DISCOVERING AND MITIGATING SOFTWARE VULNERABILITIES
THROUGH LARGE-SCALE COLLABORATION

A Dissertation in
Information Sciences and Technology
by
Mingyi Zhao

© 2016 Mingyi Zhao

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2016
The dissertation of Mingyi Zhao was reviewed and approved* by the following:

Peng Liu  
Professor of Information Sciences and Technology  
Dissertation Advisor, Chair of Committee

Jens Grossklags  
Assistant Professor of Information Sciences and Technology

Xinyu Xing  
Assistant Professor of Information Sciences and Technology

Minghui Zhu  
Assistant Professor of Electrical Engineering

Andrea Tapia  
Associate Professor of Information Sciences and Technology  
Director of Graduate Programs, College of Information Sciences and Technology

*Signatures are on file in the Graduate School.
Abstract

In today’s rapidly digitizing society, people place their trust in a wide range of digital services and systems that deliver latest news, process financial transactions, store sensitive information, etc. However, this trust does not have a solid foundation, because software code that supports this digital world has security vulnerabilities. These vulnerabilities are the root causes of many security incidents, ranging from massive user information leakages, to the damage of industrial control systems.

Security professionals have been working hard to eliminate software vulnerabilities with mainly two approaches. The first approach is to protect software programs from unknown attacks at production run-time. The second approach is to continuously review and test a software product even after release, in order to find and fix vulnerabilities before attackers can exploit them. However, both approaches are limited by the capability and resource available to an individual program instance or a single organization.

This dissertation proposes and evaluates novel approaches for discovering and mitigating software vulnerabilities based on large-scale collaborations, which overcome the limitations of an individual party. We first look at how software program instances can autonomously collaborate with each other to defend against zero-day attacks. We propose a new security defense called collaborative run-time protection, which distributes the high overhead of security monitoring to a set of software instances, and coordinates the instances so that together they can cover the potential vulnerability space. Once an instance detects a zero-day vulnerability, it quickly shares a codeless patch with other instances so that the whole group becomes immune to the threat. We implement a prototype system called HeapCRP, which protects C/C++ programs from heap memory bugs. Our evaluation shows that HeapCRP is effective against real world vulnerabilities, with low cost.

We next explore the rapidly growing vulnerability discovery ecosystem emerging from the collaboration between organizations and the global white hat hacker community. Our study shows that this collaboration has yielded tens of thousands of vulnerabilities being discovered and fixed in the past several years, for a wide range of organizations, including many famous Internet companies, financial institutions, and even government
agencies. We further provide evidence showing that white hats’ scrutiny makes finding new vulnerabilities increasingly difficult, indicating improved security for these organizations. However, we also identified several frictions that raise the cost and risk of such collaboration. Therefore, we further propose new models and policies to scale the promise of this collaboration to a larger number of organizations and white hats. We first propose a new hacker allocation mechanism to reduce the number of duplicated reports. Next, we evaluate existing bug bounty policies that aim to control the quality of submission from hackers, and propose a new policy that has several advantages over existing ones.
# Table of Contents

List of Figures .................................................. x
List of Tables ................................................... xii
Acknowledgments .................................................. xiv

Chapter 1

Introduction .................................................. 1
1.1 Motivations ................................................. 1
1.2 Approaches Overview ....................................... 3
  1.2.1 Collaborative Run-time Protection against Zero-day Heap Memory Bugs ........................................ 3
  1.2.2 Analyzing and Improving the Vulnerability Discovery Ecosystem .................................................. 5
1.3 Summary of Contributions ................................... 7

Chapter 2

Related Work .................................................... 9
2.1 Run-time Protection against Memory Corruption Vulnerabilities ............................................... 9
  2.1.1 Run-time Protection against Memory Bugs ................................................. 9
    2.1.1.1 Signature-based defense ............................................... 9
    2.1.1.2 Context sensitive defense ............................................. 10
    2.1.1.3 Guard pages .......................................................... 10
  2.1.2 Software Diversity .......................................... 11
2.2 Fuzzing and Software Vulnerability Discovery Models .................................................... 11
2.3 Vulnerability Discovery Ecosystems ................................................. 12
  2.3.1 Software Vulnerability Datasets ............................................. 12
  2.3.2 Vulnerability Discoverers ................................................. 13
  2.3.3 Vulnerability Markets .................................................. 14
Chapter 3

Collaborative Run-time Protection against Zero-day Heap Memory Bugs

3.1 Introduction ......................................................... 15
3.2 Theory ................................................................. 18
    3.2.1 Model ......................................................... 18
        3.2.1.1 Defender Model ......................................... 18
        3.2.1.2 Attacker Model ......................................... 20
        3.2.1.3 Defender Strategy ....................................... 20
    3.2.2 Analysis ...................................................... 21
    3.2.3 Results ...................................................... 25
3.3 Run-time Protection against Heap Memory Vulnerabilities .......................... 27
    3.3.1 Overview ..................................................... 27
        3.3.1.1 Setup ..................................................... 27
        3.3.1.2 Collaborative detection .................................. 28
        3.3.1.3 Diagnosis and self-patch ................................. 29
        3.3.1.4 Scale up the defense ..................................... 29
    3.3.2 Protecting an Instance with HeapTherapy ................................ 29
        3.3.2.1 Calling context sensitive defense .......................... 29
        3.3.2.2 Run-time protection against heap buffer overflow ........... 30
    3.3.3 Run-time Protection against Use-after-free .................................. 30
        3.3.3.1 Detection .................................................. 31
        3.3.3.2 Diagnosis ................................................ 33
        3.3.3.3 Patching .................................................. 33
        3.3.3.4 Double free ................................................. 33
    3.3.4 Run-time Protection against Uninitialized Pointer Access .................... 34
        3.3.4.1 Detect ..................................................... 34
        3.3.4.2 Diagnosis ................................................ 35
        3.3.4.3 Patching .................................................. 35
3.4 Collaborative Detection of Zero-day Heap Memory Vulnerabilities ............... 35
    3.4.1 Ind-guard: .................................................... 36
    3.4.2 Coord-guard .................................................. 39
    3.4.3 Improvements ................................................ 40
        3.4.3.1 Handle hot CCIDs ......................................... 40
        3.4.3.2 Defend against coordination disruption attack .............. 41
        3.4.3.3 Defend against guard probing attack ........................ 41
        3.4.3.4 Defend against guard exhaustion attack  ................... 42
3.5 Patch Sharing ..................................................... 42
3.6 Evaluation ........................................................ 45
    3.6.1 Profiling Heap Buffer Allocation CCIDs .................................. 45
    3.6.2 Measuring the Overhead of HeapCRP .................................. 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.3</td>
<td>Use HeapCRP to Defend Real World Vulnerabilities</td>
<td>47</td>
</tr>
<tr>
<td>3.7</td>
<td>Discussion</td>
<td>47</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Limitations</td>
<td>47</td>
</tr>
<tr>
<td>3.7.1.1</td>
<td>Targeted attack</td>
<td>47</td>
</tr>
<tr>
<td>3.7.1.2</td>
<td>The free rider problem</td>
<td>50</td>
</tr>
<tr>
<td>3.7.1.3</td>
<td>Attacks against CRP</td>
<td>50</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Future Work</td>
<td>51</td>
</tr>
<tr>
<td>3.7.2.1</td>
<td>Fully decentralized run-time collaboration in a untrusted environment</td>
<td>51</td>
</tr>
<tr>
<td>3.7.2.2</td>
<td>Patch Endorsement</td>
<td>51</td>
</tr>
<tr>
<td>3.7.2.3</td>
<td>Optimize the utilization of defense resources</td>
<td>51</td>
</tr>
<tr>
<td>3.8</td>
<td>Conclusion</td>
<td>52</td>
</tr>
</tbody>
</table>

Chapter 4

Empirical Analysis and Modeling of Black-box Mutational Fuzzing 53

4.1 Introduction                                                          53
4.2 BFF and Data Collection                                               54
4.3 The Long-tail Distribution of Bugs                                     56
4.4 Modeling a Fuzzing Campaign                                            59
  4.4.1 A Stochastic Model                                                  59
  4.4.2 A Simulation Model                                                  61
4.5 Analysis Results                                                       62
  4.5.1 Expected Number of Bugs Discovered                                 62
  4.5.2 The Order of Bug Discovery                                          63
  4.5.3 Exploitability                                                     66
4.6 Discussion and Future Work                                             67
  4.6.1 Apply Our Analysis to Larger Datasets                              67
  4.6.2 Generalization to Other Vulnerability Discovery Approaches         68
4.7 Conclusion                                                             68

Chapter 5

An Empirical Study of Web Vulnerability Discovery Ecosystems             70

5.1 Introduction                                                          70
5.2 Methodology                                                           73
  5.2.1 Analysis Overview                                                  73
  5.2.2 Data Collection                                                    74
    5.2.2.1 Wooyun                                                           74
    5.2.2.2 HackerOne                                                       76
5.3 Results                                                               77
  5.3.1 Vulnerability Disclosure Trends                                    77
| 5.3.1.1 | Number of Vulnerabilities | 77 |
| 5.3.1.2 | Severity Levels | 78 |
| 5.3.1.3 | Vulnerability Types | 78 |
| 5.3.2 | The White Hat Community | 79 |
| 5.3.2.1 | Size and Growth | 80 |
| 5.3.2.2 | Productivity and Accuracy | 81 |
| 5.3.2.3 | Skills and Strategies | 83 |
| 5.3.2.4 | Disclosure and Learning | 84 |
| 5.3.3 | Organizations | 86 |
| 5.3.3.1 | Size and Growth | 87 |
| 5.3.3.2 | Vulnerability Distribution | 88 |
| 5.3.3.3 | Impact on Different Sectors | 88 |
| 5.3.3.4 | Response and Resolution | 89 |
| 5.3.3.5 | Monetary Rewards | 91 |
| 5.3.3.6 | Improvements to Organizations’ Web Security | 91 |
| 5.3.3.7 | Attracting Vulnerability Reports | 94 |
| 5.4 | Discussion | 96 |
| 5.4.1 | Importance of Disclosure | 96 |
| 5.4.2 | Potential Incentive Structure Evolution | 97 |
| 5.4.3 | Encouraging White Hat Participation | 98 |
| 5.4.4 | Stimulate Participation by Organizations | 98 |
| 5.5 | Conclusion | 99 |

Chapter 6

Modeling and Organizing Bug Bounty Programs

6.1 Introduction | 100
6.2 Challenges for Current Bug-Bounty Programs | 101
6.2.1 Duplicate Reports | 101
6.2.2 Invalid Reports | 102
6.3 Model 1 - Hacker Allocation | 105
6.3.1 Bug bounty utilities. | 105
6.3.2 Bug discovery model. | 106
6.3.3 Hacker diversity. | 107
6.3.4 Expected utilities. | 107
6.3.5 Bug bounty optimization. | 109
6.4 Model 2 - Quality Control Policies | 109
6.4.1 No-restriction Policy | 111
6.4.2 Accuracy-Threshold Policy | 111
6.4.3 Report-Rate Threshold Policy | 112
6.4.4 Validation-Reward Policy | 113
List of Figures

3.1 Overview of the theoretical model. ............................................. 18
3.2 Numerical results. ............................................................... 22
3.3 The architecture of HeapCRP. .................................................. 28
3.4 The buffer layout for use-after-free detection (after allocation). ........ 31
3.5 The buffer layout for use-after-free detection (after free). Every gray page is inaccessible. ......................................................... 31
3.6 Instrumented malloc and free for use-after-free detection. ............. 32
3.7 CCID frequency distribution of the Lynx amazon instance (log-log). . 37
3.8 Distribution of CCID counts in bins in the Lynx example. ............. 38
3.9 The flowchart for patch verification. ......................................... 44
3.10 Speed overhead due to HeapCRP. Each bar represents one experimental setting. The $M$ is 1,000, 3,000 and 5,000 for low, medium and high settings, respectively. The last bar of each group is based on the related work DieHarder [1]. ......................................................... 48
4.1 The fuzzing experiment workflow. ............................................. 55
4.2 Probability distributions of bugs triggered in fuzzing campaigns. The color of a bar represents the exploitability of the bug. The meaning of colors are: exploitable (red), probably_exploitable (blue), probably_not_exploitable (yellow), not_exploitable (green), unknown (grey). 57
4.3 Plots of expected number of bugs discovered ($E[D(i)]$) and actual number of bugs discovered ($D$) overtime. We have doubled the number of fuzzing runs in order to observe how two models predict. ............................. 63
4.4 Simulated number of unique bugs discovered in two programs by the black hat under different resource advantage $A$ of the software company. 65
5.1 Structure of a web vulnerability discovery ecosystem. .................. 74
5.2 Number of vulnerabilities reported per month on Wooyun and HackerOne (public data). ......................................................... 77
5.3 Trend of vulnerabilities with different severity on Wooyun. ............. 77
5.4 Trend of top 3 vulnerability types on Wooyun. .......................... 77
5.5 Number of reports for each vulnerability type on Wooyun (log scale).  
The compact visualization uses three colors to represent the percentage of three severity levels. Note that percentages are not affected by the log scale. Types in bold font also appear in OWASP’s 2013 top 10 [2]. .......................... 79
5.6 Number of new white hats and active white hats per month on Wooyun and HackerOne. ...................................................... 80
5.7 Contribution count of white hats on Wooyun and HackerOne (log-log).  
Vertical bars: thresholds for different productivity groups on Wooyun. . 80
5.8 Distribution of follower count for vulnerabilities with different severity  
(log-log). ................................................................................... 80
5.9 Scatter plot of white hats’ vulnerability type count and targeted organization count on Wooyun. Each dot represents a white hat who has found more than 5 vulnerabilities in total. The red triangle dots are white hats of the high productivity group defined in Section 5.3.2.2. ..................... 85
5.10 Count for new and total number of organizations with vulnerability reports (per month) on Wooyun. .................................................. 87
5.11 Number of vulnerabilities by organizations on Wooyun and HackerOne. 87
5.12 Number of vulnerability reports for non-IT businesses (left), and IT-sector businesses (right) on Wooyun. .................................................. 87
5.13 Boxplots for the time of three types of response activities based on  
publicly disclosed reports on HackerOne. ......................................... 90
5.14 Trend of vulnerability report count for three organizations on Wooyun. .............................................................. 92
5.15 Trend of vulnerability report count for three organizations on HackerOne. .............................................................. 92
5.16 Scatter plots of organizations’ white hat count and vulnerability count for  
Wooyun and HackerOne public programs (excluded Yahoo and Mail.ru as outliers). .............................................................. 94

6.1 Comparing the percentages of valid reports and duplicate reports across  
different bug-bounty programs and platforms. The percentage of duplicate reports for Facebook is unknown. The percentage of duplicate reports for Google is estimated based on [1]. The percentage of duplicates for HackerOne might be underestimated, because program administrators are not required to mark a report as duplicate. ..................... 103
6.2 Trend of report types for public programs and private programs on  
HackerOne. In our paper, we consider noise and informative reports as invalid reports. .............................................................. 104
6.3 Illustration of the vulnerability discovery model. .......................... 107
6.4 Expected utilities with different number of hackers. Parameters used:  
$V = 20$, $b = 5$, $\delta = 0.99$, $c_o = c_h = 1$, $\alpha = 2$. ..................... 108
List of Tables

3.1 List of Symbols .................................................. 19
3.2 Statistics of PCC coverage in the Lynx experiment. ............. 36
3.3 Heap buffer allocation CCID profiling results. Some results are not available (-) because either the dataset is too small, or the dataset is larger than what our machine can handle (for 471.omnetpp and 483.xalancbmk). Parameters $p$ and $b$ are calculated based on Equation 3.12 and Equation 3.14, respectively. We set $M = 1,000$ for detecting each type of bugs. So in total there are $3M$ defense resource consumptions for use-after-free, uninitialized pointer access and overflow. ........................................ 46
3.4 Results of effectiveness evaluation. Time to detect means the number of instances compromised before the first detection. Numbers in bracket are the standard deviation. ............................................. 49

4.1 Seed selection and fuzzing statistics of selected programs. # cands is the number of candidate seed files we collected from the Internet. # bbls is the number of unique basic blocks recorded when parsing the candidate seed files. % bbls is the percentage of basic blocks covered by the final seed set. ................................................................. 56
4.2 Estimates of $\alpha$. .................................................. 58
4.3 Notations. ............................................................. 59
4.4 Simulated bug discovery sequence based on the ffmpeg case ($\alpha = 2.21$). Bug ids in the bold font are unique to that sequence. ................. 64
4.5 Percentages of exploitability, and correlation between bug discovery probability (log) and exploitability. A correlation is significant if the p-value is less than 0.1. ......................................................... 66
5.1 Statistics for representative bug bounty platforms sorted by their start time. The two platforms studied in this paper are highlighted. Numbers were obtained from the cited references, or platforms’ websites directly in early August of 2015. The exact definitions of each metric for different platforms may vary. For example, some platforms count registered white hats (marked with *), while others such as HackerOne count white hats that have made at least one valid contribution. 72

5.2 Comparison across three white hat groups of different productivity levels on Wooyun. 82

5.3 Frequency of IT-business types within the group of publicly available bounty programs on HackerOne. Only tags with frequency greater than 3 are shown. 88

5.4 Percentages of different types of responses by organizations on Wooyun. 90

5.5 Trend test results for organizations on Wooyun and HackerOne. The confidence level is 0.95. 93

5.6 Results of regression analysis. There are 60 observations (HackerOne). 96

6.1 Percentiles of organizations’ response efficiency on HackerOne (in days) [3]. 102

6.2 List of Symbols 110
Acknowledgments

I feel incredibly fortunate to have been under Dr. Peng Liu’s guidance during my time at Penn State. I have learned tremendously from him: research, teaching, responsibility, and much more. What also impresses me is his way of mentoring. He works very hard to give us strong support and great freedom. As a young and foolish graduate student, I was apt at digging myself into a hole on numerous occasions. Yet he has always been very generous and supportive to help me. His insights, encouragements and guidances are one of the most valuable things I obtained during this journey. I sincerely wish that one day I could achieve his level.

I give special thanks to Dr. Jens Grossklags, who essentially serves as my second advisor. His generous support and great wisdom enable me to explore a brand new field and make connections with many interesting people. I truly appreciate his invaluable mentoring and advices for my research and career.

I also want to express my sincere gratitude to Dr. Xinyu Xing, Dr. Minghui Zhu and Dr. Anna Squicciarini who have served my dissertation committee and gave me valuable feedback and support. Also, I want to thank Dr. Squicciarini for her guidance during my teaching assistant experience.

I want to give many thanks to all of my research collaborators. Particularly, I want to thank Qiang Zeng, Aron Laszka, Jorge Lobo, and Thomas Maillart, for their great insights, selflessly support, and critical comments. I also want to thank Dr. John Chuang for giving me the chance to visit University of California, Berkeley, and to learn from researchers there.

I give my thank to all past and present members of the Cyber Security Lab. They are bright and kind. I am sure that they will achieve their goals in the future. Particularly, I thank (in alphabetical order) Kai Chen, Jun Dai, Pinyao Guo, Wenhui Hu, Heqing Huang, Chuangang Ren, Xiaoyan Sun, Jun Wang, Zhi Xin, Gaoyao Xiao, Jun Xu, Tao Zhang, Bin Zhao, and Chen Zhong for all the discussions, activities and help.

I appreciate all faculty, stuff and students at the College of IST at Penn State. Particularly, I want to thank Dr. Mary Beth Rosson, Dr. Guoray Cai, Dr. Dinghao Wu, Dr. Xiaolong Zhang, Dr. Sencun Zhu, Dr. Heng Xu, Stephanie Koons, Michelle Hill, and
Sue Kelleher. The members of IST provide a great environment for me to learn and grow. I want to also thank people at Intelligent Automation Inc. and people at HackerOne for giving me internship opportunities. Particularly, HackerOne has allowed me to utilize their resources, which are very valuable for this dissertation. Among many others, I especially want to thank Foster Geng, Kelcey Morton and Ning Wang at HackerOne.

I want to give special thank to my family members who have provide me tremendous support in the past 5 years. Particularly, I fell deeply fortunate that many of the most cherishable moments in these years were spent with my fiancée Yue Zhang. Her love and support are indispensable part of my life.

Finally, I want to express my gratitude for all people that have helped me in the journey towards the Doctoral degree. Please know that I will remember your kindness forever.

This dissertation study was supported by ARO W911NF-13-1-0421 (MURI), ARO W911NF-15-1-0576, NIETP CAE Cybersecurity Grant, and NSF CNS-1422594.
Dedication

I dedicate this dissertation to my mother, for giving and nurturing my life, and my fiancée Yue Zhang, for bringing me endless love, encouragement and inspiration.
Chapter 1  
Introduction

1.1 Motivations

Human society increasingly relies on the digital domain. Within this digital world, more complex software programs have been developed and deployed throughout the Internet. We trust that these software programs will faithfully deliver latest news, process financial transactions, store critical documents, and provide many other services.

However, the foundation of this trust is not solid, because the software programs that support the digital world have bugs. While many software bugs only cause minor glitches, some bugs could lead to a failed spacecraft [4], or the loss of human lives [5]. What’s worse, certain bugs, often referred to as security bugs, or software vulnerabilities, can be exploited by malicious actors for their own profit. Activists, cyber criminals and nation-state agents are constantly discovering, collecting and purchasing [6–8] software vulnerabilities to attack a wide range of targets, including banks, hospitals, government agencies [9], or even nuclear facilities [10]. Without effective approaches to mitigate software vulnerabilities, our trust on the digital world could potentially collapse, leading to a catastrophic loss of for human society.

Software vendors have been working hard to eliminate vulnerabilities prior to the release date, using methods like code review [11], testing [12], static analysis [13–15], etc. The adoption of a security development lifecycle (SDLC) can further reduce the number of bugs in a software product [16]. However, because of the increasing software complexity, and the time-to-market pressure, released software products will still have many hidden vulnerabilities. For example, despite a tremendous amount of investment spent in security by vendors, major web browsers are still compromised
almost every year at the Pwn2Own contest due to vulnerabilities [17–19]. Our research in Chapter 5 also shows that hundreds of vulnerabilities, including ones that may cause severe consequences, are found by external security researchers in almost every mostly visited website, each year [20].

Therefore, in addition to pre-release efforts of reducing vulnerabilities, software vendors also need methods to mitigate vulnerabilities that exist post-release. Based on convention, we will refer to these vulnerabilities as zero-day vulnerabilities, since once such a vulnerability is exploited, the defender will have zero days to patch the issue. There are mainly two classes of methods for mitigating zero-day vulnerabilities. The first class of methods aims to enhance the security of a software program in production run-time. For example, address space layout randomization (ASLR) [21, 22] and data execution prevention (DEP) [23] have been widely applied to significantly limit the impact of certain types of memory corruption vulnerabilities. More powerful approaches have also been proposed to detect various kinds of memory vulnerabilities, and generate defenses (e.g., input signatures) that prevent attackers from exploiting the same vulnerabilities in the future [24–29]. However, these defenses are usually associated with very high cost, which includes performance overhead, false positives, extra effort needed in development and deployment, etc [29]. As a result, most of today’s systems are not equipped with these protections.

The second class of methods attempts to discover zero-day vulnerabilities before attackers can exploit them. For example, internal security experts of software vendors can continuously review and test their software products even after release. In addition, a software vendor might periodically hire external penetration testing teams to search for vulnerabilities. Once a vulnerability has been discovered either by its internal security team or by external pentesters, the software vendor will try to fix it as soon as possible. While these approaches are effective, they are most likely unable to completely deplete the zero-day vulnerabilities arsenal of the attackers, not to mention all vulnerabilities hidden inside the software product, as the Pwn2Own example mentioned earlier shows. The limitation of this class of methods can be explained in multiple perspectives, which we will discuss in detail in Chapter 4. At a high level, vulnerability discovery has diminishing returns. So after a certain point, the resource required to discover the next vulnerability would be prohibitively high for the vendor. On the other hand, a dedicated attacker with comparable or more resources can potentially win the arms race. Second, vulnerability discovery is a very complex task, involving different levels of expertise,
creativity, and randomness. Therefore, even a teenager, who has much less resources compared to a giant company, can still find multiple severe flaws in one of the world’s most secure web browsers [19].

1.2 Approaches Overview

A commonality of both methods for mitigating zero-day vulnerabilities discussed above is that they primarily rely on a single entity, such as an instance of the software program, or a software vendor’s internal security team. The challenges we’ve discussed above are actually caused by the limitations of a single entity. For example, a software program has limited CPU cycles and memory space, so it can only do security related checks up to a certain amount, without significantly affecting its main functionalities. A security team’s expertise and perspectives of the problem could be constrained, and thus it is only able to find a subset of zero-day vulnerabilities. Overcoming the limitations of individuals requires collaboration among them, as how the human civilization develops through expanding the scale of collaboration within and between societies, and builds increasingly complex socio-technical systems. We think that effective collaboration is also the key to solve challenging security problems, and raise the following question: can the defender utilize the power of collaboration between multiple entities to mitigate zero-day vulnerabilities?

This dissertation gives a positive answer to this question, based on two lines of research. First, we propose an approach called collaborative run-time protection (CRP), that enables a set of software instances to collaborate with each other during run-time against unknown threats. Second, we study the rapidly growing vulnerability discovery ecosystem that enables various kinds of organizations, including IT companies, financial institutions, governments, etc., to collaborate with the global white hat hacker community, using empirical analysis and theoretical modeling.

1.2.1 Collaborative Run-time Protection against Zero-day Heap Memory Bugs

The first direction of mitigating zero-day vulnerabilities is to apply run-time protection for programs. An ideal run-time protection should not only be able to detect zero-day vulnerabilities, but also generate signatures or patches which can be used to prevent
the same attacks in the future. However, a major challenge of applying such run-time protection is the high cost, particularly for memory vulnerabilities [29].

In this dissertation, we propose a new approach called collaborative run-time protection (CRP) in Chapter 3. CRP can significantly reduce the overhead of run-time protection for a single program instance, while still providing strong defense against zero-day vulnerabilities at the collective level. The key idea of CRP is to utilize the power of collaboration between individual instances of the same software program (referred to as a software species). The collaboration happens in two phases, detection and patching. In the detection phase, CRP lets individual instances of the same software program guard different subsets of the vulnerability space. Each instance only needs to spend a small amount of resources, yet overall the software species can cover a large part of or even the whole vulnerability space. Therefore, the probability that at least one instance detects the zero-day after several attacks is high.

Once a zero-day is detected by one instance, CRP moves into the patch phase. The instance that has detected the zero-day will generate a codeless patch [30], and then share that patch with other instances. Therefore, the whole species can autonomously evolve to be immune against the vulnerability. This is similar to how a biological species survives in the hostile world with new and unknown threats (e.g., predator, virus) constantly emerging. While a specific individual might be weak against a threat, the whole species has gene diversity that offers a wide range of capabilities, enabling some individuals to survive. Then through gene exchange more individuals will possess the life-saving genes, and the whole species becomes more resistant against the same threat overtime.

