Screen Capture for Sensitive Systems

Joseph A. Cooley

Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation

https://digitalcommons.dartmouth.edu/masters_theses/17

This Thesis (Master's) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Master’s Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
Screen Capture for Sensitive Systems

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Masters of Science
in
Computer Science
by
Joseph A. Cooley
(jcool@alum.dartmouth.org)

DARTMOUTH COLLEGE
Hanover, New Hampshire
May, 2011

Examining Committee:

Sean W. Smith, (chair) Advisor, Ph.D.
(sws@cs.dartmouth.edu)

Lorenzo Torresani, Committee Member, Ph.D.

Andrew Gettinger, Committee Member, MD

Ross Koppel, Committee Member, Ph.D.

Brian W. Pogue, Ph.D.
Dean of Graduate Studies
Abstract

Maintaining usable security in application domains such as healthcare or power systems requires an ongoing conversation among stakeholders such as end-users, administrators, developers, and policy makers. Each party has power to influence the design and implementation of the application and its security posture, and effective communication among stakeholders is one key to achieving influence and adapting an application to meet evolving needs.

In this thesis, we develop a system that combines keyboard/video/mouse (KVM) capture with automatic text redaction to produce precise technical content that can enrich stakeholder communications, improve end-user influence on system evolution, and help reveal the definition of “usable security.” Text-redacted screen captures reduce sensitivity of captured material and thus can facilitate timely data sharing among stakeholders.

KVM-based capture makes our system both application and operating-system independent because it eliminates software-interface dependencies on capture targets. Thus, our work can be used to instrument closed or certified systems where capture software cannot be installed or documentation and support lack. It can instrument widely-varying platforms that lack standards-compliance and interoperability or redact special document formats while displayed onscreen.

We present three techniques for redacting text from screenshots and two redaction applications. One application can capture, text redact, and edit screen video and the other can text redact and edit static screenshots. We also present empirical measurements of
redaction effectiveness and processing latency to demonstrate system performance.

When applied to our principal dataset, redaction removes text with over 93% accuracy and simultaneously preserves more than 76% of image pixels on average. Thus by default, it retains more visual context than a technique such as blindly redacting entire screenshots. Finally, our system redacts each screenshot in .1 to 21 seconds depending on which technique it applies.
Acknowledgments

This research results from a research program at the Institute for Security, Technology, and Society at Dartmouth College, supported by the National Science Foundation under Grant Award Number 0910842. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the National Science Foundation. Additionally, Google supported a portion of this work through a gift to Dartmouth College and MIT Lincoln Laboratory supported the author monetarily and in many other ways (which I will address shortly). Thank you sponsors!

There are numerous people to thank and I hope I do not miss anyone! Thank you Professor Sean Smith, my advisor. You have a gift for brainstorming, a rare depth and breadth of technical ability, an approach to thinking about security that is “spot on” for real people and real systems (anyone that wants to really learn security ought to spend time in Sean’s lab!), and you have consistently shown nothing but great patience working with me! You have also provided a lab atmosphere with amazing people and demonstrated openness to ideas, an openness to any student interested in learning, and an extremely hard work ethic in teaching courses. These are all examples for me to strive towards no matter where I end up. Thank you for supporting a non-traditional, “mature” student and providing me the freedoms you have!

Thank you Professor Lorenzo Torresani, Dr. Ross Koppel, and Andrew Gettinger MD for accepting the invitation to my Master’s committee. You provided honest feedback on
the thesis proposal, presentation, and final writeup, and I very much appreciate it. You all helped strengthen this work! Lorenzo, thank you for meeting and providing ideas to improve the computer vision aspect of this work. You provided invaluable pointers and thoughts about the problem that made a significant impact in the final product. Dr. Get-tinger, thank you for taking time away from the busy hospital transition to support me with realistic insight into software problems faced at the hospital. You were willing at any time to meet over the phone or in person to discuss material. Additionally, thank you for the dataset—it was instrumental in completing this dissertation. Ross, thank you for your realistic insights into problems associated with health information technology. In a serendipitous day, you met with our research lab and we found a mutual problem of interest! You helped strengthen this work and give it direction with candid feedback over the phone and in person. You also led me to David Hanauer MD and the second dataset, and traveled no short distance for the final presentation. Thank you! Dr. Hanauer, thank you for the second dataset. It was instrumental in the analysis I conducted for this dissertation. I am very grateful that you responded to a mailing list email and took the time to gather screenshots for me.

None of this degree would have been possible without a myriad of monetary, professional, and personal support from my employer and colleagues at MIT Lincoln Laboratory. Thank you Dr. Bill Keicher, Dr. Ken Estabrook, and the entire MIT Lincoln Scholars committee; Dr. Roger Khazan, my Scholars mentor; and Dr. Rob Cunningham, my Group Leader. You have each supported me throughout this tenure in school. Roger and Rob, I would not have reached the Scholar’s committee without either of you and would have been overwhelmed without your continual support and hands-off approach to this process. I am indebted to the Scholars committee; this academic adventure would have been impossible without your support! I would like to extend a special thanks to Bill and Ken for keeping the quarterly meetings down-to-earth, being flexible with scheduling, and just asking “how’s it going.” There are many others who have helped me along the way with admin-
istrivia, travel, and other miscellaneous needs, and they deserve thanks too! Thank you! I am very privileged to work with so many class acts both personally and professionally at MIT Lincoln Laboratory.

Labmates and pseudo labmates, thank you. Gabe Weaver, thank you for serving as a travel, gym, burger, and research companion at various times. You enlivened my experience at Dartmouth in many ways. Sergey, thank you for your yellow-card discussions, insight and perspective on computer security and life in general, and willingness to discuss any topic at a deep level. You keep the lab on its toes in a good way, are an amazing treasure trove for all those open to learning more, and bring very interesting insights to the table. I am thankful for having met you and Anna and for having learned about many topics through your lens on life. Scout, thank you for devoting time to our philosophical discussions, your willingness to help me, and general interest in my research and its progress. bx Shapiro, Jason Reeves, Ryan Speers, Baljir Baatartogtokh, and other labmates, thank you for the fresh bread, fresh food, companionship, and stimulating conversation in the lab. These are foundations of productive research! Pseudo-labmate, Tom Candon (your office is next door, we see you at lunch and during events, and whenever I randomly stop at your office!), thank you for all of your support and discussion throughout my Dartmouth tenure.

Coursemates, thank you. Ryan Kingston, Adrian Kostrubiak, Robin Chhetri, Rhadika Bhasin, Andrew Cherne, and others, thank you for spending a chunk of the year with me held up in the basement of Sudi studying one course or another, completing homework, and wrapping up projects at wee hours of the night. The lack of sleep has come to fruition, and I have had my fill of burritos, finally!

Thank you to many people in the computer science department at Dartmouth College. The department is small and its members are top notch and personable. People are extremely capable, have an open-door policy, and are very amiable. Sometimes these qualities are a “choose 2 of 3”, but not at Dartmouth College.

Thank you Anne Allen, Georgeanna Hunton, Jannine Walsh, Fr. Beckett, Fr. Maciej,
Fr. Francis and others at St. Denis Catholic Church in Hanover and The Aquinas House at Dartmouth College. You provided warmth, support, and spiritual stimulation many mornings. These are things that keep one refreshed and ready for the day. Anne, I extend a special thanks to you for your continued warmth and kindness over the past two years.

Thank you family for supporting me. Although you all live far from me, your love and support have helped lead me up to, into, and through this degree. You have always been a phone-call away and ready to listen to whatever I have to share. Thank you!

Thank you Ceci, my wife, for supporting me throughout this entire 2-year adventure—in many ways, you attended graduate school with me over the past 2 years! I had school choices closer to home, but decided a 2-hour drive to Dartmouth was the best fit for me, and you supported me unreservedly. Thank you! Being gone week after week was taxing on me and I know you would have preferred a different option too. I am indebted to you for your patience, love, and support, both while away and at home. You gave me the freedom to focus on school and let many things languish around the house! I am forever grateful!

Finally, I AM, I am nothing without you. Thank you for your abundant graces and my transformation over the past two years! I am excited to live the next step in your plan for me!
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Scope and Impact</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Vision</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Putting it All Together</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>7</td>
</tr>
<tr>
<td>1.5 Roadmap</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Related Work</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Screen Capture</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Segmenting Text</td>
<td>10</td>
</tr>
<tr>
<td>2.3 User Studies</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Deidentifying Data</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Anonymity</td>
<td>12</td>
</tr>
<tr>
<td>2.6 Document Analysis</td>
<td>12</td>
</tr>
<tr>
<td><strong>3 System Overview</strong></td>
<td>14</td>
</tr>
<tr>
<td>3.1 Architecture</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Design Goals</td>
<td>16</td>
</tr>
<tr>
<td>3.3 Functionality</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Implementation</td>
<td>17</td>
</tr>
<tr>
<td><strong>4 Screen Capture</strong></td>
<td>19</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>19</td>
</tr>
<tr>
<td>4.2 VNC-based Screen Capture</td>
<td>20</td>
</tr>
<tr>
<td><strong>5 Text Redaction</strong></td>
<td>22</td>
</tr>
<tr>
<td>5.1 Overview</td>
<td>22</td>
</tr>
<tr>
<td>5.2 Redaction Using Canny Edge Detection</td>
<td>23</td>
</tr>
<tr>
<td>5.3 Redaction using Gabor-Wavelet Filtering</td>
<td>27</td>
</tr>
<tr>
<td>5.3.1 Gabor Wavelets</td>
<td>27</td>
</tr>
<tr>
<td>5.3.2 Creating Feature Vectors</td>
<td>29</td>
</tr>
</tbody>
</table>
5.3.3 Unsupervised Classification ........................................ 31
5.3.4 Supervised Classification ............................................ 32

6 Editing Images .............................................................. 35
6.1 scrubs ........................................................................ 35
   6.1.1 Editing with scrubs ................................................. 38
6.2 five_in_one ................................................................... 39

7 Analysis ........................................................................... 49
7.1 Overview ...................................................................... 49
7.2 Image Analysis ............................................................ 50
   7.2.1 Canny-based Text Redaction .................................... 50
   7.2.2 Unsupervised Gabor-based Text Redaction .................. 52
   7.2.3 Supervised Gabor-based Text Redaction ....................... 55
   7.2.4 Layout analysis ....................................................... 57
   7.2.5 Templates ............................................................ 58
   7.2.6 Text-Redaction and Screenshot Context ....................... 59
7.3 Processing Latency ....................................................... 68
   7.3.1 Text Redaction ....................................................... 68
   7.3.2 Template Matching ................................................. 68

8 Discussion ...................................................................... 72
8.1 Text Redaction ............................................................ 72
   8.1.1 Canny ............................................................... 73
   8.1.2 Gabor ............................................................... 74
   8.1.3 Redaction Layout ................................................... 75
8.2 Templates ...................................................................... 76
   8.2.1 Matching Alerts ..................................................... 76
8.3 Overall Computational Latency ....................................... 77

9 Future Work .................................................................... 78
9.1 Big-Picture Review ....................................................... 78
9.2 Big-Picture Directions .................................................... 79
9.3 Lower-level Issues ....................................................... 82

10 Conclusions .................................................................. 87

A Supporting Libraries ....................................................... 89
A.1 boost and STL ............................................................. 89
A.2 OpenCV ................................................................. 90
   A.2.1 Matrices ............................................................ 91
   A.2.2 Template Matching .............................................. 92
A.3 liblinear .................................................................... 92
A.4 CGAL .................................................................... 94
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B Datasets</td>
<td></td>
</tr>
<tr>
<td>B.1 Processed Data</td>
<td>98</td>
</tr>
<tr>
<td>B.2 Summary of Datasets</td>
<td>99</td>
</tr>
<tr>
<td>B.2.1 EHR set 1</td>
<td>99</td>
</tr>
<tr>
<td>B.2.2 EHR set 2</td>
<td>99</td>
</tr>
<tr>
<td>B.2.3 Random laptop screenshots</td>
<td>100</td>
</tr>
<tr>
<td>C Analysis Tools</td>
<td>101</td>
</tr>
<tr>
<td>D Contact Information</td>
<td>109</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Key events processed by scrubs.</td>
<td>39</td>
</tr>
<tr>
<td>6.2</td>
<td>five_in_one editor modes.</td>
<td>41</td>
</tr>
<tr>
<td>6.3</td>
<td>Key and mouse events processed by five_in_one.</td>
<td>42</td>
</tr>
<tr>
<td>7.1</td>
<td>Unsupervised Gabor-based redaction effectiveness.</td>
<td>56</td>
</tr>
<tr>
<td>7.2</td>
<td>Latency to classify pixels for redaction.</td>
<td>69</td>
</tr>
<tr>
<td>7.3</td>
<td>Latency to generate Gabor elements.</td>
<td>69</td>
</tr>
<tr>
<td>7.4</td>
<td>Latency to load Gabor elements from a file on disk.</td>
<td>69</td>
</tr>
</tbody>
</table>
# List of Figures

3.1 Our system within the context of a larger vision ........................................... 15  
3.2 Instrumenting a computer system ............................................................... 15  
3.3 Information flow through the capture system ................................................ 16  
3.4 Sample redacted screenshot .......................................................................... 17  

4.1 VNC-based screen capture ............................................................................. 20  

5.1 Canny-based text redaction ........................................................................... 25  
5.2 More canny-based text redaction ................................................................... 26  
5.3 Gabor-based text redaction ........................................................................... 32  
5.4 More Gabor-based text redaction .................................................................. 33  

6.1 Text-redaction using scrubs ........................................................................... 36  
6.2 scrubs usage .................................................................................................. 37  
6.3 Custom redaction and unredaction with scrubs .............................................. 40  
6.4 five_in_one usage ........................................................................................... 41  
6.5 Default five_in_one look-and-feel .................................................................. 41  
6.6 Merge redaction rectangles .......................................................................... 43  
6.7 Delete redaction rectangles .......................................................................... 43  
6.8 Define and copy redaction templates ............................................................. 43  
6.9 Copy selected redaction rectangles to matching regions of an image ............... 44  
6.10 Toggle display of redaction rectangles and the underlying image .................. 45  
6.11 Draw custom redaction rectangles ............................................................... 46  
6.12 Draw a grid .................................................................................................... 47  
6.13 Toggle filled drawing of redaction rectangles ............................................... 47  
6.14 “Thin out” superfluous rectangles ............................................................... 48  

7.1 Failure of Canny-based text redaction on an EHR screenshot ......................... 50  
7.2 Relationships of redaction rectangles ............................................................. 51  
7.3 Horizontal distance between rectangle pairs ................................................... 52  
7.4 Predicate used to evaluate rectangle whitespace ............................................ 52  
7.5 Problems labeling redacted-text by visual inspection .................................... 54  
7.6 Sorted rectangle count of each screenshot in each dataset ............................... 56  
7.7 Normalized color variety of each dataset ....................................................... 57  
7.8 Effectiveness of Gabor-based text redaction ................................................... 58  
7.9 Leaking checkbox state with redaction .......................................................... 59
7.10 Leaking information with visual structure. ........................................... 60
7.11 A marked-up redaction mask using Canny-based text redaction. ............... 61
7.12 Analyzing redaction layout of an EHR screenshot. .................................... 62
7.13 Alert template matches in EHR dataset 1. .................................................. 63
7.14 Red color values of four separate EHR screenshots. ................................... 64
7.15 Redaction effect on image information. ..................................................... 65
7.16 Fraction of black-pixels within screenshot pairs. ......................................... 66
7.17 Effect of text redaction on pairwise screenshot similarity. .......................... 67
7.18 Latency of matching varying template sizes. ................................................ 70
7.19 Latency of searching for results of a template match. ................................. 71

9.1 Future information flow using Canny-based redaction. ................................. 83

A.1 Using a shared_ptr with an STL container. .................................................. 90
A.2 OpenCV-matrix file layout. .............................................................. 91
A.3 liblinear data file. .................................................................................... 93
A.4 SVM parameter set. .................................................................................. 93
A.5 Programmatically populating a liblinear feature vector. .............................. 94
A.6 Tracking state during execution of “CGAL::box_intersection_d.” ............... 96

B.1 File layout of Gabor-features. ....................................................................... 98
B.2 File layout of feature-labels. ........................................................................ 99

C.1 Count the number of rectangles in the image. ............................................ 102
C.2 Measure redaction effect over directory of images. ...................................... 102
C.3 Pixel-by-pixel comparison between redacted images in a directory. .............. 103
C.4 Compute redaction performance of a trained SVM classifier. ....................... 103
C.5 Count the number of template matches within a given image. ................. 103
C.6 Analyze computational performance of template matching......................... 104
C.7 Compute a histogram of rectangle size ratios of all images within a directory. 104
C.8 Compute a histogram of rectangle size ratios of a single image. ................... 104
C.9 Generate Canny-based redaction rectangles. .............................................. 105
C.10 Generate Gabor features for redaction. ...................................................... 105
C.11 Generate an RGB color histogram over all unredacted images in a directory. 105
C.12 Generate RGB color histograms of each unredacted image. ......................... 105
C.13 Generate layout histograms of the text-redacted image. .............................. 105
C.14 Convert features from our binary format to liblinear format. ...................... 106
C.15 Train a liblinear L1-regularized L2-loss SVM. ........................................... 106
C.16 Generate ground-truth text-redaction labels for an image. ......................... 107
C.17 Display all text-redacted images using kmeans clustering. ....................... 107
C.18 Load the features and labels for an image. clustering. ............................. 107
C.19 Compute unsupervised Gabor redaction using features and labels. .......... 107
C.20 Text-redact an image using a SVM classifier. ........................................... 107
C.21 Create, display, and store a collage of image templates. ......................... 108
C.22 Visualize a single, redacted image using a set of features and labels. ........ 108
C.23 Compute a histogram of inter-rectangle whitespace. ............................... 108
Chapter 1

Introduction

Application stakeholders seek to implement “usable security” in their systems—security that protects system access and data and promotes intended system use. While doing so, each stakeholder can ask herself, “does a system facilitate tasks it is meant to support and thwart or inhibit ‘bad things’ from happening?” The answer to this question depends on her role: is she an end-user, developer, system administrator, policy maker, or someone else? Each party has a perspective in the system, and therefore, to achieve a usable secure system, application security requires an ongoing conversation among many stakeholders.

