in transformation. All text shown in bold font is recorded in traces and \(\tau\) represents application of one or more transformation rules. Rule \(\mathcal{R}_1\) shows the most common step in this transformation that adds two statements after each assignment statement of the original program: one to propagate taint (function \texttt{taint} – discussed later) and another to generate corresponding trace (function \texttt{trace}). Certain features of PHP language such as \texttt{exit}, \texttt{return}, \texttt{break}, \texttt{continue} mandate computation of taint and trace information before the actual statement.
### TABLE VII

WAPTEC: Trace generation transformation

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u = v; )</td>
<td>( u = v; )</td>
<td>( \mathcal{R}_1 )</td>
</tr>
<tr>
<td></td>
<td>( u _t = \text{taint}(u, \text{array}(v _t)); )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{trace}(\text{&quot;PREF}_u = \text{PREF}_v;&quot;); )</td>
<td></td>
</tr>
<tr>
<td>( \text{if}(c){ )</td>
<td>( \text{if}(c){ )</td>
<td>( \mathcal{R}_2 )</td>
</tr>
<tr>
<td>( S_1; )</td>
<td>( \text{trace}(\text{&quot;if}(c){&quot;; ( \tau(S_1); \text{trace}(&quot;; )</td>
<td></td>
</tr>
<tr>
<td>} ) else { )</td>
<td>} else { )</td>
<td></td>
</tr>
<tr>
<td>( S_2; )</td>
<td>} trace(\text{&quot;if}(!c){&quot;; ( \tau(S_2); \text{trace}(&quot;; )</td>
<td></td>
</tr>
<tr>
<td>} )</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{while}(c){ )</td>
<td>( \mathcal{R}_3 )</td>
</tr>
<tr>
<td></td>
<td>( \text{trace}(\text{&quot;if}(c){&quot;; ( \tau(S); \text{trace}(&quot;; )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>} trace(\text{&quot;if}(!c){&quot;; ( \tau(S); \text{force-analyze};&quot;; )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>( u = f(v); )</td>
<td>( u = f(v, v _t); )</td>
<td>( \mathcal{R}_4 )</td>
</tr>
<tr>
<td></td>
<td>( u _t = f _ret _t; )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{trace}(\text{&quot;u = f _ret _v;&quot;}; )</td>
<td></td>
</tr>
<tr>
<td>class ( c ) { }</td>
<td>class ( c ) { }</td>
<td>( \mathcal{R}_5 )</td>
</tr>
<tr>
<td></td>
<td>( \text{var} id; )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{function} c(){ )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{this} \rightarrow id = \text{uniq}(); )</td>
<td></td>
</tr>
</tbody>
</table>
The function \texttt{trace} performs the following two tasks: a) renaming variables with a unique prefix (shown as \texttt{PREF} in Table VII) and b) replacing untainted variables with their concrete values. As discussed before unique prefixes are required to avoid name-collisions in generated traces. To generate unique prefixes, variables appearing in a function are prefixed with it's signature. For variables appearing in classes, a per object unique identifier is used to avoid name-collisions among objects. We do not show prefix in rules other than $R_1$ for ease of presentation.

Further, the most interesting aspect of the generated traces is processing of the tainted data items. We therefore replace all untainted variables with concrete runtime values. This enables us to generate traces that retain computations on tainted values in the symbolic format (thus easing analysis) whereas all computations on untainted data are concretized (thus reducing the amount of analysis to be done).

\textbf{Conditional statements.} Rule $R_2$ shows transformation of an if-else statement and captures evaluated condition with explicit if statements. In specific, for the else branch, negation of the then branch condition is captured. Similarly, for an If statement without the corresponding else clause, if the then branch is not taken, negation of the if condition is reported in the trace.

To faithfully capture conditional execution of statements in traces, the transformer enables printing of scope delimiters (opening and closing curly braces) with if statements. It retains a counter to track number of opened curly braces and uses this information to print right number of closing curly braces in traces. Certain \texttt{PHP} language features may terminate a variable number of lexical scopes (e.g., return would close all open lexical scopes in a function), and the counter is updated accordingly to ensure correct accounting of remaining scope delimiters to be reproduced in traces.
**Loops.** All looping constructs are first translated to `while` loops and rule $R_3$ shows its transformation. The generated trace captures execution of each loop iteration with the help of an `if` statement. On termination of the loop, guard condition is always false and an empty post-loop `if` captures that. To ensure that such empty `if` statements are not ignored during analysis of traces, meta information (`force-analyze`) is added.

**Function calls.** All user defined functions are transformed according to rule $R_4$ which modifies the function signature to pass taint of each argument. Further, two global variables are introduced for each function to retain its return value and the corresponding taint ($f_{ret_v}$ and $f_{ret_t}$ respectively). To inline user defined functions, corresponding trace assigns return value of the function to the left hand side variable. Notice that the generated trace would contain executed function statements immediately before this assignment.

Calls to inbuilt functions are retained in traces. Invocation of sink functions (sql query execution and file I/O) require additional processing to report their status (success / failure) in traces. This requires sink specific computation e.g., for MySQL sinks this amounts to checking the return value of `mysql_errno()` as well as analyzing return value of a query `mysql_query("show warning")`.

**Classes.** The main changes required to classes are for computing unique prefixes for variables and is shown in rule $R_5$. Here transformer adds a member variable $id$ to hold the unique identifier for each instance of the class. The constructor methods are augmented to initialize the $id$ variable to a unique value. Further, inheritance is inherently handled in this scheme as the $id$ member of inheriting class shadows the $id$ member of base class. With the help of $id$ variable, accesses to a member variable through an object ($o\to member_1$) or `this` operator ($this\to member_1$) are
uniformly transformed as $v_{\text{id.} \text{member}_1}$. This enables subsequent analysis to correctly identify accesses to a single memory location from desperate namespaces.

**Taint initialization and propagation.** To initialize taint we avoid recursive initializations

\begin{algorithm}
\caption{Taint\textup{(}u, array\textup{\{}v_1, \ldots, v_n\textup{\})}}
\begin{algorithmic}[1]
\If {$u$ is an array}
    \State $u_t := \text{array}()$;
    \ForAll {$i = 0$ to length\textup{(}u\textup{)} - 1}
        \State $u_t[i] := \text{taint\textup{(}u[i], \text{array}\textup{\{}v_1[i], \ldots, v_n[i]\textup{\})}}$
    \EndFor
\EndIf
\If {$u$ is an object}
    \ForAll {$m \in \{\text{member variables of } u\}$}
        \State $u \rightarrow m_t := \text{taint\textup{(}u \rightarrow m, \text{array}\textup{\{}v_1 \rightarrow m_t, \ldots, v_n \rightarrow m_t\textup{\})}}$
    \EndFor
\EndIf
\If {$u$ is a string or a number}
    \State $u_t := v_1_t \cup \ldots \cup v_n_t$
\EndIf
\end{algorithmic}
\end{algorithm}

by avoiding re-processing of global arrays e.g., \texttt{POST\_taint} is part of superglobal \texttt{GLOBALS} array and is ignored while initializing \texttt{GLOBALS\_taint} array. Algorithm 4 summarizes the taint propagation function \texttt{taint} that takes the \texttt{lval} of an assignment statement as the first argument. This argument dictates type of the taint variable. The second argument contains an array of taint values for variables appearing in the right-hand side of the assignment statement and is used to initialize values in the taint variable. To enable this taint propagation we add helper methods to classes that enable \texttt{taint} method to access private and static member variables. As form processing code typically uses client-supplied data to initialize variables, our transformer supports limited tracking of implicit taint. To that end, we locate variable assignments that are guarded by tainted conditionals. If such variables are used in sink operations, we mark them as tainted.
4.6 Evaluation

This section first presents evaluation of NoTamper (Section 4.6.1) followed by evaluation of WAPTEC (Section 4.6.2).

4.6.1 NoTamper

Test suite and setup. We selected 8 open source applications and 5 live websites. To choose the open source applications, we visited http://opensourcescripts.com and found applications that are heavily reliant on web forms (mainly blogs, business and management applications) and do not use AJAX. To choose the live websites, we selected forms we used personally that seemed likely to contain flaws (e.g., one of the authors has an account at the exploited bank). Table VIII provides some background details for these applications. For open source applications, columns 2 and 3 show the lines of code and number of files, respectively. Column 4 shows the type of constraints enforced by the evaluated forms and the last column shows the functionality provided by the application. We deployed the applications on a Linux Apache web server (2.8GHz Dual Intel Xeon, 6.0GB RAM) and our prototype implementation NoTamper ran under Ubuntu 9.10 on a standard desktop (2.45Ghz Quad Intel, 2.0GB RAM).

4.6.1.1 Experimental Results

4.6.1.2 Summary

Our experimental findings are summarized in Table IX. For each application (column 1), the table includes the number of forms analyzed (column 2), the number of hostile inputs NoTamper generated (column 3), the number of tampering opportunities (column 4), and whether or not we were able to confirm a vulnerability for that application (column 5). The last column lists the number of confirmed false positives.
<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of Code</th>
<th>Files</th>
<th>Source of Client-side Constraints</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ezybiz</td>
<td>186,691</td>
<td>1,103</td>
<td>HTML+JS</td>
<td>Busn Mgt</td>
</tr>
<tr>
<td>Mybloggie</td>
<td>9,431</td>
<td>59</td>
<td>HTML+JS</td>
<td>Blog</td>
</tr>
<tr>
<td>OpenDB</td>
<td>92,712</td>
<td>273</td>
<td>HTML+JS</td>
<td>Inventory</td>
</tr>
<tr>
<td>SMF</td>
<td>97,304</td>
<td>166</td>
<td>HTML+JS</td>
<td>Forum</td>
</tr>
<tr>
<td>OpenIT</td>
<td>114,959</td>
<td>335</td>
<td>HTML+JS</td>
<td>Support</td>
</tr>
<tr>
<td>Legalcase</td>
<td>58,198</td>
<td>195</td>
<td>HTML</td>
<td>Inventory</td>
</tr>
<tr>
<td>PHP-Nuke</td>
<td>228,058</td>
<td>1,745</td>
<td>HTML+JS</td>
<td>Content Mgt</td>
</tr>
<tr>
<td>B2evolution</td>
<td>167,087</td>
<td>531</td>
<td>HTML</td>
<td>Blog</td>
</tr>
<tr>
<td>smi-online.co.uk</td>
<td></td>
<td></td>
<td>HTML</td>
<td>Conference</td>
</tr>
<tr>
<td>wiley.com</td>
<td></td>
<td></td>
<td>HTML+JS</td>
<td>Library</td>
</tr>
<tr>
<td>garena.com</td>
<td></td>
<td></td>
<td>HTML</td>
<td>Gaming</td>
</tr>
<tr>
<td>selfreliance.com</td>
<td></td>
<td></td>
<td>HTML</td>
<td>Banking</td>
</tr>
<tr>
<td>codemicro.com</td>
<td></td>
<td></td>
<td>HTML+JS</td>
<td>Shopping</td>
</tr>
</tbody>
</table>

**TABLE VIII**

NOTAMPER analyzed 8 open source applications and 5 live websites

When deployed by a web developer to analyze a web application, column 4 is of primary interest. A developer need only look through those hostile inputs that were accepted by the server, and for each one manually decide whether or not the server is actually vulnerable. When deployed by testers (blackhat team), they may confirm exploits by further experimenting with the accepted hostile inputs. In a similar spirit, we tried to confirm at least one exploit in each application. The effort involved to examine 50 of the total 169 opportunities was moderate and required an undergraduate student only a week of effort. We anticipate seasoned developers and testers familiar with their applications to take much less time. During this effort, we developed working exploits in 9 out of 13 applications. Below we highlight some of the exploits we discovered.
### TABLE IX

Summary of NOTAMPER results (Opportunities: 169, Examined: 50, Confirmed exploits: 9, False Positives: 43).

<table>
<thead>
<tr>
<th>Application</th>
<th>Forms</th>
<th>Hostile Inputs</th>
<th>Tampering Opportunities</th>
<th>Confirmed Exploit?</th>
<th>Confirmed FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMF</td>
<td>5</td>
<td>56</td>
<td>42</td>
<td>✓</td>
<td>8</td>
</tr>
<tr>
<td>Ezybiz</td>
<td>3</td>
<td>37</td>
<td>35</td>
<td>✓</td>
<td>16</td>
</tr>
<tr>
<td>OpenDB</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>✓</td>
<td>1</td>
</tr>
<tr>
<td>MyBloggie</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>✓</td>
<td>7</td>
</tr>
<tr>
<td>B2evolution</td>
<td>1</td>
<td>25</td>
<td>21</td>
<td>✓</td>
<td>2</td>
</tr>
<tr>
<td>PhpNuke</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>✓</td>
<td>4</td>
</tr>
<tr>
<td>OpenIT</td>
<td>3</td>
<td>28</td>
<td>27</td>
<td>✓</td>
<td>0</td>
</tr>
<tr>
<td>LegalCase</td>
<td>2</td>
<td>13</td>
<td>9</td>
<td>✓</td>
<td>0</td>
</tr>
<tr>
<td>smi-online.co.uk</td>
<td>1</td>
<td>23</td>
<td>4</td>
<td>✓</td>
<td>2</td>
</tr>
<tr>
<td>wiley.com</td>
<td>1</td>
<td>15</td>
<td>4</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>garena.com</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>selfreliance.com</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>✓</td>
<td>0</td>
</tr>
<tr>
<td>codemicro.com</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>✓</td>
<td>0</td>
</tr>
</tbody>
</table>

#### 4.6.1.3 Details of Exploits

**Unauthorized money transfers.** The online banking website [www.selfreliance.com](http://www.selfreliance.com) allows customers to transfer money between their accounts online. A customer logs onto the web site, specifies the amount of money to transfer, uses a drop-down menu to choose the source account for the transfer, and uses another drop-down menu to choose the destination account. Both drop-down menus include all of the user’s account numbers.

It turns out that the server for this application did not validate that the account numbers provided were drawn from the drop-down menus. Thus, sending the server a request to transfer money between two arbitrary accounts succeeded, even if the user logged into the system was an owner of neither account.
When NoTamper analyzed this form, it generated a hostile input where one of the account numbers was a single zero. The server response was virtually the same as the response to the benign inputs (where the account numbers were drawn from the drop-down menus). Therefore, this input was ranked highly by NoTamper as a potential vulnerability. When we attempted to confirm the vulnerability, we were able to transfer $1 between two accounts of unrelated individuals. (Note that if the server had checked for valid account numbers but failed to ensure the user owned the chosen accounts, NoTamper would not have discovered the problem; however, if the human tester provided valid account numbers as hints, NoTamper would have identified the problem.)

