Automatic Input Rectification

by

Fan Long

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2012

Certified by ....
Martin C. Rinard
Professor
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Automatic Input Rectification

by
Fan Long

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

We present a novel technique, *automatic input rectification*, and a prototype implementation, SOAP. SOAP learns a set of constraints characterizing *typical inputs* that an application is highly likely to process correctly. When given an *atypical input* that does not satisfy these constraints, SOAP automatically *rectifies* the input (i.e., changes the input so that it satisfies the learned constraints). The goal is to automatically convert potentially dangerous inputs into typical inputs that the program is highly likely to process correctly.

Our experimental results show that, for a set of benchmark applications (Google Picasa, ImageMagick, VLC, Swfdec, and Dillo), this approach effectively converts malicious inputs (which successfully exploit vulnerabilities in the application) into benign inputs that the application processes correctly. Moreover, a manual code analysis shows that, if an input does satisfy the learned constraints, it is incapable of exploiting these vulnerabilities.

We also present the results of a user study designed to evaluate the subjective perceptual quality of outputs from benign but atypical inputs that have been automatically rectified by SOAP to conform to the learned constraints. Specifically, we obtained benign inputs that violate learned constraints, used our input rectifier to obtain rectified inputs, then paid Amazon Mechanical Turk users to provide their subjective qualitative perception of the difference between the outputs from the original and rectified inputs. The results indicate that rectification can often preserve much, and in many cases all, of the desirable data in the original input.

Thesis Supervisor: Martin C. Rinard
Title: Professor

Acknowledgements

I am grateful to my advisor Martin Rinard. It is my great pleasure to work with him in the past two years. His enthusiasm and support keep inspiring me. When I was in doubt, he can always point out the right direction for me.

I am also grateful to Vijay Ganesh, Michael Carbin, and Stelios Sidiroglou with whom I worked on the input rectification-related projects. Without their contributions, this thesis would be impossible.

I would like to thank all the members of the PAC CSAIL group, especially Sasa Misailovic, Deokhwan Kim, Jeff Perkins, and Mary McDavitt for their help and support. I really enjoy my past two years in PAC group.

Last but not least, I would like to thank my parents and Xuhang for their continuous and unconditional love and understanding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Errors and security vulnerabilities in software often occur in infrequently executed program paths triggered by atypical inputs. A standard way to ameliorate this problem is to use an anomaly detector that filters out such atypical inputs. The goal is to ensure that the program is only presented with typical inputs that it is highly likely to process without errors. A drawback of this technique is that it can filter out desirable, benign, but atypical inputs along with the malicious atypical inputs, thereby denying the user access to desirable inputs.

1.1 Input Rectification

We propose a new technique, *automatic input rectification.* Instead of rejecting atypical inputs, the input rectifier modifies the input so that it is typical, then presents the input to the application, which then processes the input. We have three goals: a) present typical inputs (which the application is highly likely to process correctly) to the application unchanged, b) render any malicious inputs harmless by eliminating any atypical input features that may trigger errors or security vulnerabilities, while c) preserving most, if not all, of the desirable behaviors for atypical benign inputs. A key empirical observation that motivates our technique is the following:

Production software is usually tested on a large number of inputs. Standard testing processes ensure that the software performs acceptably on such inputs. We refer to such inputs as *typical inputs* and the space of such typical inputs as the *comfort zone* [37] of the application. On the other hand, inputs designed to exploit security vulnerabilities (i.e., *malicious inputs*) often lie outside the comfort zone. If the rectifier is able to automatically detect inputs that lie outside the comfort zone and map these inputs to corresponding *meaningfully close inputs* within the comfort zone, then it is possible to a) prevent attackers from exploiting the vulnerability in the software, while at the same time b) preserving desirable data in atypical inputs (either benign or malicious) for the user.

We present SOAP (Sanitization Of Anomalous inPuts), an automatic input rectification system designed to prevent *overflow vulnerabilities.* SOAP first learns a set of constraints over typical inputs that characterize a comfort zone for the application that processes those inputs. It then takes the constraints and automatically generates a rectifier that, when provided with an input, automatically produces another input that satisfies the constraints. Inputs that already satisfy the constraints are passed through unchanged; inputs that do not satisfy the constraints are modified so that they do.

Potential Advantages of Automatic Input Rectification

Input rectification has several potential advantages over simply rejecting malicious or atypical inputs that lie outside the comfort zone:

- **Desirable Data in Atypical Benign Inputs:** Anomaly detectors filter out atypical inputs even if they are benign. The result is that the user is completely denied access to data in atypical inputs. Rectification, on the other hand, passes the rectified input to the application for presentation to the user. Rectification may therefore deliver much or even all of the desirable data present in the original atypical input to the user.

- **Desirable Data in Malicious Inputs:** Even a malicious input may contain data that is desirable to the user. Common examples include web pages with embedded malicious content. Rectification may eliminate the exploits while preserving most desirable input from the original input. In this case the rectifier enables the user to safely access the desirable data in the malicious input.

- **Error Nullification:** Even if they are not malicious, atypical inputs may expose errors that prevent the application from processing them successfully. In this case rectification may nullify the errors so that the application can deliver most if not all of the desirable data in the atypical input to the user.

1.2 The Input Rectification Technique

SOAP operates on the parse tree of an input, which divides the input into a collection of potentially *nested fields.* The hierarchical structure of the parse tree reflects nesting relationships between input fields. Each field may contain an integer value, a string, or unparsed raw data bytes. SOAP infers and enforces 1) *upper bound constraints* on the values of integer fields, 2) *sign constraints* that capture whether or not an integer field must be non-negative, 3) *upper bound constraints* on the lengths of string or raw data byte fields, and 4) *correlated constraints* that capture relationships between the values of integer fields and the lengths of (potentially nested) string or raw data fields.

The two main challenges in designing an automatic input rectifier such as SOAP are how to infer and enforce constraints for a given application. The execution monitor of SOAP first uses taint analysis [13, 32] to identify input fields that are related to critical operations during the execution of the application (i.e., memory allocations and memory writes). The learning engine of SOAP then automatically infers constraints on these fields based on a set of training inputs. When presented with an atypical input that violates these constraints, the SOAP rectifier automatically modifies input fields iteratively until all constraints are satisfied.

Nested Fields in Input Files

One of the key challenges in input rectification is the need to deal with nested fields. In general, input formats may contain arbitrarily nested fields, which make inferring correlated constraints hard. Our algorithm must consider relationships between multiple fields at different levels in the tree.