Standard engineering tenets suggest that to understand and tune a system, we should measure it. Therefore, an effective conversation about usable security should include data that describes system nuances, behaviors, and system effects on stakeholders. With continual data collection and ongoing conversations, stakeholders can repeatedly tune the system in facets such as source code, configurations, and policy to maintain a working notion of “usable security.” Thus, the conversation of “usable security” is in part, a data-driven communication.

To facilitate this data-driven conversation, we develop instrumentation that captures text-redacted keyboard/video/mouse (KVM) traces—the point where *humanspace and cyberspace* [46] intersect. Because “usable security” includes components from each space,
this intersection serves as a sensible target to measure. Additionally, by capturing data at the KVM interface and text-redacting images, we eliminate software interface dependencies and allow our system to remain both operating-system and application independent. This design choice may be important when instrumented systems lack standards-compliance and interoperability, are regulated and thus cannot readily accept capture software, contain many screen and document formats of interest, or lack documentation and support.

To promote repeated collections, the system should redact text effectively and with reasonable performance. Redaction helps ensure the privacy of captured data and can facilitate timely data sharing among application stakeholders, and reasonable performance reduces the capture system’s impact on end-user workflows. Therefore, we measure redaction effectiveness and processing latency.

**Collecting data**

The current process of obtaining human-computer interface (HCI) data is slow, people-intensive, *ad hoc*, and repetitive; few systematic, automated, institutionalized mechanisms exist. Collection steps include paperwork, institutional review board (IRB) approval, research manager sign-offs, and interactions with information technology staff who maintain the technical systems of interest. We intend our system to automate and simplify this process and hence lower the barrier to data capture, analysis, and application tuning.

**Sharing data**

End-users experience application problems such as missing functionality, fouled permissions, crashes, and impediments to workflow. With instrumentation, end-users could capture these issues securely, share them with appropriate application stakeholders, and have the issues redressed.
Technology

With computer vision algorithms to implement redaction, modern-day computing power, and large volumes of rich data, researchers can study usable security empirically; end-users can participate empirically in system configuration and development processes; and organizational stakeholders can begin to understand their risks empirically. Our system can facilitate these activities.

1.1 Scope and Impact

This thesis positions well within the focus of the security research and national security communities. Each group has grown to realize the important role of usability and empirical analysis in maintaining secure systems. Events such as Symposium on Usable Privacy and Security (SOUPS) and Computer Human Interaction (CHI) demonstrate this point as they have grown in size and importance.

In a recent interview, Dickie George, the Information Assurance Technical Director of the National Security Agency (NSA) stated that “fighting today’s cyber cold war depends on […] the adoption of security that is transparent to the end user” [44]. Our proposed work allows us to collect data near the end user and begin to analyze the definition of “usable security” empirically.

Additionally, Dr. Doug Maughan, Branch Chief in Homeland Security Advanced Research Projects Agency within the Science and Technology Directorate of the Department of Homeland Security, outlined a research roadmap that defines the U.S. cybersecurity R&D agenda [63]. Enterprise-level security metrics, privacy-aware security, and usable security comprise 3 of the 11 research areas listed on the roadmap. Our proposed work makes direct headway into these areas by securing and systematizing data collection in order to empirically understand usable security.
1.2 Vision

In our long-term vision, the fruit of this work can enable empirical feedback paths between application stakeholders. Developers, administrators, end-users, organizations that produce or deploy a particular system, and legislators or governance bodies that create rules to govern systems all represent different types of stakeholders. With established feedback paths, such stakeholders can begin to understand empirically the day-to-day, system-effects of their decisions.

A simple capture system can also provide direct value to end-users by endowing them with a larger, empirical role in the software maintenance cycle. They can capture, annotate, and share problems, configurations, ideas, bugs, and other captured scenarios with stakeholders. They can inform existing ad hoc stakeholder interactions such as online support forums and help-desk interactions with rich, contextual data. Additionally, end-users can use traces as visual web search keys during their own investigations.

Playing a larger role in the software maintenance cycle can motivate end-users to share their findings continually: if end-users believe and experience that their contributions make a positive difference to their workflow, end-users may be motivated to contribute further. Consequently, organizations may improve empirical insight into their information security systems and associated risk calculations. When organizations lack the expertise to analyze traces in-house, they could hire third parties to do so.

Large volumes of captured KVM data can form a new computational science concerned with usable security, application usability, debugging, and other topics that benefit from KVM evidence. The birth of such a field is akin to how computational social science becomes possible when large volumes of data from websites such as Twitter or Facebook enable study of social-interactions at macro scales. Through secured traces and new algorithms that mine large capture collections, stakeholder will have a simple means at their disposal to inform a deeper and empirical understanding of their systems.

Finally, simple KVM instrumentation could initiate the understanding of usable secu-
rity in settings such as medicine where access to software APIs may be limited. In such domains, machines may be certified or lack standards compliance and interoperability and thus not viable targets for instrumentation that requires software API access. Altogether, the secure capture system we propose can take us one step closer to the vision we describe.

1.3 Putting it All Together

Consider the following concrete example of how screen capture for sensitive systems could help improve overall system security. Note in the following discussion how our system could systematize data collection and protection to forgo instance-based IRB approvals and augment developer communications with “HIPAA-safe”\(^1\) precise, empirical findings.

A large hospital relies on an EHR system to facilitate healthcare delivery, and computer workstations exist throughout the hospital campus to promote convenient EHR access by clinicians. These end-users access hundreds of different screens and forms provided by the system and sometimes deal with slight changes in screen content when the hospital upgrades the EHR software every 6 months.

To interact with the system, end-users enter their login username and password at any workstation, and an administrative policy requires them to log off each time they walk away from their logged-in system. While clinicians lack positive control over their session, any person could walk up, view, and manipulate the system using clinician credentials. Because clinicians become busy and frequently revisit a recently logged-in workstation, and because the system loses desktop context on logout, clinicians frequently remain logged in after they walk away.

A security researcher working together with a policy administrator of the hospital wish to study clinician habits related to this policy violation and understand how current au-

\(^1\)We call the text-redacted document “HIPAA-safe”. The Health Insurance Portability and Accountability Act (HIPAA) declares that only authorized observers can view personally identifiable information such as patient name and social security number found in medical documents \(^{62}\). Redacting text from screenshots of such documents before sharing the images can implement such required protections.
The researcher conducts personnel interviews and wishes to gather system authentication logs to study data patterns and correlate them with interview findings. Before obtaining data logs from a busy information technology (IT) administrator, she composes an plan to protect and use the log data. The plan must be approved by hospital authorities and a committee of people (an IRB) who certify that proper data protections will be implemented.

In a non-technical, free-form process, the researcher answers questions on approximately ten different documents and drafts a proposal to gather, protect, and use authentication logs. The researcher has written proposals in the past and therefore has experience crafting one to avoid major rewrites during the approval process. Additionally, she has sufficient expertise to understand how authentication logs must be protected.

After completing the proposal, the researcher submits the request for approval. The entire approval process takes a minimum of one month, many document revisions, and interactions with individuals across many domains of expertise and a myriad of work schedules.

After receiving approval, the researcher collects and analyzes data and shares findings with EHR developers. Before sharing however, she submits an addendum to the proposal that outlines the developer exchange—the researcher forgot to include this step in her initial paperwork. After receiving approval in approximately two weeks, the researcher interacts with developers who implement an application update to address empirical findings. Finally, hospital IT administrators deploy the update.

This sample scenario demonstrates deficiencies in the current process of tuning an end-user application to improve overall system security. The process is manual, free-form, and time-consuming. A technical capture and redaction process could systematize data collection and protection, simplify the process of understanding system problems empirically, and facilitate rich communications with developers. With a simplified and systematized
process, many other studies could be carried out quickly and efficiently to tune EHR systems and improve their overall usability and security. Finally, this concept can applied generally to other application domains.

1.4 Contributions

In this work, we contribute a system that combines KVM capture with text redaction to produce precise technical content that can enrich communications among application stakeholders. The system automatically redacts text from screenshots to reduce sensitivity of captured material and captures at the KVM interface to eliminate software interface dependencies. Consequently, our system remains both operating-system and application independent.

1.5 Roadmap

In the remainder of this document we discuss the system in further detail. Chapter 2 describes related work on the topics of screen capture, text segmentation, user studies, de-identifying data, anonymity, and document analysis. Chapter 3 provides an overview of the system including its architecture, design goals, functionality, and implementation. Chapter 4 provides an overview of screenshot capture and the virtual-network computing (VNC) method we implemented in this work. Chapter 5 discusses the three methods this work relies on to redact text from screenshots. Chapter 6 describes two tools; one used to capture, text-redact, and edit screenshot video, and a second used to text-redact and edit static screenshots. Chapter 7 presents empirical measurements of redaction quality and latency, and Chapter 8 discusses the meaning of empirical findings. Chapter 9 describes avenues for future work, and Chapter 10 concludes the main body of this dissertation.

Four self-contained appendices follow Chapter 10. Appendix A describes supporting libraries used to implement this work, Appendix B discusses screenshot datasets that un-
derlie analysis, Appendix C presents analysis tools implemented to gather empirical measurements and manipulate screenshots, and Appendix D provides contact information for the author and advisor. We recommend skimming appendices A–C first to provide relevant background for various chapters.
Chapter 2

Related Work

Our work combines existing technologies of screen capture and computer vision with a goal of improving the quality of communications among application stakeholders and ultimately, improving our understanding of “usable security.” Many research and commercial products implement pieces of our work in isolation and for different purposes.

Section 2.1 discusses screen capture, Section 2.2 segmenting text, Section 2.3 user studies, Section 2.4 deidentifying data, Section 2.5 anonymity, and Section 2.6 overviews the topic of document analysis and closes the chapter.

2.1 Screen Capture

The MIT Sikuli research project combines computer vision and programming to enable users to create machine-independent, visually-programmed and actuated programs [68]. To write a program, users select graphical user interface (GUI) objects such as buttons, sliders, and checkboxes they wish to activate when the program is executed. Within their running program, computer vision finds the specified objects and the program actuates them. User-interface developers can use Sikuli to script GUI test suites [18].

A commercial product call eggPlant also allows developers to test GUIs with machine-independent, automation scripts [53].
Many screen capture applications such as Snipping Tool [43], Snapz Pro X [8], and xwd [25] exist. Some programs capture still screenshots, others capture both stills and video, and some allow end-users to annotate captures.

Our system captures data and modifies it with text redaction—we use screen captures for a different purpose than these works. Chapter 9 outlines future work that would give our system the ability to visually program as in Sikuli and annotate screenshots with custom user notes.

2.2 Segmenting Text

Many commercial and free-software tools such as Gimp [58], Photoshop [7], Aperture [9], Final Cut Studio [11], Pixelmator [47], and Imagemagick [21] allow one to paint, create, touch up, and modify still images and/or video. Many of these applications implement complex computer vision algorithms to allow users to isolate, select, and transform regions of an image. These applications could be used to manually redact text from a screenshot.

In Chapter 5 we describe how our work builds on existing text segmentation research [34] to redact text automatically from screenshots.

Google Goggles can extract and recognize text from natural scenery for purposes such as language translation among many others [32]. The scope of our system is limited to computer screenshots. However, screenshots taken with a camera may include angles and lighting similar to the natural scenery submitted to Google Goggles.

2.3 User Studies

In another work, Google highlights their in-house human-computer interaction (HCI) capture system called UseTube [39]. UseTube supports employees who wish to perform user studies of any network-connected computer; it simplifies the act of performing, archiving, and accessing user studies. Captures reside in a web-accessible archive to easily facilitate
real-time, browser-accessible sharing and analysis. Their system most closely relates to our system, but unlike UseTube, we redact sensitive regions from captured data.

2.4 Deidentifying Data

In the medical domain, a large body of work relates to deidentifying protected health information (PHI) in electronic documents [42]. Approaches for finding PHI within documents falls into two broad categories: comparison using a dictionary of terms and application of natural language processing (NLP) that uses machine learning and training datasets to build a language model. The MITRE Corporation developed the “Scrubber toolkit” that relies largely on the NLP approach to find and then deidentify documents [6].

Our work approaches the deidentification problem from a complementary angle. Our system does not interpret data; rather, it redacts all text within an application screenshot and allows a domain expert to unredact portions relevant to their needs. By taking this approach, we leave difficult judgment calls to humans and provide a default deny-all, “what-you-see-is-what-you-get” (WYSIWG) tool that can simplify the process of sharing screenshots among application stakeholders. Our system performs the bulk of redaction for an end-user and leaves them with any customization they wish. In future work we plan to study and augment our system with a form of deidentification found in existing scientific literature.

Document redaction products such as Rapid Redact [48] and brava! [33] exist in the commercial marketplace. These products parse document structure and can help users achieve WYSIWYG. In contrast, our system redacts material from images directly (there is typically no hidden structure); it neither interprets nor parses document structures. Our system is principally concerned with redacting screen images and videos that contain a mix of opaque image content which depends on the visual nature of open applications.

Finally, NSA has published a document that describes how to properly redact Word 2007 documents [45]. This manual process attempts to remove any hidden content from a
2.5 Anonymity

Deidentified data records can still contain visual information that reveals sensitive data. In her ground-breaking work, Sweeney recognized this problem in the context of databases that contain a mix of sensitive and unsensitive records. She proposed a technique to $k$-anonymize data so that any record within a set is indistinguishable from $(k - 1)$ other records within the set $[52]$.

Machanavajjhala et al have extended Sweeney’s work in what they call $l$-diversity to further tune sensitive, $k$-anonymized records so that $k$-anonymity exists among combinations of sensitive and non-sensitive fields $[41]$. Such additional protection can eliminate information leaks where an observer has knowledge about an unsensitive field and uses that knowledge to narrow their search results to a list of identical sensitive fields and thus reveal a sensitive value.

In some circumstances, redacted text in our system may suffer from a visual form of the $k$-anonymity problem. Visual metadata such as column and row arrangements of redacted data may help an observer learn information our system intended to hide. In Subsections 7.2.4 and 8.1.3 we discuss this problem in more detail and using empirical evidence, describe how we might add visual $k$-anonymity to the list of system features.

2.6 Document Analysis

The International Conference on Document Analysis and Recognition (ICDAR) has many papers and competitions related to the problem applying machine learning and computer vision to analyze documents analysis $[5]$. A 2003 competition sponsored by ICDAR has datasets available for optical character recognition (OCR), word recognition, text locating, and other purposes $[4]$. These datasets do not apply directly to our problem; we segment
text, in some cases have a more constrained segmentation problem, and do not apply OCR.

In general, the problem of document analysis and recognition is well-studied. As mentioned earlier, we begin to investigate screenshot structure to $k$-anonymize layout and leave in-depth document-reconstruction as subject of future work.
Chapter 3

System Overview

In this chapter, we provide an overview of the system architecture, design goals, functionality, and implementation. We begin by discussing the architecture in Section 3.1 followed by design goals in Section 3.2 and functionality in Section 3.3. We conclude with an overview of the implementation in Section 3.4.

3.1 Architecture

The architecture of our system consists of data capture and text-redaction components. Figure 3.1 describes these within the context of a larger vision and includes functionality that is subject of future work. Figure 3.2 examines more closely the subject of this work and highlights how our instrumentation augments a computer to collect keyboard/video/mouse (KVM) data.

The design described by Figure 3.2 may provide value in industries such as power where additional software installed in a real-time system can create unacceptable processing delays. Our design may also benefit industries such as medicine, where instrumented systems may lack standards-compliance and interoperability, are regulated and thus cannot readily accept capture software, contain many screen and document formats of interest, or may lack documentation and support.
Figure 3.1: Our system within the context of a larger vision. The figure includes a “Logging Service” that interacts with a monitored system, a “Sharing Service” that functions as a repository for application stakeholders to share redacted screenshots, a monitored system, an end-user who wish their system to be monitored and a web-browser through which the end-user can interact with logging and sharing services. Within this big-picture context, internal components of the bold box labeled “Logging Service” define the contribution of this work. A sample usage scenario follows the numerical labels in the following steps. 1) An administrator enables the “Logging Service.” 2) An end-user triggers the “Logging Service” to log a monitored host. 3) The “Logging Service” connects to the monitored host and begins logging. 4) The end-user interacts with monitored system. 5) The end-user reviews and possibly edits logged data. 6) The end-user publishes redacted screenshots.

Figure 3.2: Instrumenting a computer system. The capture instrumentation, which is a sub-component of the “Logging Service” in Figure 3.1 listens to keyboard and mouse input from devices and video output received by the computer. Consequently, it can remain system independent and simplify capture on closed or certified systems where special software instrumentation may not be installed. Finally, the dashed lines represent our contribution—the ability to capture KVM events.

As instrumentation captures data, it flows through the set of processing steps outlined in Figure 3.3. These steps prepare traces for sharing among application stakeholders as depicted in Figure 3.1.
Figure 3.3: Information flow through the capture system. As input, the redaction process takes raw KVM data or segmented KVM data received from a windows API. Segmented KVM data would be available only with window-system API access. After the system redacts imagery, it logs the trace to disk where it may eventually be read and transmitted to a “Sharing Service” as depicted in Figure 3.1.

3.2 Design Goals

The system should redact text from screen captures, and to facilitate API-independence, the redaction technique should operate on raw images. When an accessible windows API exists, the redaction process will simply have less work to do. This design facilitates the study of usable security in systems such as medical or power, where certification, real-time performance requirements, or lack of documentation and support limit programmatic access to the system.

Finally, our system should function without severely impacting the performance of the instrumented system or normal end-user workflows. Negative impacts may cause users to disable the instrumentation and negate its benefits. We hope to avoid this problem.

3.3 Functionality

Our system includes functionality essential to implementing screen capture for sensitive systems. The basic steps of instrumenting such systems include screen capture, image processing and editing, and data sharing. Capture through KVM, camera, or a native capture application all represent ways the system could obtain screenshot data. We implemented KVM capture because it enables systematic logging of existing systems without their modification.
After capture, the system processes an image to find and redact text as demonstrated in Figure 3.4. Additionally, the system may search for regions within the image that match a set of image snippets or “templates” and count, redact, or unredact matching regions. Finally, a user may wish to edit the image and further redact or unredact a portion of the processed screenshot. Various components of the system correspond to these possible steps and we discuss each of them in subsequent chapters.