We note that this vulnerability could have significant impact given that the bank in question has over 30,000 customers. Further, a successful exploit requires only the knowledge of victim account numbers, which are shared routinely when writing cheques. The bank was contacted about this vulnerability and fixed it in less than 24 hours, during which time the functionality for transferring money was disabled completely. Furthermore, Selfreliance had licensed the software that contained the vulnerability from ESP Solutions (www.espsolution.net), who applied a global patch for all their clients that utilized this functionality and additionally fixed similar problems in their other key product FORZA that provides online banking features.

**Unlimited shopping rebates.** The online shopping website www.codemicro.com sells computer equipment, e.g., hard drives, printers, network switches. The form in question shows the contents of the shopping cart and allows a user to modify the quantities of the selected products. The *quantity* fields employ JavaScript to restrict shoppers to enter only positive numeric values.

When NoTamper analyzed this form, it supplied a negative number for one of the *quantity* fields (and submitted through a proxy). The resulting HTML page, while containing a different
total and quantity than the benign input, was otherwise identical, and thus NOTamPER ranked it as a parameter tampering opportunity.

We were able to further develop this into another serious exploit: we were able to add an item with negative quantities by disabling JavaScript in the browser. When JavaScript was re-enabled, the application computed the total purchase price by multiplying the quantity of each product by its price. Thus, the negative quantities enabled unlimited rebates for any purchase. Furthermore, these negative quantities were successfully accepted by the server, thus permitting the user to purchase at the reduced price.

The potential of exploiting this vulnerability could have been significant as the website contains a very large inventory of computer equipment. The site administrators confirmed the vulnerability and fixed it within 24 hours.

Privilege escalation. The OpenIT application stores user profiles and employs a web form to allow users to edit their profiles. After logging in, the application provides the user with a web form for editing her profile. Included in that form is the hidden field userid, where the application stores the user’s unique identifier. When the form is submitted, the server updates the profile for the user identifier corresponding to userid. By changing userid to that of another user, it is possible to update any user’s profile.

When NOTamPER analyzed this form, it generated a hostile input where the value for userid was the number 2 (as opposed to the initial value 1). The server’s response was virtually identical to the benign input response (where the value was set to 1), and was therefore reported as a tampering opportunity.
After confirming this vulnerability, we enhanced the exploit so as to modify the profile of an admin user to include a Cross-site Scripting (XSS) payload. Every time the admin user logged in, the script would execute and send the admin cookie to a server under our control. With the help of the stolen cookie we then re-constructed and hi-jacked the admin session, thus gaining all the privileges of the admin. This experiment demonstrates that parameter tampering vulnerabilities could be used as a launch pad for other privilege escalation attacks.

**Summary of other exploits.** The NoTamper supplemental website (34) provides details of the above exploits and the others found by NoTamper. In the phpNuke application, tampering of a hidden `name` field allowed us to bypass a CAPTCHA challenge and a confirmation page during the registration process (work-flow attack). In the OpenDB application, an XSS script was injected through a tampered `country` field. In the SMF application, tampering of vote `option` radio button violated integrity of the voting results.

4.6.1.4 Other Experimental Details

**False positives.** All FPs were either (a) pertaining to the `maxlength` constraints on form inputs that couldn’t be exploited to any serious vulnerability or (b) rewritten by the server without any observable difference in HTML output (12 for the Ezybiz application).

**Categorizing potential vulnerabilities.** Table X provides more details of our experiments, categorized by application. Column 2 shows the average formula complexity for the client-side constraints, i.e., the average number of boolean connectives and atomic constraints. Column 3 shows the total number of tampering opportunities. Column 4 shows the number of potential vulnerabilities derived from HTML input controls other than hidden fields; Column 5 shows the
Details of NoTamper results.

number of potential vulnerabilities due to JavaScript; and Column 6 shows the number derived from hidden fields.

**Hostile input ranking.** For each form input NoTamper issued an HTTP request to the appropriate application and computed the difference rank (edit distance in bytes) of the response as described previously. A sorted list of the difference rank is produced for each application. In our experience, it is easy to identify the threshold limits for a potential parameter tampering opportunity, as the difference rank between inputs potentially accepted by the server tend to be at least an order of magnitude smaller than the ones potentially rejected by the server.

We use the graph in Figure 5 to illustrate the thresholds. For ease of presentation, we only chose one form from each application to be represented in this graph, although our approach tested
several forms in every application. Since we are only interested in showing a threshold, the graph plots the logarithm of the difference rank in the Y-axis, with the X-axis representing the various input points sorted according to their difference ranks. We identify the thresholds for various forms using a bold triangle, and we classify those inputs below the threshold as parameter tampering opportunities. It is clear from the graph that such thresholds exist as denoted by steep rises in the difference ranks.

**Manual intervention.** For each web form, we manually provided certain kinds of hints to **NoTamper** pertaining to information not present on the client but that a human tester might provide. For example, in the **SMF** application, the server required a *valid* login name to access the form, and so we provided such a name to **NoTamper**. Throughout all the forms, we added one of three hints: credentials or session cookies, inputs required by the server (required variables list), and variables required to be unique across invocations (unique variables list). (See Section 4.4.2 for more details.)
To discover such restrictions, we used NOTAMPER to generate an input satisfying the client-side constraints ($f_{client}$). If this input was rejected, we examined why and provided hints that ensured NOTAMPER could generate a benign input accepted by the server.

A total of 3 unique-variable hints were added in our experiments (SMF: 2, phpNuke: 1). For every application except phpNuke, we supplied a cookie with a valid session id. Further, a total of 12 required variable hints were supplied in all forms (SMF: 5 in 3 forms, phpNuke: 4, B2evolution: 1, garena.com: 2). This manual intervention is bounded by the number of input fields on a form and typically required less than 5 minutes per form. We expect this process to be simpler for a real tester who is familiar with the application being tested.

**Performance.** The most computationally expensive component of NOTAMPER was the Input Generator. The HTML / JavaScript Analyzer ran in under a second for the most elaborate form in our test suite. The Opportunity Detector ran in sub-second time for each application, ignoring the delays between consecutive HTTP requests built-in to avoid overloading the server. The most expensive step of Input Generation was constraint solving; the remainder of the Input Generation component ran in under a second. Over the 22 forms, the constraint solver solved 315 formulas in a total of 219 seconds, giving an average time of 0.7 seconds per input. Such performance is acceptable for an off-line analysis tool such as NOTAMPER.

### 4.6.2 WAPTEC

**Testsuite and setup.** We evaluated effectiveness of WAPTEC on a test suite of applications. Our test suite comprised of 6 open source PHP applications which were chosen to reflect prevalent applications that are used in commonplace settings. Table XI provides background information on these applications (lines of code, number of files, and function). The test suite was deployed
TABLE XI

Summary of WAPTEC results

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of Code</th>
<th>Files</th>
<th>Use</th>
<th>Exploits Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnipeGallery</td>
<td>9.1k</td>
<td>54</td>
<td>Image Management</td>
<td>2</td>
</tr>
<tr>
<td>SPHPBlog</td>
<td>26.5k</td>
<td>113</td>
<td>Blog</td>
<td>1</td>
</tr>
<tr>
<td>DepPortal</td>
<td>144.7k</td>
<td>484</td>
<td>Content Management</td>
<td>34</td>
</tr>
<tr>
<td>PHPNews</td>
<td>6.4k</td>
<td>21</td>
<td>News Management</td>
<td>1</td>
</tr>
<tr>
<td>Landshop</td>
<td>15.4k</td>
<td>158</td>
<td>Real Estate</td>
<td>3</td>
</tr>
<tr>
<td>MyBloggie</td>
<td>9.4k</td>
<td>59</td>
<td>Blog</td>
<td>6</td>
</tr>
</tbody>
</table>

on a Mac Mini (1.83 GHz Intel, 2.0 GB RAM) running the MAMP application suite, and we ran WAPTEC on a Ubuntu workstation (2.45Ghz Quad Intel, 2.0GB RAM).

Experiments. We evaluated our approach by conducting two sets of experiments. In the first set of experiments, we ran WAPTEC to automatically analyze chosen forms, with the aim to detect mismatches between client and server specifications and produce exploits by construction. The second set of experiments used NoTamper (Section 4.4) to quantify benefits of using whitebox analysis over blackbox analysis in this problem setup.

Results summary. The outcome of the first set of experiments is summarized in Table XI. We evaluated one form in each application. WAPTEC found a total of 47 exploits. We manually verified all of these exploits. For each application (column 1), the last column shows reported exploits. We discuss a few interesting exploits below.

4.6.2.1 Exploits

Here we describe one exploit WAPTEC found for each of the six applications we tested. The first three are given in detail, while the other three are briefly described.
**Creating arbitrary files.** The *phpnews* application allows administrators to make changes to template files via a web interface. The administrator is allowed to choose from available template files, and a page is returned with a form that displays the contents of the file. The form contains a hidden field, which specifies the name of the template file. An attacker is able to modify the hidden field to an arbitrary filename and consequently create new files or overwrite existing files on the server with malicious content.

When WAPTEC analyzed this form, it discovered a mismatch between client-side and server-side specifications. In particular, the hidden template file value is not verified for consistency on the server-side. During exploit construction, WAPTEC generated a random value for the template file parameter, which reached the success sink operation. We verified this exploit upon seeing the creation of a new file.

We analyzed the application’s server-side form processing code and discovered that the server invokes PHP I/O functions directly on the template file parameter. As a fix, the server-side should first verify that the submitted value is a valid template file. Exploring this vulnerability further, we could overwrite files of other applications on hosted the same server by developing a directory traversal attack. As demonstrated, such vulnerabilities could allow rogue users to corrupt code / data files of applications deployed on the same server.

**Denial of service.** The setup form of the *sphpblog* application allows users to configure its look and feel. Changes made through this form are written to a file `config.txt` which is consulted before loading each page of the blog. This form provides a drop-down menu showing supported languages by the blog. However, the server-side code doesn’t verify if the supplied language value is supported and applies the change. An invalid language makes the application unusable and all subsequent page
requests for all users result in blank response pages instead of requested pages. Such vulnerabilities can enable “one-request” denial of service attacks through rogue users. While the broader class of denial-of-service attacks is difficult to address in general, in this specific case it could have been easily prevented by a simple check that was missed on the server side. Furthermore, this exploit also demonstrates the ease with which such DoS attacks could be conducted by parameter tampering.

WAPTEC first generated a request consisting of a supported language. As the server-side code does not check the language parameter value, the $f_{server}$ did not contain any constraint on the language value. Our tool then generated a request with no value for the language parameter and verified it was written to the file config.txt. All subsequent attempts to access the application failed as the application could not find the language file to use. The obvious fix for this problem is to re-enforce checks on language parameter and only accept supported languages.

**Privilege escalation.** The dcpportal application allows guests to register for an account. The registration form solicits standard information, such as name, e-mail, username, password, etc. Upon normal registration, a user is provided with an account having basic privileges. When the form is submitted, the server-side form processing code validates the provided information and checks if a cookie $make\_install\_prn$ is set. When this cookie is set to 1, the user is registered with administrative privileges. By setting this cookie, it is possible for an attacker to register an account with escalated privileges (similar hidden parameters have been used for implementing backdoors in applications e.g., CVE-2007-1277).

When analyzing this form, WAPTEC demonstrated that it can uncover negative tampering vulnerabilities. While exploring additional server-side form processing code, it found a conditional that depended on $make\_install\_prn$ which is not found in the client-side formula. To explore this
branch, WAPTEC satisfied the conditional by setting the cookie \textit{make\_install\_prn} to 1. By analyzing data and control dependencies, it then determined that this branch modifies parameter values used in the sink, and therefore, reported the exploit.

After confirming the exploit, we analyzed the application to understand the root cause of this flaw. We found that the application used cookie \textit{make\_install\_prn} during initial installation to allow creation of an administrator account. To patch this vulnerability, the application can use additional server-side state (e.g., sessions) to avoid depending on the cookie value alone or have a separate form for this purpose.

\textbf{Additional exploits.} Below we briefly describe one exploit from each of the other three applications we evaluated.

- Snipegallery, a photo album application, allows users to arrange albums hierarchically by selecting a parent category for each new album from a drop down list. By selecting a value not in that list, the new album becomes invisible; furthermore, additional analysis shows that a carefully constructed parent album value leads to a SQL injection attack.

- Landshop, a real estate application, includes a form with a hidden field not pertinent to that form. When the value of this field is set to the ID of an existing listing (which are displayed prominently on the site), that listing is deleted from the application whether the user is the owner or not.

- Mybloggie, a blogging application, allows users to choose the category for each post using a drop-down list. By submitting a value not in that list, a rogue user can hijack a category that will be created in the future.
### 4.6.2.2 Comparison of WAPTEC and NoTamper results

The results of the comparison are summarized in Table XII. For each application, this table reports the number of exploits found by NoTamper (column 2) and WAPTEC (column 3). The next two columns report false positives reported by NoTamper, which were eliminated in WAPTEC, and exploits that NoTamper could not find. In total, the blackbox approach resulted in 23 false positives, 9 false negatives, and 24 fewer confirmed exploits when compared to the whitebox approach. Further, for DcpPortal and Mybloggie applications WAPTEC found several exploitable sinks for each negated disjunct of $f_{client}$ e.g., for DcpPortal column 3 shows 15(30) - hostile inputs generated by negating 15 $f_{client}$ disjuncts were used in 30 sinks and hence were exploitable. We wish to note that all these disjuncts would have contributed to one hostile each, at best, in NoTamper.

NoTamper suffers from false positives and false negatives mainly because it guesses if a given input was accepted or rejected by the server based on heuristics. Similarly, any required variables

<table>
<thead>
<tr>
<th>Application</th>
<th>Generated Exploits</th>
<th>False Positives</th>
<th>False Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BlackBox</td>
<td>WhiteBox</td>
<td>BlackBox</td>
</tr>
<tr>
<td>SnipeGallery</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>PHPBlog</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DcpPortal</td>
<td>13</td>
<td>15(30)</td>
<td>9</td>
</tr>
<tr>
<td>PHPNews</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Landshop</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Mybloggie</td>
<td>1</td>
<td>5(6)</td>
<td>12</td>
</tr>
</tbody>
</table>

**TABLE XII**

Comparing WAPTEC and NoTamper results
in $f_{server}$ can be easily identified in a whitebox approach through code analysis, but have to be heuristically determined in a blackbox approach.

**NoTamper** reported some false positives in the mybloggie application, because of missed required variables. In one example of an unsuccessful attack, the server’s demand for either the value of submit or preview was not met, and a response page containing the same form was returned. Finally, **NoTamper** is inadequate to discover exploits like the dcpportal privilege escalation, because that requires analysis of server-side form processing logic to uncover hidden functionality, which is out of scope for a blackbox tool.

Although WAPTEC results are consistently better than NoTamper, both of these approaches have their own utility. As NoTamper does not rely on analyzing server-side code, it could be employed to analyze a wider range of applications and websites. However if the source code is available, a whitebox analysis based approach like WAPTEC could be employed to perform deeper code analysis to pinpoint more security problems. Further, by ensuring production of exploits by construction, the whitebox approach can reduce the human effort in confirming exploits that may be unavoidable in blackbox approaches.