Nested input fields also complicate the rectification. Changing one field may cause the file to violate constraints associated with enclosing fields. To produce a consistent rectified input, the rectifier must therefore apply a cascading sequence of modifications to correlated fields as its constraint enforcement actions propagate up or down the tree of nested fields.

1.3 Key Questions and Experiments

We identify several key questions that are critical to the success of the input rectification technique:

- **Learning:** Is it possible to automatically learn an effective set of constraints from a set of typical benign inputs?

- **Rectification Percentage:** Given a set of learned constraints, what percentage of previously unseen benign inputs fail to satisfy the constraints and will therefore be modified by the rectifier?

- **Rectification Quality:** What is the overall quality of the outputs that the application produces when given benign inputs that SOAP has modified to enforce the constraints?

- **Security:** Does SOAP effectively protect the application against inputs that exploit errors and vulnerabilities?

We investigate these questions by applying SOAP to rectify inputs for five large software applications. The input formats of these applications include three image types (PNG, TIFF, JPEG), wave sound (WAV) and Shockwave flash video (SWF). We evaluate the effectiveness of our rectifier by performing the following experiments:

- **Benign Input Acquisition:** For each application, we acquire a set of inputs from the Internet. We run each application on each input in its set and filter out any inputs that cause the application to crash. The resulting set of inputs is the *benign inputs*. Because all of our applications are able to process all of the inputs without errors, the set of benign inputs is the same as the original set.

- **Training and Test Inputs:** We next randomly divide the collected benign inputs into two sets: the *training set* and the *test set.*

- **Potentially Malicious Inputs:** We search the CVE security database [2] and previous security papers to obtain malicious inputs designed to trigger errors and/or exploit vulnerabilities in the applications.

- **Learning:** We use the training set to automatically learn the set of constraints that characterize the comfort zone.

- **Atypical Benign Inputs:** For each application, we next compute the percentage of the benign inputs that violate at least one of the learned constraints. We call such inputs *atypical benign inputs.* In our experiments, the percentage of atypical benign inputs is less than 1.57%.

- **Quality of Rectified Atypical Inputs:** We evaluate the quality of the rectified atypical inputs by paying people on Amazon Mechanical Turk [1] to evaluate their perception of the difference between 1) the output that the application produces when given the original input and 2) the output that the application produces when given the rectified version of the original input. Specifically, we paid people to rank the difference on a scale from 0 to 3, with 0 indicating completely different outputs and 3 indicating no perceived difference. The mean scores for over 75% of the atypical inputs are greater than 2.5, indicating that Mechanical Turk workers perceive the outputs for the original and rectified inputs to be very close.

- **Security Evaluation:** We verified that the rectified versions of malicious inputs for each of these applications were processed correctly by the application.

- **Manual Code Analysis:** For each of the malicious inputs, we identified the root cause of the vulnerability that the input exploited. We examined the learned constraints and verified that if an input satisfies the constraints, then it will not be able to exploit the vulnerabilities.

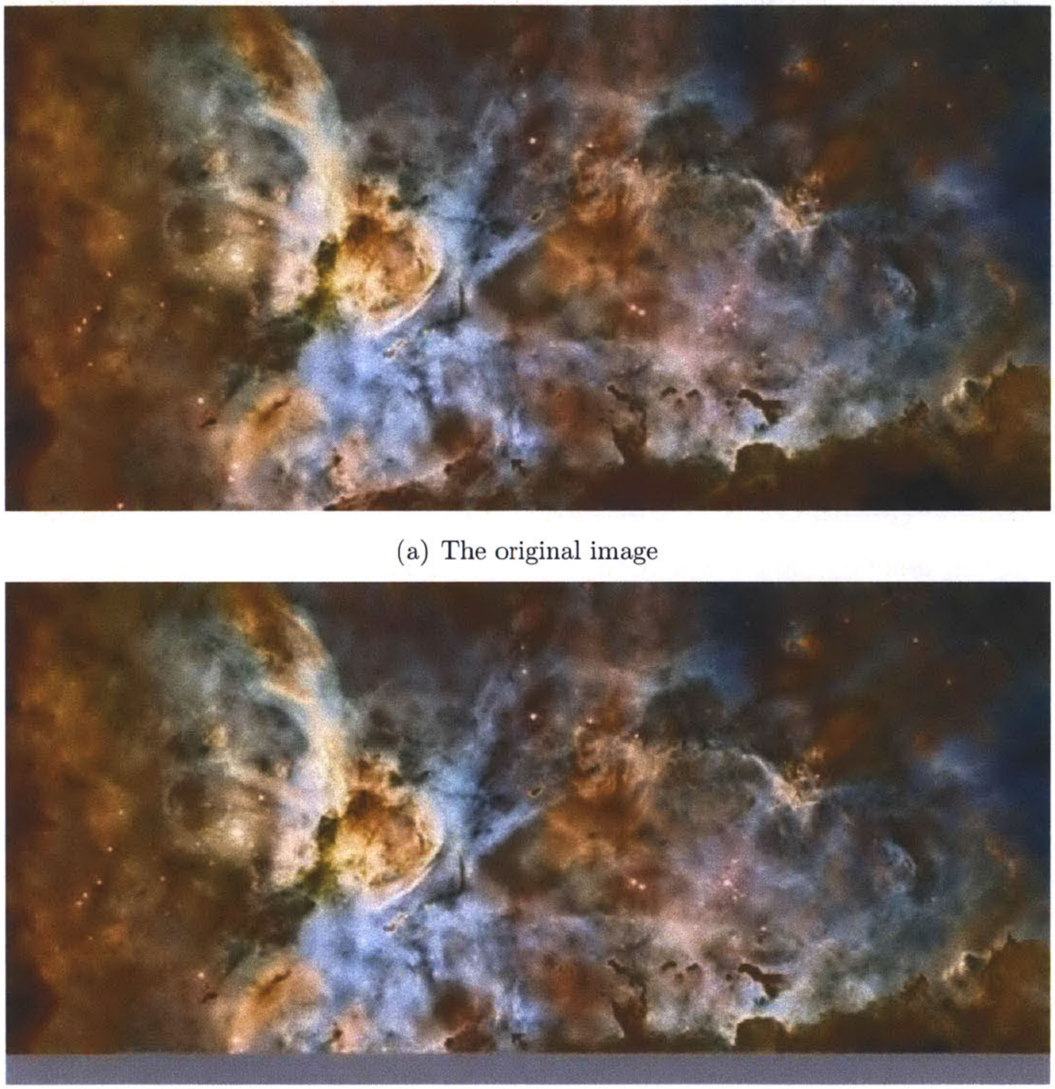
1.4 Understanding Rectification Effects

We examined the original and rectified images or videos for all test inputs that SOAP rectified. These files are available at:

http://groups.csail.mit.edu/pac/input_rectification/

For the majority of rectified inputs (83 out of 110 inputs), the original and rectified images or videos appear identical. The mean Mechanical Turk scores for such images or videos was between 2.5 and 3.0. We attribute this to the fact that the rectifier often modifies fields (such as the name of the author of the file) that are not relevant to the core functionality of the application and therefore do not visibly change the image or video presented to the user. The application must nevertheless parse and process these fields to obtain the desirable data in the input file. Furthermore, since these fields are often viewed as tangential to the primary purpose of the application, the code that handles them may be less extensively tested and therefore more likely to contain errors.