3.4 Implementation

The bulk of our system implementation relies on a mixture of C and C++ code spanning multiple open-source libraries and custom-developed libraries and applications. Appendix A describes the principal set of libraries that support our effort, which include boost [54], C++ STL [2], OpenCV [16], liblinear [28], and CGAL [1, 37, 70]. Appendix C describes applications we developed to manipulate screenshots and analyze the system.
Altogether, we implemented approximately 9000 lines of code.

We rely on a number of open-source tools to edit, build, debug, and analyze code. Emacs [59] serves as our code editor, gcc [56] compiles source, and cmake [38] and ultimately gnu make [60] function as the build system. Gdb [57] helps debug code; dtrace [50], Bash [55], and perl [24] support analysis; and gnuplot [20] generates plots.

To remain system independent, we implemented certain functionality with higher-level APIs; our development environment is a MacBook Pro running OS X 10.6 with 8 GB of memory.1 Certain, low-level OpenCV routines rely on system libraries, but these are transparent to our code—OpenCV is cross-platform.

---

1We upgraded to OS X 10.7 midway through development and analysis.
Chapter 4

Screen Capture

In the last chapter, we described a high-level architecture and design of our work. We also described how it fits within the context of a larger system that includes components to share redacted screen captures. In this chapter, we describe the capture functionality our work implements. Section 4.1 provides an overview of capture and Section 4.2 describes our capture technique.

4.1 Overview

The action of screen capture requires a subject of capture, a method to collect and communicate screenshots, and a system to process and store them. In some cases, a single computer may fulfill all three requirements. In this chapter, we discuss the second one in depth.

At least three approaches exist to collect and communicate screenshots. In the first approach, an end-user executes a capture application running directly on the capture target. The application can use system APIs to collect pixels associated with visual objects such as screens or windows. As we describe in Chapter 2, many third-party and operating system applications exist for this purpose.

Taking pictures with a hand-held camera is a second method to gather screenshots.
Screenshots collected using this method may include discoloration, natural artifacts due to the environment and camera position, screenshot orientations with complex angles and lighting, artifacts outside of the physical screen, or other problems.

Capturing content through a hardware keyboard/video/mouse (KVM) device or software-based virtual network computer (VNC) interface represents a third approach to collect and communication screenshots. Our system implements this remote-capture approach, which enables screenshot collection on instrumented systems that lack standards-compliance and interoperability, are regulated and thus cannot readily accept capture software, or may lack documentation and support.

4.2 VNC-based Screen Capture

Our system relies on a virtual network computer (VNC) arrangement to capture screen material from a remote host [49]. Figure 4.1 provides an overview of this VNC-based approach. In a nutshell, VNC defines a protocol for transporting a computer’s framebuffer, keyboard, and mouse data over the network. By building a system with this protocol, our system can capture and operate on all KVM events in a system-independent fashion.

![Figure 4.1: VNC-based screen capture. The arrangement is comprised of the capture system and its target. The capture system executes a VNC client and the target executes a VNC server. The server exports the target’s KVM over a TCP network connection as a stream of bitmap updates using the RFB protocol [49]. Initially, the client connects to the server. Then the endpoints proceed through a handshake and finally begin the screen update process. Periodically, the VNC client polls the server for updates and the server responds at its chosen time, typically when changes occur on the target.](image)

We chose software-based VNC over hardware-based KVM because the software re-
source was readily available for development. Additionally, if we build a prototype system that redacts imagery collected from software-based VNC, then the work should readily apply to hardware-based KVM. The software interface to the screen capture agent would change, but the rest of the system would remain identical.

In our test configuration, Mac OS X 10.6 functions as the “Capture System” and a Ubuntu Linux 9.10 running within a VMware instance on the “Capture System” serves as the “Capture Target.” The application x11vnc serves as the VNC server on “Capture Target,” and our code implements a VNC client as part of an application called “scrubs,” which we describe in detail in Chapter 6. The client implements read-only functionality and therefore does not pass keyboard or mouse events from the VNC client to the VNC server.

Our client connects to the VNC server using TCP and currently does not implement connection security. TLS or SSH-based security are common protocols we could use to do so. After connecting, the endpoints proceed through a handshake phase and negotiate the protocol version “RFB 003.008\n” and the “raw” pixel format to transfer screen updates from the server to the client without compression. Compressed image formats exist to reduce network traffic, and we leave their implementation as future work.
Chapter 5

Text Redaction

In the last chapter, we described how the system captures screenshots. In this chapter we explain the core system-function of text redaction (or simply “redaction”) in detail and the following three ways our system accomplishes it: Canny Edge Detection [17], Gabor-wavelet [30] filtering with unsupervised classification, and Gabor-wavelet filtering with supervised classification [34].

Section 5.1 overviews text redaction. Section 5.2 describes redaction using Canny edge detection. Section 5.3 describes Gabor-wavelet filtering with unsupervised classification. Finally, Section 5.3.4 describes Gabor-wavelet filtering with supervised classification and concludes the chapter.

5.1 Overview

Text redaction is a fundamental aspect of the system because it removes sensitive text from screen capture data, relieving the end-user from manually redacting screen captures before sharing. By default, the system implements a “deny-all” policy and thus redacts all text it finds. An end-user can then “unredact” small regions as necessary to facilitate their conversation. Instead of forcing end-users to redact large portions of the screen by hand, we redact all text and allow an end-user to unredact small portions as necessary. This naturally
reduces the workload required to share screenshots among application stakeholders.

Additionally, “deny-all” mirrors a common policy in domains such as networking where firewalls are configured typically to deny-all and whitelist (the analogue of unredact) only a small number of network ports. Because redaction affects text and a small number of icons, screen context remains despite removal of potentially sensitive data.

In a different approach to redaction, our system could simply redact an entire screen (e.g., turn the entire screen black) and the end-user could unredact whichever small piece supports their needs. We believe this approach provides too little screen context to observers. Unredacted, unsensitive screen data provides context to application stakeholders that may help focus their discussion. However, our system can support this approach without major modifications.

Image-based text redaction is comprised of two principle steps: finding text in an image, also known as text segmentation, and recoloring segmented image regions to “remove” text. After segmented pixels have been identified, a system can easily change their values to a single color such as black. Redacting images using this approach ensures that no “hidden” text or other data exists within the final redacted product—WYSIWYG. Next we discuss three approaches to accomplish WYSIWYG redaction of images.

5.2 Redaction Using Canny Edge Detection

In our first approach to segment text, we rely on the Canny edge detection algorithm [17]. Canny finds edges in an image by analyzing its intensity gradient and marking edges at gradient high points. To maintain legibility, screenshot text exists with an intensity contrast in relation to its background and thus creates gradient high points. Canny finds these high points and thus can segment screenshot text.

The entire Canny-redaction process includes multiple steps. First, the system converts a

\footnote{This statement assumes technology such as steganography does not visually hide data within an image.}
color screenshot to 8-bit gray scale. It then applies a Gaussian blur using a 3x3 window to reduce image noise—Canny output qualitatively contained less noise with this initial blurring step. Next, the system executes Canny using low and high threshold values of 100 and 300 respectively to find edges—the values provide qualitatively-reasonable redaction results for a variety of desktop screenshots. Gradient magnitudes greater than the high threshold are considered edges and traced throughout the image. Values above the low threshold denote edges that branch from an existing trace process. Together, these tunable values reduce noise during edge detection.

After executing the Canny algorithm, the system finds connected components (polygons) using Canny output and an algorithm suitable for doing so [51]. For each polygon discovered, the system computes a bounding rectangle and draws a filled version of the rectangle into an image “redaction mask.” Finally, the redaction mask is applied to the original image to produce a redacted image.

OpenCV implements Canny edge detection over 8-bit gray-scale images, finds contours within an image using a well-know method [51], and given a set of points, computes a bounding boxes. Thus, the library provides a useful toolbox for the Canny-redaction approach.

Figure 5.1 lists three screenshot snippets from the Wikipedia article about the Canny algorithm [66]. The first snippet is the article, the second is the redaction rectangles computed over Canny output, and the third is the redacted version of the article. Figure 5.2 lists the rectangles and redacted screen from snippets of a gmail inbox.

In Figure 5.1 notice how redaction includes a globe in the upper right corner. In Figure 5.2, Canny found a large rectangle that outlines the message box listing. This rectangle translates to a large black rectangle in the redacted image and destroys potentially useful screenshot context. These facts demonstrate that Canny finds any edge, whether text or not, and consequently the algorithm produces false positives for text redaction.

Additionally, Canny did not detect faint lines separating sections of the screen in Fig-
Figure 5.1: Canny-based text redaction. The top image is a screenshot snippet from the Wikipedia page about Canny edge detection [66]. The second image depicts the rectangles that result from processing the first image with Canny edge detection, polygon detection, and polygon bounding with rectangles. The third image derives from filling the rectangles in the second image and then applying the second image as a redaction mask to the first. Canny missed some true edges throughout the image (false negatives for edge detection) and added edges where text does not exist near the globe (false positive for text detection). Finally, notice whitespace between words and tiny rectangles enclosed within larger ones.

These are false negatives for edge detection, but not for text detection. In Figure 5.2, false negatives exist for text detection in the upper left corner of the screenshot.
Figure 5.2: More canny-based text redaction. The top image is a screenshot snippet from a gmail inbox. The second image depicts the rectangles computed over a gmail mailbox and the second image depicts the redacted mailbox. Canny missed text (false negative for text redaction) in the upper left portion of the screen and detected a large rectangle that includes a majority of the screen (false positive for text redaction).

Also, notice how many words in a line are separated by whitespace. Canny does not collapse or merge nearby edges. However, it does detect letters that have been visually truncated, such as those at the bottom of the screenshot snippet. Such letter fractions still produce variations in the image intensity gradient in a way that Canny detects.

Finally, the Canny algorithm sometimes finds interior edges of letters such as “p” which produce very small rectangles embedded in larger ones. We will discuss many of these issues further in Chapter [7].
5.3 Redaction using Gabor-Wavelet Filtering

The next method we explored to segment text was based on Gabor wavelets \[30\] with unsupervised classification. The underlying idea is to treat text as texture and use Gabor wavelets to segment texture. We followed the description provided by Jain and Bhattacharjee in 1992 \[34\] and modified it as necessary to produce working results. Before we delve into our approach, we first explain Gabor wavelets briefly.

5.3.1 Gabor Wavelets

In general, a wavelet is a wave with some orientation and frequency that when convolved with an image, resonates and creates a detectable signal. Gabor wavelets, which are commonly used in image processing, are comprised of a sin wave modulated by a Gaussian envelope and for our application, they use a two-dimensional envelope. Both real and complex components comprise the wavelet, but in following \[34\], we only use the real, symmetric (cos) component.

Equation 5.1 shows the wavelet equation \(h(x, y)\) that we used. Wavelength (\(\lambda\)) and orientation (\(\theta\)) comprise its tunable parameters in our application.

\[
h(x, y) = \exp\left(-\frac{1}{2}\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}\right)\right)\cos\left(\frac{2\pi}{\lambda} x_0 + \phi\right),\tag{5.1}
\]
where

\[ x_0 = x \cos(\theta) + y \sin(\theta) \]
\[ y_0 = -x \sin(\theta) + y \cos(\theta) \]
\[ \sigma_x = \frac{\lambda}{\pi} \sqrt{\frac{\log(2)}{2^{b+1}}} \frac{2^b}{2^{b-1}} \]
\[ \sigma_y = \frac{\sigma_x}{\gamma} \]
\[ b = 1.0 \]
\[ \gamma = 0.5 \]
\[ \phi = 0 \]

The wavelet filter is computed and then then convolved with the target image. The size of the filter is odd in both the \( x \) and \( y \) dimensions and its independent variables are \( n \) (the number of standard deviations of the Gaussian to consider) and the wave orientation \( \theta \). The following equations describe the filter size precisely, where the filter dimensions are \( (x_{\text{max}} - x_{\text{min}} + 1) \times (y_{\text{max}} - y_{\text{min}} + 1) \).

\[ x_{\text{max}} = \left\lceil \max \left(1, \max \left(n \sigma_x \cos(\theta), n \sigma_y \sin(\theta)\right)\right) \right\rceil \]
\[ y_{\text{max}} = \left\lceil \max \left(1, \max \left(n \sigma_x \sin(\theta), n \sigma_y \cos(\theta)\right)\right) \right\rceil \]
\[ x_{\text{min}} = -x_{\text{max}} \]
\[ y_{\text{min}} = -y_{\text{max}} \]
\[ n = 5 \]

The approach for defining the filter and its window were taken from a combination of two sources [34, 67]. We use \( n = 5 \) standard deviations as recommended by the paper [34].
5.3.2 Creating Feature Vectors

In summary, the system segments text with the following steps. First, it filters an image with each Gabor filter. Then, each filtered image is thresholded, transformed into features, and stacked so that each pixel now has a “feature vector” with dimension equal to the number of filters. Next it appends each pixel’s $x$ and $y$ position to each feature vector, shifts each vector dimension to zero mean and unit standard deviation, and classifies pixels as “text” or “not text” using either supervised or unsupervised means.

During the filter process, a collection of Gabor filters with varying parameters forms a filter bank through which an image is processed. The bank enables detection of image features of different frequencies and orientations.

When an individual filter is convolved with an image, the system extrapolates border pixels to increase the image size and prevent the filter from “falling off” the image edge. Our system relies on border replication, because the technique produces numerically useful results in combination with the Gabor filter—other extrapolation approaches failed in our experiments.

Each filter produces a “filtered image” that corresponds to one parameter combination. Through qualitative analysis, we settled on parameters

$$\lambda \in \{.5, 1.0, 2.0, 4.0, 8.0, 16.0, 32.0\}$$

and

$$\theta \in \{0.0, 45.0, 90.0, 135.0\}$$

for a filter-bank size of 28 filters ($|\lambda| \times |\theta|$).

The parameter $\theta$ varies according to the paper [34] to detect signals oriented in a uniform variety of positions. In contrast, the parameter $\lambda$ does not follow the paper. Instead the paper specifies a wave frequency $\mu_0$ and derives parameters from it to form features images. We explain more about our choice shortly.
We chose \( \lambda \) to vary by powers of 2 in order to form a dyadic collection of filters that span a collection of feature sizes. We had trouble realizing the parameter description of [34] and found that our parameters produced usable feature images. Through qualitative experiments, we found our chosen values to detect features among a collection of screenshots.

After the filter process, each filter image is thresholded using Equation 5.2 to form “thresholded images.” In the equation, \( \alpha = .25 \), \( t \) is a pixel value. The result of applying the equation to the image is that each pixel takes on its new thresholded value.

\[
\psi(t) = \frac{1 - \exp^{-2\alpha t}}{1 + \exp^{-2\alpha t}}
\]  

(5.2)

After thresholding, the windowing function defined by Equation 5.3 is applied to each thresholded image to compute the “texture energy” in small intervals about each pixel in what the paper calls a “feature image.” The \( k \)th feature image, represented by \( e_k(x, y) \) is generated by summing the thresholded values in \( M \times M \) windows \( (W_{xy}) \) about each pixel \((x, y)\). Note that \( r_k(a, b) \) is the \( k \)th filtered image, and \( \psi \) is described in Equation 5.2.

\[
e_k(x, y) = \frac{1}{M^2} \sum_{(a,b) \in W_{xy}} |\psi(r_k(a, b))|, \ 1, \ldots, n
\]  

(5.3)

To compute this summation, the system convolves a \( M \times M \) window of ones with the thresholded image and multiplies each resultant pixel value by \( \frac{1}{M^2} \). As with all other convolution operations, the system replicates the border before processing pixels that cause the window \( W_{xy} \) to fall outside the image.

Then, the feature images are stacked so that each pixel consists of a 28 dimension feature vector and the \( x \) and \( y \) pixel position of each pixel is appended to its feature vector to create a final feature-vector size 30. The vector dimensions are modified to zero mean and unit standard deviation to normalize the numeric effect of features during computations. Our system differs from the Jain and Bhattacharjee system [34] from this point on.
5.3.3 Unsupervised Classification

After deriving a collection of \((img_{rows} \cdot img_{cols})\) feature vectors as described above, the system uses the kmeans algorithm \([40]\) to cluster features into \(k\) classes, where \(k \in \{2, 3\}\). The algorithm assigns each pixel a class label \(i \in [0, k - 1]\), and one class may correspond to text if text exists. In the paper \([34]\), the authors clustered into 3 classes using a different clustering technique and chose the class labeled 2 (using 1-indexed label names) as text for all of their analysis. In our qualitative experiments, we found that some screenshots clustered better visually into \(k = 2\) classes and others into \(k = 3\) classes.

During kmeans clustering, the system relied on stopping conditions of the first of 10000 iterations or an error rate of .0001. We chose the initial cluster centers using a more recent technique \([13]\) and ran the algorithm one time to the stopping conditions before assigning labels.

After running kmeans, the label \(i\) corresponding to text must be chosen manually. The designated “text” pixels form a mask that redacts text when combined with the original image. Unlike the Canny-based approach, no polygons are found within the redaction mask, no bounding rectangles are drawn into the mask, little white space exists between adjacent words, and fewer non-text objects are labeled as text. Visually, Gabor-filtering redacts more precisely than Canny-based filtering. Figure 5.3 revisits Figure 5.1 using Gabor-based redaction where \(k = 2\) and \(i = 0\) and Figure 5.4 revisits Figure 5.2 using \(k = 2\) and \(i = 1\).

In Figure 5.3 note how Gabor-based redaction fills whitespace between words in sentences, does not redact objects such as the globe, and does redact fractional characters. It does not redact large rectangles from the screen as Canny-based redaction did in Figure 5.2.

In Figure 5.4 note how false negatives (text that should be redacted but was not) exist with certain font scales and textures in the upper left corner and also throughout lighter message-body in the message lines.
Figure 5.3: Gabor-based text redaction. The top image is the same unredacted snippet as seen in Figure 5.1. The second image depicts a Gabor-redacted version of the first image. Gabor-based redaction does not redact large objects such as the globe, connects whitespace between words in sentences, and redacts fractional characters found at the edge of the screenshot.

5.3.4 Supervised Classification

The downside to unsupervised classification is multi-fold: $k$ and $i$ are chosen manually; the approach classifies pixels into $k$ clusters whether or not text exists; and finally, the feature count can easily surpass 1 million with modern screen resolutions, and thus kmeans can be slower than possible alternatives. We will return to the topic of performance in Chapter 7, but for now, we describe the supervised classification technique our system implements.