### 4.6.2.3 Additional Experimental Data

For each evaluated application, Table XIII captures complexity of generated formulas (column 2 - client-side constraints, column 3 - server-side code constraints, column 4 - database constraints), average size of generated traces (column 5 - kilo bytes) and average time taken to run the tool (column 6 - seconds).

**Outliers.** The most notable application we tested, dcpportal, included the largest formula complexities, the largest number of exploits, and the longest running time. The larger the formula
<table>
<thead>
<tr>
<th>Application</th>
<th>Formula Complexity</th>
<th>Avg. Trace Size (KB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnipeGallery</td>
<td>11</td>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>SPHPBlog</td>
<td>37</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>DcpPortal</td>
<td>187</td>
<td>2</td>
<td>135</td>
</tr>
<tr>
<td>PHPNews</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Landshop</td>
<td>20</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>MyBloggie</td>
<td>37</td>
<td>4</td>
<td>738</td>
</tr>
</tbody>
</table>

TABLE XIII

Additional WAPTEC results

complexity, the larger and more complex the form; hence, a longer running time is to be expected. The large number of exploits is partially attributed to large formula complexity because the potential number of exploit generation attempts is larger; however, the presence of a large number of confirmed exploits points to poor server validation of inputs.

Whitebox database constraints. To assess usefulness of additionally analyzing database constraints, we (i) added the database constraints into $f_{server}$ and (ii) checked for errors/warnings on database operations. Without the former, there is the potential for false negatives, and without the latter there is the potential for false positives. For instance, during testing of the registration form for DcpPortal, database constraints helped to avoid a false positive. In this example, the hostile input was produced by negating a range constraint on the birthdate parameter in $f_{client}$, and $f_{server}$ did not contain the replicated constraint. The server’s response returned a success page, so NoTamper recognized a vulnerability. However, the default action by the database converted the invalid date to '0000-00-00’. Another example was found while testing the snipegallery application. The hostile input was produced by negating a length constraint found in $f_{client}$, and $f_{server}$ did not
contain the replicated length constraint. However, database implicitly enforced the length check and this attack did not succeed. Without considering database constraints, such false positives cannot be avoided.

**Manual intervention.** In a preliminary analysis of the chosen applications, we selected forms that contained interesting client side specifications and collected login credentials necessary to access them (in 5 applications). We also extracted form action parameters in cases where applications reused processing code between multiple forms (total of 4). These hints were necessary to facilitate automatic analysis and to restrict exploration of server-side code pertaining to other forms. Overall, it required typically less than 5 minutes to collect this data for each form.

Detailed reports of exploits discovered by WAPTEC and demo of PHP transformation are available at WAPTEC supplemental website (33).

4.7 **Summary**

This chapter presented NoTamper, a blackbox tool that identifies parameter tampering opportunities and WAPTEC, a whitebox tool that generates parameter tampering exploits by construction.

We empirically demonstrated that parameter tampering vulnerabilities are widespread in the existing web applications and these vulnerabilities could be exploited through severe attacks. A total of 17 web applications, that included open source and commercial web applications, were evaluated and a total of 52 parameter tampering vulnerabilities were confirmed. Further, starting with these vulnerabilities we were able to construct exploits that demonstrated serious security problems: unauthorized monetary transactions at a bank, unauthorized discounts added to a shopping cart, and so on. These experiments strongly suggest that parameter tampering vulnerabilities
are widespread in open source as well as commercial web applications and can be exploited through severe attacks.

NoTamper and WAPTEC provided an interesting mix of capabilities. NoTamper, being a blackbox tool did not require access to the server-side code. This allowed us to evaluate commercial web applications that we could not have evaluated otherwise. By evaluating these applications we were able to successfully find serious exploits in commercial web applications. WAPTEC allowed us to incorporate the server-side processing and generate exploits by construction. Using this tool we were not only able to reduce the manual effort required to confirm these vulnerabilities but also find exploits that could not have been found with NoTamper. In summary, NoTamper and WAPTEC provide complementary capabilities to find parameter tampering vulnerabilities in web applications.

We formulated the parameter tampering problem in terms of constraints implied on user data by the client-side code, advocated program analysis as a way of extracting those constraints. We then employed constraint solving to generate tampering opportunities as well as exploits by construction. Both of these tools illustrated that it is indeed possible to extract and use specifications of an application’s intended behavior from its own (client-side) code.

In summary, this chapter strongly highlights a significant gap between the server-side parameter validation that should occur and the server-side validation that does occur in today’s web applications. Further, it provides evidence that it is possible to extract specifications of intended behavior to automatically find these gaps.
This chapter confirms that parameter tampering vulnerabilities are widespread and pose serious threat to web applications. The next chapter presents our approach to prevent SQL injection vulnerabilities in legacy web applications.
Listing 4.2. WAPTEC: Trace generated for running example

1 // Generated trace for running example with following
2 // parameters: card = "1234−5678−9012−3456", name = "alice",
3 // address="wonderland", op = "purchase",
4 // quantity = 1
5 // line numbers shown next to trace statements
6 // refer to running example code Listing 2.2.

7 $main_ca = $POST['card']; // (line 1)
8 if ($main_ca matches '1234−5678−9012−3456' ||
9       $main_ca matches '7890−1234−5678−9012'){}
10      // (line 2)
11
12 $main_n = substr($POST['name'], 30); // (line 8)
13 if ($GET['op'] == "purchase"){
14      // (line 10)
15         // $price = 100, $shipping = 10 :
16         // concretized in the computation below
17         $main_cost = $POST['quantity1'] * 100 + 10; // (line 12)
18
19         // since 'discount' is not set,
20         // trace contains negated condition
21         if (!isset($POST['discount']){ force−analyze(); } // (line 15)
22
23         $main_q = "INSERT INTO orders ( 'na', 'co', 'ca', 'ad',
24              'de' );"; // (line 18)
25         $main_q = "INSERT INTO orders ( 'na', 'co', 'ca', 'ad',
26              'de' ) . "VALUES('" . $main_n . "', $main_cost,
27              $main_ca, '" . $POST['address'] . '"', '"
28              . $POST['delivery'] . '"');";
29         mysql_query ($main_q); // (line 22)
30
31         // query execution denoted by SUCCESS status
32         $_wb_status = "SUCCESS";
33
34      } // untained statements not reported e.g.,
35      // HTML generation (lines 3−4, 19−20)
CHAPTER 5

PREVENTION OF SQL INJECTION VULNERABILITIES

This chapter presents our work TAPS that prevents SQL injection vulnerabilities in legacy web applications. This chapter is organized as follows. Section 5.1 presents motivation behind the key design decisions made in TAPS that mainly aim to realize the high level approach outlined in Chapter 3. Section 5.2 provides a summary of advances made by this work. Section 5.3 discusses a modified running example for ease of presentation. Section 5.4 provides a high level overview of TAPS based program transformation followed by a discussion of detailed algorithms in Section 5.5. Section 5.6 narrates our experiences in evaluating TAPS on several open source web applications. Section 5.7 summarizes this chapter.

5.1 Motivation

As discussed in Chapter 3, SQL injection attacks manifest by violating intentions of web applications. A web application often computes SQL queries by combining constant strings with user inputs. While computing a query, the web application’s source code specifies a data context for each user input. By specifying these contexts, the web application encodes the intended use of user inputs. Those inputs that break out of the application intended data contexts, violate application’s intentions and result in successful SQL injection attacks.

We want to realize the high level approach outlined in Chapter 3 to prevent SQL injection vulnerabilities. To do so, we need to perform the following two tasks: a) extract the application intended data context for each user input and b) confine user inputs to their intended data contexts.
As the application source code encodes intended data contexts for user inputs, we want to analyze the source code to extract intentions. We first note that a typical query embeds user inputs (or a function of user inputs) in data arguments. Therefore, to extract an input’s intended data context we first need to identify those data arguments of a query that use the input. Once a data argument is identified, data contexts for inputs used in it could be extracted by analyzing characters enclosing the data argument. This reasoning must be done for each query that a program can compute. The intention generator component (Figure 2), therefore aims to extract intentions by analyzing web application code that computes queries.

To ensure that malicious inputs do not break out of intended data contexts, the intention utilizer component (Figure 2) requires a robust mechanism to confine user inputs to their intended data contexts. Interestingly \texttt{PREPARE} statements, a facility provided by many database platforms to construct SQL queries, provide such a mechanism. \texttt{PREPARE} statements are objects that contain precompiled SQL query structures (without data). This allows a programmer to easily isolate and confine the “data” portions of the SQL query from its “code”. In other words, \texttt{PREPARE} statements enable applications to enforce data to remain in desired data contexts. In the first step, \texttt{PREPARE} statements require a programmer to specify “structure” of the SQL query without any data. The DBMS fixes this structure and then enforces user supplied data to remain confined to data contexts in the query structure. Thus by making use of \texttt{PREPARE} statements, the intention utilizer can robustly confine user inputs to their intended data contexts.

Unfortunately, the existing practice to transform a legacy web application to make use of \texttt{PREPARE} statements requires extensive manual effort. The programmer needs to obtain a detailed understanding of the program that includes identification of all inter-procedural control and data flows.
that generate vulnerable SQL queries. Furthermore, these flows have to be analyzed to obtain the equivalent code for `PREPARE` statement generation. Each such control flow needs to be carefully transformed while ensuring that the changes do not alter semantics of the program in any undesirable fashion. Additional manual verification may be needed to ensure that the semantics of the transformed program on non-attack inputs, is the same as the original program. This process could be tedious, often error-prone, and certainly expensive for large-scale web applications.

We want to develop a sound method to automate the above transformation to `PREPARE` statements. This will overcome the deficiencies of manual approach and provide us a robust way to confine each user input to its intended data context. However, designing a sound method is extremely challenging because a completely automated method needs to replicate the human understanding of the program logic that constructs SQL queries (refer to challenges outlined in Section 3.3).

5.2 Contributions

This main contribution of TAPS is to address the above mentioned challenges by developing the first automated sound program transformation approach that retrofits a legacy web application to make use of `PREPARE` statements. We develop a new method that constructs a high-level understanding of a program’s logic directly from its low-level string operations. This method relies on a novel insight that a program’s low-level string operations along any particular control path can be viewed as a derivation of a symbolic SQL query that is parametrized by its inputs. Our method directly uses this derivation to identify and isolate any unsafe string operations that may otherwise result in injection attacks. The isolated operations are then rewritten using `PREPARE` statements, effectively eliminating the SQL injection attack vector from the web application.
Listing 5.1. SQL injection vulnerable path in server.php

```php
if ($_GET['op'] == "show"){
    $q1 = "select * from ORDERS where name like '%";
    $q2 = f($_POST['name']);
    $q3 = "% order by co";
    $q = $q1 . $q2 . $q3;
    $results = mysql_query($q);
    // construct HTML to show existing orders that
    // match given user name
}
```

TAPS is the first reported sound tool in the literature to perform this transformation. It has been successfully applied to several real world applications, including one with over 22,000 lines of code. In addition, some of these applications were vulnerable to widely publicized SQL injection attacks present in the CVE database, and our transformation renders them safe by construction.

As a concluding remark to this section, we note that there is a rich body of literature on SQL injection detection and prevention (see the related work in chapter 7). Our objective is to not propose “one more defense” to this problem. Instead, our contribution is quite the opposite: to develop an automatic method that will assist developers and system administrators to automatically retrofit their programs with the “textbook defense” for SQL injection.

5.3 Running Example

For the ease of presentation, we introduce a new control flow in the existing running example. As shown in Listing 5.1, this modified shopping cart application provides a user the ability to view her order history. If the op parameter is set to “show”, the application finds all records with
matching names by issuing a **SELECT** query. The application applies a (filter) function \( f \) on the input \( \$_POST[ \text{\'name\'}] \) and then combines it with constant strings to generate a query \( (\$q) \). This query is then executed by a **SQL sink** (query execution statement) at line 8.

This flow is vulnerable to SQL injection if the **name** parameter can be injected with malicious content and the filter function \( f \) fails to eliminate it. For example, the user input \( \text{\' OR 1=1 --} \) provided as **name** in the above example can break out of the intended string literal context and add an additional **OR** clause to the query.

Typically, user inputs such as \( \$_POST[\text{name}] \) are expected to contribute as literals in the parse structure of any query, specifically, in one of the two literal **data contexts**: (a) a string literal context which is enclosed by program supplied string delimiters (single quotes) (b) in a numeric literal context. As discussed before, SQL injection attacks violate this expectation by introducing input strings that do not remain confined to these literal data contexts and directly influence the structure of the generated queries.

**PREPARE** statement confines all query arguments to the expected data contexts. These statements allow an application to declare (and finalize) the structure of every SQL query in the application. Once constructed, the parse structure of a **PREPARE** statement is frozen and cannot be altered by malformed inputs. The following is an equivalent **PREPARE** statement based program for the running example.

1. \( \$_q = \text{\"select * from ORDERS where na LIKE ? order by co\"}; \)
2. \( \$_stmt = \text{prepare}($\$_q).bindParam(0, \text{\"s\"}, \text{\"\%\"}.f($\$_POST[\text{\'name\'}]).\text{\"\%\"}); \)
3. \( \$_stmt.execute(); \)
The question mark in the query string $q$ is a “place-holder” for the query argument $\%f(\$._-
POST[\'name\'])\%$. In the above example, providing the malicious input $\text{name} = \text{or 1=1 --}$ to the
prepared query will not result in a successful attack. This is because the actual query is parsed with
these placeholders (prepare instruction generates PREPARE statement), and the actual binding of data
to placeholders happens after the query structure is finalized (bindParam instruction). Therefore,
the malicious content from $\$._POST[\'name\']$ cannot influence the structure of query.

5.4 Approach Overview

This section provides an overview of our approach which aims to replace all queries generated
by a web application with equivalent PREPARE statements. Section 5.4.1 discusses the key challenges
faced in doing the above transformation. Section 5.4.2 presents an assumption that our approach
makes about programs to be transformed. Section 5.4.3 presents the key insights that help us in de-
signing an approach to introduce PREPARE statements and then presents a systematic transformation
of the running example.

5.4.1 The Transformation Problem

Given a large web application, making a change to PREPARE statements, is challenging. Any
efforts on the above lines, require addressing challenges outlined in the background chapter (Sec-
tion 3.3). We briefly mention the major challenges below.

The above change is tedious to achieve through manual transformation. To make the change, a
developer must consider each SQL query execution location (sink) of the program and queries that
it may execute. Depending on the control path a program may generate and execute multiple SQL
queries at a sink. Looping behavior may be used to introduce a variety of repeated operations,
such as construction of conditional clauses that involve user inputs. Sinks that can execute multiple
queries need to be transformed such that each control path gets its corresponding \texttt{PREPARE} statement. This requires a developer to consider all control flows together. Also, each such control flow may span multiple procedures and modules and thus requires an analysis spanning several procedures across the source code.