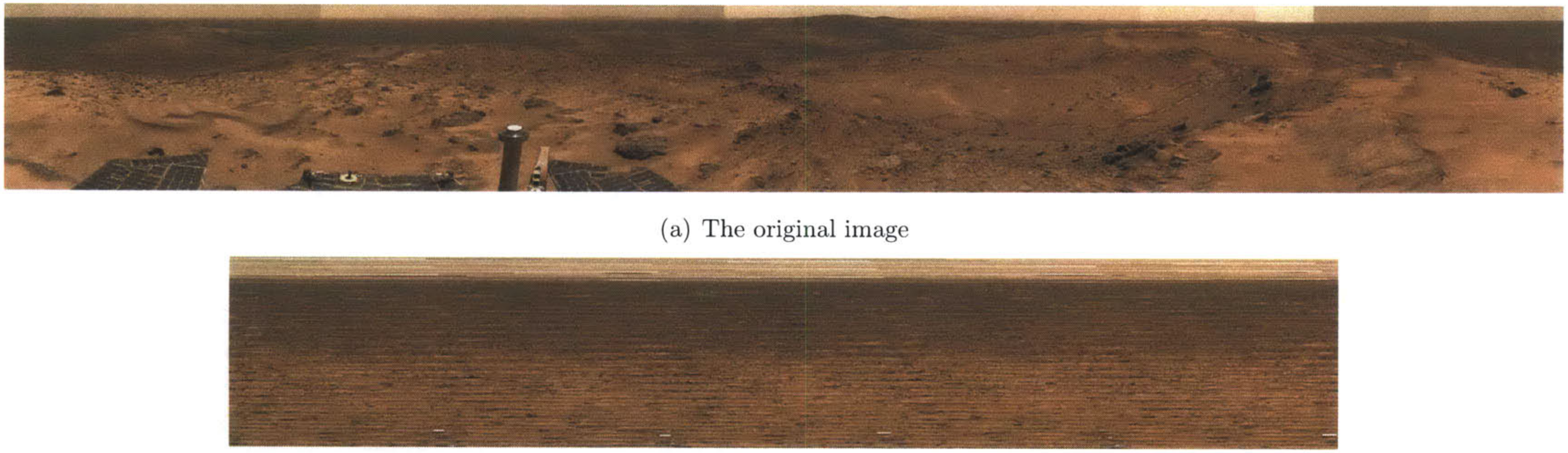
Figures 1-1, 1-2 and 1-3 present examples of images that are visibly changed by rectification. For some of the rectified images (8 of 53 inputs), the rectifier truncates part of the image, leaving a strip along the bottom of the picture (see Figure 1-1).



(a) The original image

(b) The rectified image

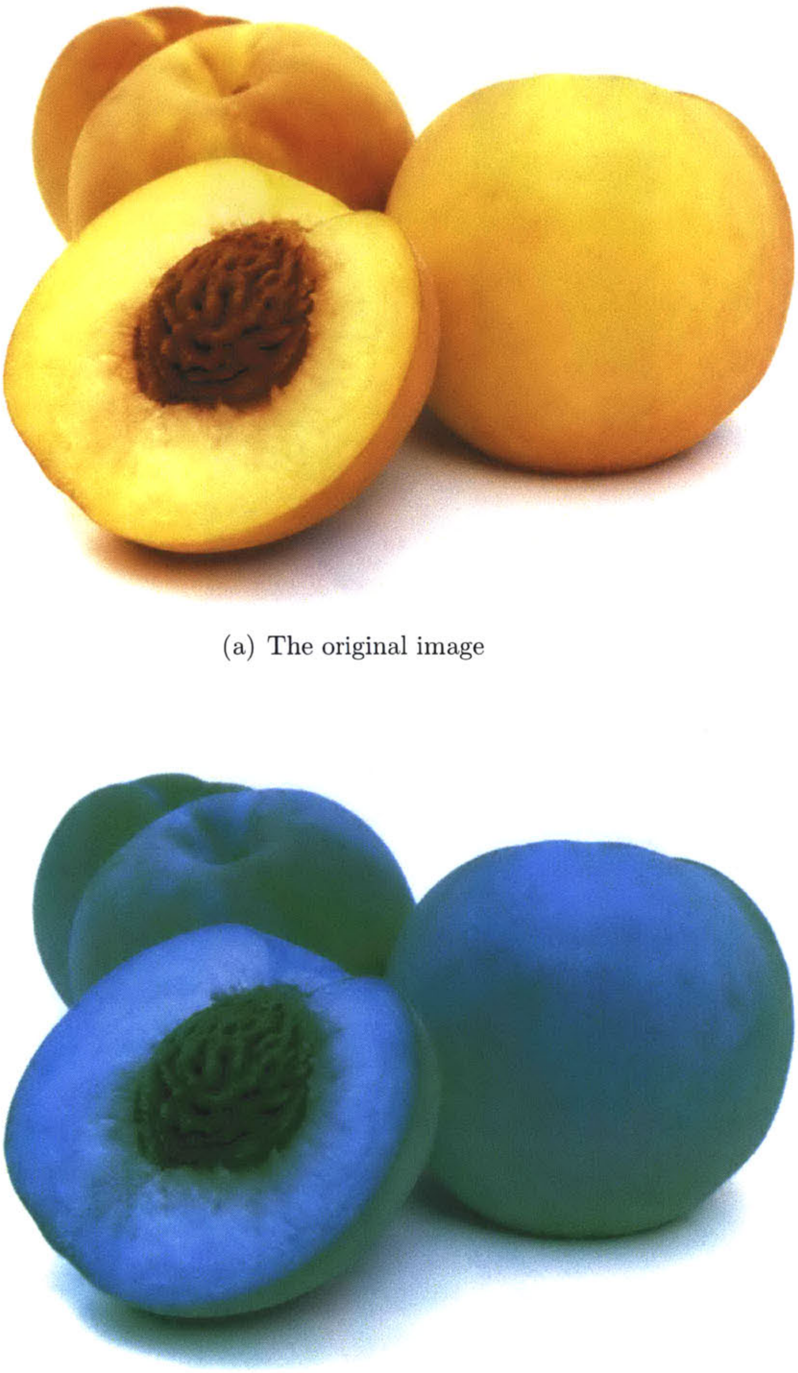
Figure 1-1: A JPEG image truncated by the rectification.