We study the feasibility and effectiveness of this idea first by theoretical analysis and modeling. Then, we design and implement a prototype system called HeapCRP, which can be used to protect C/C++ programs against various heap memory vulnerabilities, including buffer overflow, use-after-free, double-free, and uninitialized pointer access. Our evaluation shows that HeapCRP can quickly detect real-world vulnerabilities. Based on the SPEC 2006 benchmark, we further show that the overhead of HeapCRP when operating at an effective level is very low.
1.2.2 Analyzing and Improving the Vulnerability Discovery Ecosystem

The next direction of mitigating zero-day vulnerabilities is to discover and fix them before they can be exploited by the attackers. As we’ve discussed earlier, the ability of finding vulnerabilities by a single team is inherently limited. To overcome the limitations, in recent years, increasingly more organizations, from major IT companies such as Facebook and Google, to government agencies such as the U.S. Department of Defense [31], have started to collaborate with the global hacker community to find and fix security flaws through bug bounty programs [20,32,33]. While such large-scale collaboration seems to be a promising approach for security, its benefits and costs are still not well-understood and even debated [34,35], preventing more organizations to join the ecosystem and work with white hat hackers. In addition, the collaboration currently has various kinds of frictions, that raise the cost of participation, and reduce the effectiveness of the approach.

Therefore, the second half of this dissertation (Chapters 4-6) focuses on understanding, analyzing, and improving this vulnerability discovery ecosystem. We first want to analyze the characteristics and limitations of vulnerability discovery by a single entity. Furthermore, we also need to abstract the process and its characteristics into a vulnerability discovery model, which serves as a foundation and a computational tool for designing and implementing new collaboration mechanisms. While plenty of research has been done to empirically study [20,33,36–39] and model vulnerability disclosure data [40,41], very few works have actually studied vulnerability discovery data, which contain much finer-grained information about the discovery process, such as what inputs are used in each run, how many discoveries are invalid or duplicate, etc. Therefore, existing literature provides insufficient knowledge for us to understand the vulnerability discovery process and model it in detail.

As a first step towards filling this research gap, we conduct an empirical study of fuzzing, arguably the most widely used vulnerability discovery method [42–44], in Chapter 4. We collect the dataset using carefully designed experiments, so that we can clearly observe details of the discovery process. We then apply various approaches to analyze the data to observe the following characteristics of vulnerability discovery. First, the vulnerability discovery process by one entity has diminishing returns, because the discovery probability distribution is hypothesized to follow a power law distribution. We then show that the discovery process is less ordered over time, and therefore, an attacker
even with much less resources compared to the defender, can still find unique zero-day vulnerabilities. Based on the insights, we further propose stochastic vulnerability discovery models that fit well to the data.

Next, in Chapter 5, we conduct an empirical study of Web vulnerability discovery ecosystems, in order to understand their characteristics, trajectories and impact [20]. We collect data from two of the largest bug bounty platforms, HackerOne and Wooyun, and discover that there is a large-scale Internet-wide collaboration between tens of thousands of white hat hackers and organizations. This collaboration has exposed more than 70,000 vulnerabilities covering almost all types of vulnerabilities, including many highly severe ones. Based on statistical analysis, we further provide evidence that this joint effort could lead to improved security. However, we also find that the impact of this ecosystem is not always positive. For example, roughly about 80% of all submitted reports on a typical bug bounty platform are actually invalid, meaning that both white hats and organizations have spent a lot of unnecessary effort in the collaboration. Another finding is that for Wooyun, roughly 23% of all disclosed reports are not properly handled, directly exposing the vulnerabilities to malicious attackers! There are many other analyses and results, which we will discuss in detail in Chapter 5. The general takeaway is that this collaboration between white hat hackers and organizations can make significant contributions to the mitigation of zero-day vulnerabilities. However, challenges also exist for the vulnerability discovery ecosystem to sustain\(^1\) and grow.

In Chapter 6, the final part of this dissertation, we design and evaluate policies and mechanisms that can alleviate the frictions of collaboration between organizations and white hats, in order to improve its effectiveness and efficiency. The first problem we tackle is duplicated reports, which constitutes 36.23% of submissions on BugCrowd [46], one of the leading bug bounty platforms. A duplicated report contains a valid vulnerability discovery, which could be the result of hours of research by a hacker. However, because someone else has submitted the issue earlier, the effort of the second hacker will not be compensated. The risk of receiving no reward for their effort can significantly hurt the motivation of white hats to contribute. In addition, processing duplicate reports also takes time and effort from the organization side. We approach this problem from a computational perspective. Based on the vulnerability discovery model developed in Chapter 4, we build a probabilistic model for white hat hackers. We then show that by

\(^1\)On 2016/7/20, the Wooyun platform has been suspended by the government [45]. At the time of writing this dissertation, the website still hasn’t been resumed.
properly allocating hackers at different time steps, the overall utilities of both hackers and organizations can be significantly improved, compared with baseline approaches.

The second problem we consider is invalid submissions. Bug bounty programs receive a large number of invalid reports because of an incentive misalignment between organizations and hackers. While organizations only want valid reports, many hackers have small incentives to validate their findings. Instead, for them the best strategy is to submit as many reports as possible, hoping that some of them will be qualified for rewards. Addressing this issue requires designing and enforcing bug bounty policies. In Chapter 6, we summarize our efforts by theoretically evaluating existing policies, and designing a new policy [47].

1.3 Summary of Contributions

This dissertation makes the following contributions.

- We propose a novel idea called collaborative run-time protection to enhance software programs against zero-day attacks with low cost. We build a theoretical model for this protection and obtain several analytical results regarding its effectiveness and limitations. We then design and implement a collaborative run-time protection system called HeapCRP, which can probabilistically detect multiple types of zero-day heap memory bugs for C/C++ programs. Our evaluation shows that HeapCRP is effective against multiple real world vulnerabilities, and has low overhead based on the SPEC 2006 benchmark.

- We conduct fuzzing experiments to empirically study the vulnerability discovery process. We show that the fuzzing process can be modeled as a non-homogeneous Poisson process with the rates of individual bugs following a power law distribution. We then calculate the expected outcome of a fuzzing campaign. We further show that once the vulnerability discovery enters the long-tail, there will be significantly diminishing returns, and less order in the bug arrival. These effects pose challenges for the software companies that try to eliminate vulnerabilities before the black hats, and call for collaboration with white hats. We have also shared our experiment environment and datasets with the research community.

- We conduct the first empirical analysis of two major vulnerability discovery ecosystems. Our analysis provides insights of the white hat hacker community from
various aspects, including their participation, discovery strategies, productivity, and learning. We also quantitatively analyze participating organizations from several dimensions, including the vulnerability trends, the coverage of different business sectors, the response and resolve behaviors, and reward structures. Our research provides evidence showing that this collaboration between organizations and hackers has made significant impact on security. Results of this research have been covered in multiple media reports [48, 49] and presented in cross-disciplinary venues such as FTC Privacy Con 2016 and TEDxPSU. In addition, we fully published our datasets and code to support reproducible research.

• We developed theoretical models for analyzing and improving the efficiency and effectiveness of bug bounty programs. Our model defines the utilities of hackers and organizations, and also captures the vulnerability discovery process with an emphasis on the diversity of the hackers’ capability in identifying vulnerabilities. We also evaluate the strengths and weaknesses of multiple bug bounty policies, and propose a new policy enabling different white hats to exert validation effort at their individual optimal levels.
Chapter 2  
Related Work

2.1 Run-time Protection against Memory Corruption Vulnerabilities

We first examine existing run-time protections against memory bugs that are related to HeapCRP. Then we discuss related work from the closely related field of software diversity.

2.1.1 Run-time Protection against Memory Bugs

Due to extensive research in this direction, we will only cover three areas that are the most relevant to HeapCRP. For a more comprehensive survey of run-time protection against memory bugs, please refer to recent surveys [29, 50].

2.1.1.1 Signature-based defense

HeapCRP is similar to signature-based defense against zero-day attacks [24–27, 51, 52]. Such protections are able to detect attacks, generate a signature, and share that signature with other instances, so that instances with the signature become immune to the same attack in the future. However, these protections mostly focus on detecting code injection attacks by worms, which are less common nowadays. In addition, some of the protections rely on ASLR as the detection component to trigger crash diagnosis and signature generation [25, 27]. But ASLR is coarse-grained and thus the detection capability is very limited. HeapCRP applies more generic methods to detect a wider range of bugs. The focus of HeapCRP is the vulnerability, not attacks that exploits it. Therefore, compared
with signature-based approaches that suffer from false positives and polymorphic attacks, HeapCRP, based on the codeless patch [30], has zero false positives and is resilient against polymorphic attacks. In addition, with enough resources, HeapCRP can cover the whole vulnerability space. This however can hardly be achieved by signature-based defenses.

2.1.1.2 Context sensitive defense

HeapCRP is built on HeapTherapy [30], a context sensitive defense. The value of calling context beyond debugging was recognized early. For example, region-based heap allocation tags heap objects with allocation calling context information [53]. Calling context was recently used to generating context sensitive defenses [26–28, 54, 55]. However, they commonly use costly call stack walking. HeapTherapy is the first work that employs the calling context encoding technique to generate context sensitive defense [30]. It largely reduces the overhead compared to using other calling context retrieval techniques such as stack walking [56–58]. Through the calling context encoding technique HeapTherapy is able to represent the characteristics of buffers being exploited with one integer and to identify vulnerable buffers through integer comparison.

2.1.1.3 Guard pages

HeapCRP primarily relies on page protection to detect heap memory bugs. Electric Fence utilizes inaccessible guard page to detect overflow and use-after-free [59]. The main benefit of using guard page is that a zero-day bug can be discovered without the need of checking every relevant operation. However, the full enhancement for all heap buffers incurs prohibitively high overhead, so these tools are not suitable for run-time protection. Previous research has proposed an approach to significantly reduce the physical and virtual memory consumption when using guard page to detect use-after-free [60], based on a multi-to-one mapping from virtual address to physical address, and automatic pool allocation. However, this approach still incurs high overhead for allocation-intensive programs. HeapCRP, on the other hand, only applies guard pages to a subset of buffers, thus avoiding the high overhead.
2.1.2 Software Diversity

A related field of collaborative run-time protection is software diversity, whose goal is to break the software mono-culture by transforming a program into semantically equivalent but syntactically different instances [61]. Since precise knowledge of the target program is required for many kinds of attacks, diversified instances will significantly reduce the chance of a successful exploit. The idea of software diversity has been widely applied in many levels of the stacks, including instruction level [62], basic block level [63], data structure level [64], process level [65], etc.

CRP utilizes the idea of diversification as well. However, CRP only diversifies the protection layer of a program, rather than the implementation. Therefore, it avoids some obstacles in software distribution, error reporting, patching, etc., associated with other software diversity approaches [61]. Another major difference between software diversity and CRP is that the former does not consider detecting and diagnosing the vulnerability. Thus, the attacker can still repetitively cause denial of service for most software diversity approaches. On the other hand, CRP learns can learn from a crash and generate a patch to prevent future attacks.

2.2 Fuzzing and Software Vulnerability Discovery Models

Black-box mutational fuzzing has been widely used in software vulnerability discovery since the early 1990s, when Miller et al. surprisingly found out that random inputs crash 25% - 33% of Unix utilities [42]. Since then, black-box mutational fuzzing has been used to find numerous real world bugs and security vulnerabilities in various programs [66–68]. Compared with other forms of more sophisticated fuzzing approaches, such as generational fuzzing [43], whitebox fuzzing [69], taint-based fuzzing [70], etc., black-box mutational fuzzing is simpler and easier to use, but is usually inferior in terms of code coverage.

More recently, various methods were proposed to improve the effectiveness of black-box mutational fuzzing. Householder and Foote studied the problem of selecting seeds and fuzzing ratio using BFF [67]. The basic idea is to have more selection weight on parameter values that yield higher crash density in the past. Woo et al. considered a similar scheduling problem in which the target program of each fuzzing run is also
selected on the fly [71]. They designed several online scheduling algorithms and showed an average of $1.5 \times$ improvement over the one used in BFF. Rebert et al. designed and evaluated 6 seed selection algorithms [72]. The motivation of our work is different from but complementary to these works. Instead of optimizing the fuzzing process, we want to understand the fuzzing process better, based on empirical analysis and theoretical modeling.

The stochastic model built in Chapter 4 is derived from software reliability models [73–76], since fuzzing or vulnerability discovery in general is one particular approach to improve software reliability. However, different from existing work, we assume that the arrival rates of the individual bugs follow a power law distribution. This enables us to obtain similar observations on the difficulty of software reliability growth [73, 74], or in other words, diminishing returns of software vulnerability discovery. In addition, we also uniquely use the power law-based stochastic model to explain why other parties (e.g. black hats) seem always being able to discover unique vulnerabilities in Section 4.5.2. We further analyze the exploitability of bugs in Section 4.5.3, which is missing from software reliability growth models.

Long-tail distributions have been observed and discussed in various cyber security domains recently. Allodi showed that vulnerability exploitation in several common programs may follow a power law distribution [77], which can be used for vulnerability prioritization. Maillart and Sornette showed that the sizes of personal identity theft follow power law distribution [78]. Finally, Edwards et al. found that data breach size is log-normally distributed while the daily frequency of breaches can be described by a negative binomial distribution [79]. These results can be used to predict data breaches and their associated cost.

### 2.3 Vulnerability Discovery Ecosystems

#### 2.3.1 Software Vulnerability Datasets

Previous work has studied various software vulnerability datasets to understand vulnerability discovery, patching and exploitation. This research is relevant for the debate on whether vulnerability disclosure programs are beneficial to society [80]. That is, if the number of potential vulnerabilities is large with respect to the effort of white hats, and vulnerabilities are found in no particular order, then black hats could frequently
discover and exploit vulnerabilities that are not covered by white hats’ contributions; thereby questioning their effectiveness. On the one hand, Rescorla studied the ICAT dataset of 1,675 vulnerabilities and found very weak or no evidence of vulnerability depletion. He thus suggested that the vulnerability discovery efforts might not provide much social benefit [36]. On the other hand, this conclusion is challenged by Ozment and Schechter, who showed that the pool of vulnerabilities in the foundational code of OpenBSD is being depleted with strong statistical evidence [37, 38]. Ozment also found that vulnerability rediscovery is common in the OpenBSD vulnerability discovery history [37]. Therefore, they gave the opposite conclusion, i.e., vulnerability hunting by white hats is socially beneficial. More recently, Shahzad et al. [39] conducted a large-scale study of the evolution of the vulnerability life cycle using a combined dataset of NVD, OSVDB and FVDB. Their study provided three positive signs for increasing software security: (1) monthly vulnerability disclosures are decreasing since 2008, (2) exploitation difficulty of the identified vulnerabilities is increasing, and (3) software companies have become more agile in responding to discovered vulnerabilities. In another study, Frei et al. studied a security ecosystem including discoverers, vulnerability markets, criminals, vendors, security information providers and the public, based on 27,000 publicly disclosed vulnerabilities [33]. They focus on vulnerability exploits and patching of native software, while we study the ecosystem around the discovery of web vulnerabilities, and our main focus are the behaviors and dynamics of white hats and organizations that compose such ecosystems.

### 2.3.2 Vulnerability Discoverers

Most of the existing research on software security focuses on vulnerabilities, affected software products or vulnerability discovery tools. More recently, researchers started to pay attention to the humans who make vulnerability discoveries. Edmundson et al. conducted a code review experiment for a small web application with 30 subjects [11]. One of their findings is that none of the participants was able to find all 7 Web vulnerabilities embedded in the test code, but a random sample of half of the participants could cover all vulnerabilities with a probability of about 95%, indicating that a sufficiently large group of white hats is required for finding vulnerabilities effectively. This is consistent with our analysis in Section 5.3.2.2 and Section 5.3.3.7. However, the code review process they focused on is mainly conducted inside an organization with source code
available; while the vulnerability hunting focused on in this paper is conducted outside an organization. Finifter et al. provided contribution and payment statistics of participants in Google Chrome VRP and Mozilla Firefox VRP [32], and suggested that VRPs are more cost-effective compared to hiring full-time security researchers. Previous work has also reported that many discoverers primarily rely on their expertise and insights, and limited types of tools such as fuzzers and debuggers, rather than sophisticated automated vulnerability discovery tools [7, 81]. Zhao et al. conducted an initial exploratory study of white hats on Wooyun [82] and uncovered the diversity of white hat behaviors on productivity, vulnerability type specialization, and discovery transitions.

### 2.3.3 Vulnerability Markets

Böhme offers a terminology for organizational principles of vulnerability markets by comparing bug challenges, vulnerability brokers, exploit derivatives and cyber-insurance [83]. Algarni and Malaiya analyzed data of several existing vulnerability markets and showed that the black market offers higher prices for zero-day vulnerabilities, and government agencies make up a significant portion of the buyers [7]. Ozment proposed a vulnerability auction mechanism that allows a software company to measure its software quality based on the current bounty level, and to conduct vulnerability discovery at an acceptable cost [84]. This auction model can potentially be incorporated into today’s vulnerability discovery ecosystems. A panel discussion at the New Security Paradigms Workshop examined ethics and implications for vulnerability markets [80]. Finally, Kannan and Telang showed that unregulated vulnerability markets almost always perform worse than regulated ones, or even no market at all [85]. They also found that it is socially beneficial to offer rewards for benign vulnerability discoverers.
Chapter 3  |  Collaborative Run-time Protection against Zero-day Heap Memory Bugs

3.1 Introduction

As software programs get more complex and more widely deployed, the risk of zero-day bugs also increases. These bugs cause issues from annoying software errors to failures of mission critical systems. Moreover, zero-day bugs might be discovered and exploited by malicious hackers, causing large-scale panic and damage, as the recent Heartbleed [86] and Shellshock cases shown. Software companies are investing more resources to eliminate bugs through internal code review and testing, or harvest bug reports from external security researchers via bug bounty programs [20]. However, because of the increasing software complexity, these effort can only eliminate a subset of all bugs hidden inside the system.

Therefore, in addition to the huge amount of effort spent on testing before releasing programs, it is also desirable to give programs run-time protection that can detect and react to zero-day bugs, even without human intervention, which might take considerable amount of time. For memory related bugs found in C/C++ programs, many run-time protections have been proposed to enhance a single software instance [28–30, 87–92]. However, a common challenge for these run-time protections is high performance overhead [29]. For example, Exterminator can probabilistically detect and correct multiple types of heap memory bugs [28]. However, the average speed overhead of Exterminator
for allocation-intensive programs is 81.2%, and the size of the heap at least needs to be doubled. On the other hand, WIT achieves a very low overhead, but it cannot handle use-after-free and out-of-bound reads, and has issues to support modularity [29, 87]. Because of these constraints, most proposed run-time protections are not (yet) deployed widely. Commonly deployed run-time protections, such as ASLR and DEP, can only offer limited protection.

One idea to overcome the performance constraint is to let a set of instances share the overhead introduced by security protection. A similar idea has been proposed in a seminal paper [93] for cooperative testing, where each deployed software instance randomly executes a subset of run-time checks, and reports the statistics to the developer. Thus, the instrumentation cost is shared among all instances, while the developer can still statistically find zero-day bugs.

For security bugs, passively collecting data from remote instances is often not enough, because the time required to triage the bug and distribute a patch might be too long to defeat the attackers. Therefore, we also need each instance to have the ability of generating a signature or a patch that can at least temporarily fix the vulnerability, and can share the patch with other instance. Several related defenses have been proposed around 2005 [24–27], when the containment of fast spreading worms over the Internet is an urgent need. Such protections are able to detect attacks, generate a signature, and share that signature with other instances, so that instances with the signature become immune to the same attack in the future. However, these protections mostly focuses on detecting code injection attacks by worms, which are less common nowadays. In addition, some of the protections rely on ASLR as the detection component to trigger crash diagnosis and signature generation [25, 27]. But ASLR is coarse-grained and thus the detection capability is very limited. In addition, signature-based approaches suffer from false positives, and are vulnerable to polymorphic attacks.

In this paper, we propose the idea of collaborative run-time protection (CRP). The goal of CRP is to achieve high mitigation capability of zero-day bugs at the collective level, while only imposing a very low overhead onto each individual instance. We build a prototype system called HeapCRP to demonstrate and evaluate the idea of CRP. HeapCRP protects C/C++ programs against a variety of heap memory bugs, including use-after-free, double-free, buffer over-read and over-write, and uninitialized pointer access. More specifically, HeapCRP has the following three key components:

- **Model the vulnerability space:** We view all heap buffers allocated during execu-
tion as the *heap vulnerability space*. Then, we identify the same heap buffer across different instances, based on the allocation calling context encoding value (CCID) of that buffer [30].

**Divide and conquer the vulnerability space:** Each instance randomly protects a small portion of the vulnerability space. Therefore, the performance overhead is shared among all instances, and is very low for an individual instance. However, the whole software species can now cover a large area of the vulnerability space, thus leading to high probability of detecting zero-day bugs at the collective level. We further show that with enough instances or through detection coordination, even the *whole vulnerability space can be covered.*

**Patch Sharing:** Once a bug is detected by an instance, the instance will generate a patch and share it with others. Therefore, the whole software species can quickly evolve against new threats. The patch used by HeapCRP is derived from the codeless patch proposed in HeapTherapy [30]. A codeless patch only contains multiple integers, and does not require any change to the code. The only side-effect of a codeless patch is extra overhead. Therefore, compared with code-based patches, codeless patch requires less trust to be deployed. We design patch sharing and verification mechanisms in Section 3.5. Compared with signature-based defenses, codeless patch has no false positive (i.e., will not block legitimate inputs), and is also resilient against polymorphic attacks.

The key contributions of this paper are:

- We conduct a theoretical analysis of the collaborative run-time protection idea, and provide multiple results regarding its performance and limitations.
- We propose, implement, and evaluate multiple approaches for divide and conquer the heap vulnerability collaboratively.
- We implement a collaborative run-time detection system called HeapCRP, which can probabilistically detect multiple types of zero-day heap memory bugs. Our evaluation shows that HeapCRP is effective against several real vulnerabilities. The overhead of HeapCRP under medium level of protection is 12% on average, based on the SPEC 2006 benchmark.
3.2 Theory

In this section, we build a theoretical model for CRP, with two main purposes. First, we want to theoretically analyze the benefits and costs of CRP. Second, we will use the model to guide design and implementation in later sections.

3.2.1 Model

Figure 3.1 gives an overview of the model, which has three main components: a defender model, an attacker model, and defense schemes. All symbols can be found in Table 3.1.

3.2.1.1 Defender Model

Suppose a software program is deployed in \( N \) instances. For example, a web service provider runs hundreds of \( \text{Nginx} \) web servers in its data center. Each software instance
Table 3.1: List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>number of deployed instances of a software program</td>
</tr>
<tr>
<td>$C$</td>
<td>number of components created during an instance’s execution</td>
</tr>
<tr>
<td>$g_i$</td>
<td>cost of guarding component $i$</td>
</tr>
<tr>
<td>$M$</td>
<td>defense resource quota</td>
</tr>
<tr>
<td>$X$</td>
<td>number of successfully exploited instances by the attacker $i$</td>
</tr>
<tr>
<td>$p$</td>
<td>the probability of guarding a component by an instance</td>
</tr>
<tr>
<td>$F$</td>
<td>percentage of full-coverage instances</td>
</tr>
</tbody>
</table>

creates $C$ components during an execution. An example of components would be all heap buffers allocated during the program execution. For simplicity, we assume that all instances create the identical set of components in the following theoretical analysis.

The software program could have many kinds of vulnerabilities. In this work, we focus on vulnerabilities that enables an attacker to exploit individual components. For example, (contiguous) heap over-read is a type of vulnerability that enables an attacker to read data beyond a buffer. This kind of heap memory vulnerability gains strong attention, after the Heartbleed bug of OpenSSL was disclosed on the Internet. However, even tough the Heartbleed bug is patched, it is possible that another buffer is vulnerable to the same kind of attack [94]. Therefore, we define the concept of vulnerability space as follows:

**Definition 3.2.1.** The vulnerability space of an instance is the set of all potentially vulnerable components generated during run-time.

Without additional information about which components can be exploited, the defender needs to conservatively assume that all components belong to the vulnerability space. In the vulnerability space, we assume that there is one *vulnerable component* that can be exploited by the attacker. The defender does not know which component is vulnerable, so the defender needs to guard some or all of the components during run-time to catch the attack. Guarding a component $i$ is associated with a *guard cost* $g_i$. The guard cost is an abstraction of actual resource consumption, such as execution time and memory space. As a starting point, we assume that the cost is 1 for each component. Because the main goal of the user is to use the software program, rather than secure it, the defender can only spend up to $M$ resources to monitor components for each instance. We refer to $M$ as the *defense resource quota*. Of course, it is possible that some users prefer security over usability. In some special cases, such as a honeypot, the instance could be dedicated
to detecting zero-day bugs, so $M = \infty$. We will discuss this special case later.

### 3.2.1.2 Attacker Model

We assume that the attacker is the only one knowing which component is vulnerable, through vulnerability discovery methods such as fuzzing. In other words, the attacker knows a zero-day bug of the software program. It is true that the defender can also run fuzzing. However, a simulated study of a previous research shows that even if the defender has significantly more resources for fuzzing to find the bug, the attacker is still expected to find unique bugs, because of the randomness in vulnerability discovery, and the skewed distribution of discovery probabilities for different bugs [95].

The attacker then exploits this bug on all instances one by one. This sequential attack assumption captures the limits of an attacker’s capability in reconnaissance and exploitation. We assume that all instances are equally profitable for the attacker, so the attacker uniformly randomly picks the next one to exploit from all instances that have not been attacked. If the attacker touches an instance which happens to guard the vulnerable component, the zero-day vulnerability is detected by the defender. The defender can then generate a patch and spread the patch to all instances so that the bug can no longer be exploited. Therefore, the goal of the attacker is to exploit as many instances as possible before being detected. We define the number of successfully exploited instances as $X$.

### 3.2.1.3 Defender Strategy

The goal of the defender is to minimize $X$, or its expectation $E[X]$. To achieve this goal, the defender needs to decide which components to guard by each instance, under the constraint of $M$. Without information about which components are more likely to be vulnerable, the defender should make the following assumption.

**Assumption 3.2.1.** *All components are equally likely to be vulnerable.*

Later, We will discuss how this assumption can be relaxed, based on some heuristics. With this assumption, the *optimal outcome for the defender is to let each component receive the same amount of defense resources among all instances.*

Since it is very likely that a single instance cannot guard all components, multiple instances need to *collaborate* by guarding different part of the vulnerability space. A simple and implicit collaboration scheme is to let each instance randomly pick $M$ components to guard during execution. In other words, a component has probability $p = M/C$
to be covered by an instance. We will refer to this scheme as \textit{ind-guard}. Under this scheme, the defender can expect to have a diverse set of instances, who are able to cover a large set of components collectively.

Ind-guard is easy to implement and deploy, although there are several implementation challenges we encountered when building a prototype in Section 3.4). The main advantage of ind-guard is that there is no need for coordination between instances. However, ind-guard cannot achieve the optimal outcome, in which components are guarded with equal defense resources. Rather, some components are over-guarded, while other components could be under-guarded, or not guarded at all. We will quantitatively analyze the limitations of ind-guard in the following subsections.