In the supervised approach, image feature vectors are generated as in the unsupervised approach. Instead of kmeans, however, each feature is fed to a trained classifier that labels the pixel as “text” or “not text.” All pixels labeled as “text” are converted to the color black and all other pixels maintain their values.
Figure 5.4: More Gabor-based text redaction. The top image is the same unredacted snippet as seen in Figure 5.2. The second image depicts a Gabor-redacted version of the first image. Gabor-based redaction missed the large gmail text and small “by Google” text below the gmail text. It partially redacted lighter message text in the inbox. Note that Gabor-based redaction did not compute a large rectangle of false positives as Canny did in Figure 5.2.

This supervised approach to classification solves many of the problems of unsupervised Gabor. Parameters $k$ and $i$ no longer exist, so the manual step of choosing them no longer exists. The supervised approach classifies pixels as “text” or “not text” (where the class label 1 corresponds to “text” and the class label $-1$ corresponds to “not text”). Therefore, all pixels will be classified, whether or not text exists in the image. Finally, classifying pixels with a supervised classifier can be faster than running kmeans over the same set of pixels. We discuss classification performance in Subsection 7.3.1.

We chose a linear support vector machine (SVM) to label pixels as members of classes $\{-1, 1\}$. The output of our training phase tells us which if any filters do not contribute to the resultant SVM and can be removed from the filtering process. We experimented with the following two of many classifiers provided by the liblinear library [28]: (a) L1-regularized L2-loss support vector classification and (b) L1-regularized logistic regression.

We chose these classifiers because after training, they can contain a 0-valued parameter for each feature that remained unused during the training process. Such features can be
eliminated from input during future predictions and thus not computed in the first place. Their absence reduces computational overhead in the running system.

To begin machine learning, we first partition our set of screenshots into a training set and testing set. Then to train the classifier, we generate a set of ground-truth feature vectors and labels from the training set. We generate ground-truth by manually choosing the features and labels associated with “best” redaction results using the unsupervised classification technique described earlier. This ground-truth is fed into a program we implemented that interfaces liblinear to train and save the resultant classifier. The classifier can then be run on any image using another program we wrote to classify pixels as \{-1, 1\} and thus redact text.

Among the two classifiers listed, (a) used all features and (b) used all but one feature (feature 12 of 30) in the resultant classifier. Because the savings would be a single feature for a single model, we did not eliminate it.

During the SVM training process, we used default liblinear values for all SVM parameters. We experimented with cross-validation to tune the constant $C$ in the SVM expression (see liblinear for details [28]). However, we experienced minimal performance improvements and therefore relied on default values to train each classifier.
Chapter 6

Editing Images

In Chapter 5 we described in detail how our system redacts text from screenshots. Either during or after the redaction process, users may wish to edit redacted imagery. Thus, we have developed a tool called “scrubs” for doing so.

Additionally, we found visualization tools useful as we progressed in the research of “Screen Capture for Sensitive Systems.” They enable exploration of new concepts and help uncover positive findings and problems with research ideas. Thus, for the purpose of exploration and for end-user use, we developed a tool called “five_in_one.”

All tools we describe here are prototypes and would need additional modifications to meet production quality and usability standards. However, the tools demonstrate important and useful functionality that should exist in production grade tools. In this chapter we describe each in turn. Section 6.1 discusses “scrubs,” and Section 6.2 discusses “five_in_one.”

6.1 scrubs

scrubs captures and enables screenshot edits in real-time. It uses a VNC client as described in Chapter 4 to collect screenshots from a capture target. Figure 6.1 depicts an example of scrubs in action.

A user invokes scrubs on the commandline with the usage defined in Figure 6.2. Imme-
Figure 6.1: Text-redaction using scrubs. The top image depicts a screen capture target executing a Google web search about the RFB protocol. The machine runs Ubuntu 9.10 linux in VMware on a MacBook Pro with OS X 10.7. The redacted version of the web query is produced by scrubs running natively on the same MacBook Pro. The linux machine exports its screen over the network using a VNC server called x11vnc [36] and scrubs connects over the network to this server and receives screen updates from it using the RFB protocol [49]. The bottom machine reflects updates made to the top machine in real-time, but in redacted form.
Usage: scrubs [OPTIONS]
-h, --host=HOST log video of this host (default localhost)
-p, --port=PORT port to connect (default 5900)
-a, --autosave autosave every record_window/2 seconds (default 0)
-w, --record_window=TIME sliding record window
    (default 10sec; TIME < 0 for infinite window)
-f, --file=FILE avi file to store logged video
    (default [date-host].avi)
-n, --no_viewer DO NOT display a viewer while logging (default display viewer)

Help options:
-?, --help Show this help message
--usage Display brief usage message

Figure 6.2: scrubs usage. scrubs relies on libpopt to parse usage and display the content of this figure. Upon execution, scrubs connects to VNC server <host> at port <port>. <record_window> defines how many seconds of screenshot video should be save to disk during a save operation, and if autosave is enabled, scrubs will save the past <record_window> seconds of screen video to disk every <record_window>/2 seconds. <file> is the name of the file that will receive screen video whenever a file is written. scrubs writes video in raw format within an AVI container and overwrites the file on each subsequent autosave operation. Finally, <no_viewer> instructs scrubs to log without displaying the viewer (which also servers as the video editor).

Immediately upon execution, scrubs creates a thread using pthreads [61] to request and receive screen updates from the remote VNC server. After establishing the VNC connection and upon receiving updates, the VNC thread places them in a queue to be processed by the main scrubs thread.

To prevent deadlock and synchronization issues, the queue is protected by a mutex. The VNC thread acquires the mutex before it enqueues data and releases it afterwards. The main thread acquires the same mutex before it removes updates to be processed and releases it afterwards. To reduce the number of mutex operations, the main thread removes all screen updates each time it access the queue. It places them on a data structure that remains unmodified by the VNC thread.
While the VNC thread sends screen update requests and handles their subsequent receipt, the main thread iterates through an event loop where it processes any incoming screen updates, updates its display of redacted data to a compendium of all the latest screen updates, waits for keystrokes and mouse events from the user, handles any such events, and autosaves if the user enabled that option. Displaying a compendium of latest updates maintains a visually accurate and snappy viewer, while storing all updates in memory ordered by receipt from the server, enables lossless video creation.

### 6.1.1 Editing with scrubs

The scrubs user can edit video through keyboard and mouse interactions with the scrubs viewer. All edits to a video are applied to all subsequent screen updates of the current session, and a “session” is simply a commandline invocation of scrubs. Table 6.1 describes the key events processed by scrubs and Figure 6.3 demonstrates how edit mode allows a user to both redact and unredact regions of their choosing. Depending on the current operating mode, the mouse allows a user to create custom redaction rectangles on the screen.

The scrubs editor functions in two separate modes: “edit” and “record” and the ‘e’ keystroke toggles between them. Record is the default mode and displays screen updates as the capture target changes. Edit mode pauses display of screen updates and allows the user to click and drag the mouse to define custom redaction and/or unredaction rectangles. To visually denote edit mode, the system displays a shade of white over the entire viewer display. While paused for user edits, the system continually processes and maintains received screen updates in the background, and upon returning to record mode, the system displays a compilation of all updates processed during pause.

---

1. Because they are one-in-the-same, we use the term “viewer” and “editor” interchangeably.
<table>
<thead>
<tr>
<th>Key/Mouse</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘q’ or ‘Q’</td>
<td>Do not save and quit scrubs immediately.</td>
</tr>
<tr>
<td>‘e’</td>
<td>Toggle “edit mode”, which is used to create custom redaction and unredaction regions on the screen. All edits are permanent in the current viewing session after toggling out of edit mode.</td>
</tr>
<tr>
<td>‘s’</td>
<td>In record mode, save the current &lt;record_window&gt; seconds of video.</td>
</tr>
<tr>
<td></td>
<td>In “edit mode” and while the mouse is dragged, toggle the color of the current redaction rectangle between black (redact) and the image behind the redaction rectangle (unredact).</td>
</tr>
<tr>
<td>Escape or right-mouse left-drag</td>
<td>Define custom redaction regions</td>
</tr>
</tbody>
</table>

Table 6.1: Key events processed by scrubs. “Key/Mouse” is the key or mouse event that initiates an action and “Description” summarizes the action.

6.2 five_in_one

The five_in_one tool manipulates static images and also serves as a tool to explore research ideas. A user invokes five_in_one on the commandline with the usage defined in Figure 6.4. After the image is loaded, five_in_one applies Canny-based redaction as described in Section 5.2 and displays outlines of the redaction rectangles in the five_in_one editor window.

Figure 6.5 uses the screenshot from Figure 5.1 to demonstrate five_in_one look-and-feel. Table 6.2 lists editor modes a user can toggle with various keystrokes. For each mode, the table lists one or more figures that demonstrate mode functionality. Table 6.3 describes the key and mouse events processed by five_in_one—immediately upon execution, five_in_one begins an event loop and waits for user and mouse input.

The primary mode of the tool is edit, and in this mode, rectangles appear red, selected rectangles blue, matched templates green, and custom-drawn rectangles red. When drawn filled, all but redaction rectangles maintain their outlined color; red-outlined redaction rectangles become black filled rectangles.

five_in_one allows a user to create “templates” (see ‘t’ in Table 6.3) that can be saved
Figure 6.3: Custom redaction and unredaction of the top image in Figure 6.1. When a user clicks and drags the mouse in edit mode, they produce a custom unredaction region. By tapping the space key before the left mouse has been released, the region flips between redaction and unredaction colors. After a region is defined, the system searches the unredacted image for any matches to the unredacted form of the selected region. If it finds one, it copies the user’s custom region to the matching area. Note that red highlighting-rectangles disappear when a user hits “Escape” during a drag operation or when the user exits edit mode (with keystroke ’e’).

to file and redacted automatically from many images upon image load. Each template is maintained by five_in_one as an OpenCV matrix and templates are serialized into the binary format for file storage described in Subsection A.2.1 of Appendix A. The template file contains a list of these matrices preceded by a 4-byte count of templates in network byte order. Any templates that were loaded on program start are saved along with any newly defined templates. Because of this behavior, template-saving is an append operation.
Usage: five_in_one <img> [templates]

<img> the image to process
[templates] the file to retrieve templates from
and write templates into

Figure 6.4: five_in_one usage. Invoke five_in_one to process image <img>. Optionally load image templates found in file [templates] and wherever a match is found in <img>, write a redaction rectangle. No changes are every saved to the original image <img>.

Figure 6.5: Default five_in_one look-and-feel. When the editor executes, it begins in edit mode. Redaction rectangles are unfilled and painted red. A blue crosshair appears to help the user align the mouse as they manipulate the image. The user can drag the mouse to select rectangles for subsequent operations. A selected rectangle turns blue, and the “Escape” key deselects all selected rectangles and reverts their color from blue to red.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Figure(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edit</td>
<td>6.6, 6.7, 6.8, 6.9</td>
<td>The default mode used to merge selected rectangles, select and delete redaction rectangles, define templates, and copy selected rectangles to matching regions of the image.</td>
</tr>
<tr>
<td>rectangle</td>
<td>6.10</td>
<td>Display image only.</td>
</tr>
<tr>
<td>redaction-only</td>
<td>6.10</td>
<td>Display rectangles only.</td>
</tr>
<tr>
<td>draw</td>
<td>6.11</td>
<td>Draw custom redaction rectangles.</td>
</tr>
<tr>
<td>grid</td>
<td>6.12</td>
<td>Display a grid.</td>
</tr>
<tr>
<td>fill</td>
<td>6.13</td>
<td>Display filled rectangles.</td>
</tr>
</tbody>
</table>

Table 6.2: five_in_one editor modes. “mode” is the mode name, “Figures” denote which Figures demonstrate each mode, and “Description” summarizes mode functionality.
<table>
<thead>
<tr>
<th>Key/Mouse</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘m’</td>
<td>edit</td>
<td>Merge selected rectangles.</td>
</tr>
<tr>
<td>‘r’</td>
<td>rectangle</td>
<td>Enable and disable display of redaction rectangles.</td>
</tr>
<tr>
<td>‘R’</td>
<td>redaction-only</td>
<td>Enable and disable display of the underlying image.</td>
</tr>
<tr>
<td>‘f’</td>
<td>fill</td>
<td>Switch between display of filled and outlined redaction rectangles.</td>
</tr>
<tr>
<td>‘Q’ or ’q’</td>
<td>any</td>
<td>Exit five_in_one immediately without saving.</td>
</tr>
<tr>
<td>‘d’</td>
<td>draw</td>
<td>The crosshair changes color to red and dragging allows the user to create custom redaction rectangles.</td>
</tr>
<tr>
<td>‘g’</td>
<td>grid</td>
<td>Draw a black grid on the screen using a 20-pixel row and column step.</td>
</tr>
<tr>
<td>‘+’ or ‘-’</td>
<td>any</td>
<td>Increase and decrease the line width of redaction rectangles.</td>
</tr>
<tr>
<td>‘L’</td>
<td>any</td>
<td>Save image layout analysis to file. See Subsection 7.2.4 for more information about the analysis.</td>
</tr>
<tr>
<td>‘T’</td>
<td>any</td>
<td>Thin-out superfluous redaction rectangles. Those rectangles contained wholly within another rectangle will be subsumed and eliminated. Each newly created redaction rectangles grows in size with each ‘T’ operation. See Figure 6.14.</td>
</tr>
<tr>
<td>‘c’</td>
<td>edit</td>
<td>Copy selected redaction rectangles to image regions that match the unredacted image under the redaction rectangle.</td>
</tr>
<tr>
<td>‘t’</td>
<td>edit</td>
<td>Append selected redaction rectangles to the list of templates. On image load, the system draws redaction rectangles over regions where a template matches image content.</td>
</tr>
<tr>
<td>‘s’</td>
<td>any</td>
<td>Save the current screen as displayed (with all redaction rectangles but without a crosshair cursor).</td>
</tr>
<tr>
<td>‘S’</td>
<td>any</td>
<td>Save the list of templates to a file as a binary list of OpenCV matrices preceded by a 4-byte length in network-byte-order, and remove duplicate templates before saving. See Subsection A.2.1 for the binary matrix format.</td>
</tr>
<tr>
<td>Delete</td>
<td>edit</td>
<td>Delete the selected redaction rectangle.</td>
</tr>
<tr>
<td>Escape or right-mouse</td>
<td>edit</td>
<td>Unselect the current selected rectangles.</td>
</tr>
<tr>
<td>left-drag</td>
<td>edit</td>
<td>Select rectangles.</td>
</tr>
</tbody>
</table>

Table 6.3: Key and mouse events processed by five_in_one. “Key/Mouse” define the event five_in_one processes, “Mode” defines the mode associated with the event, and “Description” summarizes the action initiated by the event.
Figure 6.6: Merge redaction rectangles. The user selects a set of rectangles to merge and types the key ‘m’ to merge them into the largest rectangle that bounds the selected set. After merging, the new rectangle is automatically selected and the user can type the key “Escape” or click the mouse to deselect it.

Figure 6.7: Delete redaction rectangles. The user selects a set of rectangles to delete using the mouse and types the key “Delete.” At that point, red redaction rectangles are removed from the image.

Figure 6.8: Define and copy redaction templates. On image load, five in one matches templates found in the template file passed on the commandline against the image and places a redaction rectangle over each region that matches (see Figure 6.4 for usage details). To define templates, the user selects a set of rectangles with the mouse and types the key ‘t’. The system highlights the selected rectangles with green and internally marks them as templates. At that point, the user can save all templates to file for later use by typing the key ‘S’ (including the templates loaded at application start).
Figure 6.9: Copy selected redaction rectangles to matching regions of an image. The user selects a set of rectangles to copy to matching regions of the image and types the key ‘c’ to copy them. The system then matches the image pixel-by-pixel against the image using the OpenCV routine “matchTemplate” with the square differencing method and a threshold of 0.00006 as described in Subsection A.2.2. Then, the system draws a redaction rectangle over any matching regions. After the copy operation, any newly drawn rectangles remain selected and the user can press the key “Escape” or left or right mouse button to deselect them. In this figure, the character “n” was selected initially and five_in_one found the “n” character next to it and added a redaction rectangle around it.
Figure 6.10: Toggle display of redaction rectangles and the underlying image. The user can type the key ‘R’ to toggle between the top and middle image (“rectangles-only mode”)—display of the underlying redacted image is enabled and disabled. The key ‘r’ toggles between the top and bottom images (“rectangle mode”)—display of redaction rectangles is enabled and disabled. When the user exits edit mode, the crosshair disappears and the user no longer has the ability to manipulate redaction rectangles.
Figure 6.11: Draw custom redaction rectangles. The top image displays the redacted image in edit mode. After typing the key ‘d’, the editor switches to drawing mode. In this mode the crosshairs change to red and dragging the mouse creates new redaction rectangles. The bottom image shows one such rectangle. The user can type the “Escape” key during while dragging to undo the draw operation in progress. A rectangle can be manipulated as any other displayed rectangle after reverting to edit mode.
Figure 6.12: Draw a grid. Draw a grid over the displayed image, independent of the existing mode. Grid line spacing is 20 pixels in both row and column directions, and drawing starts in the upper left corner.

Figure 6.13: Toggle filled drawing of redaction rectangles. Typing the key ‘f’ toggles fill mode on and off. The top image shows edit mode with fill turned off. It contains redaction rectangles in red, a template highlighted in green, and selected rectangles in blue. The bottom image shows the same image drawn with fill enabled. The green and blue outlined rectangles become filled with the same color and all red rectangles become filled black ones. The green- and blue-fill are helpful when searching for rectangles or templates copied around the screen—the user can quickly and repeatedly toggle ‘f’ to make those rectangles easy to find visually.
Figure 6.14: “Thin out” superfluous rectangles. Begin in edit mode and type the key ‘T’ to subsume superfluous redaction rectangles. Between the top and bottom images, the rectangle count dropped from 969 to 237, an \( \approx 75\% \) reduction. Note how small rectangles wholly contained within larger ones are subsumed by the larger one after pressing ‘T’. Also, the remaining rectangles have grown by approximately 1 pixel in each direction. Continually pressing ‘T’ will eventually subsume all rectangles into a single rectangle that encompasses the entire screen. This behavior occurs because of how thinning is implemented. First, the system creates a filled redaction mask of the image (see key ‘f’), draws new bounding boxes around each filled rectangle, and replaces all existing redaction rectangles with the new bounding boxes. Because the new bounding boxes were created by outlining existing rectangles, the new ones are slightly larger than the ones they replace.
Chapter 7

Analysis

In the past four chapters we described the text-redaction system, screenshot capture, text-redaction, and image editing. Now we present empirical analysis of the system. In Section 7.1 we overview the analysis, including the data sets and tools we used. In Section 7.2 we analyze many aspects of the image processing, including Canny- and Gabor-based text redaction, screenshot layout, template matching, and text-redaction impact on visual context. In Section 7.3 we close the chapter by analyzing latency of Canny- and Gabor-based text redaction, generating Gabor features and loading them from disk, and template matching with different template sizes.