(a) The original image

(b) The rectified image

Figure 1-2: A JPEG image twisted by the rectification



(a) The original image

(b) The rectified image

Figure 1-3: A TIFF image whose color is changed by the rectification.

For the remaining inputs (19 of 110), the rectifier changes fields that control various aspects of core application functionality, for example, the alignment between pixels and the image size (see Figure 1-2), the image color (see Figure 1-3), or interactive aspects of videos. The mean Mechanical Turk scores for such images or videos vary depending on the severity of the effect. In all cases the application was able to process the rectified inputs without error to present the remaining data to the user.

1.5 Contributions

We make the following contributions:

- **Basic Concept:** We propose a novel technique for dealing with atypical or malicious inputs, automatic input rectification, and a prototype implementation, SOAP, which demonstrates the effectiveness of the technique.

- **Constraint Inference:** We show how to use dynamic taint analysis and a constraint inference algorithm to automatically infer safety constraints. This inference algorithm operates correctly to infer correlated constraints for hierarchically structured input files with nested fields.

- **Rectification Algorithm:** We present an input rectification algorithm that systematically enforces safety constraints on inputs while preserving as much of the benign part of the input as possible. Because it is capable of enforcing correlated constraints associated with nested input fields, this algorithm is capable of rectifying hierarchically structured input files.

- **Experimental Methodology:** We present a new experimental methodology for evaluating the significance of changes to program inputs and/or outputs. Specifically, we use Amazon Mechanical Turk [1] to evaluate the subjective perceptual quality of the outputs for rectified inputs. We present effective mechanisms to ensure the quality of the collected responses, which is a primary challenge of utilizing such crowdsourcing workforce.

- **Experimental Results:** Our results indicate that, for our set of benchmark applications and inputs, Mechanical Turk workers perceive rectified images and videos to be, in most cases, close or even identical to the original images and videos (Chapter 5). These results are consistent with our own quantitative

(Chapter 4) and qualitative (Chapter 5) evaluation of the differences between the original and rectified images and videos.

- **Explanation:** We explain (Section 1.4 and Section 5.4) why rectification often preserves much or even all of the desirable data in rectified files.

We organize the rest of this thesis as follows. Chapter 2 presents an example that illustrates how SOAP works. We describe the technical design of SOAP in Chapter 3. We present a quantitative evaluation of SOAP in Chapter 4 and a subjective human evaluation of SOAP in Chapter 5. Chapter 6 discusses related work. We conclude in Chapter 7.

Chapter 2

Motivating Example

Figure 2-1 presents source code from Dillo 2.1, a lightweight open source web browser. Dillo uses libpng to process PNG files. When Dillo starts to load a PNG file, it calls the libpng callback function Png_datainfo_callback() shown in Figure 2-1. The function contains an integer overflow vulnerability at line 20, where the multiplication calculates the size of the image buffer allocated for future callbacks. Because png→rowbytes is proportional to the image width, arithmetic integer overflow will occur when opening a PNG image with maliciously large width and height values. This error causes Dillo to allocate a significantly smaller buffer than required. Dillo eventually writes beyond the end of the buffer.

Dillo developers are well aware of the potential for overflow errors. In fact, the code contains a check of the image size at lines 10-11 to block large images. Unfortunately, the bounds check has a similar integer overflow problem. Specific large width and height values can also cause an overflow at line 10 and thus bypass the check.

SOAP can nullify this error without prior knowledge of the vulnerability itself. To use SOAP, an application developer or system administrator first provides SOAP with a set of typical benign inputs to the application. To nullify the above Dillo error, SOAP performs following steps:

- **Understand Input Format:** SOAP provides a declarative input specification interface that enables users to specify the input format. SOAP then uses this specification to automatically generate a parser, which transforms each PNG input into a collection of potentially nested input fields. Along with the other fields of a typical PNG file, the parser will identify the locations – specifically the byte offsets – of the image width and height fields.
- **Identify Critical Fields:** SOAP monitors the execution of Dillo on training

```
//Dillo’s libpng callback
static void
Png_datainfo_callback(png_structp png_ptr, ...)
{
  DilloPng *png;
  ...
  png = png_get_progressive_ptr(png_ptr);
  ...
  /* check max image size */
  if (abs(png→width*png→height) >
    IMAGE_MAX_W * IMAGE_MAX_H) {
     ...
     Png_error_handling(png_ptr, ”Aborting...”);
     ...
  }
  ...
  png→rowbytes = png_get_rowbytes(png_ptr, info_ptr);
  ...
  png→image_data = (uchar_t *) dMalloc(
    png→rowbytes * png→height);
  ...
}
```

Figure 2-1: The code snippet of Dillo libpng callback (png.c). The boldfaced code is the root cause of the overflow vulnerability.

PNG inputs and determines that values in the image width and height fields influence a critical operation, the memory allocation at line 19-20. Thus SOAP marks width and height in PNG images as critical fields which may cause dangerous overflow. In the Dillo source code, png→height contains the image height field value and the png→rowbytes is proportional to the image width field value. These two variables influence a memory allocation statement at lines 19-20.

- **Infer Constraints:** SOAP next infers constraints over the critical fields, including the height and width fields. Specifically, for each of these fields, SOAP infers an upper bound constraint by recording the largest value that appears in that field for all PNG training inputs.
- **Rectify Atypical Inputs:** SOAP performs the above three steps offline. Once SOAP generates constraints for the PNG format, it can be deployed to parse and rectify new PNG inputs. When SOAP encounters an atypical PNG input whose width or height field is larger than the inferred bound, it enforces the bound by changing the field to the bound. Note that such changes may, in turn, cause other correlated constraints (such as the length of another field involved in a correlated relation with the modified field) to be violated. SOAP therefore rectifies violated constraints iteratively until all of the learned constraints are satisfied.