To overcome the limitations of ind-guard, we propose another scheme called \textit{coord-guard}. The basic idea is that the defender can partition the whole vulnerability space into $C/M$ subsets. Then, each instance will guard a subset of components. If $N \geq C/M$, the defender knows that every component is at least covered by one instance. In addition, components are almost equally guarded, thus achieving the optimal outcome. On the other hand, coordination among instances requires more effort in implementation and deployment.

In the remainder of this section, we will analyze the two schemes. Then in Section 3.4, we will actually implement both schemes and their variations for real programs.

### 3.2.2 Analysis

Given the model, we try to answer the following two main research questions based on theoretical analysis.

- **Q1**: Can CRP quickly detect the zero-day bug with small amount of defense resource consumption on each instance?

- **Q2**: How much more effective is the coord-guard compared to ind-guard?

To answer these questions, we will first calculate the expected number of compromised instances, $E[X]$. The goal of the defender is to minimize this number. For ind-guard, it is easy to see that the probability of detecting the attack by one instance is $p = M/C$. Also, detections by individual instances are independent. Therefore, the probability of having $X$ successful attacks at the end of the game (i.e., detected the vulnerability or all instances are compromised) is:
Figure 3.2: Numerical results.

\[ P[X = x] = \begin{cases} (1 - p)^x p, & \text{if } x < N \\ (1 - p)^N, & \text{if } x = N \\ 0, & \text{if } x > N \end{cases} \]  (3.1)
The cumulative distribution function of $X$ is:

$$\Pr[X \leq x] = \begin{cases} 
1 - (1 - p)^x + 1, & \text{if } x < N \\
1, & \text{if } x \geq N.
\end{cases} \tag{3.2}$$

When $x < N$, $\Pr[X \leq x]$ is the probability of detecting the zero-day bug by no more than $x$ instances. The expected number of successful attacks is

$$E[X] = \left( \sum_{i=1}^{N-1} i(1-p)^i \right) + (1-p)^NN = \frac{(1-p)(1-(1-p)^N)}{p}. \tag{3.3}$$

We also want to know the number of unique components covered by $N$ instances with $M$ defense resource quota for each. We define this number as $Y$, and our goal is to calculate $E[Y]$. To solve this problem, we can consider the following process. Instance 1 first randomly make $M$ draws from $C$ components without replacement. Then instance 2 repeat the same process, and so on. We define $Y_{i,j}$ as the number of unique components covered by first $i-1$ instances, and the first $j$ guards of instance $i$. It is easy to see that $Y_{1,1} = 1$, and what we want is $Y = Y_{N,M}$. We know that within one instance’s process, we have:

$$E[Y_{i,j+1}|Y_{i,j}] = Y_{i,j} + \frac{C - Y_{i,j}}{C - j} \tag{3.4}$$

$$E[Y_{i,j+1}] = \frac{C - j - 1}{C - j}E[Y_{i,j}] + \frac{C}{C - j}.$$

For the transition from instance $i$ to instance $i + 1$, we have:

$$E[Y_{i+1,1}|Y_{i,M}] = Y_{i,M} + \frac{C - Y_{i,M}}{C} \tag{3.5}$$

$$E[Y_{i+1,1}] = \frac{C - 1}{C}E[Y_{i,M}] + 1.$$

Based on Equations 3.4 and 3.5, we can calculate $E[Y] = E[Y_{N,M}]$. 
To analyze coord-guard, we can view this as a special form of the urn problem: there are \( N \) balls in an urn. \( r \) of the balls are red and the rest are white. \( 0 \leq r \leq N \). The probability of drawing \( W \) white balls before drawing the first red ball, or the urn is empty, is:

\[
P[W = w] = \begin{cases} 
\left( \prod_{j=0}^{w-1} \left( 1 - \frac{r}{N-j} \right) \right) \frac{r}{N-w}, & \text{if } w \leq N - r \\
1 & \text{if } w = N \\
0 & \text{otherwise}
\end{cases} 
\]

(3.6)

It is easy to see that red balls correspond to instances that guard the vulnerable component. We will call these instances red instances. The vulnerability is detected when the attacker attacks one red instance. We know that there are at least \( \lfloor MN/C \rfloor \) red instances. There is one more red instance with probability \( q = MN/C - \lfloor MN/C \rfloor \). By representing Equation 3.6 as \( f(W, r) \), we have

\[
P[X = x] = f(x, \lceil MN/C \rceil)q + f(x, \lfloor MN/C \rfloor)(1 - q). 
\]

(3.7)

\[
P[X \leq x] = \sum_{i=0}^{x} P[X = i]. 
\]

(3.8)

The expected number of compromised instances is

\[
E[X] = \sum_{i=1}^{N} iP[X = i]. 
\]

(3.9)

The number of unique components covered is easy to calculate for coord-guard:

\[
Y = \min(NM, C). 
\]

(3.10)
3.2.3 Results

We now answer the research questions proposed previously. We first show the value of $E[X]$ under different defense resource consumption in Figure 3.2a, using ind-guard (solid line in the figure). It shows that by only guarding a small percent of the components, the expected number of compromised instances is significantly reduced. For example, when an instance only protects 2% of all components, the expected percentage of compromised instances drops from 100% to roughly 40%. The percentage of compromised instances drops further to less than 20%, if each instance guards more than 5% of all components. On the other hand, we also see a clear diminishing returns, as more defense resource investment yield less marginal security improvement.

The value of $E[X]$ also depends on $N$, the number of instances deployed. Figure 3.2b shows that the expected number of compromised instances is capped at $E^*[X] = (1 - p)/p$, when $N \to \infty$. In addition, the solid curve in Figure 3.2b is always below the diagonal line (dotted red line), meaning that the expected percentage of exploited instances monotonically decreases as more instances participate in the collaboration. In other words, adding more instances always benefit the security of the whole species.

We next compare ind-guard with coord-guard. As we have discussed previously, ind-guard cannot yield uniform distribution of defense resources over components. To illustrate this, we plot the frequency distribution of guard counts by ind-guard. As we can see, the distribution is not uniform, and a subset of the components are not guarded by any instance. Coord-guard, on the other hand, can achieve the uniform distribution, by coordinating the defense resource allocation between different instances. Therefore, we expect that it performs better than ind-guard.

In Figure 3.2a and Figure 3.2b, we can see that coord-guard (green dashed curves) can further decrease the expected number of compromised instances under the same amount of defense resources. Particularly, when $NM = C$, the difference between ind-guard and coord-guard is maximized. We will refer to this point as the coord-point, meaning that the benefit of collaboration is maximumly utilized. In addition, we think that coord-guard is very useful in some other scenarios. First, if the defender wants to achieve higher coverage of the vulnerability space, using coord-guard would be more efficient than using ind-guard, as Figure 3.2d shows. This also allows coord-guard to guarantee that the zero-day vulnerability will be detected by at least one instance, or at least one instance will survive from the attack, as long as $MN \geq C$. 

25
Second, although under the current assumptions, the improvement by coord-guard is not very significant based on the results, in some other more realistic scenarios, coord-guard’s advantage could be more obvious. For example, we currently assume that the cost of guarding a component is identical for all instances. But it is possible that guarding a component is cheaper to some instances due to their special workloads. Then, instances can “trade” with each other to more efficiently utilize the limited defense resources globally. We leave this as a future work.

Third, from the figures we see that coord-guard has the maximum performance gain over ind-guard at the coord-point \((NM = C)\). By investing more resources after this point, the ind-guard starts to catch up, and the marginal security improvement decreases. Therefore, the defender can conduct optimization and adjustment to maximize the utility of defense resources. For example, the defender can keep the run-time protection to stay around the coord-point to save defense resources. And as more instances joining the system, each instance can spend less resources. The saved resources can then be applied to detect other types of bugs.

We can also extend the model to consider different kinds of scenarios. For example, it is possible that some instances are willing to sacrifice performance for security, by setting unlimited defense resource quota \((M = \infty)\). Such instances could also be honey pots whose only purpose is to capture attacks [96]. To model this scenario, we assume that \(F\) of \(N\) instances will set \(M = \infty\) to cover all components during run-time. Then we have

\[
P[X = x] = \left( \prod_{i=0}^{x-1} \frac{N - F - i}{N - i} \right) (1 - p)^x, \quad \frac{p}{N - x} + \frac{F}{N - x} \leq x \leq N - F.
\]

(3.11)

and \(X \leq N - F\). Based on Equation 3.9, we show \(E[X]\) for \(F = 1\) and \(F = 5\) in Figure 3.2f. We observe that by only having a tiny percent of full-coverage instance, the detection capability of ind-guard can be significantly improved, particularly when \(p\) is small. For example, when there is 1 full-coverage instance and all other instances spend zero defense resources, the \(E[X]\) is still reduced by 50%, because the attacker on average is expected to hit the full-coverage instance after attacking half of the instances. Since a small percentage of more security-focused instances can have a big impact, it might be
an interesting idea to incentivize some instances to invest more resources in security, so that the whole species can be benefited. One could build an incentive mechanism similar to bug bounty [20], or similar to bitcoin mining. We will have more discussion about this in Section 3.7.

Please note that such significant improvement by full-coverage instances is obtained under the assumption that the attacker will randomly pick the next target. It is possible that the attacker can scan an instance and tell whether it is full-coverage instance or not. We will discuss this issue in later sections.

In summary, this theoretical analysis shows that CRP can significantly decrease the number of successful attacks, while only require each node to guard a small portion of the components. In addition, by coordinating between instances, the efficiency of CRP can be further increased.

### 3.3 Run-time Protection against Heap Memory Vulnerabilities

#### 3.3.1 Overview

In this section, we demonstrate how to actually implement collaborative run-time protection (CRP) for a class of real-world vulnerabilities. Specifically, we focus on zero-day heap memory vulnerabilities, including use-after-free, double free, uninitialized pointer access, and contiguous overflow (including both over-read and over-write). These vulnerabilities are serious threats for today’s software programs written in C/C++. We implement a distributed run-time protection system called HeapCRP to detect and mitigate these vulnerabilities. In this section, we will explain the design and implementation of HeapCRP. However, we will skip the collaborative detection part, which will be the sole subject of next section.

Figure 3.3 gives an overview of HeapCRP. In a high level, detecting and mitigating a zero-day vulnerability involves the following four steps.

#### 3.3.1.1 Setup

The software program is complied with an extra compiler pass for encoding calling context during run-time, and is linked with a shared library of HeapCRP for run-time
protection. Please note that we don’t diversify the program during the compilation and linking [61]. Thus the software program can still be distributed in existing ways. The developer (or the user) is also recommended to do a profiling of buffer allocation to help HeapCRP determine good parameters. We will illustrate this more in Section 3.6.1.

### 3.3.1.2 Collaborative detection

If the user decides to use ind-guard, then the program can start directly. Otherwise, HeapCRP may first connect to a coordination server to retrieve a set of components to be protected by this instance, and then starts the program. Since the exploitation of heap memory vulnerabilities always starts from a heap buffer, it is natural to define a buffer as a component. So the set of all heap buffers generated during run-time is the vulnerability space. During the program execution, HeapCRP needs to decide which buffers to guard given limited defense resources. We implement the ind-guard and coord-guard schemes, and discuss the details in Section 3.4. For each guarded buffer, HeapCRP primarily relies on page protection to trap exploits against the buffer. HeapCRP also records some metadata of guarded buffers for diagnosis. Given enough instances equipped with HeapCRP, a large portion of or even the whole vulnerability space can be covered during run-time, while keeping the overhead for an individual instance small.
3.3.1.3 Diagnosis and self-patch

If the attacker attacks an instance that happens to guard the vulnerable buffer, a segmentation fault occurs and the program is terminated. Then, HeapCRP searches through the core dump and retrieves necessary information to generate a temporary patch. The temporary patch contains the type of the bug and the identifier of the vulnerable buffer. Then in the next run, when HeapCRP encounters the vulnerable buffer allocation again, it enhances the buffer so that it cannot be exploited.

3.3.1.4 Scale up the defense

To scale up the defense, an instance shares a new patch with other instances. In a centralized scheme, the new patch is first sent to a patch server, together with a proof-of-concept (poc) input that can trigger the bug, if available. Once the patch server receives a patch, it verifies the patch in a sandbox, and publishes the patch if successful. Other instances that periodically query the patch server can download and install this new patch. Thus, once an instance detects a zero-day bug, other instances can quickly become immune to it.

3.3.2 Protecting an Instance with HeapTherapy

HeapCRP is built on HeapTherapy, which is an efficient run-time protection against heap buffer overflows, including both over-writes and over-reads [30]. HeapTherapy integrates attack detection, exploit diagnosis, and defense generation. It has very low average overhead in terms of both speed (6.2%) and memory (7.7%). We explain the two key ideas used by HeapTherapy below, and refer the reader to the previous paper for more detail [30].

3.3.2.1 Calling context sensitive defense

The first key idea is to use encoded calling context as a characterization of vulnerabilities and the guidance for applying defense, based on the observation that the allocation calling context of the vulnerable buffer does not change when the same attack recurs. The main advantage of a calling context sensitive defense is that it allows HeapTherapy to apply expensive defense only to a small set of buffers allocated under vulnerable calling contexts, thus significantly reducing protection overhead. In addition, this calling
context sensitive defense focuses on the root cause of a bug, thus is more resistant against polymorphic attacks. To minimize the cost of retrieving and comparing calling contexts, HeapTherapy applies Probabilistic Calling Context encoding (PCC) [56] that represents a calling context using one integer, referred as CCID. The idea behind PCC is simple: before each call site, the thread-local integer variable storing the current CCID will be updated by 
\[
CCID \leftarrow 3 \times CCID + cs.
\]
\(cs\) is a hash value of the call site, which is calculated from the file name and line number at compile time. Essentially, PCC can be viewed as a hash of a calling context. In this work, we extend the usage of PCC in novel ways for collaborative zero-day vulnerability detection, discussed in Section 3.4.

3.3.2.2 Run-time protection against heap buffer overflow

The original HeapTherapy was designed to defend against heap over-writes and over-reads. To detect buffer over-reads, HeapTherapy probabilistically append a guard page after the end of a buffer. The guard page has been marked as inaccessible by \texttt{mprotect}, so that an out-of-bound read from the buffer will trigger a segmentation fault. Then, based on some metadata stored at the beginning of the guard page, HeapTherapy can know the CCID of the allocation calling context of the buffer. In the next round of execution, HeapTherapy allocates extra padding space after the buffer to contain the over-read (soft defense), and append a guard page after the padding to prevent over-read (hard defense). The defense for over-read is applied to defend against over-writes as well. In addition, HeapTherapy applies a canary-based defense [97] for over-write. It wraps buffers with canaries and put other metadata, such as the allocation CCID, in the buffer header. It then periodically examines the canaries and start the diagnosis process if some canaries are damaged. Evaluation shows that these defenses can effectively detect and mitigate several real world vulnerabilities, including the Heartbleed [30].

3.3.3 Run-time Protection against Use-after-free

HeapCRP extends HeapTherapy by adding a new defense against use-after-free vulnerabilities. According to various sources, use-after-free is one of the most prevalent and severe memory corruption vulnerabilities nowadays [98].
### 3.3.3.1 Detection

The basic idea of detecting zero-day use-after-free against a specific buffer is to skip the free of the buffer, and turn the buffer inaccessible using `mprotect`. Then, any memory access to that buffer after free will trigger a segmentation fault, leading to diagnosis. Since `mprotect` works at the page level, HeapCRP needs to round the buffer size to multiple of page size, which is typically 4KB. Figure 3.4 shows the layout of an allocated buffer under protection for use-after-free.

![Buffer Layout](image)

Figure 3.4: The buffer layout for use-after-free detection (after allocation).

At free time, HeapCRP puts a unique value UAF_MAGIC in the beginning of each page of the buffer, followed by the CCID of the buffer. These metadata are used for later diagnosis. Figure 3.5 shows the layout of the buffer after free.

![Buffer Layout](image)

Figure 3.5: The buffer layout for use-after-free detection (after free). Every gray page is inaccessible.

The code snippet of instrumenting `malloc` and `free` for use-after-free detection can be found in Figure 3.6. HeapCRP also handles other memory allocation functions, including `memalign`, `realloc`, and `calloc`, which are not shown in the figure.
The major problem of this idea is the potential high space overhead. Since buffers are not freed, the memory is rapidly being exhausted. In addition, the rounding and alignment will cause internal and external memory fragmentations, which further accelerate the memory exhaustion. Luckily, HeapCRP only asks an instance to apply these steps to a small set of buffers, thus limiting the performance impact. In addition, these guard pages actually will not be used again in normal execution, so they can be swapped out to the disk and never occupy the physical memory again. Moreover, previous research has proposed to map multiple virtual addresses to the same physical page, thus almost
avoiding any overhead on the physical memory [60]. Incorporating this defense into HeapCRP is an interesting future work. Another idea of alleviating the overhead is to periodically release the guard pages after a given period of time, or when the defense quota has been used up. This is similar to a use-after-free defense introduced to Internet Explorer in 2014 [99].

3.3.3.2 Diagnosis

The goal of diagnosis is to find the CCID of the buffer that is responsible to this use-after-free attack. Once there is an access to a guarded buffer after free has been called, a segmentation fault occurs. Then, HeapCRP analyzes core dump to find the address that caused the segmentation fault. Then HeapCRP checks whether the head of the page is UAF_MAGIC. If yes, it retrieves the CCID after the UAF_MAGIC. This CCID is referred to as vulnerable CCID, or vCCID. HeapCRP next put this vCCID in a temporary patch, which also contains the type of the bug.

3.3.3.3 Patching

After obtaining the vCCID for the use-after-free vulnerability, HeapCRP can prevent attackers exploiting the same vulnerability in the future. A hard defense is to always make the vulnerable buffer inaccessible after free. This prevents the attacker from hijacking the control flow. However, the attacker can still repetitively crash the program and cause denial of services. A soft defense is to simply skip the free. Therefore, the use-after-free accesses are always confined to the same buffer. This could mitigate many exploits. But in rare conditions, the program might still not be safe. For example, if the attacker writes payload to the buffer and the payload is later read by the other code, then the program might still be compromised successfully. However, compared with no defense, the difficulty of exploiting use-after-free significantly increases.

3.3.3.4 Double free

HeapCRP treats double free as a special case of use-after-free, because when the second free is called, HeapCRP needs to access the buffer, and will trigger a segmentation fault if the buffer has been guarded for use-after-free. The hard and soft defenses for mitigating use-after-free can also be applied to double free. In addition, HeapCRP can differentiate
double free from other types of use-after-free, based on whether the function free appears in the call stack of the core dump.

### 3.3.4 Run-time Protection against Uninitialized Pointer Access

Uninitialized read is a dangerous types of vulnerability. For example, in the 2015 Pwn2Own, researchers have exploited one uninitialized read plus a privilege escalation vulnerability to compromise the Internet Explorer browser [100]. Initially, we want to use guard page to detect uninitialized read at a coarse-grained level. The basic idea is to mark a buffer as inaccessible after allocation. Then once the page is first accessed, HeapCRP's signal handler will capture the segmentation fault. If it is a write operation, the page protection will be removed and the buffer can be used normally thereafter. If it is a read operation, HeapCRP detects an initialized read bug. However, through experiments, we found that this approach has some false positives. Therefore, we decide to focus on a special case of uninitialized read, which is uninitialized pointer access [101].

The basic idea of detecting uninitialized pointer access is to fill a newly allocated buffer with poisonous values. So that an uninitialized dereference or free of a pointer in the buffer will cause segmentation fault. DieHard implements a variation of this idea by filling newly allocated buffers with random values [102]. It then compares the outputs of two instances to detect the uninitialized read. However, it is possible that a random value might actually be a legitimate address. In addition, HeapCRP needs to trace back to the CCID of the buffer allocation when an uninitialized pointer access is detected.

#### 3.3.4.1 Detect

We propose the following idea for detecting uninitialized pointer access. Whenever HeapCRP needs to guard a buffer, it fills the buffer with $0xc0000000 + i \times x$. $0xc0000000$ is the default start of kernel address in Linux, so that any access through a pointer to this area the user space will cause a segmentation fault. $i$ is id of the current buffer guarded for uninitialized pointer access. $x$ is an offset with the default value $0x10$. We use $x$ to separate different pointers, and thus reduce the false positive rate called by small offsets in a pointer access (e.g., $(p - 1)$). HeapCRP stores the CCID of the buffer in an array called $uccids$. 
3.3.4.2 Diagnosis

HeapCRP first locates the faulty address $addr$. It then gets the id of the buffer as $i = \text{round}(addr - 0xc0000000) / x$. Then, HeapCRP retrieves $uccids[i]$ as the vCCID.

3.3.4.3 Patching

Filling the poisonous value described above can be used as a hard defense against uninitialized pointer dereference. As a soft defense, HeapCRP can also zero-fill the buffer that leads to an uninitialized pointer dereference. While this could still cause a segmentation fault at dereference, in many cases the program might have code to handle such NULL pointer situations. Similar insights have been utilized by previous work as well [103, 104]. In addition, HeapCRP can benefit from existing approaches that handle NULL dereferences [105].

3.4 Collaborative Detection of Zero-day Heap Memory Vulnerabilities

As discussed in Section 3.2, there are two main schemes for collaborative run-time detection: ind-guard and coord-guard. In ind-guard, each instance independently and randomly picks a subset of components to guard; while in coord-guard, there is a coordination between instances to maximumly utilize the defense resources. We have theoretically analyzed the advantages and disadvantages of these two schemes in Section 3.2. In this section, we explain how they are implemented in HeapCRP.

Before explaining the implementation details, we first define $C$ as the total number of CCIDs an instance encounters during an execution on average. $M$ is the defense resource quota. We assume that guarding a buffer costs 1 unit of defense resource\(^1\). Since multiple buffers might belong to the same CCID, the cost of guarding different CCIDs are different, and $M$ is likely to be greater than the number of unique CCIDs guarded.

\(^1\)Buffers have different sizes, so this definition of $M$ is only coarse-grained. For the sake of simplicity, we use the current definition.
3.4.1 Ind-guard:

The simplest way of implementing ind-guard is to guard a buffer with probability \( p \) (i.e., when \( \text{rand()} < p \)). Actually, HeapTherapy uses this idea for heap over-read detection, and shows that a small \( p \) (e.g., 3\%) is already very effective against Heartbleed.

However, a major problem of this approach is that the defense resources will be mostly directed to buffers under hot allocation calling contexts. To illustrate this issue, we conduct CCID profiling based on four instances of Lynx, a text-based Web browser. Each instance visits a unique website, and records CCID statistics displayed in Table 3.2.

<table>
<thead>
<tr>
<th>Instance</th>
<th># unique CCIDs</th>
<th># allocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>1546</td>
<td>16018</td>
</tr>
<tr>
<td>Google</td>
<td>1392</td>
<td>11134</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>1496</td>
<td>15721</td>
</tr>
<tr>
<td>Yahoo</td>
<td>1525</td>
<td>18300</td>
</tr>
<tr>
<td>(Average)</td>
<td>1490</td>
<td>15293</td>
</tr>
</tbody>
</table>

Table 3.2: Statistics of PCC coverage in the Lynx experiment.

Figure 3.7 shows the frequency distribution of CCIDs of one instance\(^2\) is extremely skewed: there are a few very hot CCIDs and a large number of cold CCIDs. We formulate this observation into the following hypothesis:

**Hypothesis 1.** The frequency distribution of allocation calling contexts, or CCIDs, are highly skewed right.

We will further test this hypothesis in Section 3.6. Under this hypothesis, a hot CCID that appears hundreds of times is likely to be guarded multiple times, while a cold CCID is unlikely to be picked, leading to an undesired defense resource allocation.

To solve this problem, we can let an instance to determine whether to guard a CCID only at the first encounter with a guarding probability defined as

\[
p = \frac{M}{\bar{a}C},
\]

where \( \bar{a} \) is the average cost of guarding a CCID. Based on our definition, \( \bar{a} \) is actually the average number of buffers allocated under a CCID. Here, \( M \) is defined as the soft

\(^2\)The distributions for other instances have a very similar shape.
quota, meaning that $M$ is the just the expected defense resource consumption, while the actual consumption could be different. HeapCRP needs this flexibility because it’s hard to accurately calculate $p$. If we over estimate $p$, then CCIDs encountered later will not be guarded. We also define a hard quota, which cannot be passed. The hard quota is set to $3M$ by default. Then, if $\text{rand}() < p$, HeapCRP adds the CCID to a guard-set; otherwise, the CCID is added to the skip-set. We will call this strategy rand-select.

While rand-select solves the bias for hot CCIDs, there are two additional problems. First, the size of the skip-set could be very large for certain programs that have a large number of unique allocation calling contexts. For example, the CCID profiling in Section 3.6.1 shows that two SPEC benchmark programs have more than 100 million unique CCIDs. Second, having a function call rand() for each new allocation calling context might be expensive, particularly if the number of unique CCIDs is large.

These two problems can be solved if we can let an instance make the selection at the beginning of execution. Then HeapCRP only needs to do a simple check at each allocation. But the challenge is that it is hard to know all buffers an instance will encounter during run-time before actually executing the program. Our idea is to utilize the nature of PCC. Essentially, it is a mapping of all possible calling contexts to all 32-bit integers. In other words, we know that all the calling contexts an instance will encounter are represented as an integer in the range $[0, 2^{32} - 1]$. Then, if the mapping is uniformly random, then selecting a subset of the integers is equivalent to selecting a subset of components. We refer to this idea as interval-select.
We propose the following hypothesis for probabilistic calling context (PCC) encoding:

**Hypothesis 2.** The CCIDs of a program obtained by PCC encoding is uniformly distributed in the output space.

Figure 3.8 shows the CCID counts of the Lynx example in 128 equal-sized bins that form \([0, 2^{32} - 1]\). As we can see, the distribution appears to be uniform. To further test this hypothesis, we apply chi-squared test, and the null hypothesis is that the CCID distribution is not different from a uniform distribution. The chi-squared test gives p-value = 0.57, which is insufficient to reject the null hypothesis. Therefore, we will assume that Hypothesis 2 is true.

![Figure 3.8: Distribution of CCID counts in bins in the Lynx example.](image)

Then, given \(M\) and \(C\), an instance can pick an interval of size \(k\), which can be calculated as:

\[
k = \frac{M}{aC}2^{32}.
\]  

(3.13)

An instance also needs to randomly pick \(s\), the start of the interval. Then, the instance will only guard CCIDs in \([s, s+k]\). If \(s + k > 2^{32} - 1\), the instance needs to wrap around and use a second interval starting from 0.
3.4.2 Coord-guard

Another defense scheme discussed in Section 3.2 is coord-guard. Intuitively, the defender partitions the vulnerability space and let each instance protects a subset of components, to maximize the utilization of defense resources. This idea can be implemented based on the interval-select strategy. The defender can split $[0, 2^{32} - 1]$ into $2^{32}/k$ pieces, and then assign each piece to one instance. After all pieces have been assigned, the defender can start a new round of partition and assignment. In each round, the defender randomly picks a starting point for partition. Algorithm 1 describes this idea in detail.

input : $N$ and $k$
output : assignments of CCID intervals.
group_size = $2^{32}/k$;
for $i \leftarrow 0; i < N; i \leftarrow i + 1$ do
  if $i \%$ group_size == 0 then
    $s \leftarrow \text{random}(0, 2^{32})$;
  end
  if $s + k - 1 < 2^{32}$ then
    assign_to_instance(i, $[s, s + k - 1]$);
    $s \leftarrow s + k$;
    if $s == 2^{32}$ then
      $s \leftarrow 0$;
    end
  end
else
  $wrap \leftarrow k - 2^{32} + s - 1$;
  assign_to_instance(i, $[s, 2^{32} - 1], [0, wrap]$);
  $s \leftarrow wrap + 1$;
end

Algorithm 1: Assign CCID intervals for coord-guard

There are multiple ways to actually implement Algorithm 1. In this paper, we assume that the defender runs a coordination service, which could either locate in a private network, or on the Internet. After installation, an instance sends a request to the coordination service, which then returns a CCID interval to the instance. It would be interesting to explore other ways of assignment, for example, in a decentralized fashion. We will discuss more in Section 3.7.
3.4.3 Improvements

We now introduce several additional improvements for the two collaborative detection schemes. These improvements are either inspired by the data of real world programs, or proposed to defend against potential attacks targeting CRP.