A second issue in making this change is: for each control flow, a developer must extract query arguments from the original program statements. This requires reasoning about the data contexts. In the running example (Listing 5.1), the query argument \%f($\_\text{POST}[\text{'name'}])\% is generated at line 5, and three statements provide its value: \(f(\$\_\text{POST}[\text{'name'}])\) from line 4, and enclosing character (\%) from line 3 and 5, respectively. The above mentioned issues make the problem of isolating user input data from the original program query quite challenging.

\textbf{5.4.2 Main Assumption}

We define a \textit{partial query string variable} as a variable that holds a query fragment consisting of some string constants supplied by the program code together with user inputs e.g., in the running example, variable \$q is a partial query string variable. Our approach makes the following assumption about partial query strings.

\textit{We require the web application to be transformed, to not perform content processing or inspection of partial query string variables.}

To guarantee the correctness of our approach, we require this assumption to hold. To explain this assumption for the running example, we require that once the query string \$q is formed in line 6 of the application by concatenating filtered user input \(f(\$\_\text{POST}[\text{'name'}])\) with program generated constant strings in variables \$q1 and \$q3, it does not undergo deep string processing (i.e., splitting, character level access, etc.,) further en route to the sink.
To ensure that this assumption holds, our approach and implementation checks that the program code only performs the following operations on partial query string variables: (a) append with other program generated constant strings or program variables, (b) perform output operations (such as writing to a log file) that are independent of query construction and (c) equality comparison with string constant null. Checking the above three conditions is sufficient to guarantee that our main assumption holds.

The above conditions are in fact conservative and can be relaxed by the developer, but based on our experimental evaluation of many real world open source applications we believe that the above assumption is not very limiting. In fact, the above assumption has been implicitly held by many prior approaches for SQL injection defense. Defenses such as SQLRand (60), SQLCheck (42) are indeed applicable to real world programs because this assumption holds for their target applications. We note that all of these approaches change the original program’s data values. SQLRand randomizes the program generated keywords, SQLCheck encloses the original program’s inputs with marker tags. These approaches then require that programs do not manipulate their partial query strings in arbitrary ways. For instance, if a program splits and acts on a partial query string after its SQL keywords has been randomized, it introduces the possibility of losing the effect of randomization.

A small minority of query generation statements in some programs may not conform to our main criteria; in this case, our tool reports a warning and requires programmer involvement as discussed in Section 5.6.
5.4.3 Intuitions Behind Our Approach

As mentioned earlier, user inputs are expected to contribute to SQL queries in string and numeric data literal contexts. Our approach aims to isolate these (possibly unsafe) inputs from the query by replacing existing query locations in the source code with \texttt{PREPARE} statements, and replacing the unsafe inputs in them with safe placeholder strings. These placeholders will be bound to the unsafe inputs during program execution (at runtime).

In order to do this, we first observe that the original program’s instructions already contain the programmatic logic (in terms of string operations) to build the structure of its SQL queries. This leads to the crucial idea behind our approach: \textit{if we can precisely identify the program data variable that contributes a specific argument to a query, then replacing this variable with a safe placeholder string (?) will enable the application to programmatically confine user supplied data to their desired data contexts through \texttt{PREPARE} statements.} The above approach will work correctly if our main assumption is satisfied. We indeed can ensure that the resulting string with placeholders at the original SQL sink will have (at runtime) the body of a corresponding \texttt{PREPARE} statement.

The problem therefore reduces to precisely identifying query arguments that are computed through program instructions. In our approach, we solve this problem through symbolic execution (43), a well-known technique in program verification. Intuitively, during any run, the SQL query generated by a program can be represented as a symbolic expression over a set of program inputs (and functions over those inputs) and program-generated string constants. For instance, by symbolically executing our running example program, we obtain the following symbolic query expression:

\begin{verbatim}
SELECT * FROM ORDERS WHERE na LIKE '%f($\_POST[name])%' ORDER BY co
\end{verbatim}
Notice that the query is expressed completely by constant strings generated by the program, and (functions over) user inputs. (We will define these symbolic expressions formally later.)

Once we obtain the symbolic expression, we analyze its parse structure to identify data arguments for the `PREPARE` statement. In our running example, the only argument obtained from user input is the string `%f($_POST['name'])%`.

Our final step is to traverse the program backwards to the program statements that generate these arguments, and modify them to generate placeholder (?) instead. Now, we have changed a data variable of a program, such that the program can compute the body of the `PREPARE` statement at runtime.

In our running example, after replacing contributions of program statements that generated the query data argument `%f($_POST['name'])%` with a placeholder (?), $q$ at line 6 contains the following `PREPARE` statement body at runtime:

```
SELECT * FROM ORDERS WHERE na LIKE ? ORDER BY co, %$q2%
```

The corresponding query argument is the value `%$q2%`. Note that the query argument includes contributions from program constants (such as `%`) as well as user input (through $q2$).

**Approach overview.** Figure 6 gives an overview of our approach for the running example. For each path in the web application that leads to a query, we generate a derivation tree that represents the structure of the symbolic expression for that query. For our example, $q$ is the variable that holds the query, and step 1 of this figure shows the derivation tree rooted at $q$ that captures the query structure. The structure of this tree is analyzed to identify the contributions of user inputs and program constants to data arguments of the query, as shown in steps 2 and 3. In particular, we want to identify the subtree of this derivation tree that confines the string
Figure 6. TAPS: step (1) generates symbolic queries, steps (2-3) separate data reaching the queries, step (4) removes data from symbolic queries, and step (5) generates the transformed program. ($n$ in this figure represents $_POST['name']$).

and numeric literals, which we call the data subtree. In step 4, we transform this derivation tree to introduce the placeholder value, and isolate the data arguments. This change corresponds to a change in the original program instructions and data values. In the final step 5, the rewritten program is regenerated. The transformed program programmatically computes the body of the PREPARE statement in variable $q$ and the associated argument in variable $t$.

5.5 Algorithms and Implementation

This section provides a detailed treatment of algorithms used in introducing PREPARE statements. Section 5.5.1 first discusses handling of straight line programs with the help of a simple programming language. Using straight line programs as a building block the next two sections discuss handling
of conditionals, procedures and loops (Section 5.5.2 and Section 5.5.3). Section 5.5.4 presents implementation details of these algorithms.

5.5.1 Handling Straight Line Programs

We give a more precise description using a simple well defined programming language. We assume that all the variables in the language are string variables. Let \( \cdot \) denote string concatenation operator. The allowed statements in the language are of the following forms:

\[
x = f(), \quad x = y, \quad x = y_1 \cdot y_2
\]

where \( x \) is a variable and \( y \) is a variable or a constant, \( y_1, y_2 \) are variables or constants with the constraint that at most one of them is a constant, and \( f() \) is any function including a special function \( \text{input} \) that accepts inputs from the user. Here we describe our approach for straight line programs. Processing of more complex programs, that include conditional statements and certain type of simple loops, is presented later in this section. The approach for such complex programs uses the procedure for straight line programs as a building block.

**Derivation trees.** Now consider a straight line program \( P \) involving the above type of statements. Assume that \( P \) has \( l \) number of statements. We let \( S_i \) denote the \( i^{th} \) statement in \( P \).

With each \( i, 1 \leq i \leq l \), we define a labeled binary tree \( T_i \) as follows. Let \( x = e \) be the statement \( S_i \). Intuitively, \( T_i \) shows the derivation tree for the symbolic value of \( x \) immediately after execution.
of $S_i$. The root node $r_i$ of $T_i$ is labeled with the pair $(i, x)$ and its left and right children $(T_l, T_r)$ are defined as follows.

$$(T_l, T_r) = \begin{cases} ( (\text{label} = x), - ) & \text{if } e = f() \\ ( (\text{label} = c), - ) & \text{if } e = c \\ (T_j, -) & \text{if } e = y \\ (T_j, T_k) & \text{if } e = y \cdot z \end{cases}$$

Here $c$ is a constant, $T_j$ and $T_k$ are the derivation trees of last statements $j$ and $k$ before $i$ that update $y$ and $z$, respectively.

The derivation tree $T_i$ has two sub-trees only when $e$ is $y \cdot z$. Note that if $y$ (or $z$) is a constant then the left (or right) sub-tree is a leaf node labeled with the constant, otherwise it is a copy of some $T$ as defined above. Figure 7 gives a program and the tree $T_6$ for this program.

**Symbolic strings.** For the program $P$, we construct the trees $T_i$, for $1 \leq i \leq l$. For each tree $T_i$, we define a symbolic string, called the string generated by $T_i$, as the string obtained by concatenating the labels of leaves of $T_i$ from left to right. If $S_i$ is of the form $x = e$, then we define the symbolic value of $x$ after $S_i$ to be the symbolic string generated by $T_i$. For the program given in Figure 7, the symbolic value of $q$ after statement 6 is the string

```
select * from employee where salary = x1 + x2
```

**Data sub-strings.** Assume that the last statement of $P$ is $sql.execute(q)$ and that this is the only SQL statement in $P$. Also assume that statement $i$ is the last statement that updated $q$. We obtain the symbolic value $s$ of $q$ after statement $i$ from the tree $T_i$ and parse it using the SQL parser. If it is not successfully parsed then we reject the program. Otherwise, we do as follows. From the parse tree for $s$, we identify the sub-strings of $s$ that correspond to data portions. We call these sub-strings as data sub-strings. For each data sub-string $u$, we identify the smallest sub-tree
Figure 7. TAPS: Labeled derivation tree for symbolic values of $q$ after execution of statement 6.

$\tau_u$, called data sub-tree, of $T_i$ that generated $u$. Note that $\tau_u$ is a copy of $T_j$ for some $j \leq i$. Clearly, $u$ is a sub-string of the string generated by $\tau_u$. Now, we consider the case when the following property (*) is satisfied. (If (*) is not satisfied we transform $P$ into an equivalent program $P'$ that satisfies (*) and we invoke the following procedure on $P'$; this transformation is described later).

**Property (•):** For each data sub-string $u$, $u$ is equal to the string generated by $\tau_u$.

**Program transformation.** We modify the program so that data sub-strings in symbolic strings are replaced by “?” (Rule1) and all such data sub-strings are gathered into argument lists (Rule1 and Rule2). We achieve this as follows. For each relevant variable $x$, we introduce a new variable $args(x)$ that contains its list of arguments and initialize it to the empty lists in the beginning.

Let the root node of $T_i$ be $r_i$ and the root node of sub-tree $\tau_u$ in $T_i$ be $r_u$. We traverse the tree $T_i$ from node $r_u$ to its root and let $t_1, \ldots, t_k$ be the nodes on this path in that order. Note that
Let \( t_1 = r_u \) and \( t_k = r_i \). For each \( j, 1 \leq j \leq k \), let the label of node \( t_j \) be given by \( \langle j, \text{var}_j \rangle \) where \( \text{var}_j \) represents the variable being updated at the node \( t_j \) (note that \( t_j \) cannot be a leaf node).

**Rule1: Eliminating data subtrees.** Let \( j' \) be the smallest integer such that \( 1 < j' \leq k \) and \( t_{j'} \) has two children. Clearly, the statement \( S_{j'} \) is of the form \( \text{var}_{j'} = y' \cdot z' \). If \( \text{var}_{j'-1} = y' \) i.e., \( \tau_u \) appears in the left subtree of \( t_{j'} \). We replace \( S_{j'} : \text{var}_{j'} = y' \cdot z' \) by the following two statements.

\[
\begin{align*}
\text{args}(\text{var}_{j'}) &= \begin{cases}
[y'] & \text{if } z' \text{ is a constant} \\
[y'] \# \text{args}(z') & \text{if } z' \text{ is a variable}
\end{cases} \\
\text{var}_{j'} &= {"?"} \cdot z'
\end{align*}
\]

Note that the second statement above introduces "?" in the query and the first one adds corresponding data sub-string to the argument list. Here \([y']\) represents a list consisting of the single variable \( y' \) and operator \# represents a list concatenation operation. The operation \([y'] \# \text{args}(z')\) computes a list by concatenating the list \([y']\) and the list \(\text{args}(z')\) in that order. If \( t_{j'-1} \) is a right child of \( t_{j'} \) then Rule1 is applied in a symmetric fashion i.e., \( \text{var}_{j'} = y' \cdot \tau'' \), variable \( z' \) is used in place of \( y' \), \( \text{args}(y') \) is used in place of \( \text{args}(z') \), and \( z' \) is added at the end of the list \(\text{args}(y')\). This rule is applied to transform the lines 4 and 5 of the Figure 7.

**Rule2: Propagating arguments.** For each \( j'', j'' < j'' \leq k \), the following rule adds an additional statement immediately before the \( S_{j''} \) to propagate the argument introduced by Rule1.

\[
\begin{align*}
\text{args}(\text{var}_{j''}) &= \begin{cases}
\text{args}(z'') & \text{if } S_{j''} : \text{var}_{j''} = z'' \\
\text{args}(y_1'') \# \text{args}(y_2'') & \text{if } S_{j''} : \text{var}_{j''} = y_1'' \cdot y_2''
\end{cases}
\end{align*}
\]

The argument lists for \( \text{var}_{j''} \) is obtained by concatenating the lists \( \text{args}(y_1'') \) and \( \text{args}(y_2'') \) in that order. If either one of \( y_1'' \) or \( y_2'' \) is a constant string, the above rule sets the argument list to
be the argument list of the non-constant variable. Note that $z''$ cannot be a constant string. This rule is used to transform the line 6 in the Figure 7.

**Ensuring property (*)&.** Now we consider the case when property (*) is not satisfied. In this case, we transform the program $P$ into another equivalent program for which the property (*) is satisfied. Let $\Delta$ be the set of all data sub-strings $u$ of the query string $s$ such that property (*) is violated for them, i.e., $u$ is a strict sub-string of the string generated by $\tau_u$.

Now, observe that $r_u$ has two children, otherwise $\tau_u$ will not be the smallest sub-tree that generated $u$. Let the label of $r_u$ be $(m, y)$. Clearly $S_m$ is of the form $y = z_1 \cdot z_2$. Observe that each leaf node of $T_i$ is labeled with a constant string or the name of a variable. For each $u \in \Delta$, we transform $P$ as follows. Fix any such $u$. Chose a new variable $x_u$ and add a new statement at the beginning of $P$ initializing $x_u$ to the empty string.