Chapter 3

Design

SOAP has four components: the *input parser*, the *execution monitor*, the *learning engine*, and the *input rectifier*. The components work together cooperatively to enable automatic input rectification (see Figure 3-1). The execution monitor and the learning engine together generate safety constraints offline before the input rectifier is deployed:

- **Input parser:** The input parser understands input formats. It transforms raw input files into syntactic parse trees for the remaining components to process.

- **Execution Monitor:** The execution monitor uses taint tracing to analyze the execution of an application. It identifies critical input fields that influence critical operations (i.e., memory allocations and memory writes).

- **Learning Engine:** The learning engine starts with a set of benign training inputs and the list of identified critical fields. It infers safety constraints based on the field values in these training inputs. Safety constraints define the comfort zone of the application.

- **Input Rectifier:** The input rectifier rectifies atypical inputs to enforce safety constraints. The algorithm modifies the input iteratively until it satisfies all of the constraints.

3.1 Input Parser

As shown in Figure 3-1, the input parser transforms an arbitrary input into a general syntactic parse tree that can be easily consumed by the remaining components.

In the syntactic parse tree, only leaf fields are directly associated with input data. The hierarchical tree structure reflects nesting relationships between input fields.

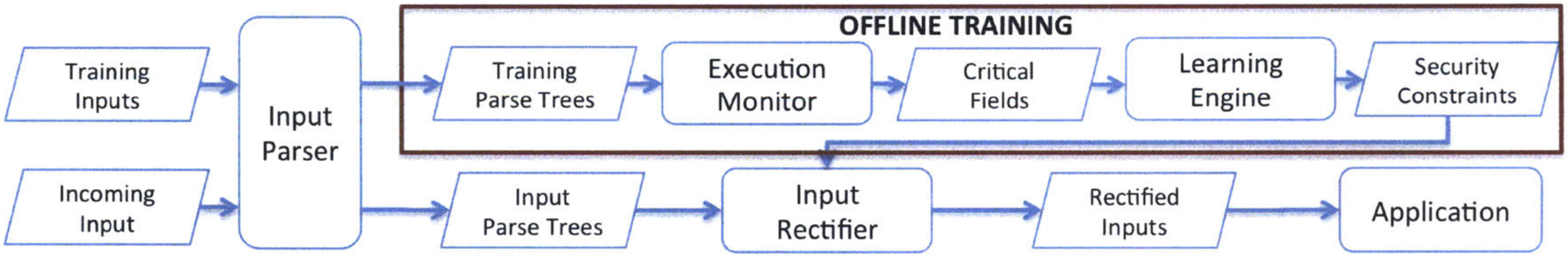


Figure 3-1: The architecture of the SOAP automatic input rectification system.

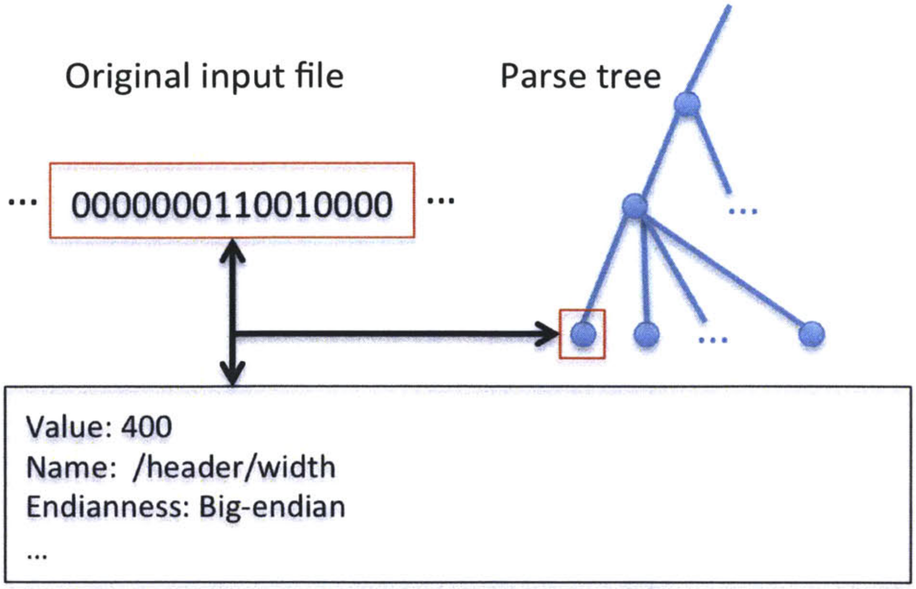


Figure 3-2: A syntax parse tree example.

Each leaf field has a type, which can be integer, string or raw bytes, while each non-leaf field is a composite field with several child fields nested inside it.

Figure 3-2 presents an example of an integer leaf field inside a parse tree for a PNG image file. The leaf field identifies the location of the data in the input file. It also contains a descriptor that specifies various aspects of the field, such as the value stored in the field, the name of the field, and the encoding information such as whether the value is stored in big endian or little endian form.

The only requirement for the input parser is the input specification for the particular format. This is reasonable as most input formats of production systems have been standardized and their specification is publicly available. In the absence of an input specification, SOAP can use one of the existing syntax inference techniques [16, 27, 11, 43, 33].

3.2 Execution Monitor

The execution monitor identifies the critical input fields that should be involved in the learned constraints (see Figure 3-1). Because large data fields may trigger memory buffer overflows, the execution monitor treats all variable-length data fields as critical. Integer fields present a more complicated scenario. Integer fields that influence the addresses of memory writes or the values used at memory allocation sites (e.g., calls to malloc() and calloc()) are relevant for our target set of errors. Other integer fields (for example, control bits or checksums) may not affect relevant program actions.

3.2.1 Taint Analysis

The execution monitor uses dynamic taint analysis [13, 32] to compute the set of critical integer fields. Specifically, SOAP considers an integer field to be critical if the dynamic taint analysis indicates that the value of the field may influence the address of memory writes or values used at memory allocation sites.

We next sketch the dynamic taint analysis in the execution monitor. The execution monitor actively tracks a mapping from memory addresses and registers to sets of input byte positions. This mapping represents the input byte positions that influence each memory address or each register.

The execution monitor dynamically instruments the binary code of the application to insert callbacks to maintain this mapping. The execution monitor also inserts callback before each critical operation (memory write or memory allocation site). This callback records the input bytes that influence the critical operation based on the current mapping information.