3.4.3.1 Handle hot CCIDs

Figure 3.7 shows that there are a few CCIDs that allocate a large number of buffers. We call them hot CCIDs. The hottest one, for example, has more than 1000 buffers. Guarding these hot CCIDs will consume a large amount of resources. In addition, although Hypothesis 2 assumes that CCIDs are distributed uniformly in intervals, the cost of guarding CCIDs in an interval is not uniformly distributed. Particularly, if the cost is high (e.g., by including the hottest CCID), then the defense resource quota $M$ will be exhausted very early, thus missing CCIDs appeared later.

Spending large amount of resources for hot CCIDs might not be ideal, for two reasons. First, it is likely that these hot calling contexts have been extensively reviewed and tested. Therefore, the chance of being vulnerable is low. Second, since the number of hot CCIDs is small, the developer can deliberately spend effort to examine or even verify these components, with reasonable cost, compared to examining all components. On the other hand, previous research has observed that cold code is more likely to contain bugs [106]. Therefore, we make the following hypothesis:

**Hypothesis 3.** Most vulnerable components appears infrequently.

Instances with a small quota $M$ can then skip hot components. We currently consider a CCID is hot if its average frequency is greater than 100. Other definitions of the threshold can be used. For example, a CCID is hot if $f > \mu + \sigma$, where $\mu$ is the mean frequency of components and $\sigma$ is the standard deviation. This formula can be adjusted based on the Chebyshev’s inequality. In our Lynx experiment, we skip 14 hot PCCs. Please note that instances (e.g., honey pots) with abundant resources (e.g., $M = \infty$) will not skip hot CCIDs. In addition, through coordination, it is possible that the defender can allocate all hot CCIDs to resource-abundant instances, thus improving the efficiency of collaboration. We leave this as a future work.
3.4.3.2 Defend against coordination disruption attack

We next discuss several potential attacks that can be used to make HeapCRP less effective. The first attack is only applicable to coord-guard, and the attacker needs to be able to access the coordination service. Then, since Algorithm 1 generates intervals sequentially, the attacker can easily predict when the interval of the vCCID will arrive. The attacker then constantly sending requests to let other benign instances skip the vulnerable interval. The attacker can also keep the intervals received by benign instances in a narrow range to reduce the diversity of instances, thus significantly limiting the effect of zero-day detection.

To prevent this attack, the actual implementation of Algorithm 1 first generates all $2^{32}/k$ intervals of a round, then randomly shuffles them, and next returns intervals one by one. Thus, the attacker cannot predict the next interval. Please note that the attacker can still disrupt the coordination, and downgrade the coord-guard to ind-guard. But the consequence is much less severe.

3.4.3.3 Defend against guard probing attack

Before sending the exploitation payload, the attacker can first try to probe whether the vulnerable component is guarded by the instance. If yes, the attacker can skip the instance. One type of probing is based on timing information, since guarding the vulnerable vCCID will delay the execution. In addition, if the instance uses interval-guard, then the attacker can probe which interval the instance is guarding, and skip intervals that contain the vCCID.

Existing defenses against timing channels could be used to address this threat [107]. In addition, we propose another way of selecting CCIDs, called bit-select, that can further reduce the predictability of HeapCRP. Different from interval-select, bit-select picks CCIDs based on a randomly generated bit pattern, $\text{pattern}$, and a randomly generated bit mask, $\text{mask}$. The instance guards a CCID if $\text{CCID} \& \text{mask} = \text{pattern}$. For example, if $\text{pattern} = 0000\ 0001$ and $\text{mask} == 1000\ 0001$, then HeapCRP only protects CCIDs whose most significant bit is 0 and least significant bit is 1.

We define the number of bits set in the $\text{mask}$ as its length $b$. We know that there will be $2^{32-b}$ CCIDs matching the pattern. Similar to Equation 3.13, we get:
\[ b = \log_2 \frac{aC}{M} \] (3.14)

Implementing coord-guard based on bit-select is easy. In each round, the coordination service first randomly generates a mask, then enumerates all patterns under the bit mask, and next assigns the patterns to different instances.

3.4.3.4 Defend against guard exhaustion attack

This attack requires three conditions: (1) the instance under attack has a finite hard defense resource quota \(3M\); (2) the attacker can allocate a large number of buffers under a guarded CCID (called \(aCCID\)); (3) the \(aCCID\) appears before the vulnerable \(vCCID\). Then, the attacker can exhaust the resource, so that when the instance encounters the actual \(vCCID\), it does not have available resources to guard the \(vCCID\) for detection.

We propose two defenses against this attack. First, under Hypothesis 3, the defender can notice the abnormal number of allocations under \(aCCID\), by keeping a list of known hot CCIDs prior to execution. Then, the defender can simply skip \(aCCID\). Or the defender can enter the “emergency mode” by setting \(M = \infty\), which at worst, reduce the bug into a denial of service attack. Third, the defender can add more randomness to the selection and assignment of CCIDs to be guarded using the bit-select strategy, thus making it harder to achieve the 3 conditions mentioned above.

3.5 Patch Sharing

After an instance detects a zero-day bug using HeapCRP, it generates a codeless patch, as we have discussed in Section 3.3. Then, the instance needs to share that patch with other instances so that they can harden themselves against the same attack. In this section, we design a patch sharing mechanism and discuss issues related to it. Since the main focus of this paper is collaborative detection, we will leave the task of implementing a full-fledged patching sharing mechanism as a future work.

First to support patch sharing and verification, we extend the codeless patch in HeapTherapy [30] to contain the following fields:

- \(vCCID\) of the vulnerability.
• **Type of the vulnerability:** this field can be use-after-free, overflow or uninitialized pointer access.

• **Other defense parameters:** The defenses provided by HeapCRP can take multiple parameters. For example, whether to use a hard or soft defense, the length of padding space for overflow, etc.

• **Software version:** the vCCID of a patch might change across different software versions. So a software version is needed in the patch.

• **PoC input:** A PoC input is an input or a script that can reproduce the bug. It is optional but highly desired.

Once the patch server receives a new patch, it verifies the patch if the PoC input is included. The whole workflow is described in Figure 3.9. Software instances can periodically check the patch server to download latest patches.

The reason to have patch verification is that if some of the instances in the species are untrusted, then the defender will assume that some of them might want to disseminate false patches to other benign instances in order to cause negative effects. However, the negative effect of a false HeapCRP patch is only some performance overhead for managing buffers allocated under the vCCID. This is an advantage over other types of defenses. For example, a false signature could block legitimate inputs, and a false code patch might enable the attacker to inject malicious payload. Therefore, implementing the patch sharing mechanism for HeapCRP requires less trust among participants, compared with other types of approaches. However, if the attacker can trick the patch server to accept a large amount of false patches, particularly if some of the false patches contain hot CCIDs, the increase of overhead for benign instances could still be large. Therefore, it is helpful to have a patch verification step for scenarios with less trust.

Capturing the PoC input at the detection site might not be easy, particularly if the input that triggers the bug requires multiple network packets or even complex user interactions. Applying record and replay methods [108] might be one solution. Even if the input can be captured, a user might have concerns to share it if it contains some sensitive information. Applying techniques like delta debugging [109] to simplify and sanitize the PoC input would be an interesting future work. In addition, an attacker could fabricate a malicious input, which might compromise the verifier’s system. Previous research proposes to use an virtual machine as a sandbox for verification [24], which is a promising approach.
A new codeless patch with a vCCID and a version.

Has PoC input?

Run the program in version' that is closest to the version in the patch, with HeapCRP in the full-coverage mode.

Select a new version and do the verification again.

Apply the PoC against the instance.

Crash?

Other versions untested?

Yes

Discard the patch.

No

Get vCCID' from the core dump.

Yes

Check and modify vulnerability type and other parameters

vCCID = vCCID'?

Yes

Release the patch mark verified.

No

version = version'?

Yes

Release the patch with vCCID', and mark verified.

No

Release the patch marked with unverified.
A new verified codeless patch will also be sent to the developer for creating a code patch, which is expected to completely remove the vulnerability. In addition, after a new version of the software is released, the patch server will test the new version with all known vulnerabilities that have a PoC. More specifically, the patch server runs each PoC against the program, similar to a regression test. If the program can still be compromised, the patch server will generate a (potentially new) vCCID. Then it publish all patches for the new version of the software.

This design can be improved in several ways. For example, it is possible to have a decentralized patch sharing mechanism without the patch server. For example, instances can share their discoveries through an overlay network [24], and at least some of the instances should have the verification capability to endorse new patches. We will have more discussion in Section 3.7.

3.6 Evaluation

We first conduct CCID profiling for programs and benchmarks used in the evaluation. Next, we use SPEC 2006 to measure the overhead of HeapCRP. Finally, we evaluate whether HeapCRP is effective against real world vulnerabilities.

3.6.1 Profiling Heap Buffer Allocation CCIDs

We first conduct a profiling of heap buffer allocation CCIDs for SPEC benchmarks used in overhead evaluation (Section 3.6.2), and programs used in the effectiveness evaluation (Section 3.6.3). The goal of is profiling is twofold. First, we want to further test Hypotheses 1-3, proposed previously. Second, we need to calculate HeapCRP parameters, including the guarding probability \( p \) for rand-select and the mask length \( b \) for bit-select. Any user of HeapCRP is recommended to do this.

Table 3.3 shows the result of CCID profiling. We make the following observations. First, we see that most programs have less than 100,000 unique allocation CCIDs, meaning that the vulnerability space can be covered with relatively small number of instances. However, two SPEC benchmark programs, omnipep and xalanbmk, have very large number of unique CCIDs. Covering such a huge vulnerability space requires much more instances. The second observation is that the frequency distribution of the CCIDs is highly skewed right, which is consistent with Hypothesis 1. This means that for
<table>
<thead>
<tr>
<th>Program</th>
<th># instances</th>
<th># unique CCIDs</th>
<th># allocations</th>
<th>skewness</th>
<th># hot CCIDs (freq. &gt; 100)</th>
<th>p</th>
<th>b</th>
<th>max. freq. of vCCID</th>
<th>p-value of chi-squared</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL 1.0.1f</td>
<td>1</td>
<td>1,656</td>
<td>5,661</td>
<td>11.20</td>
<td>7</td>
<td>0.23</td>
<td>2</td>
<td>1</td>
<td>0.86</td>
</tr>
<tr>
<td>MySQL 5.5.19</td>
<td>1</td>
<td>843</td>
<td>57,468</td>
<td>11.91</td>
<td>12</td>
<td>0.55</td>
<td>1</td>
<td>2</td>
<td>0.21</td>
</tr>
<tr>
<td>Lynx 2.8.8dev.1</td>
<td>1</td>
<td>1,284</td>
<td>11,884</td>
<td>27.54</td>
<td>14</td>
<td>0.31</td>
<td>2</td>
<td>2</td>
<td>0.39</td>
</tr>
<tr>
<td>openlitespeed 1.3.19</td>
<td>1</td>
<td>3,321</td>
<td>7,914</td>
<td>54.32</td>
<td>11</td>
<td>0.26</td>
<td>2</td>
<td>1</td>
<td>0.66</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>3</td>
<td>13,744</td>
<td>360,660,138</td>
<td>81.36</td>
<td>756</td>
<td>0.01</td>
<td>6</td>
<td>-</td>
<td>0.39</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>6</td>
<td>11</td>
<td>174</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>403.gcc</td>
<td>9</td>
<td>893,562</td>
<td>28,136,159</td>
<td>151.51</td>
<td>1655</td>
<td>0.002</td>
<td>9</td>
<td>-</td>
<td>0.45</td>
</tr>
<tr>
<td>429.mcf</td>
<td>1</td>
<td>9</td>
<td>210,500</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>5</td>
<td>5,412</td>
<td>658,595</td>
<td>38.67</td>
<td>128</td>
<td>0.05</td>
<td>4</td>
<td>-</td>
<td>0.34</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>2</td>
<td>192</td>
<td>2,474,270</td>
<td>6.83</td>
<td>21</td>
<td>0.75</td>
<td>0</td>
<td>-</td>
<td>0.24</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>1</td>
<td>11</td>
<td>180</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>3</td>
<td>262</td>
<td>179,067</td>
<td>8.01</td>
<td>27</td>
<td>0.21</td>
<td>2</td>
<td>-</td>
<td>0.38</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>1</td>
<td>157,034,048</td>
<td>267,113,956</td>
<td>-</td>
<td>394</td>
<td>4e-6</td>
<td>18</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>473.astar</td>
<td>2</td>
<td>196</td>
<td>4,799,979</td>
<td>1.49</td>
<td>83</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0.44</td>
</tr>
<tr>
<td>483.xalanckbmk</td>
<td>1</td>
<td>131,864,650</td>
<td>135,155,476</td>
<td>-</td>
<td>81</td>
<td>7e-6</td>
<td>17</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.3: Heap buffer allocation CCID profiling results. Some results are not available (-) because either the dataset is too small, or the dataset is larger than what our machine can handle (for 471.omnetpp and 483.xalanckbmk). Parameters $p$ and $b$ are calculated based on Equation 3.12 and Equation 3.14, respectively. We set $M = 1,000$ for detecting each type of bugs. So in total there are $3M$ defense resource consumptions for use-after-free, uninitialized pointer access and overflow.
each program used in the evaluation, there are a few very hot CCIDs, and a large number of cold CCIDs. We use 100 as the frequency threshold, and find that the number of hot CCIDs are very small, compared to the total number of unique CCIDs. We will skip hot CCIDs during later evaluations. Third, we see that the vCCID for each case is very cold, which supports Hypothesis 3. Finally, we conduct the chi-squared test for each case, and found no significant result based on the p-value. Therefore, the empirical result is also consistent with Hypothesis 2.

3.6.2 Measuring the Overhead of HeapCRP

We use SPEC 2006 benchmark to measure the speed overhead of using HeapCRP under different settings. Figure 3.10 shows that under the bit-low setting (i.e., HeapCRP use bit-select as the CCID selection strategy, with 1000 detection quota for each type of vulnerability), the average overhead is 11%. After tripling the quota, the average overhead is 14%. In addition, bit-select also achieves lower overhead on average, compared to rand-select.

3.6.3 Use HeapCRP to Defend Real World Vulnerabilities

We applied HeapCRP to 5 real world programs, listed in Table 3.4. Each program we use has a known vulnerability. However, please note that HeapCRP requires no knowledge of the vulnerability when performing the detection. The result shows that HeapCRP can quickly detect all vulnerabilities by only asking each instance to protect a small subset of bugs.

3.7 Discussion

3.7.1 Limitations

We discuss several limitations of CRP.

3.7.1.1 Targeted attack

CRP is not designed to be effective against a targeted attack, in which the attacker uses one or multiple zero-day vulnerabilities to only attack a very small number of instances. This is because CRP trades strong detection capability at an individual instance level for
Figure 3.10: Speed overhead due to HeapCRP. Each bar represents one experimental setting. The $M$ is 1,000, 3,000 and 5,000 for low, medium and high settings, respectively. The last bar of each group is based on the related work DieHarder [1].
<table>
<thead>
<tr>
<th>Program</th>
<th>Vulnerability</th>
<th>Reference</th>
<th>Time to detect (avg.)</th>
<th>Patched?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>M=100</td>
<td>M=200</td>
<td>M=300</td>
</tr>
<tr>
<td>OpenLiteSpeed 1.3.19</td>
<td>use-after-free</td>
<td>CVE-2014-0160</td>
<td>3.25 (3.46)</td>
<td>2.90 (3.74)</td>
</tr>
<tr>
<td>Nginx 1.3.9 + OpenSSL 1.0.1f</td>
<td>over-read</td>
<td>CVE-2014-0160</td>
<td>11.25 (7.04)</td>
<td>5.15 (3.94)</td>
</tr>
<tr>
<td>MySQL 5.5.19</td>
<td>over-write</td>
<td>CVE-2012-5612</td>
<td>4.85 (5.22)</td>
<td>2.25 (2.26)</td>
</tr>
<tr>
<td>Lynx 2.8.8dev.1</td>
<td>over-write</td>
<td>CVE-2010-2810</td>
<td>15.80 (8.60)</td>
<td>12.05 (5.59)</td>
</tr>
</tbody>
</table>

Table 3.4: Results of effectiveness evaluation. Time to detect means the number of instances compromised before the first detection. Numbers in bracket are the standard deviation.
low overhead and strong collective detection capability. However, the cost of carrying out such targeted attacks is usually very high (e.g., only affordable by nation-sponsored attackers). In addition, attacking only a small number of instances will not have sufficient financial return for many malicious actors. Therefore, we think that the main threat for Internet security is still medium or large scale attacks. In addition, CRP allows the defender to adjust the security resources investment of an instance from 0% to 100%, thus enabling it to even defend against targeted attacks at the cost of high overhead.

3.7.1.2 The free rider problem

A extremely selfish instance (or its user) can invest zero defense resources, yet still benefit from other instances detection effort by receiving latest patches. Such free rider problem has been observed in information security [110–112]. In the worst case, each user will try to free ride on the security investment of other users, thus completely invalidates CRP. One way of addressing this free rider problem is to enforce a base level of security contribution through CRP, by the software developer. Another solution, which also complements the base level enforcement, is to provide incentives for contribution. One idea for an incentive-driven approach is to introduce bug bounty. That is, whoever discovered a new zero-day can receive a considerable amount of bounty reward\(^3\). Building an infrastructure for incentive mechanisms would be an interesting future work.

3.7.1.3 Attacks against CRP

In Section 3.4, we discussed three possible attacks against the collaborative detection mechanism of CRP: the coordination disruption attack, the guard probing attack, and the guard exhaustion attack. While we proposed several solutions to these attacks, our solutions cannot fully mitigate these threats. We leave the task of further investigating and mitigating these attacks as a future work.

Since HeapCRP is built on HeapTherapy, HeapCRP also inherits the limitations of HeapTherapy in detecting and patching heap memory vulnerabilities. We refer the readers to [30] for more discussions.

\(^3\)As of 2016/09, the Internet Bug Bounty program hosted on HackerOne (https://hackerone.com/internet) offers minimum $5,000 dollar bounty for vulnerabilities related to the infrastructure of the Internet.
3.7.2 Future Work

In addition to the future work mentioned earlier, we discuss 3 additional future work directions that improve CRP and HeapCRP.

3.7.2.1 Fully decentralized run-time collaboration in a untrusted environment

The HeapCRP system we built in this work requires a centralized patching server to receive, verify and distribute patches. However, this central server is a single point of failure for CRP. The central server also needs to be trusted from all participating instances, which might be hard to achieve in the decentralized Internet. These concerns also applies to the coordination server used in the coord-guard scheme.

Therefore, an important future work is to design and develop a fully decentralized collaborative run-time protection. For example, instances can form a P2P network and share patches over it. The P2P network can either support a flat structure, in which each instance can verify an incoming patch, or a hierarchical structure, in which some instances have the ability to verify a patch, and share the patch to others. The decentralized network can also support other functionalities, such as the bug bounty mechanism for incentivizing security investment, as we have discussed in the previous subsection. The ultimate goal is to create a distributed, collaborative, and sustainable run-time protection that enables a software species to autonomously evolve against new threats.

3.7.2.2 Patch Endorsement

It is also interesting to explore whether CRP can be implemented without patch verification. For example, an instance can endorse a patch if it works against an attack, and other instances will have high trust on the patch. However, such mechanisms will likely delay the dissemination of a patch.

3.7.2.3 Optimize the utilization of defense resources

We also want to optimize the utilization of defense resources at the individual level and the collective level. At the individual level, CRP can allocate resources to components that are more likely to be vulnerable, based on vulnerability prediction methods [113–115]. For example, HeapCRP can potentially give more weights to calling contexts that are deeper, and associated with more complex code, thus relaxing Assumption 3.2.1. It is also
interesting to design more accurate profiling methods to determine CRP parameters. At the collective level, the coordination mechanism can be optimized as well. For example, the mechanism can allocate a CCID to an instance that has lower cost of guarding it. Or a high cost CCID can be allocated to a resource-abundant instance.

3.8 Conclusion

In this paper, we propose the idea of collaborative run-time protection (CRP) to enhance software programs against zero-day vulnerabilities. Theoretical analysis shows that CRP can quickly detect zero-day attacks with a small loss, and a small overhead on each individual instance. We further design and build a prototype system called HeapCRP. Evaluation based on real world vulnerabilities and SPEC 2006 benchmark show that HeapCRP is effective against real vulnerabilities at a low cost.
Chapter 4  |  Empirical Analysis and Modeling of Black-box Mutational Fuzzing

4.1 Introduction

Software vulnerability is the root cause of many security breaches. However, it has also been observed that discovering software vulnerabilities is hard. While software companies invest heavily to eliminate vulnerabilities, other parties including white hats [20] and black hats [8] are frequently able to find new vulnerabilities, even when endowed with less resources (e.g., computing power, manpower, information). In addition, investment in software security exhibits diminishing returns [116], which has also been discussed in the field of software reliability growth [74].

Understanding these phenomena has important theoretical and practical implications. Existing work on the economy of security usually involves models of software vulnerability discovery [117–119]. Such models can be improved by empirical analysis of real vulnerability discovery data. The effort of studying vulnerability discovery also help practitioners. For example, software companies can make better decisions on the level of security investment and the extent of collaboration with outside security researchers (e.g., white hats) [20]. Cyber-insurance organizations might also be able to assess the security of customers more accurately [120].

In this work, we conduct an empirical analysis and propose models for black box mutational fuzzing. Introduced in the early 1990s [42], black box mutational fuzzing remains an effective method for discovering real world vulnerabilities [121, 122]. Its basic idea is very simple. Given a program and a set of diverse seed files, the fuzzing
tool randomly mutates the files and uses the program to process them. Once the program crashes, a triaging tool identifies the underlying bug and determines its properties such as exploitability. This simplicity makes black-box mutational fuzzing not only easy to use, but also easy to analyze and model.

We first apply black-box mutational fuzzing to multiple Linux programs and collect data from each fuzzing campaign, based on the CERT Basic Fuzzing Framework (BFF) [67] (Section 4.2). Our dataset contains 60,000 fuzzing runs, 4,000 crashes and 363 unique bugs. Then, we empirically analyze the data and discuss the long-tail distribution of discovery probability (Section 4.3), as well as the distribution of exploitability of bugs (Section 4.5.3). Motivated by the empirical analysis, we propose a stochastic model of black-box mutational fuzzing (Section 4.4.1). The model is derived from software reliability growth models [73–76]. However, one unique contribution of our model is that we assume the arrival rates of individual bugs follow a power law distribution, which is consistent with our data. Together with a simulation model (Section 4.4.2), we attempt to explain phenomena discussed at the beginning of this section. First, we provide a method to estimate the expected discovery outcome, which sheds light on the diminishing return of security investment (Section 4.5.1). Next, we explain why it is hard for software companies to eliminate the vulnerability stockpile of black hats (Section 4.5.2). Finally, we discuss several potential directions for future work, including the generalization of this model to other vulnerability discovery mechanisms (Section 4.6). All scripts and data are published online\footnote{github.com/movingname/fuzzingModel} for reproducible research.

### 4.2 BFF and Data Collection

Figure 4.1 shows the workflow of black-box mutational fuzzing. We have created several Python scripts for seed collection, code coverage analysis, seed selection and data analysis. The fuzzing tool and triaging tool is from the CERT Basic Fuzzing Framework (BFF) [67]. BFF is shown to be effective in finding real vulnerabilities in various programs, and has been used in previous work on improving black-box mutational fuzzing [71, 72] as well. Next, we outline the details of our experiment.

**Step 1 - Target Selection.** By combining the lists of target programs used in the literature [67, 71, 72, 123], we have collected 18 programs that handle various types of video, audio, graphical, and document inputs. Table 4.1 lists all 9 programs in which...
BFF has successfully found bugs. We have also tried to apply fuzzing to the following programs: a2mp3, eog, gifsicle, mplayer, mp3blaster, mpg123, moc, Outside In Viewer 8.5.2, and pdf2svg. However, for any of these programs, BFF triggers less than 3 or even 0 crashes. We therefore exclude them from the following analysis.

**Step 2 - Seed Collection and Selection.** We have collected thousands of candidate seeds files, including pdf documents, mp3 files, videos and images, from search engines like Bing and Google. The # cand columns of Table 4.1 shows the number of candidate seed files for each program. We then write a script to collect the basic blocks (bbls) covered by each seed using the Intel Pin framework. In general, the higher coverage of seeds, the more vulnerabilities will be found in fuzzing [72]. Next, we select 50 seed files to form the final seed set for each program, using a simple greedy algorithm that maximizes the coverage in each iteration. Table 4.1 shows that the final seed sets still achieve similar levels of coverage (% bbls column).

**Step 3 - Fuzzing.** We use BFF as the fuzzing tool and use its default fuzzing configuration. The main configuration parameter is the seed used in each fuzzing run, and the fuzzing ratio, which indicates how many bits in the seed will be flipped. We use the default probability-based parameter selection method implemented in BFF [67]. The outcome of a fuzzing run is either a crash or nothing, while the result of a fuzzing campaign is a sequence of crashes caused by software bugs in the program. Since multiple crashes could correspond to the same bug, we need a triaging step to map a
crash to the corresponding bug.

**Step 4 - Triaging.** Once a crash is encountered, BFF will run the triaging step, which calculates the hash for the underlying bug based on the stack trace\(^2\), minimizes the input that triggers the crash, and determines whether the bug is exploitable or not. Similar to other triaging tools such as the `exploitable` for Windows OS and CrashWrangler for Mac OS X, the CERT Triage Tools in BFF assigns one of the following exploitability levels to each crash: unknown, not_exploitable, probably_not_exploitable, probably_exploitable and exploitable.

**Step 5 - Data Analysis.** At the end, we know the seed file, configuration, and outcome of each fuzzing run, as well as the hash and exploitability of each bug discovered. We then analyze the data and show statistics of the results in Table 4.1. We present our main analysis results in the next section.