The transformation outlined below removes part of $u$ that was computed in $z_1$ and stores it in $x_u$. Let $v$ be a leaf node of $\tau_u$ such that the left most element of $u$ falls in the label of $v$. The label of $v$ can be written as $s' \cdot s''$ such that $s''$ is the part that falls in $u$. Let $t_1, \ldots, t_k$ be the sequence of nodes in $\tau_u$ from the parent of $v$ to $r_u$ where $r_u$ is the root node of $\tau_u$. For $1 \leq j < k$, replace $S_j$ by $\text{New}(S_j)$ as defined below.

$$\text{New}(S_j) = \begin{cases} \{x_u = s'' \cdot x_u; \; \text{var}_j = s'\} & \text{if } j = 1 \& S_1: \text{var}_j = s' \cdot s'' \\ \{x_u = x_u \cdot z; \text{var}_j = \text{var}_{j-1}\} & \text{if } 1 < j < k \& S_j: \text{var}_j = \text{var}_{j-1} \cdot z \end{cases}$$
After this, we identify the leaf node \( w \) of \( \tau_u \) such that the right most element of \( u \) falls in the label of \( w \). \( P \) is modified in a symmetric fashion updating variable \( x_u \). Finally, we replace \( S_m \) (root of the \( \tau_u \)) by the following two statements — \( x_u = z_1 \cdot x_u; \ y = x_u \cdot z_2 \).

The above transformation is done for each \( u \in \Delta \). We say that changes corresponding to two different strings in \( \Delta \) are conflicting if both of them require different changes to the same statement of \( P \). Our handling of the cases of conflicting changes is explained in the next section. Here we assume that changes required by different strings in \( \Delta \) are non-conflicting; Let \( P' \) be the resulting program after changes corresponding to data strings in \( \Delta \) have been carried out. It can be easily shown that \( P' \) is equivalent to \( P \), i.e., the query string generated in the variable \( q \) by \( P' \) is same as the one generated by \( P \). Further more, \( P' \) can be shown to satisfy the property (*)

5.5.2 Handling of Conditionals and Procedures

In this section, we discuss our approach and implementation for programs that include branching statements, function invocations and loops.

Let us first consider branching statements. For programs that include these constructs, TAPS performs inter-procedural slicing of system dependency graphs (SDGs) (61). Intuitively, for all queries that a SQL sink may receive, the corresponding SDG captures all program statements that construct these queries (data dependencies) and control flows among these statements. TAPS then computes backward slices for SQL sinks such that each slice represents a unique control path to the sink. Each of these control paths is indeed a straight line program, and is transformed according to our approach described in the previous section. A key issue here is the possibility of conflicts: when path \( P_1 \) and \( P_2 \) of a program share an instruction (statement) \( I \) that contributes to the data argument, then instruction \( I \) may not undergo the same transformation along both paths,
and TAPS detects such conflicts. Conflict detection and resolution is described in more detail in Section 5.5.4. Also note that the inter-procedural slicing segregates unique sequences of procedures invoked to construct SQL queries. Such sequences may have multiple intra-procedural flows e.g., conditionals. These SDGs are then split further for each procedure in above construction such that each slice contains a unique control flow within a procedure.

The above discussion captures loop-free programs. Handling loops is challenging as loops in an application can result in an arbitrary number of control paths and therefore we cannot use the above approach of enumerating paths.

5.5.3 Loop Handling

First of all, let us consider programs that construct and execute the entire query inside a single iteration of the loop. Let us call the query so constructed a loop independent query. In this case, the body of the loop does not contain any intervening loops. To ensure whether a query location is loop independent, our approach checks for the following sufficient conditions: (1) the query sink is in the loop body and (2) every variable used in the loop whose value flows into the sink does not depend on any other variable from a previous iteration. Once these conditions are satisfied, our approach handles loop independent queries as described in Section 5.5.1.

However, there may be other instances where loop bodies do not generate entire queries. The most common example are query clauses that are generated by loop iterations. Consider the following example:

1. $u1 = \text{input}(); \; \; u2 = \text{input}();$
2. $q1 = \text{"select * from X where Y ="}. u1;$
3. while ( --$u2 > 0) { 
4.   $u1 = input(); 
5.   $q2 = $q2." OR Y=".$u1; 
6. } 
7. $q = $q1.$q2; 
8. sql.execute($q); 

In this case, our approach aims to summarize the contributions of the loop using the symbolic regular expressions. In the above case, at the end of the loop, our objective is to summarize the contribution of $q2 as ( OR Y=$u1)*, so that the symbolic query expression can now be expressed as 

\[
\text{select * from X where Y = $u1( OR Y=$u1)*}
\]

The goal of summarization is essentially to check whether we can introduce placeholders in loop bodies. Once we obtain a summary of the loop, if it is indeed the case that the loop contribution is present in a “repeatable” clause in the SQL grammar, we can introduce placeholders inside the loop. In the above example, since each iteration of the loop produces an OR clause in SQL, we could introduce the placeholder in statement at line 5, and generate the corresponding PREPARE statement at runtime.

Previous work (62) has shown that the body of a loop can be viewed as a grammar that represents a language contributing to certain parts of the SQL query, and a grammar can be automatically extracted from the loop body as explained there. We will need to check whether the language generated by this grammar is contained in the language spawned by the repeatable (pumped) strings generated by the SQL grammar. Note that this containment problem is not the
same as the undecidable general language containment problem for CFGs, as the SQL grammar is a fixed grammar. However, a decision procedure specific to the SQL grammar needs to be built.

We instead take an alternative approach for this problem by ensuring that the loop operations produce regular structures. To infer this we check whether each statement in the body of the loop conforms to the following conditions: (1) the statement is of the form \( q \rightarrow x \) where \( x \) is a constant or an input OR (2) it is left recursive of the form \( q \rightarrow qx \), where \( x \) itself is not recursive, i.e., resolves to a variable or a constant in each loop iteration. It can be shown that satisfaction of these conditions yields a regular language. The symbolic parser is now augmented to see if the regular structure only generates repeatable strings in the SQL language. If this condition holds, we introduce placeholders as described earlier. We find our strategy for loops quite acceptable in practice, as shown in the next section.

5.5.4 Implementation

We implemented TAPS to assess our approach on PHP applications by leveraging earlier work Pixy (63, 64) and extending it with algorithms to convert programs to Static Single Assignment (SSA) format (65), and then implementation of the transformation described earlier. We briefly discuss some key points below.

We used an off-the-shelf SQL parser and augmented it to recognize symbolic expressions in query strings. The only minor change we had to make was to recognize query strings with associative array references. An associate array access such as \( $x['member'] \) contains single quotes and may conflict with parsing of string contexts. To avoid premature termination of the data parsing context, TAPS ensures that unescaped string delimiters do not appear in any symbolic expression.

Limitations and developer intervention.
TAPS requires developer intervention if either one of the following conditions holds: (i) the main assumption is violated (Section 5.4.2) (ii) a well-formed SQL query cannot be constructed statically (e.g., use of reflection, library callbacks) (iii) the SQL query is malformed because of infeasible paths that cannot be determined statically (iv) conflicts are detected along various paths (v) query is constructed in a loop that cannot be summarized.

TAPS implements static checks for all of the above and generates reports for all untransformed control flows along with program statements that caused the failure. A developer needs to qualify a failure as: (a) generated by an infeasible path and ignore or (b) re-write of violating statements possible. The number of instances of type (a) can be reduced by more sophisticated automated analysis using decision procedures. In case of (b), TAPS can be used after making appropriate changes to the program. In certain cases, the violating statements can be re-written to assist TAPS e.g., a violating loop can be re-written to adhere to a regular structure as described earlier. The remaining cases can either be addressed manually or be selectively handled through other means e.g., dynamic prevention techniques.

In case of failures, TAPS can also be deployed to selectively transform the program such that control paths that are transformed will generate prepared queries, and those untransformed paths will continue to generate the original program’s (unsafe) SQL queries. The sufficient condition to do this in a sound manner is that the variables in untransformed part be not dependent (either directly or transitively) on the variables of the transformed paths. In this case, the transformation can be done selectively on some paths. All sinks will be transformed to PREPARE statements, and any untransformed paths will make use of the PREPARE statements (albeit with unsafe strings) to issue SQL queries with an empty argument list.
5.6 Evaluation

Our evaluation aimed to assess TAPS on two dimensions: (a) effectiveness of the approach in transforming real world applications, and (b) performance impact of transformation induced changes. We built a test suite to assess the first and a microbench to assess the latter. The time taken to transform these applications was also measured in a performance experiment. We elaborate our findings for each experiment below.

5.6.1 Effectiveness

This test aimed to assess the effectiveness of TAPS by transforming real applications and measuring the number of successfully transformed SQL query execution locations (sinks) and control flows reaching them, number of modified lines of code (LOC), and identifying exceptional cases as well as information provided in such cases.

Test suite. The test suite for this experiment was built using SQLIA vulnerable applications from another research project on static analysis (66) and was extended with several applications from Common Vulnerabilities and Exposures (CVE) list with known SQLIA vulnerabilities reported in 2009. Table XIV lists these applications with their codebase sizes in lines of code and any known CVE vulnerability identifiers. In this testsuite, the Warp CMS application was the largest application in terms of lines of code whereas the Utopia news pro outnumbered other applications in terms of number of database operations.

Transformed control flows. The result of TAPS transformation for the effectiveness testsuite is summarized in Table XV. For each application (column 1) this table lists number of analyzed SQL sinks and control flows that execute queries at SQL sinks (column 2 and 3), transformed SQL sinks and control flows (column 4 and 5) and number of control flows that
<table>
<thead>
<tr>
<th>Application</th>
<th>Size (LOC)</th>
<th>CVE Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>WarpCMS</td>
<td>22,773</td>
<td>-</td>
</tr>
<tr>
<td>Utopia NewsPro</td>
<td>7,323</td>
<td>-</td>
</tr>
<tr>
<td>AlmondSoft</td>
<td>6,633</td>
<td>CVE-2009-3226</td>
</tr>
<tr>
<td>PortalXP TE</td>
<td>5,121</td>
<td>CVE-2009-3148</td>
</tr>
<tr>
<td>Gravity Board</td>
<td>2,422</td>
<td>CVE-2009-1277</td>
</tr>
<tr>
<td>MyNews</td>
<td>1,792</td>
<td>CVE-2009-0739</td>
</tr>
<tr>
<td>Auth</td>
<td>284</td>
<td>CVE-2009-0738</td>
</tr>
<tr>
<td>BlueBird</td>
<td>288</td>
<td>CVE-2009-0740</td>
</tr>
<tr>
<td>Yap Blog</td>
<td>264</td>
<td>CVE-2009-1038</td>
</tr>
</tbody>
</table>

**TABLE XIV**

TAPS: Effectiveness suite applications

required developer intervention (column 6). In this test suite, the larger applications invoked a small number of functions to execute SQL queries. This caused the number of analyzed sinks and control flows to vary across applications.

For the three largest applications, TAPS transformed 93%, 99% and 81% of the analyzed control flows. Although smaller in LOC size, the Utopia news pro application had a greater fraction of code involving complex database operations and required analyzing more control flows than any other application. For the remaining applications, TAPS achieved a transformation rate of 100%. This table suggests that TAPS was effective in handling the many diverse ways that were employed by these applications to construct queries.

TAPS did not find any partial query string variables used in operations other than append, null checks and output generation / logging (supports main assumption from Section 5.4.2). Further, TAPS did not encounter conflicts while combining changes to program statements required for transformed control flows.
TABLE XV

Effectiveness suite applications, transformed SQL sinks and control flows: TAPS transformed over 93% and 99% of the analyzed control flows for the two largest applications.

**Untransformed control flows.** The last column of Table XV indicates that TAPS requires human intervention to transform some control flows.

As TAPS depends on symbolic evaluation, it did not transform flows that obtained queries at run time e.g., the Warp CMS application used SQL queries from a file to restore the application’s database. In two other instances, it executed query specified in a user interface. In both these cases, no meaningful `PREPARE` statement is possible as external input contributes to the query structure. If the source that supplies the query is trusted, then these flows can be allowed by the developer.

The limitations of the SQL parser implementation were responsible for two of the three failures in the Utopia news pro application and the rest are discussed below.

**Queries computed in loops.** A total of 18 control flows used loops that violated restrictions imposed by TAPS and were not transformed (11 - Warp CMS, 1 - Utopia news pro, 6 - AlmondSoft). These control flows generated queries in loop bodies that used conditional statements or
### TABLE XVI

TAPS: Transformation altered less than 5% lines for large applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Statements changed (%)</th>
<th>Args extracted Avg (max)</th>
<th>Functions traversed Avg (max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WarpCMS</td>
<td>438 (1.9%)</td>
<td>6.6 (27)</td>
<td>2.2 (3)</td>
</tr>
<tr>
<td>Utopia News Pro</td>
<td>333 (4.5%)</td>
<td>1.1 (8)</td>
<td>2.9 (6)</td>
</tr>
<tr>
<td>AlmondSoft</td>
<td>46 (0.7%)</td>
<td>1.3 (4)</td>
<td>1.3 (2)</td>
</tr>
<tr>
<td>PortalXP TE</td>
<td>332 (6.5%)</td>
<td>1.5 (9)</td>
<td>1.0 (1)</td>
</tr>
<tr>
<td>Gravity Board</td>
<td>172 (7.1%)</td>
<td>1.5 (15)</td>
<td>1.0 (1)</td>
</tr>
<tr>
<td>MyNews</td>
<td>56 (3.1%)</td>
<td>2.4 (5)</td>
<td>2.5 (3)</td>
</tr>
<tr>
<td>Auth</td>
<td>17 (6.1%)</td>
<td>3.0 (4)</td>
<td>2.0 (2)</td>
</tr>
<tr>
<td>BlueBird</td>
<td>17 (6.0%)</td>
<td>3.0 (4)</td>
<td>2.0 (2)</td>
</tr>
<tr>
<td>Yap Blog</td>
<td>8 (3.0%)</td>
<td>4.0 (7)</td>
<td>2.0 (2)</td>
</tr>
</tbody>
</table>

nested loops. We also found 23 instances of queries computed in loops, including a summarization of \texttt{implode} function, that were successfully transformed. In all such cases queries were either completely constructed and executed in each iteration of the loop or loop contributed a repeatable partial query.

For untransformed flows TAPS precisely identified statements to be analyzed e.g., the Warp CMS application required 195 LOC to be manually analyzed instead of complete codebase of 22K LOC. This is approximately two orders of magnitude reduction in LOC to be analyzed.

**Changes to applications.** Table XVI captures transformation statistics for TAPS. As shown in the second column a small fraction of original LOC was modified during transformation. The columns 3 and 4 of this table show average (maximum) number of data arguments extracted from symbolic queries and functions traversed to compute them, respectively. 2% of changes in LOC were recorded for Warp CMS - the largest application, whereas approximately 5% of lines
changed for database intensive Utopia news pro application. We noticed that a significant portion of code changes only managed propagation of the data arguments to `PREPARE` statements. Some of these changes can be eliminated by statically optimizing propagation of arguments list e.g., for all straight line flows that construct a single query, `PREPARE` statement can be directly assigned the argument list instead of propagating it through the partial queries. Overall, this small percentage of changes points to TAPS’s effectiveness in locating and extracting data from partial queries.