7.1 Overview

Our empirical analysis derives from 80 screenshots captured from two electronic health record (EHR) systems at two large healthcare providers. Datasets contain fake patient data but are still considered sensitive and therefore we do not show them within this dissertation. Appendix B describes the datasets in detail. Additionally, we developed a number of tools to analyze data and Appendix C describes them.
Figure 7.1: Failure of Canny-based text redaction on an EHR screenshot. Both left and right figures exhibit Canny-based redaction masks on an EHR screenshot from dataset 1. The left figure depicts the default mask, and the right depicts a hand-tuned version of the same mask. For many EHR screenshots, Canny-based redaction exhibits similar behavior: the system redacts all text, but the large false-positive rate eliminates potentially useful screenshot context. We used five_in_one (Section 6.2) to eliminate the large redaction rectangle from the left image and reveal the one on the right.

7.2 Image Analysis

7.2.1 Canny-based Text Redaction

Canny-based text redaction requires improvements before the system can apply it meaningfully to EHR datasets. We discuss them here but leave an improved implementation and analysis of Canny effectiveness for future work. Figure 7.1 (and 5.2) demonstrates the principal reason Canny fails: it generates redaction rectangles that subsume large fractions of a screenshot, which reduce potentially useful, non-private screenshot context.

Figure 5.1 demonstrates another Canny problem: it generates a number of superfluous rectangles during the redaction process. Reducing their number could improve processing performance of operations that apply to each rectangle.

We analyzed both problems more closely and found the trends shown in Figure 7.2. Using them as a guide, the system could automatically remove large redaction rectangles that include a large fraction of all other rectangles and large rectangles that lead to very small area ratios.

After thinning rectangles according to these trends, the system could apply the technique.
described in Figure 6.14 and reduce many other superfluous rectangles. In that example, the rectangle count dropped by approximately 75%.

![Histogram of Bounding Box Content](image)

![Histogram of Bounding-Box Area Ratios](image)

**Figure 7.2:** Relationships of redaction rectangles. The top plot displays the number of redaction rectangles contained with a given redaction rectangle. The x-axis is the fraction of total redaction rectangles in a screenshot contained wholly within a given redaction rectangle and the y-axis is a count of x-values. Rectangles that comprise the entire screenshot include all others and thus appear to the far right on the plot. The second plot displays the ratio of area for each pair of redaction rectangles in a screenshot. Its x-axis is the area ratio of rectangles within a given screenshot where one rectangle of the pair wholly contains the other, and the y-axis is a count of all x-values. Pairs where one rectangle comprises the entire screen can be found on the far left of the plot because of the extreme nature of the ratio. The system could automatically eliminate redaction rectangles that fall on the far right of the top plot and far left of the bottom. Both plots were produced using dataset 1.

In another problem demonstrated by Figures 5.1, 7.5, 7.10 or 7.11, Canny leaves whitespace between redaction rectangles and reveals word-length frequency that observers may use to reveal redacted text.

Figure 7.3 depicts a trend the system could exploit to merge rectangles within close proximity of one another and hence eliminate this whitespace. Figure 7.4 depicts the predicate that must evaluate to true before the system considers merging. The heuristic only accepts whitespace left by rectangles aligned in height and position similar to text in a line.
Figure 7.3: Horizontal distance between rectangle pairs. The x-axis is the horizontal distance in pixels between the end of one rectangle and the beginning of another, and the y-axis is the count of how many times a particular distance appeared. This histogram includes all screenshots from dataset 1 without any optimizations to eliminate or thin redaction rectangles beforehand. Note that rectangle pairs comprising the far left peak could be merged horizontally within their respective screenshot to eliminate inter-word whitespace. This figure only includes whitespace accepted by the heuristic depicted in Figure 7.4.

Figure 7.4: Predicate used to evaluate rectangle whitespace. If \((x + w) \leq x', (y - \frac{h}{2}) \leq y' \leq (y + \frac{h}{2})\), and \(y + \frac{h}{2} \leq (y' + h') \leq y + \frac{3}{2} h\), then the system includes the value \(x' - (x + w)\) in the histogram of Figure 7.3. The system evaluates all rectangles.

7.2.2 Unsupervised Gabor-based Text Redaction

Measuring quality of text-redaction techniques requires a ground-truth dataset comprised of screenshots with labels that mark each screenshot pixel as either text or not. Such labeled
data can be compared to the output of a text redaction technique and the difference between pixel labels represents error in the technique.

We lacked such ground-truth labels for our datasets and therefore proceeded down two paths to analyze redaction.

**Redaction-Analysis Path 1** First we chose a set of “best” redacted screenshots manually using Gabor-based unsupervised redaction (see the application defined in Figure C.16) and measured false negatives by visual inspection. Then we generated ground-truth labels for the entire dataset manually and used all hand-picked ground truth to measure effectiveness of Gabor-based supervised redaction.

We executed this plan for the entirety of EHR dataset 1 and followed the same procedure for our second dataset, except that we did not measure false positives by visual inspection. Instead, we trained the supervised classifier on the first dataset and then measured its application on the second.

We trained and tested in this order because we possessed dataset 1 for a period of time before obtaining dataset 2. This approach reflects a “real-world” scenario where all training variants do not exist initially. In practice, one could retrain classifiers after new datasets have been acquired, and we could have done so in our analysis. However, we chose to continue studying other aspects of the system and left such analysis as subject of future work.

To choose screenshots manually for the first path, we consider 5 output classes of the redaction process:

1. **False positives** are non-text pixels that have been redacted.

2. **False negatives** are text characters that have not been redacted. These are the worst type of error because they may reveal sensitive information directly.

3. **Partial false negative** are partial text characters that have not been redacted. Depending on the extent of a revealed character, the error may not be detrimental.
Figure 7.5: Problems labeling redacted-text by visual inspection. Many judgements apply to labeling ground-truth pixels and during quantitative assessment of various redaction techniques, such judgements may produce pixel-mismatches between ground-truth labels and redaction output and thus introduce quantitative error. However, little or no qualitative impact may exist in redaction quality. This figure demonstrates a few cases where this mismatch could occur. “Fill regions” of bounding boxes may not precisely overlap, some letters may remain partially uncovered, height differences among redacted regions may exist, and white space between words or letters may exist.

4. **True negatives** are non-text pixels that have not been redacted. This is a desirable output state.

5. **True positives** are text characters that have been redacted. This is also a desirable output state.

True positives must include some surrounding pixels, otherwise redaction and changing the display color of text are equivalent (assuming text is not changed to the background color). In general, the number of surrounding pixels that should be redacted is ill-defined, and thus qualitative judgement rather than quantitative analysis underlies a portion of “class labeling by visual inspection.” Consequently, our “ground-truth” has subjective components that qualitatively appear reasonable. Figure 7.5 demonstrates issues that arose during our analysis.

Note that we differentiate between pixels and characters when examining redaction output. In cases where we measured redaction output states of a redacted screenshot manually, we counted characters. When the system performed the measurements, it counted pixels. Additionally, we tabulated partial false negatives because partial characters are possible results of text redaction. Because pixels are an indivisible unit, the system only measured whole pixels.

We did not count false positives by visual inspection because they represent non-
characters and in many cases require counting pixels to be meaningful. With over 1 million pixels per screenshot, this proposition was beyond the scope of our efforts.

Additionally, some false positives may be non-text symbols the system should redact conditionally, and in general, false positives do not reveal information but reduce screenshot context. A clear method does not exist to quantify this latter effect without counting pixels. In combination, these issues greatly complicate counting false positives manually and led us to avoid doing so.

**Redaction-Analysis Path 2** In the second analysis path, we redacted screenshots using Gabor-based unsupervised classification with $1 < k < 3$ and chose the best redaction manually, labeled it as ground-truth, and used that ground truth to measure Gabor-based supervised classification.

We manually analyzed unsupervised Gabor-based text redaction on a small number of screenshots from our first EHR dataset. As described in Section 7.1, we tabulated false negatives and partial false negatives of text characters. For the reasons outlined there, we did not count false positives.

Each analyzed screenshot contained between 1100 and 2000 characters and we likely introduced a small number of human errors counting. To analyze a breadth of screenshot content we chose a few screenshots with low pairwise similarity measures as described in Subsection 7.2.6. Table 7.1 summarizes our findings.

### 7.2.3 Supervised Gabor-based Text Redaction

We trained the supervised classifier on dataset 1, which may have been fortuitous. Qualitatively, the dataset appears visually more complex than the dataset 2. Figure 7.6 examines the rectangle count for each screenshot of each dataset, which supports one measure of complexity. Figure 7.7 compares color distributions of datasets 1 and 2 and demonstrates another.
Table 7.1: Unsupervised Gabor-based redaction effectiveness. False negatives ranged 
[.005, .03] and partial false negatives ranged from [.02, .045]. Note that we only counted false 
negatives if characters were unredacted entirely. Qualitatively, some partial false negatives 
revealed no more than one or two pixels of the underlying text and in other cases only a few 
redaction pixels covered text.

<table>
<thead>
<tr>
<th>Total Characters</th>
<th>False Negatives (Count/Fraction)</th>
<th>Partial False Negatives (Count/Fraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1149</td>
<td>5 / .0044</td>
<td>22 / .0191</td>
</tr>
<tr>
<td>1102</td>
<td>4 / .0036</td>
<td>19 / .0172</td>
</tr>
<tr>
<td>1094</td>
<td>14 / .0128</td>
<td>41 / .0375</td>
</tr>
<tr>
<td>2145</td>
<td>58 / .0270</td>
<td>94 / .0438</td>
</tr>
<tr>
<td>1779</td>
<td>22 / .0124</td>
<td>58 / .0326</td>
</tr>
</tbody>
</table>

Figure 7.6: Sorted rectangle count of each screenshot in each dataset. The x-axis is the screen- 
shot number and the y-axis is the count of redaction rectangles that exist within the screenshot. 
To derive these values, an application redacts each screenshot using Canny-based redaction as 
described in Section 5.2, but instead of generating and drawing bounding boxes around each 
contour found in each screenshot, it simply counts them as described in Figure C.1. The first 
dataset generally contains more rectangles than the second, which supports the qualitative 
notion that dataset 1 is visually more complex than dataset 2.

Figure 7.8 highlights the effectiveness of supervised Gabor-based text redaction applied 
to the second dataset after being trained on the first. The ground-truth labels of the second 
dataset were chosen manually from a set of unsupervised Gabor-based redaction alterna-
Figure 7.7: Normalized color variety of each dataset. The x-axes are color values and y-axes are fraction of total pixels. Note how color variation in dataset 1 has less variation than dataset 2. This suggests a more even color distribution in dataset 1 and supports the qualitative notion that dataset 1 is visually more complex than dataset 2.

7.2.4 Layout analysis

In some circumstances, Gabor- or Canny-based redaction can hide text but leave screen patterns that reveal sensitive information to observers. Figure 7.9 demonstrates one such problem with a Gabor-redacted screenshot. In the example, viewers can distinguish between objects whose redacted and unredacted versions connote similar information.

In our dataset, such objects appeared within rows and columns of like objects—some were checked and some not. Additionally, some rows and columns of text existed as in Figure 7.10, which depicts a Canny-redacted EHR screenshot mask. Such missing row or column elements may provide helpful information to a knowledgeable observer.

Figures 7.11 and 7.12 demonstrate layout trends the system could exploit to \( k \)-anonymize screenshots visually and reduce the aforementioned problems. In doing
so, the system would fill-in rows, columns, and normalize the state of certain objects such as checkboxes so they all appear similar—redacted, uncheck, checked, etc. Some screenshots may require this level of protection, but probably not all.

7.2.5 Templates

EHR records contain visual “alerts” such as red exclamation points or yellow-highlighted text that connote an exceptional situation to the viewer. Automatically finding alerts within EHR records can support bulk screenshot analysis such as determining which and how many alerts exist, whether they correspond correctly to other metadata within the record.
Figure 7.9: Leaking checkbox state with redaction. The screenshot depicts a plain screenshot and its Gabor-redacted counterpart. Toggled checkboxes are distinguishable in the redacted version and the problem could be addressed by “normalizing” the column so that each box appears identical.

set, etc. To demonstrate how our system can support such analysis, Figure 7.13 depicts the counts of 18 unique alerts chosen from the 51 screenshots of dataset 1.

Red Pixel Analysis

Professionals in the healthcare field presented the idea that alerts contain red pixels, and so we hypothesized that by examining histograms of red screenshot pixels, one could discover the existence and location of alerts within a screenshot. Figure 7.14 summarizes our findings.

7.2.6 Text-Redaction and Screenshot Context

Visual similarities between a screenshot and its redacted version or differences between two redacted screenshots enable observers to recognize captured application windows despite

59
the presence of text redaction. Such useful context is a source of differentiating “information” that observers unconsciously exploit to distinguish screenshots from one another and recognize them in the first place.

We assert that such differentiating information or “screenshot context” is important to maintain when redacting text because it may provide useful metadata to knowledgeable observers. Thus, our system does not blindly redact entire screenshots and leave the end-user to unredact a majority of context (in addition to any other screenshot features they wish to unredact). The left-hand screenshot of Figure 7.1 depicts such a full-screen redaction.

We devised a metric to quantify the “information” that exists among screenshot pairs, and the application listed in Figure C.2 computes the value. The application computes the
Figure 7.11: A marked-up redaction mask using Canny-based text redaction. This figure depicts a Canny-based redaction mask with a layout grid. The underlying screenshot is an EHR screenshot taken from dataset 1, and black horizontal and vertical lines were drawn every 20 pixels starting in the upper left corner. We outlined a column of redaction rectangles at pixel $x \approx 180$ and a row of redaction rectangles at pixel $y \approx 40$. These values can be found empirically using the measurements depicted by Figure 7.12. Note that the markup described here is not limited to Canny-based redaction.

The application begins computing in the upper-left corner of each screenshot and calculates from left to right over each row before analyzing the next row. Calculations are not made for pixel addresses of one screenshot that lie outside of the other because no pair exists.

The application accumulates changes when it text redacts pairwise-pixel values $S(x, y)$.
Figure 7.12: Analyzing redaction layout of an EHR screenshot. We analyzed the EHR screenshot that underlies Figure 7.11 to derive the histograms depicted here. From the top down, the first and second histograms depict the fraction of black redaction pixels that exist at particular $x$ and $y$ coordinates respectively. The third and fourth histograms depict the fraction of redaction rectangles that exist at particular $x$ and $y$ coordinates respectively, considering only the upper-left corner of each rectangle. The first and third arrows correspond to the column outlined at pixel $x \approx 180$ in Figure 7.11, and the second and fourth arrows correspond to the row outlined at pixel $y \approx 40$. Minor x-tics in each plot are spaced every 20-pixels, and the upper-left corner of Figure 7.11 is $(0, 0)$. 

62
Figure 7.13: Alert template matches in EHR dataset 1. The x-axis depicts the alert number and the axis the total count. We hand-picked alerts from a small number of screenshots in dataset 1 using five_in_one tool to save templates (see Figure 6.8 for details about saving templates). Then we fed the file to count_matching_templates depicted in Figure C.5 to count the total number of templates that exist in dataset 1.

and \( S'(x, y) \) of screenshots \( S \) and \( S' \) in pair \((S, S')\) with the redaction operation \( R(p) \). The following equation defines how the application counts the total number of pairwise changes induced by text redaction:

\[
\text{total changes} = \sum_{x,y} \begin{cases} 
1 & x \in S \land x \in S' \land y \in S \land y \in S' \\
& \land (S(x, y) \neq 0 \lor S'(x, y) \neq 0) \\
& \land (R(S'(x, y)) = 0 \land R(S'(x, y)) = 0), \\
0 & \text{otherwise}
\end{cases}
\]

Changes accumulate only when a pair of pixels exist, at least one pixel of the pair begins non-black, and both pixels are redacted.
Figure 7.14: Red color values of four separate EHR screenshots. Each x-axis depicts the value and each y-axis depicts the fraction of screenshot pixels that possessed a given red channel value. We analyzed red-pixel histograms of four screenshots from dataset 1 and found no obvious trend that suggests red-pixel count can reveal whether an alert exists within a screenshot or not. Upon visual inspection, we found that alerts contained varying numbers and shades of red pixels, and non-alert regions of some screenshots (including text regions) contained red pixels. In one instance, text was written in rows with alternating dark- and light-pink backgrounds, which are comprised of red pixel values. The first screenshot had one alert, the second pink rows, and the final two, many alerts. Our findings do not preclude more complex signal processing from discovering alerts based on red pixel counts.

Next the application computes the pairwise fraction for screenshot pair \((S, S')\) as

\[
\frac{\text{total changes}}{\text{total pixels}}
\]

where

\[
\text{total pixels} = \max \text{ width} \times \max \text{ height} \tag{7.1}
\]

of both screenshots of the pair.

When one or both pixels begin non-black and the pixels correspond to text, redaction
removes differentiating information by converting both values to black. Removing information reduces differentiating screenshot context. Taken to the limit, redaction blackens each screenshot entirely and leaves no differentiating information. Except for size and metadata unrelated to pixel values, screenshot context disappears in this case.

Figure 7.15 depicts the effect of redaction on image pairs and Figure 7.16 depicts the number of matching black pixels within each screenshot pair \((S, S')\) before redaction is applied.