The execution monitor currently tracks data dependences only. This approach works well for our set of benchmark applications, eliminating 58.3%-88.7% of integer fields from consideration. It would be possible to use a taint system that tracks control dependences [10] as well.

3.2.2 Training Input Selection

Dynamic taint analysis generally imposes 10x-100x overhead on the application execution. Given a limited time budget for offline training, the execution monitor may not be able to run the application on all training inputs.

The execution monitor uses an automated greedy algorithm to select a subset of the training inputs for the runs that determine the critical integer fields. The goal is to select a small set of training inputs that 1) reduce the total execution time on the execution monitor and 2) together cover all of the integer fields that may appear in the input files. Our algorithm assumes that the running time of an input is proportional to the size of the input. Therefore it attempts to minimize the total size of selected training inputs.

The greedy algorithm contains a main loop. For each training input I, the algorithm computes the number of new integer fields the input I covers (C_I) and the size

```
1 /header/width <= 1920
2 /header/width >= 0
3 sizebits(/text/text) <= 21112
4 /text/size * 8 == sizebits(/text/keyword)
5     + sizebits(/text/text)
```

Figure 3-3: A subset of constraints generated by SOAP for PNG image files.

of the input I (S_I). At each iteration, the algorithm select a new training input I that maximize C_I/S_I, until all integer fields have been covered.

3.3 Learning Engine

The learning engine works with the parse trees of the training inputs and the list of critical fields as identified by the execution monitor. It infers safety constraints over critical fields (see the offline training box in Figure 3-1).

3.3.1 Safety Constraints

Overflow vulnerabilities are typically exploited by large data fields, extreme values, negative entries or inconsistencies involving multiple fields. Figure 3-3 presents several examples of constraints that SOAP infers for PNG image files. Specifically, SOAP infers upper bound constraints of integer fields (line 1 in Figure 3-3), sign constraints of integer fields (line 2), upper bound constraints of data field lengths (line 3), and correlated constraints between values and lengths of parse tree fields (lines 4-5).

These kinds of constraints enable the rectification system to eliminate extreme values in integer fields, overly long data fields, and inconsistencies between the specified and actual lengths of data fields in the input. When properly inferred and enforced, these constraints enable the rectifier to nullify the target set of errors and vulnerabilities in our benchmark.

Note that once SOAP infers a set of safety constraints for one input format, it may use these constraints to rectify inputs for any application that reads inputs in that format. This is useful when different applications are vulnerable to the same exploit. For example, both Picasa [6] and ImageMagick [5] are vulnerable to the same overflow exploit (see Section 4). A single set of inferred constraints enables SOAP to nullify the vulnerability for both applications.

3.3.2 Inferring Bound Constraints

SOAP infers three kinds of bound constraints: upper bound constraints of integer fields, sign constraints of integer fields, and upper bound constraints of data field lengths. SOAP learns the maximum value of an integer field in training inputs as the upper bound of its value. SOAP learns an integer field to be non-negative if it is never negative in all training inputs. SOAP also learns the maximum length of a data field that appeared in training inputs as the upper bound of its length. SOAP infers all these constraints with a single pass over all the parse trees of the training inputs.

3.3.3 Inferring Correlated Constraints

SOAP infers correlated constraints in which an integer field f indicates the total length of *consecutive* children of the parent field of f. Lines 4-5 in Figure 3-3 present an example. A text data chunk in PNG image file contains five different fields: size, tag, keyword, text, and crc32 in this order. The constraint states that the value of the field "/text/size" is the total length in bytes of the field "/text/keyword" and the field "/text/text", which are two consecutive nested fields inside the field "/text".

Figure 3-4 shows the pseudo-code of the inference algorithm for length indicator constraints in SOAP. The SOAP learning algorithm first enumerates all possible field combinations for correlated constraints initially assuming that all of these constraints are true. When processing each training input, the algorithm eliminates constraints that do not hold in the input.

The pseudo-code in Figure 3-4 uses a map R to track valid length indicator constraints associated with each integer field. At lines 1-12, the pseudo-code initializes R with all possible length indicator constraints. fields in the code represents data fields whose lengths are controlled by the integer field. scale identifies whether the length field counts the size in bits or in bytes[1]. At lines 14-27, the algorithm tests each candidate constraint on the parse tree of each training input. After the algorithm processes all training inputs, the inference algorithm outputs the length indicator constraints that still remain in R.

[1] For convenience, the pseudo-code here assumes that the length values count lengths either in bits or bytes. Our algorithm extends to handle arbitrary measures.

```
// Initialization
FOR f IN all integer fields:
  R[f] $\leftarrow$ $\emptyset$
  p $\leftarrow$ f.parent
  FOR S IN subsets of consecutive children of p:
    FOR scale IN $\{1, 8\}$:
      relation.fields $\leftarrow$ S
      relation.scale $\leftarrow$ scale
      R[f].add(relation)
    END FOR
  END FOR
END FOR

// Checking relations against each parse tree
FOR parse_tree IN training_input_set:
  FOR f IN parse_tree.integer_fields:
    FOR relation IN R[f]:
      sum $\leftarrow$ 0
      FOR data_field IN relation.fields:
        sum $\leftarrow$ sum + sizeinbit(data_field)
      END FOR
      IF f.value * relation.scale $\neq$ sum:
        R[f].erase(relation)
      END IF
    END FOR
  END FOR
END FOR
```

Figure 3-4: Inference algorithm for correlated constraints.

3.4 Input Rectifier

Given safety constraints generated by the learning engine and a new input, the input rectifier rectifies the input if it violates the safety constraints (see Figure 3-1). The main challenge in designing the input rectifier is enforcing safety constraints while preserving as much desirable data as possible.

3.4.1 Rectification Algorithm

Our algorithm is designed around two principles: 1) It enforces constraints only by modifying integer fields or truncating data fields—it does not change the parse tree structure of the input. 2) At each step, it attempts to minimize the value difference of the modified integer field or the amount of truncated data. It finds a single violated constraint and applies a minimum modification or truncation to enforce the constraint.

Nested input fields further complicate rectification, because changing one field may cause the file to violate correlated constraints associated with enclosing or enclosed fields at other levels. Our algorithm therefore iteratively continues the rectification process until there are no more violated constraints. In our experiments, SOAP enforces as many as 86 constraints on some rectified input files.