<table>
<thead>
<tr>
<th>Program</th>
<th># cand</th>
<th># bbls</th>
<th>% bbls</th>
<th># runs</th>
<th># crashes</th>
<th># bugs</th>
<th>max_freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>xpdf 3.02-2</td>
<td>2,161</td>
<td>188,023</td>
<td>93.1%</td>
<td>4,303</td>
<td>185</td>
<td>37</td>
<td>73</td>
</tr>
<tr>
<td>mupdf (^3)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9,900</td>
<td>201</td>
<td>25</td>
<td>61</td>
</tr>
<tr>
<td>convert 5.2.0 (^4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>79,636</td>
<td>32,161</td>
<td>134</td>
<td>3,197</td>
</tr>
<tr>
<td>ffmpeg 0.8</td>
<td>787</td>
<td>121,875</td>
<td>86.7%</td>
<td>16,055</td>
<td>3,872</td>
<td>96</td>
<td>863</td>
</tr>
<tr>
<td>autotrace 0.31.1</td>
<td>149 (^5)</td>
<td>-</td>
<td>100%</td>
<td>29,729</td>
<td>2,548</td>
<td>23</td>
<td>593</td>
</tr>
<tr>
<td>jpegtran 1.2.0</td>
<td>320</td>
<td>6,837</td>
<td>99.4%</td>
<td>303,898</td>
<td>116</td>
<td>33</td>
<td>31</td>
</tr>
<tr>
<td>gif2png 2.5.4-2</td>
<td>1,084</td>
<td>12,772</td>
<td>99.8%</td>
<td>136,768</td>
<td>2,305</td>
<td>7</td>
<td>34</td>
</tr>
<tr>
<td>feh 2.2</td>
<td>1,332</td>
<td>56,266</td>
<td>94.8%</td>
<td>5,209</td>
<td>159</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>mp3gain 1.5.2</td>
<td>214</td>
<td>7,224</td>
<td>99.9%</td>
<td>1,369</td>
<td>1,451 (^6)</td>
<td>7</td>
<td>861</td>
</tr>
</tbody>
</table>

Table 4.1: Seed selection and fuzzing statistics of selected programs. # cands is the number of candidate seed files we collected from the Internet. # bbls is the number of unique basic blocks recorded when parsing the candidate seed files. % bbls is the percentage of basic blocks covered by the final seed set.

### 4.3 The Long-tail Distribution of Bugs

The major goal of fuzzing and any bug discovery effort is to find as many bugs as possible. Moreover, it has been observed that the ease of discovering different bugs is different. In

\(^2\)The method used to generate the hash is an extension of the fuzzy stack hash method proposed in the literature [124].
black-box mutational fuzzing, we can quantify ease of discovering bug $i$ as its discovery probability ($\lambda_i$):

$$\lambda_i = \frac{c_i}{t}$$

(4.1)

where $c_i$ is the number of crashes caused by bug $i$, and $t$ is the number of fuzzing runs in the fuzzing campaign. Then the question is, what is the distribution for bug discovery probability?

In Figure 4.2, we plot the empirical probability distribution of bugs for all 6 programs with more than 20 bugs discovered. We see that these distributions all have the long-tail shape; that is, a few bugs trigger a large number of crashes, while most bugs have only triggered crashes a few times. Many such distributions [125], including vulnerability exploitation [77], have been proposed to follow the power law distribution. We thus propose the following hypothesis:

**Hypothesis 1.** The discovery probability of bugs in a program follows a power law
distribution.

More specifically, we assume the following discrete power law distribution [125]:

\[
P(\text{discover bug } i \text{ in a fuzzing run}) = \lambda_i = \frac{i^{-\alpha}}{\zeta(\alpha)}
\] (4.2)

where \(\alpha\) is the scaling factor of the power law distribution, \(\zeta(\alpha)\) is the Riemann \(\zeta\)-function as the normalizer. As we will show in Section 4.5, a smaller \(\alpha\) leads to more bugs discovered in the same number of fuzzing runs. \(i\) is the rank id of the bug among all bugs sorted by their discovery probability inside the program. A bug with a larger rank id (lower rank) has lower probability to be discovered, as Equation 4.2 tells. To complete the probability distribution, we also use \(\lambda_1\) to represent the probability of no crash. We can think about no crash as a special bug, and it has the highest probability in these 6 fuzzing campaigns.

We next need to estimate the scaling factor \(\alpha\) of a power law distribution from the empirical distribution. The most common approach is to use Maximum Likelihood Estimators (MLEs) [125,126]. However, we could not apply these estimators because we do not know the true rank id of a bug discovered in fuzzing. We only know a bug’s rank among all discovered bugs. For example, the 20th bug in the empirical data could have the true rank id of 100.

We propose a simulation method to estimate \(\alpha\). We could think a fuzzing campaign with \(t\) runs as generating \(t\) values form the corresponding power law distribution. We then choose the \(\alpha\) that minimizes the difference between the number of unique bugs discovered in the experiment and the number of unique values generated from the distribution. Table 4.2 shows the estimates of \(\alpha\).

<table>
<thead>
<tr>
<th>Program</th>
<th>(\alpha)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xpdf</td>
<td>2.39</td>
</tr>
<tr>
<td>mupdf</td>
<td>2.88</td>
</tr>
<tr>
<td>convert</td>
<td>2.38</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>2.21</td>
</tr>
<tr>
<td>autotrace</td>
<td>3.25</td>
</tr>
<tr>
<td>jpegtran</td>
<td>3.53</td>
</tr>
</tbody>
</table>

Table 4.2: Estimates of \(\alpha\).

Because we do not know the true rank id of bugs discovered, it is also difficult to
apply goodness-of-fit tests, either through bootstrapping or by comparing with alternative distributions [77, 126]. In this work, we will test the estimates of $\alpha$ by comparing the predicted number of bugs discovered with the actual number of bugs discovered in Section 4.5.1. More rigorous methods of estimating $\alpha$ and testing the goodness-of-fit are left as future work. In the following sections, we will show that this power law hypothesis enables us to answer some interesting questions related to vulnerability discovery and software security.

4.4 Modeling a Fuzzing Campaign

In this section, we build models for a fuzzing campaign. First, we propose a stochastic model based on the existing software reliability literature [73–76], in Section 4.4.1. Although expressive, this stochastic model has two assumptions that might not be realistic. We remove one assumption by proposing a simulation model in Section 4.4.2.

4.4.1 A Stochastic Model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N(t)$</td>
<td>The Poisson process for number of crashes in a fuzzing campaign.</td>
</tr>
<tr>
<td>$N_i(t)$</td>
<td>The Poisson process corresponding to bug $i$.</td>
</tr>
<tr>
<td>$\lambda_i$</td>
<td>The rate for $N_i(t)$ and the discovery probability of bug $i$.</td>
</tr>
<tr>
<td>$n$</td>
<td>Total number of bugs in the program.</td>
</tr>
<tr>
<td>$N'(t)$</td>
<td>The non-homogeneous Poisson process for number of unique bugs.</td>
</tr>
<tr>
<td>$\mathcal{D}(t)$</td>
<td>The set of discovered bugs by time $t$.</td>
</tr>
<tr>
<td>$D(t)$</td>
<td>Number of discovered bugs by time $t$. $D(t) =</td>
</tr>
<tr>
<td>$\mathcal{L}(t)$</td>
<td>The set of remaining bugs by time $t$.</td>
</tr>
</tbody>
</table>

Table 4.3: Notations.

Since each fuzzing run is independent from other runs, and the outcome of a fuzzing run is either 1 (crashed) or 0 (not crashed), it is natural to consider the fuzzing process as a Poisson Process $\{N(t), t \geq 0\}$, where $N(t)$ is the number of crashes seen till time $t$. Furthermore, since crashes are caused by different bugs, we can expressed $N(t)$ as:

$$N(t) = \sum_{i=2}^{\infty} N_i(t)$$  \hspace{1cm} (4.3)
Here, $i$ is the rank id of a bug and $t$ is the number of fuzzing runs. $\{N_i(t), t \geq 0\}$ is the corresponding Poisson process for the $i$-th bug, and $N_i(t)$ is the number of crashes for the $i$-th bug we have seen till time $t$. We can see that the discovery probability of the $i$-th bug we have discussed in the previous subsection is actually the rate $\lambda_i$ of the Poisson process $N_i(t)$. A larger $\lambda_i$ means that bug $i$ causes crashes more frequently.

In a fuzzing campaign, we are mostly interested in the first crash of a bug. This is equivalent to the assumption in the software reliability models that a bug is found and instantly fixed, while the fix does not influence the discovery of other remaining bugs [73]. We define $\mathcal{D}(t)$ as the set of bugs that have already been found by time $t$, and $\mathcal{L}(t)$ as the set of remaining bugs. So we have:

$$\lambda'(t) = \sum_{i \in \mathcal{L}(t)} \lambda_i = \sum_{i=2}^{\infty} \lambda_i - \sum_{i \in \mathcal{D}(t)} \lambda_i$$  \hspace{1cm} (4.4)

Therefore, we obtain a new non-homogeneous Poisson process, $N'(t)$, for the discovery of unique bugs. $\lambda'(t)$ is the arrival rate of new bugs, and the expected time to discover the next bug is $1/\lambda'(t)$.

We currently do not know how to solve Equation 4.4 analytically. Thus when doing the calculation, we replace $\infty$ with $n$, in order to obtain an approximate result. Intuitively, we assume there are $n$ bugs in total inside the program. By choosing a larger $n$, we can further approximate the true result. In our following analysis, we set $n = 1000$. The probability that $i > 1000$ is only $1.3e-4$ for ffmpeg ($\alpha = 2.21$), and $9.03e-9$ for jpegtran ($\alpha = 3.53$).

In addition, this stochastic model relies on the following two assumptions:

**Assumption 4.4.1.** *In one fuzzing run, multiple bugs can be triggered.*

However, in BFF, each fuzzing run stops at the first crash, which is then triaged to one bug. Thus, with Assumption 4.4.1, the model will slightly overestimate the number of bugs discovered, as we will see in Section 4.5.1. But we expected that this effect is small because most bugs have low discovery probability (Figure 4.2), and the chance that multiple bugs are triggered in the same fuzzing run is even lower.

**Assumption 4.4.2.** *The discovery probability distribution is the same for all fuzzing runs in a fuzzing campaign.*
This assumption also oversimplifies the reality. Since the fuzzing seeds and fuzzing ratio are different among fuzzing runs, each fuzzing run will explore a unique input space and be able to trigger a different subset of all latent bugs. We will discuss this more in Section 4.6.

Improving this stochastic model by relaxing these two assumptions is challenging, which is left as a future work. In the next sub section, we propose a simulation model that removes Assumption 4.4.1.

### 4.4.2 A Simulation Model

Similar to the discussion in Section 4.3, we could think of a fuzzing campaign with $t$ runs as generating $t$ values from the corresponding power law distribution. Algorithm 2 returns a simulated bug discovery sequence as well as unique bugs discovered, given $\alpha$ and $t$ as the inputs. Step 1 and 2 can be implemented using an existing software package [126]. In step 5, we add the condition $id > n$ because we will compare the simulation model with the stochastic model.

```python
input : \alpha of the bug distribution, and $t$, the number of fuzzing runs
output : Simulated bug discovery sequence and unique bugs discovered
dist = powerlaw(\alpha, xmin=1, discrete=True);
seq = dist.gen_random($t$);
bugs = {};
   foreach id \in bugs do
      if $id == 1$ or $id > n$ then
         continue;
      end
      if $id \notin$ bugs then
         bugs.add(id);
      end
   end
return seq, bugs;
```

**Algorithm 2:** Simulate a fuzzing campaign.

In this simulation model, we remove Assumption 4.4.1 since each fuzzing run only yields at most one bug discovery. In Section 4.5.1, we will compare the predicted numbers of bugs discovered by these two models, and the actual number of bugs discovered.
4.5 Analysis Results

We present 3 analysis results in this section. We first use the models presented in the last section to calculate the expected number of bugs discovered, and discuss the diminishing returns in software security. We then examine the order of bug discovery to explain why different parties are likely to find different bugs. Finally, we empirically study the exploitability of bugs and discuss its implications.

4.5.1 Expected Number of Bugs Discovered

The first question is, what is the expected number of unique bugs found by time \( t \)? Under the stochastic model proposed in Section 4.4.1, we know that the time of the first occurrence of each bug follows the exponential distribution with parameter \( \lambda_i \). Therefore, the probability of bug \( i \) undiscovered by time \( t \) is \( e^{-\lambda_i t} \), and the expected number of undiscovered bugs at time \( t \) is \( \sum_{i=2}^{n} e^{-\lambda_i t} \). We then know that the number of expected bugs discovered by time \( t \) is:

\[
E[D(t)] = n - \sum_{i=2}^{n} e^{-\lambda_i t} = n - \sum_{i=2}^{n} e^{-\frac{i-a}{\zeta(a)} t}
\]  

(4.5)

We also use the simulation model proposed in Section 4.4.2 to obtain \( E[D(t)] \). We repeat the simulation 10 times and take the average number of bugs discovered by time \( t \) as \( E[D(t)] \). As we have discussed in Section 4.4.1, we set \( n = 1000 \) for both models.

In Figure 4.3, we show the plots of expected bugs discovered based on the Poisson process and the simulation, and the real trajectory, of 6 fuzzing campaigns. We see that the predicted curves from both models are close to the real curve, except for autotrace. We suspect that the large prediction error for autotrace is due to a poor fit of power law to its empirical distribution. We plan to further investigate this in the future. In addition, the curve of the stochastic model is generally above the other two. This can be partly explained by Assumption 4.4.1, as we have discussed in Section 4.4.1. In general, the simulation model gives a more accurate prediction for the 6 fuzzing campaigns than the stochastic model.

The concave shape of all curves show the diminishing returns: as the fuzzing campaign enters the long tail, the rate of discovery (\( \lambda'(t) \)) decreases, and the number of bugs

62
discovered in the same amount of time reduces. This diminishing return is consistent with our experience of fuzzing and software reliability growth [75, 76]. A software company can use the two models to decide how long the fuzzing campaign shall run. First, the company needs to run a fuzzing campaign for a limited amount of time, in order to estimate $\alpha$. Then, the company needs to define the reliability and security utility gain of finding a bug, and the fuzzing cost, which might include computing resource consumption, delayed product release, etc. Next, the company can generate the accumulated utility curve and the accumulated cost curve based on the curve of expected bug discovery ($E[D(t)]$) proposed in this section. At the point when the utility of fuzzing is below the cost, the fuzzing campaign should be terminated.

### 4.5.2 The Order of Bug Discovery

The diminishing return discussed in Section 4.5.1 might appear to be a good thing for security. If there is a strong order of bug discovery, then bugs with larger discovery probability will almost always be eliminated first. Thus, as long as the software company invests more resources than other parties, including black hats [8] and white hats [20], in
vulnerability discovery, these other parties are less likely to find new vulnerabilities.

However, in reality, we see that many vulnerabilities of famous software have been discovered by outside parties, many of whom are just individuals [19, 20, 44, 127]. There are multiple reasons to explain this. In this work, we propose one explanation based on the power law hypothesis. The basic idea is that the order of bug discovery is weak in the long-tail part of the distribution.

To further explain this, we first define the order of bug discovery. At the end of a fuzzing campaign, the expected sequence of rank id (S) of discovered k bugs is 2, 3, ..., k, because a higher ranked bug has higher discovery probability, and thus is expected to be discovered earlier. However, due to the randomness, the actual id sequence (\(\hat{S}\)) would be different from the expected sequence S. We can calculate the edit distance \(D(S, \hat{S})\) between these two sequences, and define the order of bug discovery as:

\[
\text{order}(\hat{S}) = k - D(S, \hat{S})
\] (4.6)

Intuitively, the bug discovery is strongly/weakly ordered if the distance between S and \(\hat{S}\) is small/large. However, since we do not know the true rank id of bugs discovered, we cannot calculate the order of empirical sequences directly. Instead, we run a simulation to generate 5 sequences in Table 4.4. We see bugs discovered in the beginning are more ordered, and tend to be rediscovered in other sequences.

| Seq 1: | 19 | 3 | 2 | 9 | 4 | 5 | 12 | 14 | 6 | 84 | 10 | 7 | 85 | 95 | 24 |
| Seq 2: | 2 | 3 | 7 | 4 | 5 | 17 | 10 | 13 | 40 | 8 | 6 | 49 | 12 | 11 | 9 |
| Seq 3: | 2 | 4 | 5 | 28 | 3 | 6 | 7 | 18 | 9 | 12 | 13 | 20 | 11 | 10 | 21 |
| Seq 4: | 2 | 5 | 6 | 3 | 4 | 9 | 15 | 12 | 99 | 10 | 8 | 46 | 7 | 225 | 20 |
| Seq 5: | 3 | 2 | 4 | 7 | 8 | 5 | 27 | 10 | 11 | 6 | 9 | 23 | 82 | 14 | 12 |

Table 4.4: Simulated bug discovery sequence based on the ffmpeg case (\(\alpha = 2.21\)). Bug ids in the bold font are unique to that sequence.

We can use the stochastic model to explain this. The probability that the next new discovery is bug \(i\) (assuming \(i \in \mathcal{L}(t)\)) is:

\[
P(\text{bug } i \text{ is the next one after time } t) = \frac{\lambda_i}{\left( \sum_{j \in \mathcal{L}(t)} \lambda_j \right)} \propto \lambda_i \propto i^{-\alpha}
\] (4.7)
For bug $i$ and bug $i+1$ (assuming $i+1 \in \mathcal{L}(t)$), we have:

$$P_i - P_{i+1} \propto i^{-\alpha} - (i+1)^{-\alpha}$$

(4.8)

which decreases to 0 as $i \to \infty$. This means that when $i$ is small (the fuzzing process is in the “head part” of the distribution), a bug with higher discovery probability is much more likely to be discovered first, and the fuzzing process has a stronger order. However, as $i$ increases and the fuzzing process enters the long-tail, which vulnerability will come next is harder to predict. In addition, a smaller $\alpha$ will make the fuzzing outcome less ordered, while a larger $\alpha$ makes the process more ordered.

To understand its implication, we consider a “fuzzing competition” between a software company and a black hat. Both sides run fuzzing and try to find as many bugs as possible. We assume that the software company has a resource advantage $A$ over the black hat. That is, while the black hat can conduct a fuzzing campaign with $t$ runs, the company can do $At$ runs, by having a larger fuzzing server farm. We want to know how many unique bugs can the black hat find.

![Figure 4.4: Simulated number of unique bugs discovered in two programs by the black hat under different resource advantage $A$ of the software company.](image)

We simulate 10,000 fuzzing runs for the black hat, and simulate $10,000 \times A$ runs for the software company. The two curves in Figure 4.4 show the number of unique bugs found by the black hat for two programs. We observe that although in the beginning, the software company can quickly reduce the bug pool of the black hat by investing more resources, the return of investment quickly diminishes as $A$ further grows. When the
software company has 30 times more fuzzing resources, the black hat is still able to find 2 unique bugs for ffmpeg and 1 unique bug for xpdf on average. Intuitively, it means that when the fuzzing enters the long-tail, the outcome is more random, so the company is less capable of interfering with the black hat’s outcome. This partly explains why in reality, outside parties such as black hats and white hats are able to find security holes, despite software companies having already spent significant effort in software security. From Figure 4.4, we can also see that when $\alpha$ is smaller, more unique bugs can be found by the black hat.

In summary, the power law hypothesis favors attackers, since they are able to find vulnerabilities even if the defender has much more resources. In addition, there is an asymmetry between attackers and defenders: the attackers only need to find a few exploitable bugs to succeed, while the defenders have to patch all holes. On the other hand, this result also encourages software companies to collaborate with outside benign white hats, through vulnerability disclosure and bug bounty programs [20]. We will discuss this more in Section 4.6.2. But before that, we need to ask one more question: are these unique bugs discovered by the black hat exploitable?

### 4.5.3 Exploitability

Table 4.5 shows the distribution of bug exploitability in the data. We see that a significant portion of the bugs are either exploitable or probably exploitable.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>xpdf</td>
<td>27%</td>
<td>32%</td>
<td>22%</td>
<td>0</td>
<td>19%</td>
<td>-0.34</td>
<td>0.07</td>
</tr>
<tr>
<td>mupdf</td>
<td>24%</td>
<td>0</td>
<td>64%</td>
<td>0</td>
<td>12%</td>
<td>-0.23</td>
<td>0.30</td>
</tr>
<tr>
<td>convert</td>
<td>33%</td>
<td>3%</td>
<td>7%</td>
<td>0</td>
<td>57%</td>
<td>-0.02</td>
<td>0.88</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>8%</td>
<td>17%</td>
<td>29%</td>
<td>0</td>
<td>46%</td>
<td>-0.15</td>
<td>0.28</td>
</tr>
<tr>
<td>autotrace</td>
<td>39%</td>
<td>4%</td>
<td>4%</td>
<td>0</td>
<td>52%</td>
<td>0.22</td>
<td>0.52</td>
</tr>
<tr>
<td>jpegtran</td>
<td>79%</td>
<td>6%</td>
<td>0</td>
<td>0</td>
<td>15%</td>
<td>-0.08</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Table 4.5: Percentages of exploitability, and correlation between bug discovery probability (log) and exploitability. A correlation is significant if the p-value is less than 0.1.

Then, we further ask the question: is there any correlation between discovery probability and exportability? If there is a positive correlation, then it means that harder to be discovered bugs are harder to be exploited, which favors the software company side. To
answer this question, we calculate the Spearman correlation between the logarithm of discovery probability, and the exploitability which is mapped to a 1-4 scale, with 1 meaning not exploitable and 4 means exploitable. We exclude bugs of unknown exploitability. The result is shown in Table 4.5. We see that although 5 out of 6 programs have a weak negative correlation (i.e., harder to be discovered bugs are easier to be exploited), there is only one that is statistically significant (xpdf). We thus propose the following hypothesis:

**Hypothesis 2.** Bug discovery probability and exploitability do not have a strong correlation.

Hypothesis 2 has several implications. First, it indicates that the next bug to be found could be exploitable, no matter how many runs have been conducted before. This gives an advantage for black hats, who not only are likely to find unique bugs, but are also able to find exploitable ones. Second, by assuming the independence between discovery probability and exploitability, one can predict the exploitability of the next bug based on the empirical exploitability distribution in Table 4.5. For example, in the case of xpdf and $A = 30$ in Figure 4.4, we can predict that the 1 unique bug discovered by the black hat has roughly 25% probability of being exploitable. By combining the vulnerability discovery models and the exploitability distribution, the software company can thus better forecast potential attacks and allocate defense resources accordingly.

### 4.6 Discussion and Future Work

#### 4.6.1 Apply Our Analysis to Larger Datasets

Although our dataset includes most of the programs studied in previous work [67, 71, 72, 123], it is still not enough to fully test the hypotheses we proposed. Therefore, an important future work is to increase the scope of analysis to other programs, other platforms (e.g., Microsoft Windows and Mac OS), and other fuzzing frameworks [43]. It would also be helpful to run the fuzzing campaign for much longer time.

Another important direction is to apply our analysis to different fuzzing configurations, which include the selection of fuzzing ratio, seeds, etc. It is possible that the same bug’s discovery probability might be significantly different in different configurations. This diversity gives an additional explanation to why other parties are likely to find unique bugs, in addition to our discussion in Section 4.5.2. That is, different parties tend to
have different configurations, and thus the discovery probability distribution is distinct to each of them. However, although the discovery probability of a bug might be different under different fuzzing configurations, we hypothesize that the discovery probability distribution will still be a power law distribution:

**Hypothesis 3.** The bug discovery probability under different fuzzing configurations follows power law distributions.

### 4.6.2 Generalization to Other Vulnerability Discovery Approaches

We choose to study black-box mutational fuzzing first because it is probably the simplest vulnerability discovery method. However, black-box mutational fuzzing is just one method in the vulnerability discovery toolbox. Other methods include code review, static analysis, symbolic execution, dynamic analysis, etc. We hypothesize that these approaches might resemble fuzzing, and thus the bug discovery “ease”, a generalization of the discovery probability, could also follow the power law distribution. Collecting empirical data from these approaches and applying similar analysis would be an interesting future work.

Some other vulnerability discovery paradigms also share similarities with black box mutational fuzzing. For example, many companies today collaborate with a large number of outside security researchers (or white hats) through vulnerability disclosure and bug bounty programs [20]. Actually, our discussion in Section 4.5.2 provides one explanation of why such collaboration is necessary. In addition, these white hats, with diverse background and skill levels, will often test different parts of the system, or using various testing payload. This is similar to the seed mutation in a black-box mutational fuzzing, although the distribution of inputs might be more complex than random bit flipping. Therefore, we could possibly generalize the proposed models to understand and analyze data from these bug bounty programs.

### 4.7 Conclusion

Understanding the process of vulnerability discovery and why software security is hard has important practical implications. In this work, we have collected empirical data of black-box mutational fuzzing. We show that the fuzzing process can be modeled as a non-homogeneous Poisson process with the rates of individual bugs following a power
law distribution. We then show how to calculate the expected outcome of a fuzzing campaign. We further show that once the vulnerability discovery enters the long-tail, there will be significant diminishing returns, and less order in the bug arrival. These effects pose challenges for the software companies that try to eliminate vulnerabilities before the black hats, and call for collaboration with white hats. Finally, we show that the model can potentially be extended to other vulnerability discovery mechanisms, such as bug bounty programs, that have diversity and randomness.
Chapter 5  
An Empirical Study of Web Vulnerability Discovery Ecosystems

5.1 Introduction

Websites are critical pathways to facilitate e-commerce, customer service, input procurement, and employee connectivity, and they continue to reach significant penetration in various business sectors. Most large businesses are hosting web services, and over 50% of small businesses are now offering web accessibility [128]. As such, web security has become critically important for most organizations, and the prevention of security compromises enabled by web vulnerabilities is gaining increasingly the attention of company leadership and the broader security community. Nevertheless, web vulnerabilities are the likely causes of many recent security breaches contributing to massive disclosure of user data, leakage of business information, and other losses.

To reduce the number of web vulnerabilities, organizations can use automated web vulnerability scanners which however have been shown to only have limited coverage [129, 130]. In response, organizations more recently started to directly collaborate with or indirectly benefit from outside security researchers. These so-called white hat researchers spend time to analyze organizations’ web systems and report vulnerabilities to self-run bug bounty programs of organizations such as Facebook, Github and PayPal, or to corresponding programs on third-party bug bounty platforms such as Wooyun, HackerOne, BugCrowd, Cobalt, etc.

White hats contribute in many positive ways to the discovery of web vulnerabilities. First, they can complement the limitations of automated scanners [129] by reaching
deeper states of web applications, and may better understand the application logic. Second, with a mindset comparable to attackers, white hats are good at finding many exploitable vulnerabilities of high severity. Third, the large and diverse group of potential white hat contributors outnumbers internal security teams or penetration testing teams and could therefore cover a wider range of security issues.

White hats’ considerable efforts are rewarded in different ways. Organizations or bug bounty platforms may provide monetary incentives based on severity and originality of the discovered issue, or publicize white hats’ contributions to enhance their reputations. Previous studies and reports have shown that the cost of utilizing the white hat community may be lower compared with hiring internal security researchers [32] or using services from penetration testing companies [131].

The resulting interactions extend beyond organizational boundaries and form web vulnerability discovery ecosystems including businesses/organizations, white hats, and third-party vulnerability disclosure reward/bounty programs (Figure 5.1). These ecosystems have been growing rapidly and are becoming more prominent in the battle against malicious actors on the Internet. However, detailed studies of these web vulnerability ecosystems to understand their characteristics, trajectories, and impact are notably absent.