Further, as columns 3 and 4 suggest, TAPS extracted a large number of data arguments from symbolic queries constructed in several non-trivial inter-procedural flows. For a manual transformation both of these vectors may lead to increased effort and human mistakes and may require substantial application domain expertise. For successfully transformed symbolic queries the deepest construction spanned 6 functions in the Utopia news pro application and a maximum of 27 arguments (in a single query) were extracted for the Warp CMS application, demonstrating robust identification of arguments.

5.6.2 Performance Experiment

**Performance of transformed applications.** TAPS was assessed for performance overhead on a microbench that consisted of an application to issue an `insert` query. This application did not contain tasks that typically interleave query executions e.g., HTML generation, formatting. Further, the test setup was over a LAN and lacked typical Internet latencies. Overall, the microbench provided a worst case scenario for performance measurement.

We measured end-to-end response times for 10 iterations each with TAPS transformed and original application and varied sizes of data arguments to `insert` queries from 256B to 2KB. In some instances TAPS transformed application outperformed the original application. However, we
did not find any noteworthy trend in such differences and both applications showed same response times in most cases. It is important to note here that dynamic approaches typically increase this overhead by 50% or more (67). Whereas, TAPS transformed application’s performance did not show any differences in response times. Overall, this experiment suggested that TAPS transformed applications do not have any overheads.

**Performance of the tool.** We profiled TAPS to measure the time spent in the following phases of transformation: conversion of program to SSA format, enumeration of control flows, static checks for violations described earlier, derivation tree generation and changing the program. The time taken by each phase is summarized in the last four columns of Table XVII. The largest application took around 2 hours to transform whereas the rest took less than an hour. The smallest three applications were transformed in less than 5 seconds. For large applications TAPS spent a majority of time in the SSA conversion. The only exception to this case occurred for AlmondSoft
application which had smaller functions in comparison to other applications and hence SSA conversion took lesser time. We wish to note here that TAPS is currently not optimized. A faster SSA conversion implementation may improve performance of the tool and by summarizing basic blocks some redundant computations can be removed. For a static transformation these numbers are acceptable.

5.7 Summary

This chapter presented TAPS, a static program transformation tool that modifies web applications to make use of PREPARE statements. By making use of PREPARE statements TAPS extends “textbook defense” offered by PREPARE statements to legacy web applications.

We demonstrated the effectiveness of TAPS by automatically transforming several open source web applications. TAPS automatically transformed over 90% of SQL queries in large applications it was evaluated on. Further, it addressed several challenges by successfully reasoning about many complex ways in which applications compute these queries. As TAPS automates a majority of tasks in transitioning legacy web applications to make use of PREPARE statements, it provides an effective tool to combat the serious risks of SQLIA in legacy web applications.

A small fraction of queries could not be transformed with TAPS due to one of the following two reasons: a) queries were unavailable statically and b) queries were computed in complex loops. The former cannot be fixed with a static method and does require human intervention. The latter poses interesting research problems and will be investigated to extend TAPS to complex loops. For such untransformed queries TAPS precisely identifies statements computing them thus enabling a less effort intensive manual transformation.
In summary, TAPS provides strong evidence that it is possible to successfully design retrofitting techniques that guarantee security (by construction) in legacy applications, and eliminate well known attacks.

This chapter presented TAPS a sound program transformation technique to retrofit legacy web applications with PREPARE statements. The next chapter provides guidance to future efforts towards prevention of parameter tampering vulnerabilities and identifies challenges that must be addressed.
CHAPTER 6

DISCUSSION

As parameter tampering vulnerabilities are pervasive and represent serious security problems, this thesis establishes the need for further research to prevent these vulnerabilities. One potential solution is to automatically generate patches for these vulnerabilities. In this chapter we identify challenges involved in generating patches that any follow-up research should address.

This chapter is organized as follows. Section 6.1 elaborates patch generation goal, Section 6.2 provides intuitions for computing patches and Section 6.3 enumerates challenges that need to be addressed.

6.1 Goal

As noted before, parameter tampering vulnerabilities arise when the server-side code for processing forms fails to enforce intentions encoded by the corresponding client-side code. Specifically, the client-side code for processing forms encodes intended validation by placing semantic restrictions on user inputs which the server-side code fails to replicate. We aim to generate patches that would enable the server-side code to enforce these intentions.

We first note that parameter tampering vulnerabilities are present in the server-side code. Thus patching these vulnerabilities entails generating additional code that would enable the server-side code to thwart parameter tampering attacks. Specifically, as parameter tampering vulnerabilities arise from missing checks on inputs, the patch would essentially comprise of any missing checks on
user inputs. Thus we aim to augment the server-side validation such that it is at least as strong as the corresponding client-side validation.

6.2 **Intuition for Patching**

We revisit the running example from the background chapter and focus on the parameter $quantity_1$ (Section 2.1). The client-side JavaScript code for this application requires the end user to specify a positive value for the parameter $quantity_1$, thus encoding intent of the application with respect to this parameter (Listing 2.1). However, the corresponding server-side code does not check value of the parameter $quantity_1$ and fails to enforce the intent (Listing 2.2). This lapse in enforcing intentions was subsequently exploited by supplying a negative value for the parameter $quantity_1$.

The intuition for patch generation is to locate and fix such missing enforcement of intentions by augmenting the server-side code. In our running example, parameter tampering vulnerability of the parameter $quantity_1$ can be patched by augmenting the server-side code to check and reject negative values of the parameter $quantity_1$. Intuitively, this requires us to augment the server-side code with the following check: $\text{exit if } (quantity_1 < 0)$ i.e., forbid execution if value of the parameter $quantity_1$ is negative.

To fix parameter tampering vulnerabilities in the running example, the following patch needs to be generated:
exit if

(  \iffalse input-verifier

quantity_1 < 0 \text{ OR } quantity_2 < 0 \text{ OR }

\text{card } \notin (1234-5678-9012-3456 \text{ OR } 7890-1234-5678-9012)

\fi )

\text{AND}

(  \iffalse path-selector

\text{op = “purchase” AND not-set discount}

\fi )

The above intuitive patch contains two parts: a) input-verifier and b) path-selector. The input-verifier part of the patch builds on $f_{client}$ and $f_{server}$ for the running example and replicates client-side checks that the server-side code failed to enforce. The input-verifier fixes parameter tampering vulnerabilities of the running example by replicating missing checks on the parameters $quantity_1$, $quantity_2$ and $\text{card}$. Note that the above patch does not replicate client-side check on the parameter $name$ as its value is truncated by the server-side code thus implicitly enforcing the corresponding client-side check (Listing 2.2).

The path-selector part of the patch builds on $f_{server}$ and aims to trigger execution of this patch only when these parameter tampering vulnerabilities will be exploited. We first note that the generated $f_{server}$ corresponds to a run of the web application. Thus the parameter tampering vulnerabilities are applicable to only that run of the web application. The path-selector aims to limit the applicability of the patch to the analyzed run of the web application.
Generating the above patch is very challenging and the next section highlights the key challenges.

6.3 Challenges

This section elaborates the major challenges in generating patches for parameter tampering vulnerabilities: a) generating patches to only forbid parameter tampering attacks (Section 6.3.1), b) incorporating server-side processing of inputs to generate sound patches (Section 6.3.2) and c) generating generic patches that factor-in user-specific or session-specific information in forms (Section 6.3.3).

6.3.1 Patching the Vulnerable Path

The first challenge is to make sure that the patch is only activated for the path in the server-side code that was identified to be vulnerable. This challenge stems from the way $f_{server}$ is computed. The $f_{server}$ is computed from the trace generated for one run of the web application. Such a trace only captures checks performed in a single control path that the server followed in the run. As a result, the $f_{server}$ computed by analyzing the trace only captures checks done by the application in this particular control path. Subsequently, the parameter tampering vulnerabilities identified by using such an $f_{server}$ are only applicable to a single control path in the server-side code and any patch to fix these vulnerabilities must also be path specific.

The key intuition to address this challenge is based on the following observation: $f_{server}$ contains conditions that must be satisfied to re-run the vulnerable path. First, we note that our trace generation algorithm only reports decision points (conditional statements) in the server-side code that act on user inputs (Section 4.5.2.3). By doing so, the resulting trace precisely captures how the control flow proceeded at each decision point with respect to user inputs. If we supply inputs that satisfy these conditions captured by $f_{server}$, the server-side code would execute the same control
path that was executed to generate the trace itself. Intuitively, the trace captures footprints of the server-side code as it traverses various decision points in the code. This traversal is captured by conditions in $f_{server}$ and inputs that satisfy these conditions, would guide the server-side code execution to re-trace its footprints.

Let's suppose that the running example server-side code is first executed with the following parameters:

\[
\begin{align*}
    \text{quantity}_1 &= 1, \quad \text{quantity}_2 = 1, \quad \text{card} = "1234-5678-9012-3456", \\
    \text{name} &= "Alice", \quad \text{address} = "wonderland", \quad \text{delivery} = "none", \\
    \text{op} &= "purchase"
\end{align*}
\]

The resulting $f_{server}$ by analyzing the produced trace is:

\[
\begin{align*}
    \text{op} = "purchase" \ \text{AND} \ \neg \text{isset} \ \text{discount} \ \text{AND} \\
    \leq \text{len} \ (\text{name}) \ 30 \ \text{AND} \ \neg = \text{address} \ \text{NULL}
\end{align*}
\]

In this formula, the last conjunct is contributed by the database analysis and hence does not dictate the choices made at decisions points. Similarly, the second last conjunct is generated by WAPTEC to capture the sanitization done by $\text{substr}$ function and hence does not involve decision points. A satisfying assignment to the other two conjuncts would produce $\text{op} = "purchase"$ as a satisfying assignment to "$\neg \text{isset} \ \text{discount}$" yields no value assignment to the parameter $\text{discount}$.

If we execute the server-side code with the value $\text{op} = "purchase"$ (Listing 2.2), it follows the exact same control path as the run triggered by the inputs above.

The above discussion provides the key intuition behind generation of the path-selector part in the patch shown before i.e., the conditions checked by the $f_{server}$ help application of the patch to the vulnerable path. For our running example the path-selector is computed as

\[
\begin{align*}
    \text{op} = "purchase" \ \text{AND} \ \neg \text{isset} \ \text{discount}
\end{align*}
\]
6.3.2 Incorporating Server-side Changes to Inputs

This challenge stems from the possibility that the server-side code may alter user inputs before using them in sensitive operations and we discuss it with the help of an example. Suppose a web application accepts a parameter name from user and then the server-side code splits the name into a first and a last name. Further, the server-side code requires that the length of the first and the last names is less than 10 characters. The challenge for patch generation is to reason about how constraints on the first and the last name translate to constraints on the user input name and if they are sufficient to prevent parameter tampering of the name parameter. What we are pointing to with the help of this example is a deeper question: how does the processing of inputs at the server-side code impact identification of parameter tampering vulnerabilities and subsequently correctness of patches generated to prevent these vulnerabilities.

We note that the above discussion points to inherent limitations in reasoning done by WAPTEC with respect to processing of inputs at the server-side. Specifically WAPTEC may report false alarms if it cannot reason about processing done on user inputs due to the above problem or insufficient support from underlying constraint solver platform. In our evaluation we have not encountered such instances. However, the goal of WAPTEC was to establish prevalence of parameter tampering vulnerabilities and not to attain completeness or soundness. The results produced by WAPTEC perhaps justify its current approach.

Although some progress has been made in WAPTEC towards addressing this challenge, by and large it remains an open problem. WAPTEC makes progress towards addressing this challenge as it captures symbolic expressions on user inputs that are used in sensitive operations. For example, the running example server-side code truncates all names to ensure that they contain 30 or less
characters which is captured as a constraint $\leq \text{len } (\text{name}) 30$ in the $f_{server}$. Although the approach adopted by WAPTEC was sufficient to establish prevalence of parameter tampering vulnerabilities, to generate sound patches this challenge must be further researched and addressed through a more precise approach.

6.3.3 Factoring in Dynamism in Form Generation

Let us revisit the intuitive patch shown at the end of Section 6.2. Such a patch, if deployed as is, may cause the following two serious problems for the web application.

- **User Specific Patch.** This patch may render the application unusable for other users. Lets imagine that the intuitive patch shown before was generated for a user Alice who used cards card1 and card2 in the past. The employed patch thus demands Alice to specify one of card1 or card2 in the future as well. Unfortunately, the patch does not distinguish between users. As a result, when a different user say Bob who used cards card3 and card4 in the past attempts to use this web application, the deployed patch rejects the request as it does not specify either card1 or card2. Thus deployment of this patch will make the application unusable to all but Alice.

- **Session Specific Patch.** Unfortunately, such a patch may render the application unusable to Alice as well. After the cards card1 and card2 expire, this patch will not allow Alice to specify a new card.

The above narration points to a deeper problem: applications may utilize user-specific or time-sensitive information while generating forms and patches should be oblivious to such changing information.
It is evident that an approach to generate patches must incorporate such dynamic information in generating patches. More research is needed to find a satisfactory answer to this challenge.

This chapter identified challenges that need to be addressed to automatically generate patches for parameter tampering vulnerabilities. The next chapter reviews the related work.
CHAPTER 7

RELATED WORK

This chapter covers contemporary research work in the area of improving security of web applications. We classify these approaches broadly under the following four groups: (a) vulnerability detection techniques that warn the programmer of possible attacks (Section 7.1), and (b) vulnerability prevention techniques that detect vulnerabilities and simultaneously prevent them (Section 7.2), and (c) detection / prevention techniques based on extracting specification of safe behavior from programs (Section 7.3), and (d) new paradigms that aim to incorporate security ground up (Section 7.4). In the rest of this section we provide detailed treatment of each of these groups followed by discussions that place contributions of this thesis into the broader context of each group.

7.1 Vulnerability Detection Techniques

Vulnerability detection techniques provide the crucial facility to identify vulnerabilities in web applications. These techniques can be employed post-deployment as well as during development to help programmers in avoiding critical mistakes. In this section, we broadly highlight efforts in the following broad categories: static analysis based techniques (§7.1.1), dynamic analysis based techniques (§7.1.2) and test input generation (§7.1.3) and then discuss how these efforts relate to the contributions of this thesis (§7.1.4).

7.1.1 Static Analysis Based Techniques

There has been extensive research on static analysis to detect vulnerabilities in web applications (31, 46, 56, 62, 66, 68, 69, 70, 71, 72, 73, 74, 75, 76). The most common theme of detection
approaches is to reason about sources (user inputs) and their influence on query strings issued at sinks (sensitive operations) or intermediate points (sanitization routines). These techniques aim to check whether every source-to-sink data flow passes through an input validation function. Livshits and Lam (68) propose a flow-insensitive analysis to discover such flows that employ improper validation while Xie and Aiken (69) propose a flow-sensitive analysis. Halfond et al. (31, 71) use static analysis to model all possible SQL queries reaching at a sink, and enforce compliance at runtime. Several others (62, 66, 72) use formal languages to model outputs reaching at sinks and detect vulnerabilities by computing intersection with predefined languages that represent attacks. Balzarotti et al. (73) construct an extended state to reason about multi-module vulnerabilities in the programs. Fu et al. (74) use static analysis to identify and generate test cases for SQL injection vulnerabilities in applications. Tripp et al. (75) employ static taint analysis to discover vulnerabilities in web applications. Kiezun et al. (46) use repositories of attack payloads to find SQL injection and cross-site scripting vulnerabilities in web applications. New developments in this area employ static analysis to detect JavaScript malware (76) and to analyze JavaScript (56).