Figure 3-5 presents the pseudo-code of the SOAP rectification algorithm. upbound maps an integer field or a data field to the corresponding upper bound of its value or its length. If a field f has no upper bound, upbound[f]=∞. may_neg stores the set of integer fields that have been observed to allow negative values. R stores length indicator constraints (see Section 3.3).

Our algorithm has a main loop that iteratively checks the input against learned constraints. The loop exits when the input no longer violates any constraint. At each iteration, it applies a rectification action depending on the violated constraint:

- **Upper bounds of integer fields:** At lines 4-8 in Figure 3-5, if the input violates the upper bound constraint of an integer field, our algorithm changes the value of the field to the learned upper bound.

- **Sign Constraints of integer fields:** At lines 9-13 in Figure 3-5, if the input violates the sign constraint of an integer field, our algorithm changes the value of the field to zero.

```
REPEAT
 violated $\leftarrow$ false;
 FOR f IN input.integer_fields:
  // Checking against upper bounds
  IF f.value $>$ upbound[f]:
   f.value $\leftarrow$ upbound[f]
   violated $\leftarrow$ true
  END IF
  // Checking against non-negativeness
  IF f.value $<$ 0 and f $\notin$ may_neg:
   f.value $\leftarrow$ 0
   violated $\leftarrow$ true
  END IF
 END FOR

 FOR f IN input.data_fields:
  // Checking against length upper bounds
  IF f.size $>$ upbound[data_field.name]:
   truncate f to size upbound[data_field.name];
   violated $\leftarrow$ true
  END IF
 END FOR

 FOR f IN input.integer_fields:
  FOR rel IN R[f]:
   // Checking against length indicator constraints
   IF f.value * rel.scale $>$ sizeinbit(rel.fields):
    f.value $\leftarrow$ sizeinbit(rel.fields) / rel.scale
    violated $\leftarrow$ true
   ELSE IF f.value * rel.scale $<$ sizeinbit(rel.fields):
    truncate rel.fields to size f.value * rel.scale
    violated $\leftarrow$ true
   END IF
  END FOR
 END FOR
UNTIL violated = false
```

Figure 3-5: The rectification algorithm in SOAP.

- **Upper bounds of data field lengths:** At lines 16-22 in Figure 3-5, if the input violates the upper bound constraint of the length of a data field, our algorithm truncates the data field to its length upper bound.
- **Correlated constraints:** At lines 27-29 in Figure 3-5, if the value of a length indicator field is greater than the actual length of corresponding data fields, our algorithm changes the value to the actual length.

  At lines 30-32 in Figure 3-5, if the total length of a set of data fields is longer than the length indicated by a corresponding integer field, our algorithm truncates one or more data fields to ensure that the input satisfies the constraint. Note that correlated constraints may be violated due to previous enforcements of other constraints. To avoid violating previously enforced constraints, our algorithm does not increase the value of the length indicator field or increase the field length, which may roll back previous changes.

3.4.2 Properties of the Algorithm

Note that this algorithm will always terminate, because the absolute values of integer fields and the lengths of data fields always decrease at each iteration. The algorithm may discard data when truncating data fields to enforce upper bounds constraints of data fields (See lines 16-22) or correlated constraints (See lines 30-32). Thus we can use the amount of truncated data to measure the data loss after rectification. Because the algorithm truncates a minimum amount of data each iteration, it produces a locally optimal rectified input, i.e., if the algorithm returns a rectified input I_a, there is no other rectified input I_b such that I_b contains all data of I_a.

Note also that rectifying an input to enforce correlated constraints with minimum data truncation is NP-hard in general. We can easily prove the NP-hardness by reduction from the set cover problem [24], which is known to be NP-complete. We next sketch the proof.

The definition of the set cover problem is the following: given an universe set $\{1, \ldots, m\}$ and n sets $S_1, \ldots, S_n$ whose union comprises the universe, the question is whether it is possible to select k or less sets from the given n sets such that the union of the selected sets still comprises the universe.

We can construct a rectification problem instance from a set cover problem instance as follows. For each set S_i, we create a corresponding data field $data_i$ with length 1 in the input. For each universe element i, we create a corresponding integer field int_i. Assume $S_{a_1}, S_{a_2}, \ldots, S_{a_k}$ are sets that contain this element i. We set the original value of int_i to $k - 1$ and create a correlated constraint, $int_i = size(data_{a_1}) + \cdots + size(data_{a_k})$. We also create a bound constraint, $int_i \leq k - 1$.

Note that we can construct the solution for the set cover problem instance from the solution of this constructed rectification problem instance. To obtain the solution for the set cover problem, we simply select all sets whose corresponding data fields are truncated in the rectification problem solution. The correlated constraints guarantee that for each universe element we selected at least one set that covers this element. Therefore we obtain a valid solution for the set cover problem instance. This reduction shows that the input rectification problem with correlated constraints is NP-hard.

3.4.3 Checksum

SOAP appropriately updates checksums after the rectification. SOAP currently relies on the input parser to identify the fields that store checksums and the method used to compute checksums. After the rectification algorithm terminates, SOAP calculates the new checksums and appropriately updates checksum fields. SOAP could use the checksum repair technique in TaintScope [41] to further automate this step.

3.5 Implementation

The SOAP learning engine and input rectifier are implemented in Python. The execution monitor is implemented in C based on Valgrind [30], a dynamic binary instrumentation framework. The input parser is implemented with Hachoir [4], a manually maintained Python library for parsing binary streams in various formats. SOAP is able to process any file format that Hachoir supports. Because SOAP implements an extensible framework, it can work with additional parser components implemented in the declarative specification interface of Hachoir to support other input formats.

Chapter 4

Quantitative Evaluation

We next present a quantitative evaluation of SOAP using five popular media applications. Specifically, the following questions drive our evaluation:

1. Is SOAP effective in nullifying errors?

2. How much desirable data does rectification preserve?

3. How does the amount of training data affect SOAP's ability to preserve desirable data?

4.1 Experimental Setup

Applications and Errors

We use SOAP to rectify inputs for five applications: Swfdec 0.5.5 (a shockwave player) [7], Dillo 2.1 (a browser) [3], ImageMagick 6.5.2-8 (an image processing toolbox) [5], Google Picasa 3.5 (a photo managing application) [6], and VLC 0.8.6h (a media player) [8].

Table 4.1 presents a description of each error in each application. In Table 4.1, "Source" presents the source where we collected this vulnerability. "Fault" and "Format" present the fault type and the format of malicious inputs that can trigger this error. "Position" presents the source code file and/or line positions that are related to the root cause. "Related constraints" presents constraints generated by SOAP that nullify the vulnerability.