In this work, we conduct the first empirical study of two major web vulnerability discovery ecosystems. We base our analyses on publicly available data. The first dataset is collected from Wooyun¹, the predominant and likely the oldest web vulnerability discovery ecosystem in China. Our data contains 64,134 vulnerabilities affecting a total of 17,328 organizations including almost all popular Chinese web companies. We additionally collect publicly available data from HackerOne², a US-based start-up company which hosts bug bounty programs for hundreds of organizations, such as Yahoo, Mail.ru and Twitter, from many parts of the world. The Wooyun dataset is larger due to its coercive participation model for involving organizations, and also contains more detailed vulnerability information due to its delayed full disclosure policy. The HackerOne dataset is smaller and not all of its reports can be accessed. However, it covers a different set of organizations and also contains monetary reward information that does not exist for the Wooyun dataset. By combining these two complementary datasets, we are able to explore a wide range of topics and gain a better understanding of the structure and dynamics of such ecosystems and their impact on Internet security. We anticipate that our study will

¹www.wooyun.org
²hackerone.com
<table>
<thead>
<tr>
<th>Platforms</th>
<th>Start</th>
<th>HQ</th>
<th># Vuln.</th>
<th># WHat</th>
<th># Org.</th>
<th>Bounty Paid</th>
<th>Disclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wooyun</td>
<td>2010-07</td>
<td>China</td>
<td>64,134</td>
<td>7,744</td>
<td>17,328</td>
<td>Unknown</td>
<td>Full</td>
</tr>
<tr>
<td>Facebook (2013) [132]</td>
<td>2011-08</td>
<td>US</td>
<td>687</td>
<td>330</td>
<td>1</td>
<td>$1.5M</td>
<td>No</td>
</tr>
<tr>
<td>BugCrowd [133]</td>
<td>2012-09</td>
<td>US</td>
<td>7,958</td>
<td>566</td>
<td>166</td>
<td>$0.7M</td>
<td>No</td>
</tr>
<tr>
<td>Loudong 360</td>
<td>2013-03</td>
<td>China</td>
<td>54,727</td>
<td>14,104</td>
<td>2,271</td>
<td>$0.7M</td>
<td>Partial</td>
</tr>
<tr>
<td>Cobalt</td>
<td>2013-07</td>
<td>US</td>
<td>8,119</td>
<td>2,600*</td>
<td>230</td>
<td>Unknown</td>
<td>Partial</td>
</tr>
<tr>
<td><strong>HackerOne</strong></td>
<td><strong>2013-11</strong></td>
<td>US</td>
<td><strong>10,997</strong></td>
<td><strong>1,653</strong></td>
<td><strong>99 (Public)</strong></td>
<td><strong>$3.64M</strong></td>
<td><strong>Partial</strong></td>
</tr>
<tr>
<td>Vulbox</td>
<td>2014-05</td>
<td>China</td>
<td>10,000</td>
<td>20,000*</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Partial</td>
</tr>
<tr>
<td>Sobug</td>
<td>2014-05</td>
<td>China</td>
<td>3,270</td>
<td>8,611*</td>
<td>285</td>
<td>$0.8M (Budget)</td>
<td>Partial</td>
</tr>
</tbody>
</table>

Table 5.1: Statistics for representative bug bounty platforms sorted by their start time. The two platforms studied in this paper are highlighted. Numbers were obtained from the cited references, or platforms’ websites directly in early August of 2015. The exact definitions of each metric for different platforms may vary. For example, some platforms count registered white hats (marked with *), while others such as HackerOne count white hats that have made at least one valid contribution.
be a valuable reference for organizations who want to create or optimize their existing bounty programs.

We make the following contributions:

• Our analysis shows how many white hats have been attracted by these ecosystems and how the number of contributing white hats evolves over time. We further assess their diversity in terms of productivity and breadth of vulnerability discovery (e.g., types of vulnerabilities and affected organizations) by studying individual contributions but also contributions by groups of white hats with high/medium/low productivity. We also analyze the potential (learning) value of disclosing vulnerabilities to the white hat community.

• We then quantitatively analyze participating organizations from several dimensions, including the vulnerability trends, the coverage of different business sectors, the response and resolve behaviors, and reward structures. We evaluate the trend of reported vulnerabilities for representative organizations.

• Our study further measures the impact of different factors on vulnerability discovery. In particular, we quantify the effect of offering monetary incentives for attracting white hats and reporting discovered vulnerabilities. Based on these analyses, we discuss the benefits of disclosing vulnerability information, offer suggestions on how to improve the effectiveness of the collaboration between white hats and organizations, discuss insights for relevant policy making (e.g., the Wassenaar Arrangement), and identify important research questions for future studies.

We proceed as follows. In Section 2.3, we discuss related work. In Section 5.2, we provide background information about Wooyun and HackerOne, and discuss the collection of the datasets. We present our data analysis results in Section 5.3, and provide a discussion in Section 5.4. We offer concluding remarks in Section 5.5.

5.2 Methodology

5.2.1 Analysis Overview

We organize our analysis around three components: the vulnerabilities disclosed, the white hats, and the involved businesses/organizations. Figure 5.1 outlines the structure of
a representative web vulnerability discovery ecosystem. In the following, we describe our data collection efforts.

5.2.2 Data Collection

We have collected publicly available data from Wooyun and HackerOne. The processed data and related Python scripts can be shared upon request in order to reproduce and extend our research.

5.2.2.1 Wooyun

Wooyun is the predominant web vulnerability disclosure program in China launched in May 2010. It has attracted 7,744 white hats who contributed 64,134 vulnerability reports related to 17,328 organizations. In most cases, Wooyun does not offer monetary rewards.

We choose Wooyun as one of the data sources for our study for several reasons. First, Wooyun insists on a delayed full disclosure policy, which states that the vulnerability will be disclosed 45 days after the submission of the report, irrespective of whether the organization has addressed the issue or not. To the best of our knowledge, it is the only platform that has such a disclosure policy. We will focus on this aspect in Section 5.4.1.

Second, Wooyun covers the longest period of time and the largest number of contributions compared with other platforms (Table 4.1). It also includes a large number of organiza-
tions from several different sectors, as we will discuss in Section 5.3.3.3. This is because Wooyun has a very relaxed submission rule compared with other US-based platforms: white hats can submit a vulnerability report to Wooyun for almost any organization, and Wooyun will publish it as long as the report is considered valid.

We crawled the vulnerability reports on Wooyun published from May 2010 to early August 2015. For each vulnerability report, we collected the following data fields: (1) white hat’s registration name, (2) target organization, (3) vulnerability type, (4) severity and (5) submission time. We further explain key data types below.

**Vulnerability type:** Each vulnerability report on Wooyun has a vulnerability type from a predefined list. However, we also observe that for some reports, the vulnerability types used are not in the list, possibly due to mistakes. We manually corrected these instances. We also translated the types from Chinese into English and list them in Figure 5.5.

**Severity:** The severity level of a vulnerability reflects its impact on the target organization. There are three levels: high, medium and low. We mainly use the severity level assigned by the affected organization or by the Wooyun platform. If this information is missing (e.g., when the organization does not respond to the report), we will use the severity level provided by the white hat reporter.

We have also collected the following data:

**Organization website’s URL and Alexa rank:** To examine whether a website’s popularity is related to vulnerability discovery, we collected the website’s rank from the Alexa Top Sites service. Since Wooyun does not provide the URL for all organizations, we wrote a script that queries the organization’s name on Google and takes the first result as the URL. We then retrieved the Alexa rank of all websites from the Alexa Top Sites service. Since most websites on Wooyun are Chinese, we use the Chinese Alexa rank, rather than the global rank.

**Organization sector:** We also categorized organizations into different sectors. The definition of sectors are based on previous studies [134, 135]. The categorization is initially based on patterns in the organization’s name. For example, universities have names like “XX university” or “university of XX”. After this step, we further manually categorized the remaining organizations that have received more than 40 vulnerabilities into different sectors.

Our dataset cannot contain all vulnerabilities discovered by white hats for organizations. First, due to the large volume of vulnerability reports received, Wooyun may ignore vulnerabilities that are considered irrelevant or of very low importance, such as
many reflected XSS vulnerabilities [136]. The impact of this initial expert selection is ambiguous, but we expect that our analysis may benefit from a heightened focus on valuable contributions. Second, white hats are starting to use alternative platforms such as Vulbox which do not have a public disclosure policy. As Wooyun remains the dominant platform for Chinese website vulnerabilities, we anticipate that the latter effect is relatively small.

5.2.2.2 HackerOne

HackerOne is a US-based bug bounty platform started in November 2013. As of early August 2015, it facilitates 99 public bug bounty programs for global companies such as Yahoo, Mail.ru and Twitter. Unlike Wooyun, white hats on HackerOne can only submit reports for these organizations. HackerOne also hosts invitation-only programs. To be eligible white hats must reach a reputation score threshold. Similar programs, such as BugCrowd also separate bounty programs into public and invitation-only [133]. Unfortunately, invitation-only programs cannot be accessed publicly, so our dataset only includes public programs.

Our HackerOne dataset includes contributions from 1,653 white hats. An organization can either reward white hats with reputation scores or monetarily compensate them. Unlike Wooyun, HackerOne does not have a delayed public disclosure policy. A vulnerability report can only be disclosed if both the white hat and the organization commit to its publication. As a result, only a small fraction (732 of 10,997) of all reports are publicly disclosed. For other reports, we only know limited metadata, including submission times, white hat identifiers, and the names of the affected organizations for each vulnerability. We are able to collect the metadata of 6,876 reports from public bounty programs in total. They constitute 62.5% of all resolved reports. We assume the remainder to be reports for invitation-only programs. In addition, HackerOne hosts bounty programs for several open source software projects, such as Perl, Python, OpenSSL. We exclude 69 reports for these bounty programs since they are not related to web vulnerabilities. Our data includes 3,886 reports with bounties paid during the study period. However, some organizations choose not to disclose the bounty amount; i.e., only 1,638 reports have exact monetary payment information. We calculate the average amount of monetary reward paid by an organization, and refer to this value as the expected reward.
5.3 Results

5.3.1 Vulnerability Disclosure Trends

We first provide an overview of the disclosed vulnerabilities. Since the HackerOne dataset does not include data about the vulnerability type and severity, we will mainly focus on the Wooyun dataset.

5.3.1.1 Number of Vulnerabilities

The number of vulnerabilities accepted by the bug bounty platforms provides an initial overview of the productivity of the web vulnerability discovery ecosystems, and also reflects the time trend of web security. Table 4.1 shows that each of the major bug bounty platforms has published a large number of vulnerability reports. Figure 5.2 further displays the number of vulnerabilities accepted by Wooyun and HackerOne every month.
For Wooyun, the number of vulnerabilities accepted per month continues to grow rapidly in the 5-year span. After an initial growth, the number of vulnerabilities for HackerOne’s public bounty programs is relatively stable at around 400 per month. We suspect that an inclusion of data for invitation-only programs would also result in an upward trend for the HackerOne trajectory.

5.3.1.2 Severity Levels

We break down the overall vulnerability trend on Wooyun by severity in Figure 5.3. While the percentage of low severity vulnerabilities is decreasing, the percentage of published high severity reports is increasing over time. One known reason is the intentional omission of certain low severity reports, as we have discussed in Section 5.2.2.1. It is also possible that white hats are becoming more skilled in finding severe vulnerabilities over time. Another hypothesis is that low severity vulnerabilities are easier to discover and thus are usually reported well before more severe problems. Further investigation of these possible causes would be an interesting research question. Overall, the displayed trend indicates that organizations inside this ecosystem are still at risk, and more efforts from both the white hat community and the involved organizations are required.

5.3.1.3 Vulnerability Types

We next examine vulnerability reports on Wooyun according to their types. Figure 5.4 shows the trend for the top 3 most common vulnerability types. While the percentage of XSS reports is decreasing (possibly due to filtering as mentioned previously), we observe a small relative increase of SQL injection reports. The high amount of XSS is expected for web applications; other platforms, such as BugCrowd, have also reported that XSS is the most common vulnerability type (17.9%) [133]. In contrast, the high amount of SQL injection vulnerabilities on Wooyun is particularly surprising, since SQL injection vulnerabilities are not common on other platforms such as BugCrowd (only 1.3%) [133]. A recent study also reveals that many Chinese websites are generally less secure [134]. However, the observed differences could also be caused by the particular organization participation model of Wooyun, which is able to cover much more poorly secured websites. We will discuss more on this in Section 5.4.4.

Figure 5.5 further shows the number of published reports, and the breakdown in severity categories for all vulnerability types on Wooyun. The distribution across vulnerability
types is comparable to other sources [2, 133]. We also observe that some types have a larger proportion of high severity vulnerabilities; for example, SQL injection attacks and malicious file uploads may frequently open up a direct pathway to sensitive data.

In summary, data from bug bounty platforms can be used to meaningfully aggregate valuable security information. Disclosing such information, even at the aggregate level, can help the defense side to update its strategies and to allocate resources against different types of threats.

### 5.3.2 The White Hat Community

In this section, we first look at the size and growth of the white hat communities on Wooyun and HackerOne. Then, we discuss significant differences regarding productivity and accuracy among white hats using the two datasets. Next, we investigate different
skills and strategies of white hats. Finally, we analyze how disclosure of reports can have positive effects on the white hat community.

5.3.2.1 Size and Growth

The outcome of a web vulnerability discovery ecosystem is closely related to the size of the white hat community, who is the “supplier” of vulnerability reports. Table 4.1 shows that these ecosystems have accumulated large white hat communities with tens of thousands of contributors, who may come from all over the world [132, 133]. Later, we will analyze how the size and the diversity within the white hat community correlate with vulnerability discovery outcomes.

We first examine how the size of the white hat community changes over time, using two metrics: the number of white hats who reported at least one vulnerability in each
month (active white hats), and the number of white hats who submitted their first vulnerability in each month (new white hats). The difference between the number of active white hats and the number of new white hats is the number of repeat contributors. We report these two metrics for Wooyun and HackerOne in Figure 5.6. For Wooyun, the number of active white hats per month gradually grows to 700 per month. The number of new white hats per month is about 200 in the past 2 years, which means that there is a relatively constant flow of newcomers joining the ecosystem. The trend for the public programs of HackerOne is similar. In summary, both platforms attract a relatively constant number of white hats who contribute in a given month, while the overall size of the white hat community keeps increasing.

5.3.2.2 Productivity and Accuracy

While the size of the community matters, we also care about the individual productivity of a white hat, i.e., the number of vulnerabilities found by each white hat. In Figure 5.7, we plot the distribution of vulnerabilities found by individual white hats on both Wooyun and HackerOne. We observe that the distributions on both platforms are very skewed. Of 7,744 white hats on Wooyun, the top 1 has found 521 vulnerabilities, the top 100 have published more than 147 reports per person on average, but 3725 of the white hats have contributed only once. Similar observations can be made for white hats on HackerOne. Such long-tail pattern has also been found in other domains, such as scientific productivity [137].

Another important aspect associated with productivity is accuracy. Many existing public bounty programs have complained about the low signal-to-noise ratio and the effort required to deal with a large amount of invalid reports, which generally include duplications, non-security issues, out-of-scope, false positives, or even spam [132, 133, 138, 139]. The signal-to-noise ratio is roughly 20% for platforms such as HackerOne and BugCrowd, and even lower for individually hosted bounty programs by Facebook and Github [133, 139]. In addition, HackerOne has reported that in general more productive researchers have a higher signal-to-noise ratio [139]. Based on [139], we estimate that the top 1% researchers on HackerOne have an average ratio of 0.54, while the bottom 50% only have an average ratio of 0.03, indicating that approximately among 100 reports submitted by them, only 3 are expected to be valid vulnerability reports. Bug bounty platforms have introduced various data-driven approaches, including reputation systems and rate limiting, to improve the signal-to-noise ratio [139]. This partly explains the
higher signal-to-noise ratio of bounty platforms over individually hosted bounty programs. However, the low signal-to-noise ratio remains a key challenge for effective vulnerability discovery and requires more research effort.

The long-tailed distribution of contribution levels as well as concerns about accuracy lead to an increased focus on the top contributors in today’s bug bounty programs, since they are on average much more productive, and more accurate. As a result, existing bounty platforms such as HackerOne and BugCrowd have created private bounty programs that only invite a small number of top contributors [133, 138, 139]. In some cases, the top contributors were directly hired by organizations or bounty platforms [132, 140].

Less attention is given to white hats with lower productivity. However, taken as a group, they contribute a sizable number of accepted reports. As such, the question arises how to evaluate their contributions. To do an initial comparative assessment, we split the white hat community on Wooyun into three groups of different levels of productivity. The two thresholds, displayed in Figure 5.7, are chosen so that the three groups have approximately the same number of reports, thus allowing us to compare other dimensions of their contributions.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Productivity Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High</td>
</tr>
<tr>
<td># white hats</td>
<td>142</td>
</tr>
<tr>
<td>Total # vuln.</td>
<td>17,611</td>
</tr>
<tr>
<td>Average # vuln.</td>
<td>124</td>
</tr>
<tr>
<td># contributed org.</td>
<td>4,727</td>
</tr>
<tr>
<td>Alexa 1-200 (%)</td>
<td>32.5</td>
</tr>
<tr>
<td>Alexa 201-2000 (%)</td>
<td>32.4</td>
</tr>
<tr>
<td>Alexa &gt; 2000 (%)</td>
<td>33.7</td>
</tr>
<tr>
<td>Severity High (%)</td>
<td>38.4</td>
</tr>
<tr>
<td>Severity Medium (%)</td>
<td>31.3</td>
</tr>
<tr>
<td>Severity Low (%)</td>
<td>25.1</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison across three white hat groups of different productivity levels on Wooyun.

We report the results in Table 5.2. Unsurprisingly, the average number of accepted reports differs substantially across these groups. In contrast, an interesting observation is that the less productive groups have contributed reports for a considerably larger number of organizations. There could be multiple reasons to explain this difference. First, the less productive groups have many more white hats, leading to more “manpower” and
more diverse interests covering a wider range of websites. Meanwhile, white hats in the highly productive group have more limited attention or may benefit from an increased focus on a specific set of websites. Second, some websites may have been particularly popular targets for white hats, and easy-to-be-found vulnerabilities are already removed. For many low productive white hats who may also have limited expertise, spending effort on such websites might not be cost-effective. Thus, they shift their attention to other websites, which are more likely to yield discoveries.

The broader coverage of websites by less productive white hats has a positive impact on the security of the Internet, since even less popular sites still receive a considerable amount of visitors every day. In addition, the security of organizations is rather connected in many ways [112, 141]. For example, a user could use the same username and password across multiple sites, and the compromise of one of them will jeopardize others. Therefore, by complementing the limited attention of top white hats, the less productive white hat groups make different but important contributions.

We further break down the contributions of each group by target websites’ popularity and by vulnerability severity. Rows 5 - 7 of Table 5.2 show that for the different popularity categories the contributions (in %) across the three productivity groups are remarkably consistent. In particular, the least productive group also reports a significant percentage of discoveries for popular websites. Row 8 shows that more productive white hats have a larger percentage of contributions with high severity vulnerabilities, but 28.1% of high severity vulnerabilities were still discovered by the least productive white hats.

In summary, the results support the existence of a substantial expertise and productivity gap on an individual level, but from a collective perspective the difference is smaller than perhaps expected. How to better utilize the potential of these different groups of white hats is an interesting challenge. In particular, it would be useful to think about how to boost the productivity of less productive white hats through better incentives, training, and other measures.

5.3.2.3 Skills and Strategies

Next to productivity, we measure two additional metrics: the number of different organizations an individual white hat investigated, and the number of different vulnerability types an individual white hat reported. These two metrics partly reflect the skills, experiences and strategies of white hats. Figure 5.9 shows the distribution of these two metrics for white hats on Wooyun with more than 5 discoveries. The average number of
organizations investigated by a white hat of this group is 18, while the average number of vulnerability types found is 7. The most productive individuals (i.e., red triangles in the figure) generally surpass others in both metrics which partially explains the productivity difference. First, top white hats’ broad knowledge of different types of vulnerabilities may enable them to discover more vulnerabilities. Second, they may find more vulnerabilities because their strategy is to investigate a larger number of websites. Furthermore, we hypothesize that there is a trade-off between exploration vs. exploitation: to find more vulnerabilities, a white hat must develop a good balance between spending effort at one particular website and exploring opportunities on other sites. However, different successful strategies coexist. For example, our dataset includes several white hats in the bottom left corner of Figure 5.9 that is much more focused on exploitation. Similarly, a white hat named ‘meals’ ranked 4th on HackerOne only focuses on Yahoo’s bounty platform, and has to-date found 155 vulnerabilities.

Investigating the optimal degree of strategy diversification during web vulnerability hunting is an interesting area for future work.

5.3.2.4 Disclosure and Learning

A primary consideration of previous research was to understand how vulnerability disclosure pushes software vendors to fix flaws in their products [39, 142]. However, when considering the whole ecosystem, we question whether vulnerability disclosures also have positive effects on the white hat community itself. One possible effect is to enable white hats to learn valuable technical insights and skills from others’ findings. Another effect is to obtain valuable strategic information for their own vulnerability discovery activities, such as which organizations to investigate. Both effects likely improve white hats’ productivity and accuracy. In addition, the software engineering community and peer organizations can also learn valuable lessons from vulnerability reports to avoid making similar mistakes in the future. While the latter factor may be of high practical relevance, we are unaware of related research. In this paper, we investigate the first effect using data from Wooyun.

The Wooyun platform allows white hats to mark and follow a particular report. Therefore, we can use the number of followers of a vulnerability report as an approximate indicator of its learning value to white hats. In Figure 5.8, we plot the distribution of this follower count for all vulnerabilities on Wooyun, and also break down the data by different severity levels. We observe that the distribution is very skewed. There are
Figure 5.9: Scatter plot of white hats’ vulnerability type count and targeted organization count on Wooyun. Each dot represents a white hat who has found more than 5 vulnerabilities in total. The red triangle dots are white hats of the high productivity group defined in Section 5.3.2.2.

9,489 reports that have at least 10 followers, indicating that white hats have been actively learning from a broad portion of reports. On average, high severity vulnerabilities have more followers, which is not surprising, as more severe vulnerabilities tend to have a more significant security impact, and higher discovery and exploit complexity. What might be counter-intuitive is that some low severity vulnerabilities still receive more than 100 followers.
To examine why some vulnerabilities have received much more attention than others, and why some low severity vulnerabilities are followed by many, we selected the 30 most followed vulnerabilities from each severity level. We then manually examined these 90 vulnerabilities. We find that these vulnerabilities mostly belong to one or more of the following categories: (1) Vulnerabilities with significant impact (e.g., with a potential for massive user data leakage, or an XSS inside the site statistics javascript code from a major search engine company); (2) Vulnerabilities that are associated with novel discovery or exploitation techniques; (3) Vulnerabilities of widely used web applications, such as CMS; (4) Vulnerabilities that are explicitly organized as tutorials. We found 21 such tutorial-style reports belonging to a series about XSS, which are all of low severity, yet they still receive a lot of attention because of the emphasis on learning. We also examined a subset of disclosed reports from HackerOne and have discovered that some organizations make disclosures\(^3\) to teach the writing of concise reports.

In summary, our analysis provides evidence of how white hats are learning from vulnerability reports; a typically overlooked benefit of vulnerability disclosure to the white hat community. We will discuss additional facets of disclosure in Section 5.4.1.

### 5.3.3 Organizations

We now shift our focus to the organizations who have participated in vulnerability discovery ecosystems. These organizations harvest vulnerability reports from the white hat community, fix security flaws, and thereby ultimately improve the security of the whole Internet (e.g., by reducing the impact of security interdependencies [112, 141]). However, collecting data about them is non-trivial because many organizations, such as banks, are still reluctant to collaborate with white hats due to various concerns [143]. In addition, for many organizations who joined platforms such as HackerOne, data about discovered vulnerabilities, monetary rewards and other important factors is often not publicly disclosed.

Wooyun provides a valuable opportunity to study the impact of such ecosystems on organizations; and not only because of the existence of the delayed public full disclosure policy. More importantly, an organization is rather coerced to join this ecosystem once a white hat publishes a vulnerability on Wooyun affecting the organization. This coercive model is different from most other platforms which only host bounty programs for

---

\(^3\)For example: https://hackerone.com/reports/32825.
organizations that agree to participate (i.e., voluntary model). Due to the diversity of the large white hat community, Wooyun covers a broad range of organizations from many sectors, as we will show in Section 5.3.3.3. As a result, observations made from this dataset do not only help us understand the web vulnerability discovery ecosystem in China, and the general security status of the Chinese web, but also help us to envision the impact of the bug bounty model for organizations in other parts of the world.

### 5.3.3.1 Size and Growth

Table 4.1 lists the number of organizations participating in representative vulnerability discovery ecosystems. We observe that Wooyun affects a larger number of organizations compared with US-based platforms, who typically have tens or hundreds of participating organizations. The difference is partly due to the coercive versus voluntary ways of involving organizations. Therefore, the Wooyun ecosystem roughly represents an upper
bound of coverage (growth) for other ecosystems. We also investigate the trajectory of
the growth of the number of organizations covered on Wooyun. Figure 5.10 shows that
in every month, there are about 300 organizations benefiting from white hats’ efforts.
Around 150 of them are new organizations, which implies that the white hat community
is continuously broadening its horizon. It would be interesting to understand whether
this effect relies on the fact that new businesses are founded (or new websites become
public), or that white hats are moving to already established but previously unresearched
websites.

5.3.3.2 Vulnerability Distribution

For both Wooyun and HackerOne, Figure 5.11 shows that only few organizations re-
ceive a high number of vulnerability reports, while most organizations receive very few
vulnerability reports. We hypothesize that the number of vulnerabilities received by
organizations is related to multiple factors, such as the complexity of the web system,
the existence of monetary incentives, the popularity of the website, etc. We will further
investigate the relation between these factors and the number of published vulnerability
reports in Section 5.3.3.7.

5.3.3.3 Impact on Different Sectors

To investigate the diversity within participating organizations, we have manually tagged
organizations on HackerOne based on their business types. We find that all participating
companies are IT-focused and cater to different business/consumer needs which are
shown in Table 5.3.

<table>
<thead>
<tr>
<th>Business Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>social network</td>
<td>13</td>
</tr>
<tr>
<td>security</td>
<td>9</td>
</tr>
<tr>
<td>content sharing</td>
<td>9</td>
</tr>
<tr>
<td>payment</td>
<td>8</td>
</tr>
<tr>
<td>communication</td>
<td>8</td>
</tr>
<tr>
<td>bitcoin</td>
<td>6</td>
</tr>
<tr>
<td>cloud</td>
<td>5</td>
</tr>
<tr>
<td>customer management</td>
<td>5</td>
</tr>
<tr>
<td>site builder</td>
<td>5</td>
</tr>
<tr>
<td>finance</td>
<td>4</td>
</tr>
<tr>
<td>ecommerce</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.3: Frequency of IT-business types within the group of publicly available bounty
programs on HackerOne. Only tags with frequency greater than 3 are shown.