![Redaction Effect on Pairwise Image Information](image)

**Figure 7.15**: Redaction effect on image information. These figures depict the effect of redaction on screenshot pairs from dataset 1. The x-axis of both plots is a pair identifier and both y-axes are the fraction text-redacted pixels the images have in common. The top figure depicts all image pairs in dataset 1, including those with identical screenshots. The bottom figure depicts pairs where screenshots are paired with themselves. Pairing an image with itself computes the effect of redaction on a single image. For all pairs, the mean fraction of overlapping redacted text is 0.093 with a stddev of 0.035, and for pairs of identical screenshots, the mean is 0.237 with a stddev of 0.036. Redaction preserves 90% of differentiating information in all pairs and 76% in pairs of identical screenshots—on average, redaction affects no more than 24% of the pixels in any screenshot. Based on the maximum value plotted, the system redacts no more than 35% of any screenshot in dataset 1. Thus, the system preserves a large fraction of potentially useful screenshot context.
Figure 7.16: Fraction of black-pixels within screenshot pairs. This figure depicts screenshot pairs from dataset 1, excluding pairs where pair elements are identical screenshots. The x-axis is a pair identifier (which does not correlate to the x-axis in Figure 7.15) and the y-axis is the fraction of pixel pairs where both pixels in the pair share a black value. As Figure 7.15 demonstrates the effects of redaction, only pairs of black pixels are left unaffected by redaction—they are already black. Within dataset 1, a small number of black pixels exist at the same location in different screenshots. 151 screenshot pairs have no overlapping black pixels.

Pairwise-Screenshot Similarity

In another technique the system computes a “distance” between two screenshots by counting the number of pixels that match within the pair. As above, counting begins in the upper left of each screenshot and moves across columns before moving to the next row. However, all non-overlapping pixel values (due to a screenshot-size mismatch) count as differences and reduce similarity. The total pixel count is identical to equation 7.1 and the fraction of matching pixels within a pair is defined as the following:

\[
\frac{\text{similar count}}{\text{total pixels}}
\]
The application listed in Figure C.3 computes this distance and Figure 7.17 depicts how text redaction does not completely eliminate pairwise differences.

Because EHR screenshots are nearly identical in size and aligned in content, e.g., items such as menus are not pixel-shifted among screenshots, this measurement gives a notion of similarity that enables useful pairwise-screenshot comparisons. We qualitatively validated our technique visually; screenshots that matched (or not) with a large fraction of pixels according to the metric were visually comparable (or not).

Figure 7.17: Effect of text redaction on pairwise screenshot similarity. We computed the effect of text redaction over 1275 screenshot pairs of dataset 1 (we excluded pairs of identical screenshots). The x-axis of both plots is the screenshot pair identifier. The top plot depicts the change in pixel similarity among pairwise images after redaction, and the bottom plot displays the fraction of identical pixels among image pairs, both before and after redaction. In the top plot, fractions that fall above the horizontal line correspond to screenshots that are more similar and those below the line correspond to screenshots that are less similar. The latter case occurs when pixels that matched before text redaction do not match afterwards. Overall, redaction has little impact on pairwise screenshot similarity with changes ranging from 2 – 15%. Text-redaction retains potentially important screenshot context in EHR dataset 1.
7.3 Processing Latency

The principal computational components of our system include text redaction and template matching, and we empirically quantify their latency characteristics in this section.

A MacBook Pro running Mac OS X 10.7 with 8 GB of memory serves as the experimental platform. An AES-256-encrypted disk image stores image, feature, and label files associated with redaction. See Appendix B for more information about these files. All file loads are measured using a cold file cache; remounting the encrypted storage volume before each set of file-access sensitive measurements flushes buffers and ensures a consistent, cold cache. The dtrace [50] tool generates some measurements and test applications generate others by programmatically printing timing information to the screen (C.6, C.9, C.10, C.18, C.19, C.20).

7.3.1 Text Redaction

We analyzed numerous aspects of the computational latencies associated with text redaction. We ran computations against all images in dataset 1 to derive measurements, and for each data point we compute a mean, standard deviation, and standard deviation / mean. The last value provides a “normalized” notion of variability with respect to the mean.

Table 7.2 compares the cost of Canny-, supervised Gabor-, and unsupervised Gabor-based text redaction, Table 7.3 describes latencies required to generate Gabor features and labels, and finally, Table 7.4 describes latencies to load Gabor features and labels from disk.

7.3.2 Template Matching

Finally, we analyzed the computational cost of matching templates against a screenshot. First the system compares a template against all pixels in an image and then it searches through comparison results to find matches. Figure 7.18 depicts the latency of comparison operations and Figure 7.19 depicts the latency of searching through results to find template
<table>
<thead>
<tr>
<th>Redaction Type</th>
<th>Mean Latency (seconds)</th>
<th>stddev (seconds)</th>
<th>stddev / Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canny</td>
<td>0.096732016</td>
<td>0.009194057</td>
<td>0.095047</td>
</tr>
<tr>
<td>Unsupervised Gabor</td>
<td>2.077065258</td>
<td>0.722214985</td>
<td>0.347709</td>
</tr>
<tr>
<td>Supervised Gabor</td>
<td>2.077105231</td>
<td>0.016699164</td>
<td>0.008040</td>
</tr>
</tbody>
</table>

Table 7.2: Latency to classify pixels for redaction. All measurements include the time required to build a redaction mask used by the system to paint a redacted image. For Canny, we measured the time required to find redaction rectangles within an image, construct bounding boxes, and add this information to a list. For unsupervised Gabor, we measured the time required to segment Gabor features using kmeans classification and build a redaction mask used to paint a redacted image. For supervised Gabor, we measured the time required to classify each pixel as “text” or “not text.”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mean Latency (seconds)</th>
<th>stddev (seconds)</th>
<th>stddev / Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>0.364528181</td>
<td>0.004057357</td>
<td>0.011130</td>
</tr>
<tr>
<td>Build</td>
<td>13.114747233</td>
<td>0.180450313</td>
<td>0.013759</td>
</tr>
<tr>
<td>Normalize</td>
<td>4.568932003</td>
<td>0.033349181</td>
<td>0.007299</td>
</tr>
</tbody>
</table>

Table 7.3: Latency to generate Gabor elements. We measure the time to setup all 28 filters, build feature vectors, and normalize their values. Filtering the image, applying a threshold transform, computing features, and setting up a data structure comprise “building” feature vectors. Subsection 5.3.1 describes Gabor-based text redaction in detail.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mean Latency (seconds)</th>
<th>stddev (seconds)</th>
<th>stddev / Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>5.986632176</td>
<td>0.406650891</td>
<td>0.067926</td>
</tr>
<tr>
<td>Labels</td>
<td>0.226349412</td>
<td>0.068724034</td>
<td>0.303619</td>
</tr>
</tbody>
</table>

Table 7.4: Latency to load Gabor elements from a file on disk. After generating Gabor features, our system can store them persistently to disk. The first row lists the latency of loading Gabor features from disk, which includes both features and their vector coordinates as described in Section B.1. The second row lists the latency of loading feature labels from disk.
matches. We computed figures with square templates of varying sizes and side lengths in pixels that were powers of two.
Figure 7.19: Latency of searching for results of a template match. The x-axis is the side length of a template (note the log scale). The y-axis of the top plot is the mean time in seconds required to search through the results of a template match operation. The y-axis of the bottom plot is the stddev of the latency in the top plot as a fraction of the mean. Each point is computed over all images in dataset 1. Note that a few images vary in size by 2 pixels in height, which does not significantly impact these calculations. See Section B.2 for information about the datasets.
Chapter 8

Discussion

To improve the ability of our system to redact correctly and with reasonable performance, we first need to understand its performance envelope and strengths and weaknesses. In the last chapter, we described empirical aspects of image analysis and computational latencies that begin to provide insight into these issues. In this chapter, we discuss the empirical measurements provided in Chapter 7 and how one might exploit them to improve system performance.

Section 8.1 discusses Canny- and Gabor-based text redaction and layout of redacted text, Section 8.2 discusses image templates and alert matching, and Section 8.3 closes the chapter with a brief discussion on overall computational latency.

8.1 Text Redaction

In this dissertation we describe three text-redaction techniques: Canny, unsupervised Gabor, and supervised Gabor. According to Table 7.2, Canny performs over $20 \times$ faster than either Gabor-based technique, but evidence in the same chapter suggests that it requires tuning before it can be used use in a “real-world” system.
8.1.1 Canny

Subsection 7.2.1 describes how Canny generates as many as four times more redaction rectangles than necessary, which can slow subsequent processing steps that touch every rectangle. Figure 7.1 describes how Canny generates redaction rectangles that cover the entire screen, which reduces non-private visual context for observers. Figure 7.3 describes how Canny leaves whitespace between words, which may enable word-based frequency analysis that reveals redacted text. Finally, the introduction of Canny in Section 5.2 and Figures 5.1 and 5.2 demonstrate these problems and their tendency to miss some text (false negatives) while redacting some non-text (false positives). False positives and negatives can create more work for the end-user relying on our system to prepare a screenshot for sharing. At the potential cost of additional processing, we demonstrated trends that could be exploited to reduce many of Canny-related problems.

Large Canny rectangles that include non-private data can reduce an observer's screenshot context. To handle this problem (described fully in Figure 7.1), the system could exploit the trends shown in Figure 7.2 to eliminate large redaction rectangles that fall within certain analytical bounds. The system could eliminate large rectangles that wholly contain a large fraction of others and larger rectangles whose pairwise area ratio is near zero.

Reducing rectangle count can reduce the latency of subsequent processing steps that involve all Canny rectangles, such as rendering rectangles in an image or analyzing and merging adjacent words. In Figure 6.14, the five_in_one tool demonstrates a technique to reduce superfluous rectangles by merging rectangles where one wholly contains another. In that sample screenshot, the merging technique reduced the count of redaction rectangles by 4-fold (at the cost of an additional processing step). Before merging, all-inclusive, large rectangles should be reduced using the trends depicted in Figure 7.2.

In combination, exploiting word length, ordering of word length, and contextual topic, e.g., medicine or any number of subtopics, may enable frequency analysis and reduce pri-
vacy of redacted text. Merging adjacent words could reduce this problem. The system can merge rectangles horizontally with inter-word spaces that fall within certain analytical bounds by exploiting the trend depicted in Figure 7.3.

8.1.2 Gabor

Unlike Canny, we empirically assessed the ability of Gabor to redact text pixels correctly; Canny required additional fine-tuning before we could meaningfully assess the technique (see Figure 7.1 for an example of a Canny failure). We found no ground-truth datasets of screenshots that label pixels either text or non-text. Thus we began by visually inspecting and analyzing false negative and partial false negative rates of unsupervised Gabor redaction as described in Subsection 7.2.2. We built on this assessment by hand-picking ground-truth for all images of dataset 1 and 2 using unsupervised Gabor redaction. Then we split this ground-truth dataset to serve as a train- and test- set for supervised Gabor redaction. According to the measurements of redaction effectiveness described in Table 7.1 and Figure 7.8, each form of the Gabor approach performs well.

We trained the supervised Gabor classifier on dataset 1 because we possessed it before dataset 2. As demonstrated by the performance of the classifier trained on dataset 1 and tested on dataset 2, this ordering produced a classification accuracy of greater than 90% (see Figure 7.8). Rather than run a battery of experiments such as swapping the testing and training sets or mix the two datasets before testing and training, we chose to continue with other experiments. However, we hypothesized that greater visual complexity present in dataset 1 led to the high classification results of dataset 2. Figures 7.6 and 7.7 attempt to capture empirically how datasets vary from one another in visual complexity. In dataset 1, the number of image features found using Canny and the smooth color gradation suggest, for some notion of “visual complexity,” that dataset 1 has more visual complexity than dataset 2.

1Additionally, an attacker may have external knowledge about particular redacted words and use that information to simplify their frequency analysis problem using such ground-truth.
As demonstrated by Figure 5.3, Gabor-based text redaction leaves much less whitespace between words and appears to target text well. Additionally, supervised Gabor guarantees a pixel label of “text” or “not-text” independent of image content and end-user involvement at a one-time training cost. As described in Subsection 5.3.3, unsupervised Gabor requires an end-user to choose a suitable “$k$” and “$i$” for each classified image.

In combination with the supervised and unsupervised latencies outlined in Section 7.2, supervised Gabor appears to be the suitable technique for performing text redaction. However, both under-perform Canny latency by $20 \times$, without including the costs to generate or load Gabor features depicted by Tables 7.3 and 7.4. After perfecting its accuracy, Canny may be the technique we employ by default.

Additionally, both supervised and unsupervised Gabor require the system to generate feature pixels before classifying. Compare Tables 7.3 and 7.4 to see that recomputing features is more expensive than storing and reloading from disk, even when feature files and labels are stored in a software-encrypted disk image on a laptop disk. Generating Gabor features requires multiple, computational-intensive steps for each Gabor filter in the employed filterbank and therefore requires the most time of any Gabor-precessing step.

### 8.1.3 Redaction Layout

Visual objects such as checkboxes, column and row structure, and varying field widths embody screenshot content that may be invariant to the effects of text redaction. See Figures 7.9 and 7.10 for examples. Figures 7.11 and 7.12 demonstrate trends the system could exploit to “normalize” such invariant content by making row and column entries even in count and width. By examining pixel and rectangle layouts, the system can determine where rows and columns exist and their visual characteristics, such as length and height.

---

2 Mean latencies of supervised versus unsupervised Gabor are approximately equivalent but the standard deviation of unsupervised Gabor is much higher.
Together, this visual metadata could underlie a process that visually \( k \)-anonymizes screenshots.

### 8.2 Templates

Template-matching allows the system to unredact user-defined regions of the screen (Figures 6.3, 6.8, and 6.9); count alerts (Figure 7.13); and perform other predicate-based operations automatically (i.e., if a screen matches a template, perform action \( x \)). OpenCV implements the matching method as described in Subsection A.2.2.

Figure 7.18 depicts performance measurements of the system’s ability to match templates. The two-step matching operation can be expensive when tens or hundreds of templates per screenshot must be matched. To perform a match, the system compares a small patch of template pixels against a target image and fills a matrix with comparison results. Then, the system searches the result matrix for matches. Searching can be two orders of magnitude less expensive than matching. Overall, the performance of template matching is proportional to image size, depth, and target size, and the system could downsample both template and target to improve the matching rate.

#### 8.2.1 Matching Alerts

One important aspect of template matching is counting matches or taking action when the system finds one or more matches. To avoid costly template matches and empirically corroborate a notion that alerts contain red pixels, we measured red image pixels. Figure 7.14 demonstrates our findings on four images that contain alerts and no immediate trends were prevalent. Red pixels of varying shades and volume exist in screenshot regions that do not contain alert images. Further experimentation may lead to useful results. For example, the system could search for red-valued pixels only within Canny rectangles and cross-correlate findings to discover useful patterns.
8.3 Overall Computational Latency

Redacting screenshots at the rate of full-motion video, 30 frames-per-second or 0.03 seconds-per-frame, would ensure smooth operation of scrubs-based deployments or server-based configurations where the system bulk-processes a collection of screenshots to redact text or match templates. However, none of the described processing techniques operate at this rate yet. Despite this fact, we have discussed how the system can provide text-redacted captures at human interaction rates (scrubs) and in static form (Gabor-based redaction and five_in_one).
Chapter 9

Future Work

In past chapters, we described many technical aspects of our proof-of-concept system (Chapters 3-6), empirical measurements of its performance characteristics (Chapter 7), and how one might exploit findings to improve performance (Chapter 8). In this chapter, we consider future work. We begin by reviewing the “big-picture” of our system in Section 9.1 and then share directions for “big-picture” work in Section 9.2. Finally, we close the chapter with directions for smaller technical details in Section 9.3.

9.1 Big-Picture Review

Recall Section 1.3, where we described how screen capture for sensitive systems could help improve overall system security. In the example scenario, hospital EHR software impeded clinician workflow and thus may have negatively affected the security of patient data records and physical well-being of patients. A security researcher working together with a hospital administrator wished to study the impact of the technology on clinician workflow in the hospital setting.

Our work provided a technical mechanism that, together with components described in Figure 3.1, systematized data collection and protection, simplified the process of understanding system problems empirically, and facilitated rich communications with develop-
ers. With a simplified and systematized process, many other studies could be carried out quickly and efficiently to tune EHR systems and improve their overall usability and security.

### 9.2 Big-Picture Directions

**Implement the Vision** To fulfill the vision described in Figure 3.1 we can implement the sharing components. The system currently implements screen capture and text redaction but does not include system components to share data.

**Command-line Interpreter and Rich User Interface** The system currently relies on keystrokes and a limited user interface (UI) to interact with and manipulate screenshot imagery and video. We would like to improve the UI by incorporating menus and OS-native versions of edit systems. Additionally, the edit and analysis applications could be combined with a command-line front-end to expedite our development and analysis of the system.

**Complete Web Front-End** In an alternative approach for end-users, the sharing system, screenshot viewer, and screenshot editor could all be web-based, platform independent, and centrally managed. A web-enabled system could enable real-time visualization, sharing of links among end-users, and downloading material to local formats, etc.

One obvious downside to this arrangement may be that unfettered access to a central redaction system could provide access to screenshots before final edits have been implemented. Another issue may be managing access control to web-accessible material.\(^1\)

**Remote Management** In some cases, it may be useful to allow authenticated capture controls through a smartphone or web browser. Actions such as “start” or “stop capture”

\(^1\)“Web-accessible” may not equate to “Internet-accessible.” Data may be accessible via an organization-internal network or any other number of arrangements.
and “publish redacted screenshots” could exist as simple commands accessible via those means.

**Camera-based Image Upload** Supporting input from ad hoc sources may further improve an end-users ability to capture important information in a timely fashion. End-users could collect screenshots with a camera mounted on a mobile device and upload data securely to a network-accessible system interface.

**User studies** Ultimately, we hope our system improves data-driven communications among application stakeholders such as end-users, developers, administrators, and policy-makers. To test how well the system achieves this goal, we should run user studies. We would like to understand how end-users use the capture system, how the system performs in their environment, which application features are useful and which are not, how end-users include the instrumentation in a concept of operations, how much unredacted context provides value, and many other issues. Healthcare is one domain that could benefit largely from this work and can serve as a proving ground to tune the existing system and guide evolution of its feature set.

**Predicate Matching** Predicate matching is an important building block of a programmable system that responds differently to varied inputs. In future work, we plan to implement support for predicate matching to allow programs to be built on top of the basic image processing capabilities provided by our system. For example, a predicate might say “if rectangle x matches in the upper left corner of the screen, then perform action y.” Predicate matching is closely related to OCR and language-driven operations described next.