7.1.2 Dynamic Analysis Based Techniques

Authors of FLAX (77) take a dynamic approach to detect cross-site scripting (XSS) vulnerabilities. This work identifies slices of web application client-side code that accept user input and can produce JavaScript output. The FLAX tool injects randomized data inputs to the slice (i.e., fuzzing), attempting to cause JavaScript output by exploiting insufficient input validation. Vulnerabilities detected this way can be leveraged to mount XSS attacks. Felmetsger et al. (39) propose a dynamic analysis based technique to find logic vulnerabilities in web applications. This work employs dynamic analysis to observe the normal operation of web application to infer a set of
behavioral specifications and then uses static analysis to find flows that may violate these specifications. Balduzzi et al. (78) propose a dynamic analysis based technique to identify HTTP parameter pollution vulnerabilities in web applications.

7.1.3 Test Input Generation

A rich literature exists on automating the task of test input generation (47, 48, 56, 79, 80, 81, 82). Saxena et al. (56) present an automated approach for testing JavaScript applications, implemented in the tool Kudzu. It combines the use of random test generation to explore the applications event space (i.e., the possible sequences of user-interface actions) with the use of symbolic execution for systematically exploring an applications value space (i.e., how the execution of control flow paths depends on input values). The main goal of their work is to find code injection vulnerabilities in the client-side code that result from untrusted data provided as arguments to, for example, eval. Artzi et al. (79) propose feedback-directed random testing heuristics to generate tests for JavaScript applications. Babić et al. (80) propose a technique for exploiting static analysis to guide dynamic automated test generation for binary programs, prioritizing the paths to be explored. Sen et al. (82) propose a technique that combines concrete and symbolic execution to avoid redundant test cases as well as false warnings. Authors of (48, 81) propose techniques to record an actual run of the program under test on either a well-formed input (81) or random inputs (48), symbolically evaluate the recorded trace, and gather constraints on inputs capturing how the program uses these. The collected constraints are then negated one by one and solved with a constraint solver, producing new inputs that exercise different control paths in the program.
7.1.4 Discussion

Static analysis provides a powerful means to build understanding of the program logic. By allowing analysis techniques to reason about all possible control flows in a program, it is very effective in generating wholistic view of programs. However, most detection techniques that employ static analysis do not address challenges mentioned in the background chapter (Section 3.3). First, these techniques often require the user to verify if discovered vulnerabilities indeed exist. Second, they require the end user to manually identify and declare the sanitizing blocks of code for each application, and hence these approaches are not fully automatable. More importantly, the tools do not in any way help the user in fixing the identified vulnerabilities. Thus many of these approaches fall short of addressing “scalability” (Section 3.3.0.7) and “Factoring in Input Validation / Sanitization” challenges (Section 3.3.0.4).

In the context of static analysis based detection techniques, our contributions are complementary. NoTamper and WAPTEC employ static analysis of the client-side code to extract application intended validation checks and subsequently find missing checks on the server-side. TAPS employs static analysis to extract application intended queries and then employs novel program transformation to enforce these intentions. However, NoTamper and WAPTEC realize a systematic methodology that uses program intent to guide discovery of vulnerabilities. NoTamper required moderate manual intervention for each analyzed form and discovered vulnerabilities. WAPTEC reduced these manual interventions by analyzing the server-side code and generated exploits that confirmed vulnerabilities by construction. Further, for each reported exploit, these tools offer a precise description of tamperable parameter along with constraints that the server-side code failed to enforce. Such reports will facilitate developers in understanding and fixing these vul-
nerabilities. Further, NoTamper and WAPTEC differ from attack-specific vulnerability detection techniques as they provide evidence that intention based vulnerability detection can be used as a “template” to reason about other attacks.

Automated test generation techniques aim to automate an important activity. In that sense, NoTamper and WAPTEC are similar to these approaches as they automatically generate hostile inputs to discover vulnerabilities. However, our formulation of the parameter tampering problem as one checking the consistency of the server and the client code bases and development of server-side methods specialized to this problem makes it distinctive from these approaches.

7.2 Vulnerability Prevention Techniques

Vulnerability prevention techniques are the most desired as they aim to accomplish a difficult and desired goal of preventing vulnerabilities in web applications. These techniques can safeguard legacy as well as freshly written web applications thus help improve security of the Web in general. In this section, we broadly highlight efforts in the following broad categories: learning based white-box defenses (§7.2.1) learning based blackbox defenses (§7.2.2) and then discuss how these efforts relate to the contributions of this thesis (§7.2.3).

7.2.1 Learning Based Whitebox Defenses

**Input Validation and Sanitization.** Filtering is a widely deployed solution for input validation and sanitization. Filters typically examine user supplied data before the server-side code processes it and aim to identify malicious patterns in them. Although an effective layer of defense, specifying and checking all malicious patterns is an effort-intensive and error-prone task (31,32). Additionally, techniques like alternate encoding of characters further complicates this task.
Consequently, writing effective filters requires deep understanding of attacks which developers often lack.

Finally, filters lack contextual awareness of how the application uses an input. In general, they examine each input in segregation which is insufficient to identify malicious payloads arising from combining these inputs. For example, a filter that forbids a SQL keyword `SELECT` may individually examine and allow two inputs with values `SEL` and `ECT`. However, if the application combines these inputs, it generates forbidden keyword `SELECT` from user supplied data. Such evasion techniques have been used to compromise security mechanisms of real world applications (9). Although convoluted, this example points to a fundamental limitation of filtering mechanisms that they lack awareness of how the application uses inputs.

**Tracking User Inputs.** A popular and powerful idea is to track use of untrusted inputs in applications (41, 42, 46, 60, 67, 83, 84). To do so, taint based approaches (46, 67, 83, 84) associate a flag (taint) with untrusted data, propagate the taint as applications compute arguments to sensitive operations and then reason about taintedness of arguments to sensitive operations. Several other approaches (41, 42, 60) propose techniques to learn use of untrusted inputs at sinks through bracketing inputs (41, 42) to reason about SQL keywords appearing within brackets, employing modified PHP platform to assign and track meta data to user inputs (85) and randomizing SQL keywords used in query computation to differentiate them from user input injected keywords (60).

**Code Re-writing.** The option to re-write vulnerable applications has been explored in other security relevant contexts (86) as well as non-security contexts (87, 88). Brumley and Song (86) proposed a static analysis based technique to split the application for privilege separation. Livshits and Emre (87) proposed a technique to analyze application workloads to automatically rewrite the
existing Web 2.0 applications for performance optimization. Volta (88) recompiles nondistributed
executables into functionally equivalent distributed form, inserting remoting and synchronization
boilerplate code and facilitating post hoc instrumentation.

Some recent works have also explored the idea of computing a model of intended behavior
and enforcing it by re-writing outputs (89, 90). Blueprint (89) computes a model of intended
scripts in web pages and enforces it by re-writing these scripts to prevent XSS attacks in existing
browsers. AdJail (90) re-writes web pages to enforce policies specified by advertisement publishers
on advertisements rendered in browsers.

**Automated PREPARE Statement Generation.** The following two approaches (91, 92) also
proposed employing PREPARE statements in legacy web applications through program re-writing.
Dysart and Sherriff (92) propose parsing plain-text SQL statements (symbolic queries) to identify
and extract inputs from them. Thomas et al. (91) propose a technique that requires a developer
to identify SQL code embedded in program strings. It then adds a reference monitor to generate
structure of PREPARE statements at runtime. Both of these approaches require significant human
intervention and thus will not scale to the task of transforming legacy web applications.

### 7.2.2 Learning Based Blackbox Defenses

Several approaches (93, 94, 95, 96, 97, 98, 99, 100, 101) analyze inputs and outputs of applications to
detect vulnerabilities and prevent attacks. Sekar (93) proposed a blackbox approach that intercepts
requests and responses to web applications and then applies syntax- and taint-aware policies to
detect / prevent code injection attacks. Web application firewalls (WAFs) usually incorporate
some combination of misuse and anomaly detection techniques to protect web applications from
attack (94, 95, 96, 97, 98). Rietta (99) and Valeur et al. (100) independently proposed anomaly
detection systems that compute a profile of normal queries to detect successful SQL injection attacks. Spectator (101) performs distributed data tainting to identify and prevent propagation of JavaScript worms by observing and tagging the traffic between a browser and a web application.

7.2.3 Discussion

Filtering techniques provide a first layer of defense for attacks on web applications. Consistent use of these techniques can eliminate “low hanging” vulnerabilities of these applications. However, as mentioned above these techniques cannot prevent subtle attacks owing to their lack of context awareness. Solutions developed as part of this thesis are compatible with existing filters in web applications. TAPS computes query arguments that incorporate affect of sanitization / validation functions. However, as it offers a fundamentally robust defense against SQL injection attacks, an application employed filtering for SQL injection attacks is redundant. WAPTEC captures existing validation / sanitization of user inputs in the server-side code ($f_{server}$ contains calls to in-built sanitization routines). However, WAPTEC’s reasoning of filter functions is inherently restricted by limitations of the underlying constraint solver.

The broader contributions of this thesis are also complementary with that of vulnerability prevention techniques. However, defenses developed in this thesis make advances in the area of precisely generating specifications as well as fundamentally enforcing them. We revisit challenges discussed in the background chapter (Section 3.3). On the precision front, existing approaches suffer from lack of: a) deep inter component analysis (41, 42, 60, 67, 83, 84), b) flow sensitivity (84), c) factoring in input validation / sanitization (41, 42, 60), d) preserving semantics of original application (41, 42, 60) and e) scalability (91, 92). We wish to note that all blackbox approaches for prevention (93, 94, 95, 96, 97, 98, 99, 100, 101) inherently suffer from one or more of these problems.
As discussed in Chapter 5 and Chapter 4, techniques developed in this thesis offer program analysis techniques that do not suffer from the above limitations and thus extract more precise specifications of program intent than any of these approaches. On the enforcement front, a key difference is that TAPS extends robustness of PREPARE statements (well known defense for freshly written code) to legacy web applications instead of employing ad-hoc enforcement mechanism that may miss certain cases thus leaving an application vulnerable.

Blackbox defense approaches provide interesting options. They may be the only recourse to validate / defend an application for which the source-code is not available. Also, as these approaches do not require apriori knowledge of the structure or implementation of a web application, they can be used on a larger set of applications than an equivalent whitebox approach. Whitebox defenses often tightly couple with technologies like PHP, JSP, ASP, etc. that are used to implement web applications. However, blackbox approaches lack the contextual awareness that whitebox approaches garner by analyzing the source code. Hence blackbox approaches may result in solutions that are both overly restrictive (forbidding benign inputs) and insufficiently restrictive (allowing hostile inputs). By trading off precision for scalability, blackbox defenses provide an interesting option to secure web applications.

7.3 Specification Extraction for Security

There exists a rich literature for inferring specification of program behavior for program verification (102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112) and many recent works have focused on inferring such specifications for web applications (39, 77, 113, 114) with the goal of vulnerability detection and prevention.
Several works infer types of variables in weakly typed languages (115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125). Static type inference has the potential to provide the benefits of static typing – well-typed programs don’t go wrong (126). Further, the type information enables static and dynamic verification of certain properties in programs as well as early detection of runtime errors. These approaches have employed symbolic execution and other static analysis techniques (115, 116, 117, 118) as well as monitoring of program execution (119, 120, 121, 122, 123, 124, 125).

A more closely related body of work to that of this thesis is the automated extraction of program invariants or other properties. Daikon (102) infers likely program invariants automatically by using statistical inference from a program’s execution traces. Engler et al. (103) detect security bugs in C programs by mining temporal safety patterns and checking for inconsistencies. Ammons et al. (104) employ a machine learning approach that observes a program’s execution and concisely summarizes the frequent interaction patterns as state machines that capture both temporal and data dependence. Whaley et al. (127) employ dynamic and static analysis to infer sequencing information for various programming interfaces and use this specification to find illegal call sequences in programs. DIDUCE (105) assists developers in finding programming errors by dynamically inferring simple invariants over the values of program variables. Alur et al. (106) employ predicate abstraction and symbolic model checking to automatically extract temporal specifications of correct sequencing of method calls for Java classes. Similarly, Weimer et al. (128) learn temporal specifications from error handling code and use it in error detection. Perracotta (121) infers temporal properties of programs by analyzing dynamic program traces and utilizes them in error detection. DySy (107) infers pre-conditions and post-conditions for program methods by combining symbolic execution with dynamic testing.
Baliga et al. (108) propose a technique to detect rootkits by externally observing the execution of the kernel during an inference phase and hypothesize invariants on kernel data structures. A rootkit detection phase uses these invariants as specifications of data structure integrity. Program invariants have also been inferred to enforce consistent use of data structures (109) and to automatically retrofit legacy system code to enforce authorization policies (110). AutoISES (111) is an approach for C program bug detection that mines for common security-related patterns and identifies deviations from these as vulnerabilities. Srivastava (129) et al. exploit the difference between multiple implementations of the same application programming interface to detect security violations. Livshits et al. (113) propose a technique to infer information flow specification with a goal to detect vulnerabilities such as SQL injection and cross-site scripting. This approach analyzes data propagation graph, which represents inter-procedural flow of information in the program. Authors of (39) as described before learn specifications of normal behavior to find logic vulnerabilities in web applications. Authors of (114) parse SQL queries generated by an application and identify HTTP parameters that appear in these queries. SQL injection vulnerabilities are then identified by injecting SQL injection payloads in such HTTP parameters and analyzing resulting SQL queries.

7.3.1 Discussion

Automatically extracting specification of safe behavior is a powerful idea. Such specification can enable vulnerability detection as well as their prevention. Many security relevant specification extraction works discussed above were concurrently developed with techniques of this thesis. The key differences offered by this thesis are two fold: a) techniques developed in this thesis offer deeper analysis to yield better specifications and b) we use specification in detection as well as prevention of vulnerabilities thus offering a crucial step beyond extracting specifications of safe
behavior. NoTamper and WAPTEC, offer analysis of the server-side code that also reasons about constraints embedded in database schemas thus providing better specifications. Also in the context of parameter tampering vulnerabilities, we are analyzing the two distinctive code bases of a single web application and have developed techniques to check consistencies between these two code bases. TAPS extracts specifications of SQL queries and robustly enforces them with the help of PREPARE statements.