Due to its coercive model for involving organizations, the Wooyun dataset includes
a larger and more diverse set of organizations (see Figure 5.12). Further, it shows that
white hats do not exclusively focus on certain business sectors.
For non-IT organizations, two sectors with many vulnerability reports are government and finance. We consider this finding surprising since these sectors have robust incentives for security investments. While the finance sector, and possibly the government sector as well, are often not willing to collaborate with non-commercial white hats [143], we infer from the Wooyun data that they can disproportionally benefit from the involvement of the white hat community. Participating in disclosure programs may also reduce the likelihood that vulnerabilities flow into the black market [33].

Portal sites, telecommunication and e-commerce organizations have the highest number of vulnerability reports in the IT-sector. A possible explanation is that web services and systems from these domains are large and complex which increases the amount of latent vulnerabilities. Further, these companies serve substantial user populations which increases their desirability for vulnerability researchers.

5.3.3.4 Response and Resolution

After the initial submission of a vulnerability report, the typical follow-up process on most bounty platforms contains the following steps: triage/confirm, resolve, and disclose. During this process, the white hat and the security/development team of the organization may collaborate together to address the identified problem. A delayed response likely increases the risk of a security breach, since it increases the time frame for rediscovery of the vulnerability, and stealthy exploitation of stockpiled vulnerabilities by malicious agents. Given its full disclosure policy, a delayed response to a submission on Wooyun may be even more serious because details will be disclosed publicly after 45 days.

Our data allows us to examine how organizations respond and resolve vulnerability reports in the studied ecosystems. HackerOne maintains a detailed handling history for each vulnerability report. Unfortunately, only a small portion of all resolved reports (732 of 10,997) are publicly disclosed. For these disclosed reports, we determined the time distribution for three types of response activities (see Figure 5.13). The median time for the first response (e.g., a confirmation of receiving the report) is 0.18 days, and the median time for triage is 0.88 days. The median resolve time is 6.49 days, and 75% of the disclosed reports are resolved in 25 days. However, one should be cautious when generalizing from these observations since the data is possibly biased. Particularly, the analysis likely underestimates the time required for triaging and resolving vulnerabilities, since the organizations that are willing to disclose vulnerabilities may be more efficient in handling reports and may have more experience in running bounty programs.
Figure 5.13: Boxplots for the time of three types of response activities based on publicly disclosed reports on HackerOne.

Wooyun shows four types of responses by organizations: confirmed by organization (CO), confirmed and handled by a third party such as CNCERT (CT), ignored by the organization (IG), and no response (NO). Since all reports are classified in this way, the Wooyun response data is considerably larger, but provides less details. For example, it is difficult to discern whether the organization eventually fixed the vulnerability (or not), but the first two types of response can serve as an indication that the organizations recognizes the problem. The third type of response suggests that the organization considers the vulnerability report invalid. The fourth type means that the organization did not respond to the report at all. We use the count of the fourth type as a rough estimate for the number of cases when an organization fails to address a vulnerability report, and consider the other three types of responses as situations when the vulnerability is likely being handled.

<table>
<thead>
<tr>
<th></th>
<th>CO</th>
<th>CT</th>
<th>IG</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall (%)</td>
<td>40</td>
<td>34</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>Organizations:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Alexa 1 - 200 (%)</td>
<td>71</td>
<td>13</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>- Alexa 201 - 2000 (%)</td>
<td>57</td>
<td>18</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>- Alexa &gt; 2000 (%)</td>
<td>28</td>
<td>44</td>
<td>1</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 5.4: Percentages of different types of responses by organizations on Wooyun.

Table 5.4 shows the percentages for the different types of response as a breakdown by the popularity of the websites. We observe that overall, the majority (77%) of the
vulnerability reports have been handled. Popular websites address more vulnerabilities by themselves, while less popular websites rely more often on third parties. In addition, less popular websites have a higher rate of no response, possibly due to limited resources for vulnerability management.

5.3.3.5 Monetary Rewards

We also examine the role of monetary rewards offered by some organizations. We observe that in their absence, white hats still make contributions to Wooyun and HackerOne for the purpose of making the Internet safer and for reputation gains. For example, 33 of the public programs on HackerOne do not provide monetary rewards, yet they still have received 1201 valid reports from the white hat community. But as Table 4.1 shows, most platforms offer monetary rewards as an additional incentive for white hats to contribute their time and expertise.

We conduct a preliminary analysis based on the disclosed bounties for public programs on HackerOne. Given a total of 3886 bounties, 1638 have the amount information disclosed. The maximum bounty is $7560, paid by Twitter, and the average bounty amount is $424 which varies considerably by organizations. Yahoo pays $800 on average, followed by Dropbox ($702) and Twitter ($611). We hypothesize that the current reward level is attractive to many white hats, and we explore this topic in more detail with a regression study in Section 5.3.3.7.

5.3.3.6 Improvements to Organizations’ Web Security

The participation in a bug bounty program should over time improve the web security of an organization in a noticeable way. In particular, it is reasonable to expect that the number of latent vulnerabilities in an average organization’s web systems (and the stockpile of web vulnerabilities held by black hats) would gradually diminish. Our data allows us to investigate whether the number of vulnerability reports per month is changing over time which is a relevant metric in this context. Moreover, it is a type of analysis that can be conducted by external evaluators if the bug bounty program is public, and provides stakeholders an indication of the web security of an organization. For example, the Cobalt bounty platform offers security seals for organizations that use their services, which is expected to improve the public perception of the organization’s security [130]. Other services (e.g., cyber-insurance companies), can also benefit from
such security assessments (e.g., for the determination of insurance premiums) [118, 144].

Figure 5.14: Trend of vulnerability report count for three organizations on Wooyun.

To initially explore this question, we show the vulnerability report trends for three large organizations on Wooyun in Figure 5.14. While one notices a slight decreasing trend for Tencent, it is hard to observe a clear tendency for the other two organizations. More importantly, Wooyun may not exclusively host these organizations’ bounty programs which could influence the analysis.

Figure 5.15: Trend of vulnerability report count for three organizations on HackerOne.

In contrast, HackerOne is tasked to exclusively host bounty programs for participating organizations which ensures a more reliable analysis. We show the vulnerability trends for the three organizations with the most vulnerabilities on HackerOne (Figure 5.15).
Interestingly, these organizations have received a large volume of vulnerability reports right after the launch of their bounty programs. We propose three possible explanations. First, the monetary compensation offered by HackerOne provides stronger incentives for white hats to compete for vulnerability discoveries in the early stage of a bounty program since the bounty program only rewards the first discoverer. Second, the target range for white hats on HackerOne is much more limited compared to Wooyun, thus concentrating white hats’ focus. Third, some white hats might have stockpiled vulnerabilities to offload them for reward in anticipation of the opening of new reward programs. After these initial spikes, the number of vulnerability reports on HackerOne drops significantly, possibly because the difficulty of finding new vulnerabilities is increasing. However, even though we observe decreasing trends, these organizations still receive a positive number of vulnerability reports every month. These additional discoveries may either be related to further latent vulnerabilities in existing code or stem from new code. Therefore, we suggest that organizations continuously collaborate with white hats.

To further examine the vulnerability trends for organizations, we apply the Laplace test [38] to the vulnerability history of organizations who have received at least 50 reports and have a bounty program for more than 4 months. We also excluded data before 2012-02 and 2014-02 (the initial growth periods), for Wooyun data and HackerOne data, respectively. This test indicates whether there is an increasing trend, a decreasing trend, or no trend for the number of reported vulnerabilities for a given organization (Table 5.5).

<table>
<thead>
<tr>
<th>Platform</th>
<th>Decrease</th>
<th>Increase</th>
<th>No Trend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wooyun</td>
<td>11</td>
<td>81</td>
<td>17</td>
</tr>
<tr>
<td>HackerOne</td>
<td>32</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.5: Trend test results for organizations on Wooyun and HackerOne. The confidence level is 0.95.

Only 11 of the 109 organizations on Wooyun (which match the criteria) fit a decreasing trend, while most selected organizations have an increasing trend for the number of vulnerability reports. The data omission bias discussed previously could be one reason of the result. A sufficiently large pool of latent vulnerabilities in combination with increasing activity on Wooyun could serve as an alternative explanation. For organizations on HackerOne, 32 of 49 have a decreasing trend indicating a positive effect of the vulnerability discovery ecosystem.

The trend test, however, cannot completely assess the web security status of an
organization for several reasons which we have partly discussed above. Further, as a possible part of their vulnerability discovery strategy (see Section 5.3.2.3), white hats might switch to new organizations or newly deployed web systems which are expected to have more low hanging fruits. In general, we suggest that a reliable assessment requires careful modeling and statistical analysis of the whole ecosystem which is an important area for future work.

5.3.3.7 Attracting Vulnerability Reports

How can an organization harvest more vulnerability reports from the white hat community to improve its web security? To address this question, we first study the correlation between the number of vulnerability reports per organization and the number of contributing white hats.

![Scatter plots of organizations’ white hat count and vulnerability count for Wooyun and HackerOne public programs (excluded Yahoo and Mail.ru as outliers).](image)

Figure 5.16: Scatter plots of organizations’ white hat count and vulnerability count for Wooyun and HackerOne public programs (excluded Yahoo and Mail.ru as outliers).

Figure 5.16 plots the number of white hats that have made at least one discovery, and the number of vulnerabilities, for each organizations (with at least 20 vulnerability reports) on Wooyun and HackerOne. We observe very strong positive linear (Pearson) correlations for these measures (as shown in the figures). Therefore, the following strategies are likely beneficial: (1) While paying special attention to top contributors is a useful strategy, it is
also important to increase the total number of contributors. A possible reason to explain the observed effect is that vulnerability discovery requires diversity, i.e., investigators with different expertise using different tools may find different vulnerabilities; (2) It is important to incentivize new participation, for example, by offering an extra bonus (e.g., badge or money) for the first valid submission of a white hat to a platform or specific program.

Other factors such as the popularity of the target, the expected bounty amount, and the number of alternative choices are all related to a bounty program’s attractiveness to white hats. To better understand these factors, we conduct a linear regression by taking the number of vulnerability reports as the dependent variable and other factors as independent variables, as the following equation shows:

\[
V_i = \beta_0 + \beta_1 R_i + \beta_2 A_i + \beta_3 M_i + \epsilon_i
\]

where for each organization, \(V_i\) is the average number of vulnerabilities per month, \(R_i\) is the expected reward, \(A_i\) is the log Alexa rank of \(i\)'s website, and \(M_i\) is the average platform manpower during the lifetime of organization \(i\)'s bounty program. \(M_i\) is defined as the time-weighted number of white hats divided by the time-weighted number of peer organizations during the lifetime of \(i\)'s bounty program:

\[
M_i = \frac{NW_i T_i + \sum_{k=2}^{T_i} (NW_k - NW_{k-1})(T_i - k + 1)}{NO_i T_i + \sum_{k=2}^{T_i} (NO_k - NO_{k-1})(T_i - k + 1)}
\]

Here, \(T_i\) is the number of months for \(i\)'s bounty program. \(NW_k\) and \(NO_k\) are the accumulated number of white hats and the number of peer organizations on the whole platform at the \(k\)th month for organization \(i\), respectively.

Table 5.6 shows three variations of the regression model. In all three models, we find a highly significant positive correlation between the expected reward offered and the number of vulnerabilities received by that organization per month. Roughly speaking, a $100 increase in the expected vulnerability reward is associated with an additional 3 vulnerabilities reported per month. We also find a significant negative correlation between the Alexa rank and the number of vulnerabilities in models (2) and (3) suggesting that rank determines the attractiveness of a website to white hats. However, it is also possible
<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. Bounty ($R_i$)</td>
<td>0.04***</td>
<td>0.03***</td>
<td>0.03***</td>
</tr>
<tr>
<td></td>
<td>(0.01)</td>
<td>(0.01)</td>
<td>(0.01)</td>
</tr>
<tr>
<td>Alexa [log] ($A_i$)</td>
<td>-2.52*</td>
<td>-2.70**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1.20)</td>
<td>(1.21)</td>
<td></td>
</tr>
<tr>
<td>Platform Manpower ($M_i$)</td>
<td>10.54</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(10.14)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>3.21*</td>
<td>16.12**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1.88)</td>
<td>(6.39)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-133.05</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(143.66)</td>
<td></td>
</tr>
<tr>
<td>R-squared</td>
<td>0.35</td>
<td>0.39</td>
<td>0.40</td>
</tr>
</tbody>
</table>

Standard errors in parentheses  
*** p<0.01, ** p<0.05, * p<0.1

Table 5.6: Results of regression analysis. There are 60 observations (HackerOne).

that less popular websites are in general less complex in design and implementation, and thus contain less vulnerabilities. For model (3), we expect that with higher average platform manpower, an organization will receive more attention from white hats and thus will have more vulnerability reports. However, the analysis does not yield a conclusive answer, possibly due to the omission of invitation-only programs and limited sample size.

The quantified model can be used by organizations when determining their bug bounty policies and attracting an effective white hat following. In particular, offering higher rewards and running the program for a longer time contributes to a higher number of reports. The model also contributes to the security assessment question in Section 5.3.3.6. Nevertheless, our regression model is only a first step towards modeling the dynamics of the web vulnerability discovery ecosystem. It could be extended with more independent variables, such as the business type of organizations (see Section 5.3.3.3), or the expected rewards from peer organizations in the ecosystem.

5.4 Discussion

5.4.1 Importance of Disclosure

Based on our analysis, we believe that disclosing important information about vulnerability discovery (such as the resolve time for each vulnerability, bounty amounts, and even
the detailed reports) is important for the success of a web vulnerability discovery ecosystem. For the white hat community, disclosing more vulnerability information not only enables them to learn and improve, but also potentially allows to make better decisions on target selection, as we have discussed in Section 5.3.2.4. The transparency associated with disclosure could also reduce conflicts between organizations and white hats on issues like the validity of a report or the reasonableness of a bounty amount. For organizations, disclosing more information enables the public (e.g., Internet users, or cyber-insurance providers) to better assess the security of an organization (Section 5.3.3.6). Disclosure is also vital for the research community to tackle some of the challenging issues and future research questions we have discussed. In addition, a platform such as Wooyun with a delayed full disclosure policy also pushes organizations to fix their reports sooner.

However, there are also potentially less desirable consequences of disclosing vulnerabilities about organizations’ web systems, such as the leakage of critical information that can be utilized by black hats. An ideal disclosure policy has to balance the potential benefits and disadvantages to the ecosystem or a specific organization. Several disclosure programs are moving towards this direction. For example, some programs on HackerOne disclose only a subset of their vulnerabilities to the public. The Github bounty program discloses data about every vulnerability discovered by white hats, yet intentionally redacts certain details. Further analyzing the benefits and risks of disclosing vulnerability information, and designing improved disclosure policies is important future work.

5.4.2 Potential Incentive Structure Evolution

Our study shows that monetary incentives increase the number of vulnerability reports (Section 5.3.3.7). We anticipate that more organizations will start paying bounties, as more organizations are joining vulnerability disclosure ecosystems and are competing for the limited attention of white hats. The amount of an average bounty will likely rise not only for the purpose of attracting more white hats, but also for compensating the increasing cost incurred by white hats to discover vulnerabilities (e.g., to compete with black hats). In addition, high reward amounts can also be a positive signal of an organization’s security practices to the public, similar to the proposal in [84].

Many organizations will continue to not offer bounties. For small organizations with limited revenues, maintaining a competitive bounty level could be challenging.
It has also been suggested that an organization could start with no bounty first, and gradually increase the reward level, to alleviate the initial surge of reports (including many invalid submissions) [145], as we have shown in Figure 5.15. Organizations that cannot afford paying bounties can resort to other forms of incentives, such as reputation scores, hall-of-fame memberships, or even public disclosure.

### 5.4.3 Encouraging White Hat Participation

Increasing the size of the white hat community allows more organizations to be covered and more vulnerabilities to be found (see Section 5.3.2.2). A larger white hat community might also decrease the cost of running bounty programs for organizations, similar to the increase of supply in any economic market. Therefore, potential regulations that hinder the collaboration between white hats and organizations are likely detrimental. One such example is the proposed update of the Wassenaar Arrangement, which aims to control the export of intrusion software. The utilized overly broad definition of intrusion software could easily limit the participation of white hats [146], particularly considering the global nature of the white hat community [132, 133].

To encourage more white hats to join the ecosystem, organizations can try to offer a first time bonus (see Section 5.3.3.7), organize capture-the-flag activities, etc. In addition, by analyzing the behavioral patterns and dynamics of white hats (e.g., Section 5.3.2.3), bounty platforms can design customized services for white hats, such as target selection or recommender systems, which match white hats’ skills and organizations’ requirements. For this purpose, it would be helpful to further investigate vulnerability discovery by white hats (e.g., tool usage) through interview or survey studies [81].

### 5.4.4 Stimulate Participation by Organizations

Our study results provide incentives for organizations to join vulnerability discovery ecosystems and to benefit from white hats’ efforts. In addition, government agencies such as the Federal Trade Commission also encourage organizations to have a process of receiving and addressing vulnerability reports [147], which can be achieved by running a bounty program.

In our work, we contrast two participation models from the organizations’ perspective. The first one is the *coercive participation model*, represented by China-based platforms such as Wooyun. That is, an organization is coerced to join the ecosystem once a white
hat has submitted a vulnerability for that organization. The second participation model, represented by US-based platforms including HackerOne, is voluntary, i.e., companies explicitly authorize external researchers to study the security of their web systems. Both models have their advantages and disadvantages. Our results show that the first model is capable of covering a wider range of organizations, although varying legal conditions in different countries might not allow for such an approach (see, for example, [148]). Also, this model might allow many severe vulnerabilities to be found earlier in websites that are not willing to participate bug bounty and are poorly secured. This could partly explains the high percentage of SQL injection on Wooyun in Section 5.1. The coercive model might be more attractive to white hats, for example, since they may feel more in control. The second model clearly grows more slowly when considering the number of participating organizations. However, voluntary participation likely encourages a better response behavior to vulnerability reports, as we have discussed in Section 5.3.1.3. To encourage organizations to participate in the voluntary model, future work is needed to identify and address organizations’ concerns including the perceived lack of trustworthiness of the white hat population [143], misuse of automated vulnerability scanners, and time wasted due to false reports [139].

5.5 Conclusion

In this chapter, we have studied emerging web vulnerability discovery ecosystems, which include white hats, organizations and bug bounty platforms, based on publicly available data from Wooyun and HackerOne. The data shows that white hat security researchers have been making significant contributions to the security of tens of thousands of organizations on the Internet.

We conducted quantitative analyses for different aspects of the web vulnerability discovery ecosystem. Based on our results, we suggest that organizations should continuously collaborate with white hats, actively seek to enlarge the contributor base, and design their recognition and reward structure based on multiple factors. We have also proposed future work directions to help to increase the impact and coverage of these ecosystems.
Chapter 6
Modeling and Organizing Bug Bounty Programs

6.1 Introduction

From the empirical study of two major vulnerability discovery ecosystems in Chapter 5, we see that the collaboration between organizations and the white hat hacker community has discovered and fixed significant amount of vulnerabilities. However, we have also identified several frictions between participants that increase the cost of operation for both sides, hinder participation, or even harm the sustainability of the ecosystems.

In this Chapter, we look into two challenges of today’s bug bounty program, duplicated reports and invalid reports. These two types of reports constitute the majority of the reports received by organizations. However, both of them provide little or no value, but can actually significantly increase the cost and risk of collaboration for both hackers and organizations. In Section 6.2, we will have an in-depth analysis of these two issues.

We propose new mechanisms and policies to tackle these two challenges. The duplicated reports problem is partly caused by inefficient allocation of hackers. For example, if an organization with a small attack surface invites too many hackers, then it is very likely that these hackers will contribute a lot of duplicated discoveries. Therefore, we propose a new hacker allocation mechanism that aims to optimize the utilities of both organizations and hackers. This new allocation mechanism applies the vulnerability discovery model proposed in Chapter 4 to predict the outcome of bug bounty under an allocation. Then, it iterates through possible allocations and pick the one with highest utility.
The invalid report problem has multiple causes, which we will further explain in Section 6.2.2. At the core of this problem, we see a misalignment of incentives between organizations and white hat hackers [47]. While an organization only want to receive valid reports at a low cost, many hackers only want to maximize their rewards. Therefore, some hackers adopt a shotgun approach by submitting as many reports as possible, and hoping that some of the reports will hit reward. Unfortunately, such strategy incurs high cost on the organization side, and even forced some organizations to shut down their bug bounty programs.

Existing bug bounty platforms have enforced multiple policies to control the quality of submission [139, 149, 150]. However, while empirical evidences show that these policies have reduced the amount of invalid reports to a certain degree, a comprehensive understanding of their effect is missing. In addition, some of these policies restrict certain hacker’s submission, and potentially hurt the diversity of participants. As we have discussed in Chapter 5, the power of bug bounty partly comes from the fact that it can utilize the wisdom of a diverse set of hackers. To understand the impact of bug bounty policies, we conduct a theoretical analysis based on an economic model. We further propose a new policy that incentivize white hat hackers to spend validation effort at their individual level, thus achieving a balance between diversity and quality. We will summarize our effort in Section 6.4 and refer readers to [47] for the complete result.

6.2 Challenges for Current Bug-Bounty Programs

Our paper discusses two major challenges for today’s bug-bounty programs: a significant amount of duplicated effort, and a high percentage of invalid reports.

6.2.1 Duplicate Reports

A key challenge for bug-bounty programs are duplicate reports. After a bug is first reported to an organization, it usually takes 1 or 2 months for the organization to fix the issue. Therefore, before the fix is completed, other white hats might spend effort to discover and report the same issue. However, according to the present rule, only the first discoverer will be rewarded, while other reporters’ efforts are not properly compensated (and may only receive some small form of recognition). In addition, the organization also needs to spend effort triaging these duplicated reports, and interacting with the reporting
Table 6.1: Percentiles of organizations’ response efficiency on HackerOne (in days) [3].

<table>
<thead>
<tr>
<th></th>
<th>25th</th>
<th>50th</th>
<th>75th</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Response</td>
<td>0.1</td>
<td>0.8</td>
<td>4.1</td>
</tr>
<tr>
<td>Triage</td>
<td>0.4</td>
<td>1.8</td>
<td>7.1</td>
</tr>
<tr>
<td>Bounty</td>
<td>4.6</td>
<td>16.7</td>
<td>52.7</td>
</tr>
<tr>
<td>Fix</td>
<td>4.7</td>
<td>20.9</td>
<td>66.6</td>
</tr>
</tbody>
</table>

white hats.

The percentage of duplicates is quite high for bug-bounty programs, as we can see in Figures 6.1 and 6.2. For Google and BugCrowd, the percentage of duplicates is higher than the percentage of valid reports. The duplicate rate is lower on HackerOne. However, since HackerOne does not force organizations to mark duplication, it is possible that the reported number underestimates the true magnitude of the problem. Unfortunately, we do not know the duplicate rate of Facebook’s bug-bounty program. But Facebook reported one interesting case in an annual report [151]: when messenger.com was launched, within minutes, 15 hackers filed a report for the same Cross-site Request Forgery issue. While this case shows how powerful bug-bounty programs are in finding vulnerabilities quickly, it also demonstrates the inefficiency in utilizing hacker’s valuable manpower.

One obvious way of reducing the number of duplicates is to shorten the time window between a bug being reported and the same bug being fixed. However, addressing a vulnerability usually requires coordination between multiple departments of the affected organization. Each security patch will also need to go through a multi-step process before it can be applied or released. Considering these factors, it would be hard to expect an organization, particularly a larger one, to fix bugs very rapidly. HackerOne has reported several statistics about how long it takes for organizations to respond and fix a bug [3], shown in Table 6.1. As we can see, the median time to address a vulnerability is 21 days, and the 75th percentile is more than 2 months. Within the time period, it is not unlikely for other hackers to find the same bug, particularly if the incentives are high, like in the case of Facebook, which pays more than $1700 per bug on average in 2015 [151].

6.2.2 Invalid Reports

A key contribution of bug-bounty programs is the utilization of the wisdom of the large globally-distributed white hat security research community to find vulnerabilities. Through bug bounty programs, the white hat community has played a significant part
in improving Internet security [20, 32]. However, at the same time, organizations that run bug-bounty programs could be easily overwhelmed by invalid reports, which include spam (i.e., completely irrelevant reports), false positives (i.e., issues that do not actually exist or have no security impact), and out-of-scope reports (i.e., issues that do exist but are explicitly excluded by bug-bounty program rules, for example, such as bugs in software products from recently acquired companies or affiliates). In fact, bug-bounty platforms acknowledge that the key challenge “companies face in running a public program at scale is managing noise, or the proportion of low-value reports they receive” [150].

In practice, the number of invalid reports is significant. Figure 6.1 shows some relevant statistics of two independent bug-bounty programs run by Facebook [151] and Google [152], and two bug-bounty platforms, HackerOne and BugCrowd [46]. We can observe that the percentage of valid reports is lower than 25% for all programs and platforms. The value is particularly low for the independent bug-bounty programs ran by Facebook and Google, and higher for bug-bounty programs hosted on platforms like HackerOne and BugCrowd. One reason behind this difference is that the platforms have better identification practices for participants, accumulate rich data about white hat researchers, and can thus enforce various kinds of quality-control policies. In addition, some bug-bounty programs on these platforms are private, meaning that they are only open to well-established and experienced white hats, who are more likely to submit valid reports.

![Figure 6.1: Comparing the percentages of valid reports and duplicate reports across different bug-bounty programs and platforms. The percentage of duplicate reports for Facebook is unknown. The percentage of duplicate reports for Google is estimated based on [1]. The percentage of duplicates for HackerOne might be underestimated, because program administrators are not required to mark a report as duplicate.](image-url)
We also worked with HackerOne to obtain additional data of its public bug-bounty programs and private programs. Figure 6.2 shows that private programs have a much higher percentage of valid reports (around 45%). However, private programs are not designed to efficiently utilize the wider crowd of white hats, which has also been shown to be a valuable contributor of vulnerability discoveries with higher severity levels [20]. For that reason, even though the public bug-bounty programs of Facebook and Google have a low signal percentage, the absolute number of valid reports submitted to their programs is high. Therefore, these companies have been strongly supporting bug-bounty in the past several years. Figure 6.2 also suggests that since April 2015, there is a continuous decrease of noise, and a continuous increase of valid reports on HackerOne. A main reason behind this trend is HackerOne’s constant effort in improving the signal ratio [139,149,150]. Nevertheless, we see that there is still ample space for improvements, particularly for public programs.

![Figure 6.2](image)

Figure 6.2: Trend of report types for public programs and private programs on HackerOne.

In our paper, we consider noise and informative reports as invalid reports.

Invalid reports may be the result of imprecise research approaches or lack of thorough validation by white hats. For example, some hackers utilize automated vulnerability scanners in the discovery process, which typically have high false-positive rates [129]. Since filtering out false positives is costly, some hackers may prefer to send the outputs of an automated scanner to the bug-bounty program, hoping that some of them may be recognized with a reward. Further, some discoveries may initially appear to be valid, but after further inspection they may fail to prove a genuine security vulnerability. Another important facet is that the hacker should clearly explain in the report what the discovered flaw is, and how it can lead to a security problem. Failure of articulating the issue could result in the report being marked as invalid, or the report being accepted only after
many rounds of communication between the hacker and an organization’s security team. However, writing a clear report takes time and effort from the hackers, who might not be willing to spend the effort.