**Include Optical Character Recognition (OCR)** To support higher-level semantic operations such as predicate matching, we should include OCR in a future version of the
system. Such processing would also enable the use of natural language processing (NLP) to selectively remove sensitive text from screenshots (Chapter 2).

**Policy-based Operations** Currently, the capture system redacts text blindly using a default-deny policy (Section 5.1). In future work, we could explore the combination of predicate matching, OCR, and a processing language to intelligently redact regions of the screen or perform other operations. The system could search for and count alerts, cross-correlate accuracy of screenshot content without access to the underlying application programming interface.

**Screen rewriting** With the ability to recognize regions of the screen and perform predicate-driven actions based on content, our system could be used to rewrite the screen in real-time to fine-tune the end-user experience. This work could build on existing research in universal access for disadvantaged users (e.g., [31]).

**Motion Analytics** We would like to apply analytics to understand mouse and navigation motions in a capture video. Such quantitative findings could inform application stakeholders how an application is used in practice and influence future improvements.

**User Notes** As users customize redacted screenshots, they may wish to include textual notations, circles, arrows, and other basic markups to highlight points. The system currently does not support these but can in a future version of the system.

**Secured Captures** Because many existing image viewers do not support encrypted image data, the current system does implement cryptographic protections on images. However, some users may wish to secure images cryptographically with integrity, confidentiality, non-repudiation, or some other properties. Therefore, a future version of the system can include cryptographic protections, perhaps as “metadata” carried in common image file formats.
In theory, protections such as valid image signatures can represent authentic provenance, image content, and “trustworthy” redaction operations, but many subtle security details would remain to be considered. For example, many security problems can arise when signed and viewed data differ or images contain active content [35].

**Metadata Watermarks**  Certain metadata can be watermarked directly into the image. Coupled with cryptographic integrity protections, such information can help form an audit trail related to a capture.

**Updating a Redacted Image**  The system could include secured metadata with each image to enable unredact operations of existing redacted text. Password protection or another form of encrypted protection could be used to manage metadata access control.

**Anonymized Identifiers**  Instead of fully redacting text, the system could match and replace it with anonymous identifiers in a way that preserves correlation but deidentifies individuals.

As an example, suppose a system captures login screens and a researcher wishes to study login access patterns without associating a particular pattern with a specific person. Each username could be remapped to an identifier that replaces the actual username in captured login screen with a consistent but anonymous identifier. The researcher could then correlate login screens without gaining user identity.

### 9.3 Lower-level Issues

**Redaction**  Canny-based redaction clearly outperforms Gabor in computational performance (Table 7.2), but it requires tuning before the technique can be practically useful. We presented a number of promising avenues for future work. The processing steps in Figure 9.1 collect a number of them into a coherent processing chain and include a new, Gabor-
Figure 9.1: Future information flow using Canny-based redaction. Revisiting Figure 3.3, we expand the redaction step to incorporate material from Chapters 7 and 8. In that step, the system would first filter out large rectangles to avoid fully-redacted screens (Subsection 8.1.1). Then, it would reduce significantly the number of rectangles managed during the redaction processing by merging redacted rectangles (Subsection 8.1.1). Next, the system would apply supervised Gabor redaction over a random sampling of pixels from each redaction rectangle to eliminate false-positives—a lack of Gabor-detected text would induce the system to discard the redaction rectangle. After this, it would merge whitespace (Section 8.1), and finally, the system could visually k-anonymize rows, columns, and objects such as checkboxes (Subsection 8.1.3).

Based step worth further investigation. We can apply supervised Gabor-based text redaction after thinning superfluous rectangles to eliminate Canny-based false-positives such as the one demonstrated in Figure 5.1. As a result, we may be able to increase Canny’s sensitivity threshold to redact lower-contrast text while maintaining a low false positive rate.

Remote Monitoring In this work the VNC protocol established a remote monitoring connection to a target workstation (Section 4.2). In future work, the system could secure the connection using encryption and authentication via the TLS [26] or SSH [69] protocols, support compression of transported screenshots, support hardware-based KVM systems, and implement a Citrix-based [19] connection protocol to monitor remote systems.

Metadata and Revision Control Aside from Gabor features and labels, the current system does not store a comprehensive set of metadata such as Canny-based redaction rectangles with each image. In a future area of work, the system could maintain such metadata along with revision control to support undo and redo operations. Additionally, the system could manage file saving intelligently. For example, certain existing applications write files to default file names and leave renaming to the user. Altogether, these features would have been useful during our development process.
Training the Supervised Classifier  In this work, we trained on dataset 1 and tested on dataset 2. We can explore the classification performance of a different order or training and testing mixtures that includes some features and labels from each dataset. Additionally, we can establish a process to retrain and incorporate new datasets as they become available.

Building a Ground-truth Dataset  The current technique of visually counting characters to produce a basic level of ground-truth is manually intensive and counts characters instead of pixels. We could use five_in_one and libraries of templates to help automate the process of labeling ground-truth. Canny and template matching can label large fractions of each image automatically and leave a small number of remaining elements to label by hand. Marked pixels can be saved persistently as ground-truth text-labels and incorporated into other processing steps. Additionally, unsupervised Gabor-based redaction can be used to “accept” true-positives suggested by Canny.

Document Analysis and k-anonymity  We introduced a foray into visual k-anonymity founded on document analysis concepts (Subsection 8.1.3). Building a screenshot model, such as the object model a web browser does for web documents, and using it to recognize and k-anonymize visual structure may further reduce the workload required by an end-user to share screenshots with application stakeholders.

Tune Gabor  Currently, Gabor parameters are tuned heuristically by starting with a parameter set that worked for others and hand-tuning variables until the system provides “useful” results. To improve rigor, we can explore works that describe design of Gabor filter banks [27,65].

Replace Gabor  The concept of steerable filters developed by Freeman and Adelson [29] could reduce the computational load associated with Gabor filtering by reducing the filter-bank size. Two steerable filters can replace four Gabor orientations and therefore reduce
the overall filterbank size by one half.

**Replace Canny**  Jensen-Shannon divergence may be a simple, high speed alternative to our use of Canny [14][15]. The technique finds image contours.

**Template Matching**  Currently, the system matches a template against a full resolution screenshot. Instead we could down-sample the template and screenshot multiple times, match at low resolutions, and rematch at higher resolutions only where low resolution matches exist. This may reduce matching time, albeit down-sampling includes a filter application which could reduce its computational benefit.

**Color Analysis**  We can study the effect of screenshot color shades on redaction effectiveness. End-users of applications such as EHR systems have the ability to change color schemes, and color schemes of such systems can vary widely among different vendors. Both of these factors can affect redaction algorithms. For example, our Canny technique relies on an image intensity gradient to detect edges within an image. The Canny algorithm will not detect text with low tonal variation compared to its background.

A parallel output of this work may be color-palette guidance for developers to maximize the effectiveness of text redaction.

**Finding Alerts**  We used template matching to find alerts within images and explored red-pixel histograms to search for screenshot alerts without relying on expensive template matching (Subsection [8.2.1]). We found no immediate trends. In a next step, we could examine red histograms in redacted text only and eliminate less “interesting” screen regions that may contain red pixels.

In another approach, the system could find redaction rectangles, paint the entire screen white except for the red pixels found within redaction rectangles, and analyze the remaining spatial structure.
“Live” Analytics  Analysis tools could include plots and statistics that update in real-time based on editor state and user actions. Chapter 7 includes analytics that may be relevant. A “real-time” display of information such as layout or color histograms would expedite our task in developing the system and assist end-users as they fine-tune redacted screenshots.
Chapter 10

Conclusions

We have designed, built, described, and empirically analyzed a system that allows end-users to take screen captures on sensitive systems. The system automatically redacts screenshot text and allows end-users to fine-tune redacted results for their needs. The automated redaction process requires no end-user intervention.

To redact, our system implements three different techniques. The first is based on the Canny edge detection algorithm that finds and marks changes in the screenshot intensity gradients as edges. Because screenshot text exists with an intensity difference compared to its background, Canny detects text. Before this technique can be used by our system, it requires perfecting to reduce false-positive rectangles that include the entire screen. In our experiments, Canny performs $20\times$ faster than the following two redaction approaches without counting setup time of the other techniques.

The second technique is based an unsupervised Gabor-filter technique. In this technique, we treat text as a texture and apply a Gabor filter bank of 28 filters to generate a feature vector for each pixel. A kmeans classifier then segments the feature list into $k$ classes, one of which may be text. This technique requires human intervention to choose $k$ for each image and then the $i$th of $k$ classes that correspond to redacted text.

In the third redaction technique, the system uses a supervised Gabor-filter technique. We
use unsupervised Gabor to generate hand-chosen ground-truth and rely on ground-truth to train a supervised, SVM classifier. The supervised classifier always labels pixels as text or not.

We performed the majority of our redaction experiments using 80 screenshots from electronic health record systems. Two large medical facilities provided the data.

We analyzed numerous aspects of the system to understand its performance envelope and strengths and weaknesses. Supervised Gabor performed with greater than 90% accuracy and preserved 76% of image pixels on average. Thus, our text redaction scheme preserves screenshot context.

We highlighted trends that the system could exploit to reduce or eliminate problems experienced with Canny-based text redaction among other issues. Some trends included rectangle area ratios, the fraction of total screenshot rectangles that a particular rectangle contains, rectangle layout, and inter-word whitespace.

We presented a number of avenues of future work that can build on this research. We hope to improve Canny for general-purpose use, implement predicate matching to process screenshots according to logical conditions, build a larger ground-truth data corpus, build system components for sharing redacted screenshots, deploy the system in a real user environment, and study its effectiveness in improving end-user interactions with technology.

Ultimately, our redaction system can facilitate data-driven communications among application stakeholders and guide system evolution to address stakeholder needs. With accurate and timely tuning enabled by our work, stakeholders can achieve and maintain usable and secure systems in practice.
Appendix A

Supporting Libraries

This work relies on a number of external libraries developed and maintained by various open source communities. Here we provide background for some of the more important ones so the reader who may be unfamiliar with them can better understand discussion throughout this dissertation.

We rely on the C++ template libraries boost [54], which is community developed and maintained, and standard template library (STL) defined in the C++ specification [2]; OpenCV, a feature-rich open-source computer vision library originally developed by Intel and now community developed and maintained [16]; liblinear, a machine learning library for support vector machines (SVM) developed and maintained by National Taiwan University [28]; and CGAL, a computational geometry library used to efficiently find intersections among rectangles [1, 37, 70]. CGAL is a compilation of contributions from numerous researchers across the field of computational geometry study and community maintained.

A.1 boost and STL

Our work relies heavily on C++ for its implementation. It uses STL containers such as “vector”s for storing instantiated class objects and the boost “shared_ptr” to manage heap-allocated objects. The “shared_ptr” also reduces overhead associated with storing C++
objects within STL containers. Figure A.1 demonstrates the common idiom in use.

```cpp
1 class Foo {
2    ...
3    int some_foo_method (void);
4 }
5
typedef shared_ptr <Foo> foo_ptr;
6 ...
7 vector<foo_ptr> v;
8 v.push_back (foo_ptr (new Foo ()));
9 int x = v.back ()->some_foo_method ();
10 ...
```

Figure A.1: Using a shared_ptr with an STL container. This code snippet demonstrates how shared_ptr <Foo> objects are stored in an STL vector and accessed as if they were pointers to class Foo. Note that shared_ptr implements an “T* operator->()” as demonstrated in line 10, so that accessing the object held by a shared_ptr is analogous to accessing the underlying object directly.

The shared_ptr class maintains a reference count to its underlying heap-allocated memory object and when the reference count drops to 0, it calls the “delete” operator on its managed object. This approach simplifies greatly the need to manually manage heap-allocated memory.

Additionally, STL containers that store “shared_ptr”s can function with greater efficiency that those that store objects directly. Data structure operations such as insert invoke a simple “shared_ptr” copy constructor versus a potentially expensive copy constructor for a complex C++ class.

A.2 OpenCV

The OpenCV library is a mature, popular, system-independent, open-source computer vision library implemented in a mix of C and C++. It exposes bindings for both languages and the python [23] scripting language; we relied principally on the C++ bindings. The library contains a rich set of documentation and examples.
OpenCV can rely on system-native, low-level graphics routines for rendering and graphics operations. For example, on our OS X development platform, OpenCV relies on Quicktime [12] and core animation components [10] to perform low-level graphics operations.

Our system relies on OpenCV’s core, image processing, user-interface, and machine learning components throughout. They components provide functionality that underlie redaction and the machinery for defining redaction templates as described in Chapter 5. They enable our system to load, store, filter, and otherwise process images; manipulate pixels individually; draw into images; display them on a screen and handle mouse and keyboard input events; and execute the kmeans machine-learning algorithm also described in Chapter 5.

A.2.1 Matrices

OpenCV relies on a structure called “Mat” to represent images in memory. Conceptually, “Mat” is a matrix and OpenCV processing algorithms operate on it to manipulate images. At various places throughout our system we store “Mat” structures in files for later processing. Figure A.2 defines the file layout that we defined for the objects.

| rows | cols | step | type | (rows*step) bytes |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| <-------- header --------> | <----- data ----> | <-------- header --------> | <----- data ----> |

Figure A.2: OpenCV-matrix file layout. Each matrix of stored data, whether features, feature coordinates, or feature labels, is stored as depicted in this figure. Each header field is an unsigned 4-byte integer. “rows” is the number of rows in the matrix, “cols” is the columns, step is the number of bytes stored per row and may be larger than cols to improve data alignment, and type is an OpenCV matrix type $\in \{CV\_U8C1, CV\_F32C1\}$ OpenCV uses matrices to store images and in OpenCV parlance, each pixel is represented by one matrix element and the matrix “type” governs how OpenCV interprets each matrix element or pixel. CV\_8C1 corresponds to one unsigned 8-bit value per element and CV\_F32C1 corresponds to one signed 32-bit floating value per element.
A.2.2 Template Matching

Our system relies on OpenCV’s ability to find regions in an image that match a given template. Templates are small images such as icons that may exist within a larger image. The OpenCV function “matchTemplate” implements this feature with a variety of options that define “match.” Our system relies on the following definition, which for each pixel computes the sum of the square of the difference between the template and target image aligned at a point.

\[ R(x, y) = \sum_{x',y'} (T(x', y') - I(x + x', y + y'))^2 \]

\(R\) is the resultant calculation at each pixel \(x\) and \(y\), \(T\) is the template, \(I\) is the image, and \(x'\) and \(y'\) are valid pixel positions within the template. See the OpenCV documentation for other “match” techniques \[22]\.

The result of all calculations are written into a result matrix. The system normalizes the matrix and then searches for all values below a user-defined threshold. We use the value 0.00006 in the five_in_one tool (Section 6.2) and a tool for counting matching templates (Figure C.5). We use 0.00005 within the scrubs tool (Section 6.1). We determined these values through qualitative analysis.

A.3 liblinear

liblinear implements a set of high-speed SVM classifiers. It provides commandline applications for training and classifying data, good documentation and guidance for using the library to achieve reasonable classification performance, and a documented data format for training and testing data sets. Figure A.3 provides a sample file and a description of the file format, which includes feature vectors and their labels. Figure A.4 provides a sample SVM parameter set generated by training a model using labeled data. liblinear uses the
parameter set to make predictions on related, unlabeled data points.

-1 1:-0.00135181 2:-0.00134854 3:0.000421816 4:0.000444805
1 1:-0.00135181 2:-0.00132873 3:0.000289231 5:0.00034581
-1 3:0.000290677 4:0.000338868 5:0.000347394 7:-0.000188436
-1 4:0.000338977 5:0.000347591 6:0.000339016 7:-0.000189546

Figure A.3: liblinear data file. Data files consist of feature vectors and their labels in a text-based, sparse-data format. Each row begins with a feature label chosen from \{-1, 1\}, and a list of 1-indexed, index:feature pairs. Whitespace separates each line item and lines end with a newline. Feature values of 0 are not written and therefore, the sparse data format more efficiently represents sparse data. Our system realizes no gain using this approach because our features are dense with 30 features per label.

```plaintext
1 solver_type L1R_LR
2 nr_class 2
3 label -1 1
4 nr_feature 30
5 bias -1
6 w
7 -21.52096798121508
8 -30.24049436071538
9 .
10 . [values elided]
11 .
12 -558.8403947460139
13 -914.9739787865601
14 0
15 -164.7544799904056
16 .
17 . [values elided]
18 .
```

Figure A.4: SVM parameter set. liblinear creates a classifier of this nature after training on a labeled data set. This particular one is an L1-regularized logistic regression classifier. Thirty features are associated with this SVM and two classes of labels exist, drawn from values \{-1, 1\}. liblinear’s commandline tools or API can be used to generate and use this file. Note that any parameters values of “0” (line 14) represent input features that are not used by the model and can be removed from the input data set.

We used their commandline applications as API guides to build a trainer and classifier that interfaces the binary data formats of our system. While using the API, we found two interesting points to remember while programmatically populating a liblinear feature vector. First, feature vectors are 1-indexed, so any 0-valued indices are incorrect. Second,
for (u32 i = 0; i < n; i++) {
    ...  
    for (u32 j = 0; j < features->cols; j++) {
        prob.x[i][j].index = j + 1;  // 1-indexed
        prob.x[i][j].value = features->at <float> (row, j);
    }

    prob.x[i][features->cols].index = -1;  // end marker
    prob.x[i][features->cols].value = 0;
}

Figure A.5: Programatically populating a liblinear feature vector. This code snippet demonstrates the feature indices must be 1-indexed and the last feature index must be −1. Without these, liblinear will behave erratically. In more detail, “prob” on lines 4-5 and 7-8 is a liblinear “struct problem” data structure, x is an array of feature vectors represented by a two-dimensional array of liblinear “feature_node” structures. x[i] is a single feature vector, and x[i][j] is a single feature. Each feature has an index and corresponding value.

each feature vector represented by a liblinear “feature_node” must end with an index value of −1. liblinear uses a sparse data representation and represents the end of a feature vector internally with a −1 marker. Listing A.5 demonstrates these points with a code snippet from our implementation.