7.4 New Paradigms

This section covers literature that proposes new mechanisms to enable development of fundamentally secure web applications or infrastructure such as new browsers that aim to mitigate damages caused by vulnerable web applications. In this section, we broadly highlight efforts in the following broad categories: security by construction (§7.4.1), new browser platforms (§7.4.2), prevention architecture (§7.4.3) and then discuss how these efforts relate to the contributions of this thesis (§7.4.4).

7.4.1 Developing Secure Applications

This category covers frameworks and techniques that enable development of fundamentally secure web applications.

Several existing (130,131,132,133,134) as well as newly proposed web application development frameworks (135,136) aim to generate web applications that are secure by construction. With the help of developer annotations, existing frameworks such as Apache Struts (130), ASP.Net (131) and Ruby on Rails (132) are able to automatically generate applications that are resilient to Cross-site Request Forgery (XSRF) attacks. Other web application development frameworks such as Django
(134) and Google's ctemplate (133) provide features that enable developers to conduct context sensitive sanitization of web documents.

Robertson and Vigna (135) propose a web application development framework that uses strong type systems to statically enforce a separation between the structure and content of both web documents and database queries generated by a web application, effectively generating applications that are immune to cross-site scripting and SQL injection vulnerabilities. Livshits and Erlingsson (136) propose a scheme for eliminating a wide range of script injection vulnerabilities in applications built on top of popular AJAX development frameworks such as the Dojo Toolkit, prototype.js, and AJAX.NET.

PREPARE statements, a facility provided by DBMS vendors, offer robust prevention of SQL injection attacks for freshly written web applications that issue SQL queries. These statements incorporate type enforcement mechanisms that confine user inputs to their data contexts. Several other proposals provide type safe APIs that strive to achieve PREPARE statements like confinement for dynamically constructed SQL queries (137, 138, 139, 140): SQL DOM (137) and Safe Query Objects (138), provide executable mechanisms that enable the user to construct queries that isolate user input whereas the other two works (139, 140) propose type systems and language extensions to enable typesafe querying of SQL tables.

7.4.2 New Browser Platforms

There has been a lot of proposals on new browser platforms (141, 142, 143, 144, 145, 146, 147). Development of these platforms is motivated by inconsistencies of existing browsers that present significant hurdles for web programmers to build robust and secure web applications. The key theme of these new browser platforms is to achieve varying degrees of isolation to restrict damages caused
by successful attacks. Chrome (142) and IE 8 (144) employ process sandboxing i.e., instantiate a separate process for each browser instance thus aiming to protect the host machine from the browser and the Web. The OP web browser (143) uses processes to isolate browser components (i.e., HTML engine, JavaScript interpreter, rendering engine) as well as pages of the same origin. Tahoma (145) uses virtual machines (VMs) to completely isolate web applications, disallowing any communications between the VMs. The Building a Secure Web Browser project (146, 147) uses SubOS processes to isolate content downloading, display, and browser instances. SubOS processes are similar to Unix processes except that instead of a user ID, each process has a SubOS ID with OS support for isolation between objects with different SubOS IDs. Gazelle (141) is centered around protecting principals from one another by separating their respective resources into OS-enforced protection domains.

7.4.3 Prevention Architectures

Recent works have also aimed at ensuring that the server side of a web application remains protected from malicious clients. Ripley (45) aims to detect malicious activities at the client by replicating the client execution in a trusted environment. SWIFT (44) uses information flow analysis during the development of new applications to ensure that constraints regarding information flow confidentiality and integrity will be met in client side code. Bethea et al. (148) discuss enforcement strategies for misbehaving clients in the context of online games.

7.4.4 Discussion

The solutions discussed under new paradigms provide much needed progress towards achieving security in the long run. Specifically by providing necessary support through frameworks and
languages, these research efforts are making progress in the direction of developing secure web applications.

We also wish to note that these new paradigms aim to achieve a much challenging goal “secure web application development”. There are several hurdles in achieving this goal. First, adoption / commercialization of these efforts may take time. Second, success of these techniques relies on human intervention in the form of annotations or consistent use of provided APIs. Achieving higher level of abstractions in these techniques, that minimize such human interventions, remains an interesting and challenging research problem.

The contributions of this thesis are complementary to these developments as we are interested in securing existing web applications. First, new paradigms do not offer any support for safeguarding legacy web applications. This thesis bridges this gap by extending power of new paradigms to legacy web applications. For example, \texttt{PREPARE} statements offer a robust defense for SQL injection attacks to freshly written applications. TAPS extends that defense to legacy web applications by retrofitting them to make use of \texttt{PREPARE} statements. Second, until new paradigms are adopted and consistently used, techniques developed in this thesis can provide a “stop-gap” solution by mitigating risks in web applications.

The valuable lessons learned in this thesis provide critical feedback to developers of new paradigms. For example, current frameworks offer no support for avoiding parameter tampering vulnerabilities in new web applications even though our results strongly indicate that such vulnerabilities are pervasive in legacy web applications. Web application development framework could easily adopt mechanisms to generate parameter tampering resilient applications. A simple strategy would be to
automatically generate the client-side and the server-side validation code from a common specification, thus ensuring that the server-side validation is as strong as the client-side validation.

This chapter discussed related work and broadly placed contributions of this thesis in their context. The next chapter concludes this thesis.
CHAPTER 8

CONCLUSION

In this thesis we underscored the importance of improving security of legacy web applications to improve the overall security of the Web. The Web contains a large number of vulnerable web applications that are exploited for profit. The process of web application development fails to incorporate security measures at various levels and lack of security awareness on the part of end-users and developers further contributes to exploitability of web applications. For these reasons discovery of robust techniques to find and fix security flaws in legacy web applications is of great importance.

Chapter 4 discussed NoTamper and WAPTEC to automatically identify parameter tampering vulnerabilities in web applications. This chapter highlighted lack of studies to detect or prevent parameter tampering vulnerabilities in contemporary literature and set out to find if these vulnerabilities indeed existed in legacy web applications. NoTamper and WAPTEC systematically analyzed the client-side form processing code to infer intended validation checks on user inputs. By negating these checks, NoTamper and WAPTEC proposed techniques to identify parameter tampering vulnerabilities. These projects studied a total of 17 web applications, including open source and commercial web applications, and identified 52 parameter tampering vulnerabilities. Starting with these vulnerabilities we generated exploits that demonstrate serious security problems: unauthorized monetary transactions at a bank, unauthorized discounts added to a shopping cart, and so on. In summary, NoTamper and WAPTEC empirically established that parameter tampering vulnerabilities pose a serious threat to web applications.
Chapter 5 discussed TAPS that prevent SQL injection vulnerabilities. TAPS offered the first sound method to transform existing web applications to employ \texttt{PREPARE} statements. \texttt{PREPARE} statements offer robust defense for SQL injection attacks and are widely adopted for developing new web applications. TAPS builds a high-level understanding of a program’s logic directly from its low-level string operations. TAPS uses this high-level understanding in identifying untrusted data arguments in query computations and then isolates them using \texttt{PREPARE} statements. Our evaluation of TAPS over several real world applications demonstrated effectiveness of TAPS as it automatically transformed over 90\% of queries in large web applications.

The empirical results produced by \textsc{NoTamper} and \textsc{WAPTEC} point to less explored areas of web application security and pervasiveness of vulnerabilities in these areas. Further, they disclose widespread malpractice of inconsistently replicating security relevant code across different modules of web applications. Such problems may be addressed by better frameworks that can automate replication tasks thus eliminating errors that naturally occur when multiple developers are involved. By identifying threats posed by parameter tampering vulnerabilities, studies such as \textsc{NoTamper} and \textsc{WAPTEC} provide vital feedback to researchers and the web application development community.

At a broader level, TAPS provides evidence that it is possible to retrofit legacy web applications with new robust defenses. \texttt{PREPARE} statements are mainly useful for freshly written applications as manually transforming legacy web applications to make use of \texttt{PREPARE} statements is both expensive and error-prone. New robust techniques such as \texttt{PREPARE} statements are much needed solutions to improve security of the freshly written code. However, they do not contribute towards improving
security of legacy web applications. TAPS extends security guarantees of \texttt{PREPARE} statements to legacy web applications by retrofitting them to make use of \texttt{PREPARE} statements.

In the long run, it is vital to develop robust new defenses and web application development tools and techniques that incorporate security in a ground up fashion. By automating and assisting developers in error-prone tasks such efforts may help reduce vulnerabilities in new applications. Given that web applications provide many essential services and are highly popular, such developments are a must to attain security in the long run. As new robust defenses are proposed, techniques such as TAPS can provide a bridge between these new robust defenses and legacy web applications thus improving security of already deployed part of the Web. By providing assistance in improving security of existing as well as new web applications, this thesis makes vital contributions towards improving security of the Web.
CITED LITERATURE


33. WAPTEC Supplementary Website. [http://sisl.rites.uic.edu/waptec](http://sisl.rites.uic.edu/waptec).

34. NOTAMPER Supplementary Website. [http://sisl.rites.uic.edu/notamper](http://sisl.rites.uic.edu/notamper).

35. TAPS Supplementary Website. [http://sisl.rites.uic.edu/taps](http://sisl.rites.uic.edu/taps).


VITA
895 Westmoreland Dr, Apt 21, Vernon Hills, Illinois 60061

Prithvi Bisht
☎ 3 1 2 - 8 1 0 - 1 8 1 2 ✉ p b i s h t @ c s . u i c . e d u

www.cs.uic.edu/~pbisht

Education

Doctor of Philosophy, CS
University of Illinois at Chicago
GPA: 4.0/4.0
Aug 2006 - Jun 2011 (expected)

Master of Technology, CS
Indian Institute of Technology, Kanpur
CPA: 7.71/10.0
2000 - 2002

Bachelor of Engineering, CS
G.B. Pant Engineering College
Percentage: 80% (honors)
1996 - 2000

Publications

Refereed Journal Articles


Refereed Conference Papers


**Refereed Workshop Papers**

9. **Analysis of Hypertext Isolation Techniques for Cross-site Scripting Prevention.**
   Mike Ter Louw, Prithvi Bisht and V.N. Venkatakrishnan. In *2nd Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, **Acceptance Rate = 14 / 45, 31%**.

**Invited Papers**


**Book Chapters**


**Presentations**

  - Dasient, Sunnyvale, USA, Mar 2011
  - AT&T Security Research Center, New York, USA, Feb 2011
- SRI International, Computer Science Lab Seminar, Menlo Park, USA, Dec 2010

- **NoTamper**: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications
  - Poster presentation, Computer Security Awareness Week (CSAW), NY, USA, Oct 2010
  - Paper presentation, CCS Conference, Chicago, USA, Oct 2010
  - Rump session presentation, Usenix Security Symposium, Washington, USA, Aug 2010

- **TAPS**: Automatically Preparing Safe SQL Queries
  - Paper presentation, FC Conference, Tenerife, Spain, Jan 2010
  - Poster presentation, CCS Conference, Chicago, USA, Oct 2010


- **Candid**: Preventing SQL Code Injection Attacks
  - Poster presentation, Midwest Security Workshop, Chicago, Oct 2007

**Additional Research Projects**

- Automated Discovery of Zero-day Attacks through Binary Program Analysis (with Phillip Porras and Vinod Yegneswaran - SRI International Lab, V.N. Venkatakrishnan - UIC)

- Reduction of Data Lifetime (with Kalpana Gondi, Praveen Venkatachari, A. Prasad Sistla and V.N. Venkatakrishnan - UIC)

**Professional Experience**
Doctoral Intern May, 2010 – Aug, 2010

SRI International, Computer Science Lab, Menlo Park, CA, USA
- Analyzed malicious Flash applications and prepared a categorized knowledge base.
- Studied existing literature on security analysis of binary applications.
- Proposed a novel scheme to find and prevent Zero-day attacks in binary applications.


University of Illinois at Chicago, Department of Computer Science, Chicago, IL, USA
- Studied security issues in Web applications.
- Proposed solutions for mitigation of top security threats including SQL-injection, Cross-site scripting and Cross-site request forgery.
- Proposed novel ways of finding high impact vulnerabilities in commercial web applications (online banking / shopping).
- Published research papers in top tier security conferences and participated in preparation of grant proposals to NSF.
- Peer reviewed academic conference papers and journal articles.
- Prototyped and evaluated several research ideas.

Teaching Assistant Aug, 2006 – Dec, 2006

University of Illinois at Chicago, Department of Computer Science, Chicago, IL, USA
- Mentored tutorial sessions for undergraduate class “Introduction to Programming”.
- Designed problems for weekly assignments and graded submissions.
- Graded assignments for graduate class “Introduction to Algorithms”.

Intel Corporation, Bangalore, India

- Designed and developed software for concept platforms of Intel.
- Proposed novel ideas that showcased hardware strength.
- Prototyped and prepared demos for higher management to get seed money for projects.
- Interfaced with Bluetooth stack vendors (Toshiba Japan, IVT China) as the sole technical contact.
- Published patentable ideas at http://www.ip.com.


Novell Inc., Bangalore, India

- Developed software to provide location independent secure access to the corporate information.

Research Student  Jul, 2000 – Feb, 2002

Indian Institute of Technology Kanpur, Department of Computer Science, Kanpur, India

- Developed an architecture-independent disassembler.
- Studied hands-on security (buffer overflows, trojan horses, packet sniffers).

Professional Activities

- Peer-reviewed research articles for:
Recent Advances in Intrusion Detection (RAID): 2008, 2010


Journal of Software Practice and Experience (JSPE): 2008

Web 2.0 Security and Privacy (W2SP): 2011

IET Information Security Journal: 2011

TECHNICAL REPORTS AND PATENTS

PATENTS

1. Automatically Generating Safe SQL Queries [Submitted to US Patent Office].

TECHNICAL REPORTS

2. Designing secure SDKs. [Published at www.ip.com].

3. A Bluetooth based WiFi Access Point Management [Published at www.ip.com].

HONORS AND DISTINCTIONS

- NoTamper project is among the 10 finalists in NYU-Polytechnic Computer Security Awareness Week competition 2010 (open to all students in the Continental USA).

- Research work featured in news

  - Oct 2009: http://www.uic.edu/htbin/cgiwrap/bin/uicnews/articledetail.cgi?id=13593
• Student travel grants
  • 18th Usenix Security Symposium, Montreal, 2009.
• All India Rank 52, Graduate Aptitude Test of Engineering, India, 2000 (99.06 percentile).

SECURITY RELEVANT COURSEWORK AT UIC

| Codes & Cryptography | Computer Systems Security |

SKILLS

Research: Problem identification, theoretical analysis, solution development and concept prototyping. Author academic literature and research proposals. Collaboration. Knowledge extraction and critical review of academic literature.

Software Engineering: Conception, design, implementation, optimization, debugging, documentation, support and growth of small to large, long-term software development projects

Computer Languages: C, C++, HTML, \LaTeX, Java, JavaScript, Perl, Shell script, PHP, SQL, VHDL