As a direct response, bug-bounty platforms have started to offer multiple policies that participating organizations can use for reducing the number of invalid reports. For example, HackerOne has introduced “Signal Requirements” and “Rate Limiter” mechanisms, which organizations can use to increase the quality of reports [150]. The former allows only those hackers to submit reports who maintain a given ratio of valid to invalid submissions, while the latter limits the number of reports that a hacker can make in some time interval. These policies aim to incentivize hackers to engage in consistent efforts to validate their reports. According to HackerOne [150], these measures together have decreased the percentage of noise to around 25%.

Unfortunately, policies may also prevent some hackers, who could contribute valid reports, from participating and may force others to waste effort by being overly meticulous. Consequently, strict policies will result not only in a reduced number of invalid reports, but also in a lower number of valid reports. In summary, finding the right policies and their optimal configuration is a challenging problem since white hat hackers need to be incentivized to produce and submit valid reports, but at the same time, discouraged from submitting invalid reports.

6.3 Model 1 - Hacker Allocation

6.3.1 Bug bounty utilities.

We assume that there are unknown bugs\(^1\) waiting to be discovered in a software product (e.g., a website). The organization responsible for this product creates a bug bounty program. It then invites participants from a pool of hackers \(H\) at the start of each time step \(t (t = 1, 2, \ldots)\). We define an allocation plan as a vector \(A = \{H_1, H_2, \ldots\}\), where \(H_t \subseteq H\) is the set of hackers invited for time step \(t\). At the end of \(t\), invited hackers submit \(r_t\) bug reports in total to the program. Among these reports, some are duplicates because multiple hackers could find the same issue. On the other hand, the organization is only interested in unique discoveries, whose number is denoted by \(u_t\), and \(u_t \leq r_t\). \(r_t\) and \(u_t\) are

\(^1\)Usually, bug bounty programs only focus on security bugs, or vulnerabilities. For brevity, we will use the more general word “bug.” In addition, we will assume that all bugs have equal impact. This assumption will be relaxed in future work.
calculated by the bug discovery model to be discussed shortly. The organization rewards each unique bug discovery with bounty $b$ (e.g., average around $424$ in 2015 [20]), and fixes all discovered bugs at the end of each time step. The organization also incurs cost $c_o$ for processing a submitted report. We also assume that the organization gains $V$ value by fixing a bug. However, the value decreases over time since it is more likely that malicious parties will find and exploit the bug. We use $\delta \in (0, 1)$ to model this time discount. We can write the utility function of an organization from bug bounty as

$$U_o = \sum_{t=1}^{\infty} \left( (\delta^{t-1} V - b)u_t - c_o r_t \right).$$  \hspace{1cm} (6.1)$$

Similarly, we assume that it costs $c_h$ for a hacker to find and submit a bug. So the utility function of all invited hackers is

$$U_h = \sum_{t=1}^{\infty} (bu_t - c_h r_t).$$  \hspace{1cm} (6.2)$$

### 6.3.2 Bug discovery model.

To calculate the expected utilities, we first need to establish a bug discovery model. A bug can be represented as a single or a group of inputs that triggers a specific error in the software or hardware system. Since the input space of any non-trivial system is prohibitively large, a hacker usually discovers bugs based on tools with randomization (e.g., fuzzing), heuristics, experiences, and luck. We propose a bug discovery model. We assume that, for bug $i$, each invited hacker discovers it with probability $p_i$ independently in one time step, as long as bug $i$ has not been discovered in previous time steps. Probability $p_i$ not only models the randomness in bug discovery, but also captures the difficulty of discovering a bug. Previous research has shown that bugs have different discovery difficulty [74, 95]. Particularly, we inferred that in practice $p_i$ follows a discrete power law distribution [95]:

$$p_i = \frac{i^{-\alpha}}{\zeta(\alpha)},$$  \hspace{1cm} (6.3)$$

where $\alpha$ is the scaling factor and $\zeta$ is the Riemann Zeta function. $\alpha$ reflects the size of the system’s attack surface and the system’s security quality, and can potentially be estimated from factors like codebase size, maturity of the security development life cycle, etc. Also, we implicitly sort all bugs in descending order of their discovery probability,
so bug 1 is the easiest to be found. In addition, a bug belongs to one type from a set of vulnerability types denoted by $S$ [20]. We assume that the probability of a bug being of type $s$ is $q_s$, where $q_s$ is exogenous and known, and obviously $\sum_{s \in S} q_s = 1$. $q_s$ can be estimated from earlier bug discovery data, obtained through internal security testing or from similar organizations.

$$
\begin{array}{c}
\text{h}_1 \\
\text{s}_1 \\
p_1 \\
\text{b}_1 \\
\text{h}_2 \\
\text{s}_2 \\
p_2 \\
\text{b}_2 \\
\text{h}_3 \\
\text{s}_3 \\
p_3 \\
\text{b}_3 \\
\vdots \\
\text{\text{hackers}} \\
\text{\text{bug types}} \\
\text{\text{bugs}}
\end{array}
$$

Figure 6.3: Illustration of the vulnerability discovery model.

### 6.3.3 Hacker diversity.

Existing literature has revealed that hackers have diverse expertise, use different tools, etc., so they are good at discovering different types of bugs [11, 20, 44, 153]. We let $S_h$ be the set of vulnerability types that hacker $h \in H$ can discover, so the probability that hacker $h$ discovers bug $i$ is $p_i$ if $s \in S_h$, where $s$ is the type of bug $i$, and it is 0 if $s \not\in S_h$. $S_h$ can be obtained from data accumulated on bug bounty platforms. Figure 6.3 illustrates the vulnerability discovery model.

### 6.3.4 Expected utilities.

We calculate expected utilities of the organization and hackers as follows. First, we define $H_{t,s}$ as the set of hackers allocated for time $t$ with the expertise to find bugs of type $s$. In other words, $H_{t,s} = \{h \in H_t | s \in S_h\}$. We then define $ND_{t,i} := (1 - p_i)^{|H_{t,s}|}$ as the probability that none of the hackers in $H_t$ discover bug $i$ at $t$. For the organization, the expected utility $E[U_{oi}|\text{bug } i \text{ is of type } s]$ for discovering bug $i$ of type $s$, and the total expected utility $E[U_o]$ from a bug bounty program given an allocation plan are, respectively:
\begin{align*}
E[U_{oi}] & \text{bug i is of type s} = \sum_{t=1}^{\infty} \left( \prod_{k=1}^{t-1} ND_{i,k} \right) \left( (\delta^{t-1}V - b) (1 - ND_{i,t}) - c_o (|H_{t,s}|p_i) \right) \\
E[U_o] &= \sum_{i=1}^{\infty} \sum_{s \in S} q_s E[U_{oi}] \text{bug i is of type s}. \quad \text{(6.4)} \\
\sum_{i=1}^{\infty} \sum_{s \in S} q_s E[U_{hi}] \text{bug i is of type s}. \quad \text{(6.5)} \\
E[U_h] &= \sum_{i=1}^{\infty} \sum_{s \in S} q_s E[U_{hi}] \text{bug i is of type s}. \quad \text{(6.6)}
\end{align*}

Similarly, for all hackers, we have

\begin{align*}
E[U_{hi}] & \text{bug i is of type s} = \sum_{t=1}^{\infty} \left( \prod_{k=1}^{t-1} ND_{i,k} \right) \left( b (1 - ND_{i,t}) - c_h (|H_{t,s}|p_i) \right) \\
E[U_h] &= \sum_{i=1}^{\infty} \sum_{s \in S} q_s E[U_{hi}] \text{bug i is of type s}. \quad \text{(6.7)}
\end{align*}

Figure 6.4: Expected utilities with different number of hackers. Parameters used: $V = 20, b = 5, \delta = 0.99, c_o = c_h = 1, \alpha = 2.$
6.3.5 Bug bounty optimization.

The practical goal of our work is to help organizations optimize their bug bounty programs. A basic, but often voiced idea is to attract as many hackers as possible. We evaluate this notion using our model and data collected from one bug bounty platform, i.e., Wooyun [20]. Figure 6.4 shows that the expected utilities of the inviting organization and the invited hackers exhibit inverted U-shapes, and do not scale linearly with the number of hackers. Rather, they start to decrease after a certain number of hackers have joined. The reason is that as more hackers are invited, the number of duplicates increases, which raises the cost of processing reports by the organization, and also decreases the expected bounty received by hackers. This result suggests that, for bug bounty programs and possibly for some other crowdsourcing scenarios that require expertise and competition, more participation is not always better. Instead, the bug bounty program shall carefully design its allocation plan to control the competition among participants and to diversify its workforce. In addition, the bug bounty program also needs to offer enough reward for a bug, such that the expected utility of hackers is greater than zero, even as discovering bugs is getting harder over time. We are defining this problem as \( \max_{A, b, E[U_h]} E[U_h] \geq 0 \), which is subject of our ongoing work.

6.4 Model 2 - Quality Control Policies

We now focus on invalid reports, the second challenges for bug bounty programs. As we have discussed in Section 6.2.2, invalid reports account for a large portion of submissions for today’s bug bounty programs, and thus significantly burden participating companies. Bug bounty platforms have worked hard to reduce the number of invalid reports, and have proposed various quality control policies [139, 149, 150]. To evaluate the strengths and weaknesses of these policies, we build the following economic model. In this section, we provide a summary of our model and results. The complete version can be found in [47].

Table 6.2 lists the symbols of this model. The model considers one organization that runs a bug bounty program, and multiple types of hackers that may participate. The utility of hackers of type \( i \) is

\[
U_{H_i}(b, t_i, v_i) = b \cdot \Phi_i \cdot D_i(t_i) - W_i \cdot (t_i + v_i),
\]  

(6.8)
Table 6.2: List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Constants and Functions</strong></td>
</tr>
<tr>
<td>$V$</td>
<td>average value of a valid report for the organization</td>
</tr>
<tr>
<td>$C$</td>
<td>average cost of processing a report for the organization</td>
</tr>
<tr>
<td>$W_i$</td>
<td>value of time for hackers of type $i$</td>
</tr>
<tr>
<td>$\Phi_i$</td>
<td>fraction of discoveries by hackers of type $i$ that are valid vulnerabilities</td>
</tr>
<tr>
<td>$D_i(t_i)$</td>
<td>number of potential vulnerabilities discovered by hackers of type $i$</td>
</tr>
<tr>
<td>$I_i(v_i)$</td>
<td>number of discoveries validated by hackers of type $i$</td>
</tr>
<tr>
<td></td>
<td><strong>Variables</strong></td>
</tr>
<tr>
<td>$b$</td>
<td>average bounty paid for a valid report</td>
</tr>
<tr>
<td>$t_i$</td>
<td>time spent on vulnerability discovery by hackers of type $i$</td>
</tr>
<tr>
<td>$v_i$</td>
<td>time spent on validating discoveries by hackers of type $i$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>accuracy threshold imposed on participating hackers</td>
</tr>
<tr>
<td>$\rho$</td>
<td>report-rate threshold imposed on participating hackers</td>
</tr>
<tr>
<td>$\delta$</td>
<td>validation reward for participating hackers</td>
</tr>
</tbody>
</table>

where $b$ is the average bounty that the organization pays for a valid report, and $W_i > 0$ is the hacker’s utility for spending time on other activities. In other words, $W_i$ is the opportunity cost of the hacker’s time.

The organization’s utility is

$$\mathcal{U}_O(b, t, v) = \sum_i (V - b) \Phi_i D_i(t_i) - C \cdot (\Phi_i D_i(t_i) + (1 - \Phi_i) (D_i(t_i) - I_i(v_i))), \quad (6.9)$$

where $V > 0$ is the average value of a valid report for the organization, and $C > 0$ is the average cost of processing a report. Note that $V$ can incorporate a variety of factors, such as a difference between the processing costs of valid and invalid reports, cost of patching a vulnerability, etc. By letting $\hat{V} = V - C$, we can express the organization’s utility as

$$\mathcal{U}_O(b, t, v) = \sum_i (\hat{V} - b) \Phi_i D_i(t_i) - C \cdot (1 - \Phi_i) (D_i(t_i) - I_i(v_i)). \quad (6.10)$$

We then derive theoretical results for canonical policies.
6.4.1 No-restriction Policy

We first consider a baseline case, in which the bug bounty program enforces no restriction on the submissions. This type of policy is actually used by individually-run bug bounty programs, such as Facebook’s bug bounty program. While this policy can maximize the number of contributions from hackers, they also lead to a very low percent of valid reports (e.g., 4% for Facebook), because hackers have no incentive to validate their submissions at all.

Proposition 6.4.1. Without no-restriction Policy policy, hackers of type \( i \) will spend

\[
t^*_i(b) = \begin{cases} 
(D'_i)^{-1} \left( \frac{W_i}{b \Phi_i} \right) & \text{if } D'_i(0) > \frac{W_i}{b \Phi_i} \\
0 & \text{otherwise}
\end{cases}
\]

\hspace{1cm} (6.11)

time on vulnerability discovery and \( v^*_i = 0 \) time on validating their discoveries.

6.4.2 Accuracy-Threshold Policy

A bug bounty program can also set an accuracy threshold so that only hackers whose accuracy is above the threshold can submit reports. The accuracy of a hacker can be measured based on the hacker’s activity data on the whole bug bounty platform [149]. Existing bug bounty platforms all have private (or invitation-only) programs, which is essentially an accuracy threshold policy. HackerOne further allows a public program administrator to set signal requirements, which can be used to prevent low-accuracy hackers from participating as well [150]. We formally define this policy as follows.

Definition 6.4.1 (Accuracy-Threshold Policy). Under an accuracy-threshold policy with threshold \( \alpha \in [0, 1] \), the hackers’ choices must satisfy

\[
\Phi_i \cdot D_i(t_i) \geq \alpha.
\]

\hspace{1cm} (6.12)

The following proposition characterizes the hackers’ responses to the accuracy-threshold policy when \( \alpha > \Phi_i \) (when \( \alpha \leq \Phi_i \) their responses are obviously the same as without a policy).
Proposition 6.4.2. Under an accuracy-threshold policy, if $\alpha > \Phi_i$, hackers of type $i$ will spend

$$t^*_i(b, \alpha) = \begin{cases} 0 & \text{if } D^*_i(0) \leq \frac{W_i}{b \cdot \Phi_i - W_i I'_i(0)} \frac{\alpha - \Phi_i}{\alpha (1 - \Phi_i)} \\ \tilde{t}_i & \text{otherwise} \end{cases}$$

(6.13)

time on vulnerability discovery, where $\tilde{t}_i$ is the unique solution to

$$D^*_i(\tilde{t}_i) \left( b \cdot \Phi_i - W_i I'_i \left( \frac{1}{I_i^* \left( D^*_i(\tilde{t}_i) \frac{\alpha - \Phi_i}{\alpha (1 - \Phi_i)} \right) \right) \alpha - \Phi_i \right) = W_i.$$  

(6.14)

In addition, they will spend

$$v^*_i(b, \alpha) = I_i^{-1} \left( D_i(t^*_i) \frac{\alpha - \Phi_i}{\alpha (1 - \Phi_i)} \right)$$

(6.15)

time on validating their discoveries.

Note that even though we cannot express the solution of Equation (6.14) in closed form, it can be found easily numerically since the left-hand side is strictly decreasing or negative (see the proof for details). Furthermore, this also holds for the remaining propositions (Propositions 6.4.3 and 6.4.4).

6.4.3 Report-Rate Threshold Policy

An alternative policy of controlling submission quality is to limit the number of reports that each hacker can submit in some fixed time interval. The rate limit for a type of hacker is based its past accuracy, so that more accurate hackers are allowed to submit more reports than inaccurate hackers. HackerOne has implemented such a policy called Rate Limiter primarily for public programs [150]. We formally define this policy as follows.

Definition 6.4.2 (Rate-Threshold Policy). Under a rate-threshold policy with threshold $\rho > 0$, the hackers’ choices must satisfy

$$D_i(t_i) - (1 - \Phi_i) I_i(v_i) \leq \rho.$$  

(6.16)

The following proposition characterizes the hackers’ responses to the rate-threshold
policy.

**Proposition 6.4.3.** Under a rate-threshold policy, hackers of type \( i \) will spend

\[
t^*_i(b, \rho) = \begin{cases} 
0 & \text{if } D'_i(0) \leq \frac{W_i}{b \cdot \rho} \\
D_i^{-1}(\rho/\Phi_i) & \text{if } D'_i(D_i^{-1}(\rho/\Phi_i)) \geq \frac{W_i}{b \cdot \Phi_i - W_i \cdot (D_i^{-1}(\rho/\Phi_i))} \\
\tilde{t}_i & \text{otherwise}
\end{cases}
\]  

(6.17)

time on vulnerability discovery, where \( \tilde{t}_i \) is the unique solution to \( \frac{d}{dt} U_{H_i} = 0 \). In addition, they will spend

\[
v^*_i(b, \rho) = \begin{cases} 
0 & \text{if } D_i(t^*_i) \leq \rho \\
D_i^{-1}(\rho/\Phi_i) & \text{otherwise}
\end{cases}
\]

(6.18)
time on validating their discoveries.

### 6.4.4 Validation-Reward Policy

One of the primary reasons for the large number of invalid reports is the misalignment of incentives: hackers are only interested in increasing the number of valid reports, while organizations are also interested in decreasing the number of invalid reports. Existing approaches try to solve this problem by imposing constraints on the hackers’ choices (e.g., imposing a threshold on their accuracy or on their report rate). Here, we propose a novel, alternative approach, which incentivizes hackers to reduce the number of invalid reports by rewarding their validation efforts. The advantage of this approach is that it does not impose strict constraints on the hackers’ choices, but instead aligns their incentives with those of the organization, and allows the hackers to optimize their productivity.

A validation-reward policy can be formulated in multiple ways. For example, the organization could slightly lower bounties for valid reports, but give a bonus based on the submitter’s accuracy. Alternatively, it could raise bounties, but deduct from the payment based on the submitter’s rate of invalid reports. Here, we will study the latter approach since it allows us to align the hackers’ incentives with those of the organization in a very straightforward way.

In practice, this policy can be easily implemented in the same way as an accuracy or rate threshold, by keeping track of each hacker’s valid and invalid reports. Similar to the rate-threshold policy, we will assume for ease of presentation that each hacker type
contains only a single hacker.

We define the validation-reward policy as follows.

**Definition 6.4.3** (Validation-Reward Policy). Under a validation-reward policy with incentive \( \delta > 0 \), a hacker’s utility is

\[
\mathcal{U}_H(b, \delta, t_i, v_i) = b \cdot \Phi_i \cdot D_i(t_i) - W_i \cdot (t_i + v_i) - \delta \cdot (1 - \Phi_i) \cdot (D_i(t_i) - I_i(v_i)),
\]

and the organization’s utility is

\[
\mathcal{U}_O(b, \delta, t, v) = \sum_i (\hat{V} - b) \Phi_i \cdot D_i(t_i) - (C - \delta) \cdot (1 - \Phi_i) \cdot (D_i(t_i) - I_i(v_i)).
\]

The following proposition characterizes the hackers’ responses to the validation-reward policy.

**Proposition 6.4.4.** Let

\[
\hat{v}_i = \begin{cases} 
0 & \text{if } I'_i(0) \leq \frac{W_i}{\delta(1 - \Phi_i)} \\
\left( I'_i \right)^{-1} \left( \frac{W_i}{\delta(1 - \Phi_i)} \right) & \text{otherwise}.
\end{cases}
\]

Under a validation-reward policy, hackers of type \( i \) will spend

\[
t^+_i(b, \delta) = \begin{cases} 
0 & \text{if } \hat{v}_i = 0 \text{ and } D'_i(0) \leq \frac{W_i}{b \cdot \Phi_i - \delta(1 - \Phi_i)} \\
0 & \text{if } \hat{v}_i > 0 \text{ and } D'_i(0) \leq \frac{W_i}{b \cdot \Phi_i - \frac{W_i}{\hat{v}_i(0)}} \\
\tilde{t}_i & \text{otherwise}
\end{cases}
\]

time on vulnerability discovery, where \( \tilde{t}_i \) is the unique solution to \( \frac{d}{dt} U_{H_i} = 0 \). In addition, they will spend

\[
v^+_i(b, \delta) = \min \left\{ \hat{v}_i, I^{-1}_i(D_i(t^+_i)) \right\}
\]

time on validating their discoveries.

**6.4.5 Comparing Different Bug Bounty Policies**

We summarize the numerical results obtained for these policies in [47]. For the vulnerability-discovery function \( D(t) \), we use an instance of Anderson’s thermodynamic model [?]:
\[ D(t) = \ln(10 \cdot t + 1). \] Note that we added 1 to the argument so that \( D(0) = 0. \) We instantiate the remainder of our model with the following parameters: \( V = 10, \ C = 1, \) and a single hacker type with \( W_1 = 1, \Phi_i = 0.2, \) and \( I_1(v_1) = \ln(20 \cdot v_1 + 1). \)

For homogeneous hackers:

- Under the no-restriction policy, the organization attains maximum utility at \( b = 2.07: \) with lower bounties, hackers dedicate significantly less time to vulnerability discovery (zero time when \( b < 0.31), \) while with higher bounties, the cost of running the program becomes prohibitively high.

- The accuracy-threshold policy is very effective and robust: the organization’s utility increases steeply with the threshold \( \alpha, \) reaches a 70% improvement at \( \alpha = 0.74, \) and declines negligibly after that.

- The rate-threshold policy is considerably less reliable: the organization’s utility is improved by 55% at \( \rho = 0.2, \) but it decreases rapidly as the threshold decreases or increases, and it may reach significantly lower values than without a policy. Thus, the organization must implement this policy with great care in order to avoid suppressing productivity.

- The validation-reward policy is robust: even though the organization’s utility does not increase until the threshold reaches \( \delta < 0.66, \) it increases steeply after that, reaching and maintaining a 69% improvement.

For heterogeneous hackers:

- Similar to the result under homogeneous hackers, the rate-threshold policy must be implemented carefully since overzealous limiting may significantly decrease the organization’s utility, while lenient limiting is ineffective.

- The accuracy-threshold and validation-reward policies have large “plateaus” around the optimal values, which make them more robust to changes in configuration or parameter values. Nonetheless, if the bounty value is very low, even these policies – especially the validation-reward policy – may be too strict and deter hackers from participating.

Finally, we examine the organization’s maximum attainable utility under various policies with two types of hackers. Compared to this baseline, the accuracy-threshold,
rate-threshold, and validation-reward policies can attain 31%, 13%, and 52% improvement, respectively. However, if the bounty value is not high enough, none of the policies can improve the organization’s utility. Finally, offering validation rewards outperforms the other policies significantly, since it is able to incentivize heterogeneous hackers to operate at their individual maxima instead of forcing them towards a uniform strategy.

6.5 Conclusion

In this Chapter, we study two major challenges, duplicated reports and invalid reports, for the collaboration between white hat hackers and organizations. These two types of reports provide little or no value, yet according to our empirical results, they constitute majority of the submissions. We propose a hacker allocation model to tackle the duplicated reports problem. The model defines utilities of both hackers and organizations, and captures the vulnerability discovery process. Based on the model, we propose new allocation algorithms that improve the utilities of both sides. We also propose a second model for evaluating quality control policies that can reduce the number of invalid reports. We show that all of the considered policies may substantially improve an organization’s utility, which explains their widespread use [150]. However, their effectiveness and reliability vary significantly. We found that the rate-threshold policy is not only less effective than the other two, but it must also be configured more carefully. In contrast, the accuracy-threshold and validation-reward policies are less sensitive to changes in parameter and configuration values, and they can also be more effective. However, without adequate bounties, even these policies might “backfire” and actually deter hackers from dedicating time to vulnerability discovery. Finally, we found that the validation-reward policy may significantly outperform the other two when hackers are not homogeneous, since it allows hackers to operate at their individual optima.
Chapter 7  |  Conclusion

In this dissertation, we perform a two-dimensional research study focusing on large-scale collaborative effort for discovering and mitigating zero-day vulnerabilities.

In Chapter 3, we propose the idea of collaborative run-time protection (CRP) to enhance software programs against zero-day vulnerabilities. Theoretical analysis shows that CRP can quickly detect zero-day attacks with a small loss, and a small overhead on each individual instance. We further design and build a prototype system called HeapCRP. Evaluation based on real-world vulnerabilities and SPEC 2006 benchmark show that HeapCRP is effective against real vulnerabilities at a low cost.

In Chapter 4, we have collected empirical data of black-box mutational fuzzing. We show that the fuzzing process can be modeled as a non-homogeneous Poisson process with the rates of individual bugs following a power-law distribution. We then show how to calculate the expected outcome of a fuzzing campaign. We further show that once the vulnerability discovery enters the long-tail, there will be significant diminishing returns, and less order in the bug arrival. These effects pose challenges for the software companies that try to eliminate vulnerabilities before the black hats, and call for collaboration with white hats. Finally, we show that the model can potentially be extended to other vulnerability discovery mechanisms, such as bug bounty programs, that have diversity and randomness.

In Chapter 5, we have studied emerging web vulnerability discovery ecosystems, which include white hats, organizations, and bug bounty platforms, based on publicly available data from Wooyun and HackerOne. The data show that white hat security researchers have been making significant contributions to the security of tens of thousands of organizations on the Internet. We conducted quantitative analyses for different aspects of web vulnerability discovery ecosystems. Based on our results, we suggest that
organizations should continuously collaborate with white hats, actively seek to enlarge the contributor base, and design their recognition and reward structure based on multiple factors. We have also proposed future work directions to help to increase the impact and coverage of these ecosystems.

In Chapter 6, we study two major challenges, duplicated reports and invalid reports, for the collaboration between white hat hackers and organizations. These two types of reports provide little or no value, yet according to our empirical results, they constitute majority of the submissions. We propose a hacker allocation model to tackle the duplicated reports problem. We also propose a second model for evaluating quality control policies that can reduce the number of invalid reports. We show that all of the considered policies may substantially improve an organization’s utility, which explains their widespread use. However, their effectiveness and reliability vary significantly. We found that the validation-reward policy may significantly outperform the other two when hackers are not homogeneous, since it allows hackers to operate at their individual optima.

In conclusion, we provide empirical and theoretical evidences for the effectiveness of large-scale collaboration in discovering and mitigating zero-day vulnerabilities. We further design and implement new systems and policies for improving existing security ecosystems. We believe that software security will increasingly depend on large-scale collaborations between program instances, and between various human participants. We hope our work can facilitate further studies and practices on this topic.
Bibliography


121


[116] JACKSON, W., “Has secure software development reached its limits?” GCN.


Mingyi Zhao enrolled in the Ph.D. program in Information Sciences and Technology at Pennsylvania State University in August 2011. Prior to that, he received his B.S. degree in Computer Science from University of Science and Technology of China in June 2011. His research interests include software vulnerability analysis and mitigation, and security data science. His work has led to 10 peer-reviewed papers published in conferences and journals including ACM CCS, DSN, IEEE TDSC, ESORICS, WEIS, etc. He is a recipient of the 2016 RSA Conference Security Scholarship, and travel grants from ACM CCS 2015, WEIS 2014/2016 and SafeConfig 2012. He is a member of ACM.