A.4 CGAL

The Computations Geometry Algorithms Library (CGAL) implements an fast box intersection algorithm [70] that we rely on in various tools and analysis applications. The library contains extensive documentation, samples, and references to academic works that it implements. CGAL relies heavily on C++ templates and thus debugging can be difficult when the compiler displays a page of template arguments associated with one line of code.

We found that working from the manual’s sample code [37] provided a workable solution for our box intersection problem. CGAL routines interact naturally with STL containers, which simplifies greatly their integration into our code.

We discovered two points worth sharing about CGAL and the box-intersection routine. First, the routine does not support “shared_ptr.” The manual describes storing their “Box”
objects or “Box *” pointers directly within a container. Although we worked around this issue, it would be natural to include support for C++ draft standard “shared_ptr” [3]. Second, upon call of “CGAL::box_intersection_d,” the copy constructor of the callback argument is invoked. Thus, if the callback is used to track data while the intersection algorithm executes, the callback object whose copy constructor was invoke will not experience changes to internal data structures unless those structures are pointers or references to data from a different scope. Figure A.6 demonstrates this issue with an example.
class Box_Intersection_Handler {
public:
    Box_Intersection_Handler (int &intersection_counter) :
        intersection_counter_ (intersection_counter) {}
    Box_Intersection_Handler (const Box_Intersection_Handler &h) :
        intersection_counter_ (h.intersection_counter_) {}
    // invoked on intersection by CGAL::box_intersection_d
    void operator () (const Box *a, const Box *query) {
        intersection_counter_++;
    }
    int count (void) const { return intersection_counter_; }
private:
    int &intersection_counter_;
};

int intersection_counter = 0;
Box query[] = {bbox};
Box *qp[] = {query};
Box_Intersection_Handler callback (intersection_counter_);

// 'callback' copy ctr invoked, but object
// maintains reference to intersection_counter
CGAL::box_intersection_d (boxes_ptrs_.begin (),
    boxes_ptrs_.end (),
    qp,  
    qp+1,  
    callback);

// Because Box_Intersection_Handler maintains a reference
// to intersection_counter, this will print a correct value.
cout << 'Intersection count: ' << callback.count () << endl;

Figure A.6: Tracking state during execution of “CGAL::box_intersection_d.”
“Box_Intersection_Handler” maintains a reference “intersection_counter_” on line 14
to “intersection_counter” on line 19. Therefore, when the “CGAL::box_intersection_d”
is invoked on line 26 and “callback”’s copy constructor (line 5) is invoked within
“CGAL::box_intersection_d,” updates to “intersection_counter_” on line 9 in the new
copy will be visible in the original “callback” object. Consequently, the last print statement
on line 34 will function as expected.
Appendix B

Datasets

Our dataset is comprised of three screenshot collections and two derived files for each screenshot. Because healthcare was an important motivating application domain for this work, we obtained two datasets of electronic health record (EHR) screenshots, each from a different hospital. The third collection is a set of random desktop screenshots and screenshot snippets from a MacBook Pro laptop running OS X 10.6. Finally, for each EHR screenshot we compute files used to aid redaction processing and analysis.

The EHR screenshots contain fake patient data but are still considered sensitive. Consequently, we store them in a 256-bit AES encrypted disk volume. Additionally, we do not display their unredacted form, limit their redacted display throughout this dissertation, and reveal only aggregate analysis and comments. Finally, because we perform system analysis on these sets, and the size of each dataset is tens of files, we use a file naming convention conducive to scripting. Names exist in the form “n.png,” where $n \in [1, |\text{dataset}|]$.

Some of the random desk screenshots also contain sensitive information. Therefore, we do not display unredacted information where relevant.
B.1 Processed Data

Along with each EHR dataset image, we generate and store Gabor features, feature metadata, and labels used for unsupervised and supervised pixel classification as described in Chapter 5. Features and metadata reside within one binary file and labels within another. Features, metadata, and labels are each represented by a OpenCV matrix data structure, and matrices are stored on disk as described in Appendix A, Figure A.2.

Features:

| # pixels | 30 | 4*30 | CV_F32C1 | (# pixels*4*30) bytes |

ith row of features:

| feat_0 | feat_1 | ... | feat_28 | row # | col # |

Feature coordinates (begins immediately following Features):

| # pixels | 2 | 4*2 | CV_u8C1 | (# pixels*4*2) bytes |

ith row of coordinates:

| row # | col # |

Figure B.1: File layout of Gabor-features. A “features” file contains a matrix of Gabor features (as described in Chapter 5) and matrix of feature coordinates. Each contains one row for each image pixel. In the feature matrix, each row contains 30×4-byte floating-point elements for each pixel, and rows do not exist in image-pixel order. Twenty-eight elements of each feature are normalized, floating-point Gabor-feature values and two are normalized floating-point pixel-coordinate values used during Gabor analysis. The feature-coordinates matrix exists after the features matrix in the features file, and each row in the coordinates matrix consists of 2×4-byte unsigned integer fields. The ith row of the coordinates matrix consists of the row and column pixel value for the ith feature vector stored much earlier in the same file.
Feature labels:
---------------------------------------------------------------
| k | i | # pixels | 1 | 4*1 | CV_u8C1 | (# pixels*4*1) bytes |
|--------------------------------|
| label meta data |
-------- |
| matrix of label data |
-------------------------------------------------------

ith row of labels:
----------------- 
| label \in [0, k-1] |
-------------------

Figure B.2: File layout of feature-labels. The “labels” file contains a kmeans label (as described in Chapter 5) for each pixel. The \( j \)th row corresponds to the \( j \)th image pixel and contains a 1-byte integer pixel label drawn from the range \([0, k - 1]\). The \( k \) value used to generate the label set is listed first in the file as an unsigned 4-byte integer, followed by a separate unsigned 4-byte integer \( i \) that represents which label class corresponds to text.

B.2 Summary of Datasets

B.2.1 EHR set 1

This dataset contains 51 screenshots from an EHR system. Screenshots vary significantly in visual content. Image attributes include the following: PNG format, RGB color, 1502 \( \times \) 1901 pixels in size, and 150 pixels/inch. A few images have a slightly larger size of 1502 \( \times \) 1093 pixels. Image files range in size from 1-1.5 MB, feature files each contain approximately 201 MB, and label files each contain approximately 6.3 MB.

B.2.2 EHR set 2

This dataset contains 29 screenshots from a separate EHR system. Screenshots vary in visual content but not as significantly as the first set. Additionally, screenshots contain a small border artifact beyond the size of the captured application window. Image attributes include the following: PNG format, RGB color, 1680 \( \times \) 1050 pixels in size, and 72 pixels/inch. Image files range in size from approximately 230-390 KB, feature files each contain
B.2.3 Random laptop screenshots

This dataset contains approximately 17 screenshots from a separate EHR system. Screen-
shots vary widely in visual content and size. Some screenshots are application windows
and others are snippets of larger screenshots. Image attributes include the following: PNG
and JPG formats, RGB color, $397 \times 101$ through $1440 \times 900$ pixels in size, and a mix of
72 and 150 pixels/inch. Image files range in size from approximately 58-595 KB, feature
files range in size from 54-112 MB, and corresponding label files range from 1.7-3.5 MB.
We did not generate a feature file for each non-EHR screenshot.
Appendix C

Analysis Tools

In this appendix, we describe a number of small tools developed to support development and analysis of this system. They are built with functionality developed over Chapters 3–6 and support discussion in Chapter 7. Next we describe the functionality and commandline usage of each tool.
Usage:
count_rects <img>

<img> The image file to redact and count rectangles.

Figure C.1: Count the number of rectangles in the image. Canny-redact the image, count, and return the number of subsequent contours present in the image.

Usage:
measure_redaction_effect <img dir> <low img id>  
   <high img id (inclusive)> <low frac>  
   <high frac> <step>

<img dir> The director holding the images to compare.  
<low img id> The lowest numbered image filename.  
   (Integer file names are assumed.)  
<high img id> The highest numbered image filename, inclusive.  
<low frac> The starting fraction of image pixels to redact uniform-at-random (UAR).  
<high frac> The ending fraction of image pixels to redact UAR.  
<step> The step for each loop iteration to get get form <low frac> to <high frac>

Figure C.2: Measure redaction effect over directory of images. For each image pair within a directory, compute the fraction of pairwise, text-redaction pixels that overlap. A pixel pair consists of one pixel from the same location in each image of a pair. Pairs that include two black pixels before redaction and due to image size mismatches, pixels without a pair never change value after redaction. When both pixels in a pair correspond to text, text redaction converts non-black pixel values to black (the redaction color).
Usage:
compare_similarity_of_all <img dir> <low img id> 
<high img id (inclusive)> <low frac> 
<high frac> <step>

<img dir>  The director holding the images to compare. 
<low img id> The lowest numbered image filename. 
(Integer file names are assumed.) 
<high img id> The highest numbered image filename, inclusive. 
<low frac> The starting fraction of image pixels to redact uniform-at-random (UAR). 
<high frac> The ending fraction of image pixels to redact UAR. 
<step> The step for each loop iteration to get get form <low frac> to <high frac>

Figure C.3: Pixel-by-pixel comparison between redacted images in a directory. Iterating over all image pairs in a directory (except those of identical images), the application first supervised-Gabor redacts images, chooses <low frac> pixels uniform-at-random (UAR) to color black, colors those pixels in each image, and counts how many pixel values at a identical image positions match one another. The Gabor-feature files and a liblinear-defined SVM models are used to redact images, and the <low frac>, <high frac> <step> values define the loop used to incrementally increase the number of pixels redacted UAR. The fraction does not account for already redacted pixels and therefore may overlap with them. We only analyzed output associated with supervised-Gabor redaction and left unused the computations associated with additional UAR redaction.

Usage:
compare_prediction_performance <features> <labels> <model>

<feature> Gabor image features. 
<labels> Ground-truth labels that define which Gabor-image features are redacted pixels and which are not. 
<model> A liblinear trained classifier to use to redact images before comparing.

Figure C.4: Compute redaction performance of a trained SVM classifier. Using the trained SVM <model>, classify each feature as text (1) or not text (−1). Compare against the given feature labels to compute the true positive, true negative, false positive, and false negative rates of the classifier.
Usage:
\texttt{count\_matching\_templates <img> <templates> [thresh]}

\begin{itemize}
\item <\texttt{img}> The image to search for templates.
\item <\texttt{templates}> I file of templates to search for and count within <\texttt{img}>
\item [\texttt{thresh}] The threshold below which a positive match is identified.
\end{itemize}

\textbf{Figure C.5:} Count the number of template matches within a given image. Templates can be defined and stored using the five\_in\_one tool described in Section \ref{sec:five_in_one}. Subsection \ref{sec:matching} describes the matching function. The default matching threshold is 0.00006, which was qualitatively derived to provide a visual balance between false positive and false negative matches within our datasets.

Usage:
\texttt{compute\_matching\_perf <img>}

\begin{itemize}
\item <\texttt{img}> The image to search for templates.
\end{itemize}

\textbf{Figure C.6:} Analyze computational performance of template matching. Generate templates of size $M \times M$ where $M = 2^i$, $i \in \left[1, 2^{\min(\log_2(w), \log_2(h))}\right]$, $w =$ image width, and $h =$ image height; match each against the image; and evaluate match results.

Usage:
\texttt{count\_subset\_frac <dir> <first> <last>}

\begin{itemize}
\item <\texttt{dir}> Directory of images to examine
\item <\texttt{first}> The name of the first file. File names are assumed to be "[integer].png", where [integer] is a number $\geq 0$.
\item <\texttt{last}> The name of the last file numerically. The value is inclusive.
\end{itemize}

\textbf{Figure C.7:} Compute a histogram of redaction rectangle size ratios of all images within a directory. For each image, apply Canny-based redaction to the image as described in Section \ref{sec:redaction}, iterate over all redaction rectangles, and for each, compute the fraction of total rectangle count that resides wholly within its bounds.

Usage:
\texttt{count\_subsets <img>}

\begin{itemize}
\item <\texttt{img}> The image to count redaction rectangles.
\end{itemize}

\textbf{Figure C.8:} Compute a histogram of redaction rectangle size ratios of a single image within a directory. Do the same operation as described in Figure \ref{fig:C.7}, but only with a single image.
Usage:
canny_redact <img>

<img> The image to redact using a Canny-based technique.

Figure C.9: Generate Canny-based redaction rectangles. Used to analyze computational performance of the Canny redaction technique.

Usage:
generate_gabor_features <img> <features>

<img> The image over which to generate gabor features.
<features> The feature file to store generated gabor features.

Figure C.10: Generate Gabor features for redaction. Features are written to the output file named <features> and Subsection 5.3.2 describes the method used to generate the features.

Usage:
generate_color_variety <img dir> <low img id> <high img id (inclusive)>

<img dir> The director holding the images to analyze.
<low img id> The lowest numbered image filename.
(Integer file names are assumed.)
<high img id> The highest numbered image filename, inclusive.

Figure C.11: Generate an RGB color histogram over all unredacted images in a directory.

Usage:
generate_color_histos <img>

<img> The image over which to compute RGB color histograms. Image text is first redacted.

Figure C.12: Generate RGB color histograms of each unredacted image.

Usage:
generate_layout_histos <img> <features> <labels>

<img> The image over which to compute layout histograms. Image text is first redacted.
<features> The feature file associated <img>.
<labels> The feature labels of <img>.

Figure C.13: Generate layout histograms of the text-redacted image. Text-redact the image and compute histograms of the x and y coordinates of the upper left corner of each redaction rectangle and each rectangle's width and height.
Usage:

```
ground_truth2liblinear <features> <labels> <seed>
  <train frac> <ofile (append)>
```

- `<features>`: An image feature file.
- `<labels>`: Feature labels for the same image.
- `<seed>`: The pseudo-random number generator (PRNG) seed.
- `<train frac>`: The fraction of features to output as training features.
  
  
  $(0 < \text{<train frac>} \leq 1.0)$
- `<ofile (append)>`: The file to write the liblinear-formatted features.

Figure C.14: Convert features from our binary format to liblinear format. The number of features sub-selected from the feature list are appended to the file `<ofile>`. The application selects features uniform-at-random using a PRNG with seed `<seed>`. Section B.1 describes our file formats for features and labels, which is binary. Section A.3 describes the liblinear file format, which is text-based.

Usage:

```
ground_truth2support_vectors <features> <labels>
  <train frac> <nfold> <C> <seed> <ofile>
```

- `<features>`: An image feature file.
- `<labels>`: Feature labels for the same image.
- `<train frac>`: The fraction of features to use while training the SVM.
  
  
  $(0 < \text{<train frac>} \leq 1.0)$
- `<nfold>`: The n-fold cross validation to use during training.
- `<C>`: The C constant to use for the SVM.
- `<seed>`: The pseudo-random number generator (PRNG) seed.
- `<ofile>`: The file to write the trained liblinear SVM.

Figure C.15: Train a liblinear L1-regularized L2-loss SVM. Use a fraction of the features and their corresponding labels, `<nfold>` cross-validation, and the SVM constant $C=\text{<C>}$ to generate a SVM. During training, choose the training features UAR, and seed the PRNG function using `<seed>`. Feature and label files should exist in the binary format described in Section B.1.
Usage:
label_ground_truth <img> <features> <labels>

<img> An image from which to label ground truth.
<features> Image features.
<labels> File to hold feature labels.

Figure C.16: Generate ground-truth text-redaction labels for an image. Using Gabor redaction and unsupervised classification of a single image, cycle through $i \in [0, 1]$ for $k = 2$ and $i \in [0, 2]$ for $k = 3$ and allow the user to save pixel labels corresponding to the set that provides the best text-redaction visually.

Usage:
load_features_and_segment <img> <features> [k]

<img> An image from which to label ground truth.
<features> Image features.
<k> The number of unsupervised classes.

Figure C.17: Display all text-redacted images using kmeans clustering. An image will be displayed with each class of $i \in [0, k - 1]$ chosen as the redacted pixels.

Usage:
load_features_and_labels <features> <labels>

<features> Image features.
<labels> Feature labels.

Figure C.18: Load features and labels for an image. Used to analyze computational performance of loading features from a file versus generating them.

Usage:
load_features_and_segment_labeled <img> <features> <labels>

<img> The image to redact.
<features> Image features.
<labels> Feature labels.

Figure C.19: Compute unsupervised Gabor redaction using features and labels. Used to analyze computational performance of unsupervised Gabor-redaction technique.

Usage:
redact_using_model <img> <features> <model>

<img> An image from which to label ground truth.
<features> Image features.
$model> A liblinear trained classifier used to text-redact the image.

Figure C.20: Text-redact an image using a SVM classifier. The redacted image is displayed on the screen.
Usage:
templates2img <templates> <oimg>

<templates> A file of template images.
{oimg} The image file to store the image collage of templates.

Figure C.21: Create, display, and store a collage of image templates. Convert a binary file of image templates to a PNG-formatted image with visually magnified template images and store them in an output file.

Usage:
visualize_ground_truth <img> <features> <labels>

<img> An image file.
<features> Gabor features corresponding to the image file.
<labels> Feature labels that describe which pixels to redact.

Figure C.22: Visualize a single, redacted image using a set of features and labels. Used to visualize ground truth produced using label_ground_truth.

Usage:
analyze_ws <dir> <first> <last>

<dir> Directory of images to examine whitespace
<first> The name of the first file. File names are assumed to be "[integer].png", where [integer] is a number >= 0.
<last> The name of the last file numerically. The value is inclusive.

Figure C.23: Compute a histogram of inter-rectangle whitespace. For each image, apply Canny-based redaction to the image as described in Section 5.2, iterate over all redaction rectangles, and for each, evaluate the x-pixel distance between it and all other rectangles. Limit the evaluation to pairs that satisfy the predicate defined in Figure 7.4.
Appendix D

Contact Information

Please use the following information to contact the author or advisor with questions or comments about the work and inquiries about source code.

<table>
<thead>
<tr>
<th>name</th>
<th>email</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe Cooley</td>
<td><a href="mailto:jcool@alum.dartmouth.org">jcool@alum.dartmouth.org</a></td>
<td>author</td>
</tr>
<tr>
<td>Dr. Sean Smith</td>
<td><a href="mailto:sws@cs.dartmouth.edu">sws@cs.dartmouth.edu</a></td>
<td>advisor</td>
</tr>
</tbody>
</table>
Bibliography