These applications consume inputs that (if crafted) may cause the applications to incorrectly allocate memory or perform an invalid memory access. The input formats

Application	Source	Fault	Format	Position	Related constraints
Swfdec	Buzzfuzz	X11 crash	SWF	XCreatePixMap	/rect/xmax ≤ 57600 /rect/ymax ≤ 51000
Swfdec	Buzzfuzz	overflow/crash	SWF	jpeg.c:192	/sub jpeg/.../width ≤ 6020 /sub jpeg/.../height ≤ 2351
Dillo	CVE-2009-2294	overflow/crash	PNG	png.c:142 png.c:203	/header/width ≤ 1920 /header/height ≤ 1080
ImageMagick	CVE-2009-1882	overflow/crash	JPEG,TIFF	xwindow.c:5619	/ifd[..]/img_width/value ≤ 14764 /ifd[..]/img_height/value ≤ 24576 /start_frame/content/width ≤ 15941 /start_frame/content/height ≤ 29803
Picasa	TaintScope	overflow/crash	JPEG,TIFF	N/A	
VLC	CVE-2008-2430	overflow/crash	WAV	wav.c:147	/format/size ≤ 150

Table 4.1: The six errors used in our experiments. SOAP successfully nullifies all of these errors.

Input	Application	Train	Test	Field	Distinct
SWF	Swfdec	3620	3620	5550.2	98.17
PNG	Dillo	1496	1497	306.8	32.3
JPEG	IMK, Picasa	3025	3024	298.2	75.5
TIFF	IMK, Picasa	870	872	333.5	84.5
WAV	VLC	5488	5488	17.1	16.8

Table 4.2: Applications and collected inputs for our experiments.

for these errors are the SWF Shockwave Flash format; the PNG, JPEG, and TIF image formats; and the WAV sound format.

Malicious Inputs

We obtained six malicious input files from the CVE database [2], the Buzzfuzz project [21] and the TaintScope project [41]. Each input targets a distinct error (see Table 4.1) in at least one of the examined applications. We obtained three of these inputs from the CVE database [2], two from the example inputs of the Buzzfuzz project [21], and one from the example inputs of the TaintScope project [41].

Benign Inputs

We implemented a web crawler to collect input files for each format (see Table 4.2 for the number of collected inputs for each input format). The "Input" column presents the input file format. The "Application" column presents the application name (here "IMK" is an abbreviation of ImageMagick). The "Train" column presents the number of training inputs. The "Test" column presents the number of test inputs. The "Field" column presents the mean number of fields in each test input of each format. The "Distinct" column presents the mean number of semantically distinct fields (i.e., fields that have different names) in each test input.

Our web crawler uses Google's search interface to acquire a list of pages that contain at least one link to a file of a specified format (e.g., SWF, JPEG, or WAV). The crawler then downloads each file linked within each page. We verified that all of these inputs are benign, i.e., the corresponding applications successfully process these inputs. For each format, we randomly partitioned these inputs into two sets, the training set and the test set (see Table 4.2).

Training

For each input format and application, we trained SOAP with the training set to obtain safety constraints.

Picasa is a Windows application and because our execution monitor does not support Windows we, therefore, cannot monitor the execution of Picasa to identify critical fields for JPEG. However, because ImageMagick processes both TIF and JPEG files, we were able monitor the execution of ImageMagick on JPEG files to determine critical fields and then evaluate the effectiveness of the derived constraints for Picasa.

Rectification

Once SOAP learned the safety constraint for each application, we deploy the SOAP input rectifier as front-end to the application test it with the malicious input and all benign test inputs of the application. All rectification experiments were run on a 3.33 GHz machine with 24 GB of RAM.

4.2 Nullifying Vulnerabilities

We next evaluate the effectiveness of SOAP in nullifying six vulnerabilities in our benchmark (see Table 4.1). We applied the rectifier to the obtained malicious inputs. The rectifier detected that all of these inputs violated at least one constraint. It enforced all constraints to produce six corresponding rectified inputs. We verified that the applications processed the rectified inputs without error and that none of the rectified inputs exploited the vulnerabilities. We next discuss the interactions between the inputs and the root cause of each vulnerability.

4.2.1 Flash Video

The root cause of the X11 crash error in Swfdec is a failure to check for large Swfdec viewing window sizes as specified in the input file. If this window size is very large, the X11 library will allocate an extremely large buffer for the window and Swfdec will eventually crash. SOAP nullifies this error by enforcing the constraints /rect/xmax $\leq$ 57600 and /rect/ymax $\leq$ 51000, which limit the window to a size that Swfdec can handle. In this way, SOAP ensures that no rectified input will be able to exploit this error in Swfdec.

The integer overflow vulnerabilities in Swfdec occurs when Swfdec calculates the required size of the memory buffer for JPEG images embedded within the SWF file. If the SWF input file contains a JPEG image with abnormally large specified width and height values, this calculation will overflow and Swfdec will allocate a buffer significantly smaller than the required size. When SOAP enforces the learned constraints, it nullifies the error by limiting the size of the embedded image. No rectified input will be able to exploit this error.

4.2.2 Image

Errors in Dillo, ImageMagick and Picasa have similar root causes. A large PNG image with crafted width and height can exploit the integer overflow vulnerability in Dillo (see Chapter 2). The same malicious JPEG and TIFF images can exploit vulnerabilities in both ImageMagick (running on Linux) and Picasa Photo Viewer (running on Windows). ImageMagick does not check the size of images when allocating an image buffer for display at magick/xwindow.c:5619 in function XMakeImage(). Picasa Photo Viewer also mishandles large image files [41]. By enforcing the safety constraints, SOAP limits the size of input images and nullifies these vulnerabilities (across applications and operating systems).

4.2.3 Sound

VLC has an overflow vulnerability when processing the format chunk of a WAV file. The integer field /format/size specifies the size of the format chunk (which is less than 150 in typical WAV files). VLC allocates a memory buffer to hold the format chunk with the size of the buffer equal to the value of the field /format/size plus two. A malicious input with a large value (such as 0xfffffffe) in this field can exploit this overflow vulnerability. By enforcing the constraint /format/size ≤ 150, SOAP limits the size of the format chunk in WAV files and nullifies this vulnerability.

These results indicate that SOAP effectively nullifies all six vulnerabilities. Our code inspection proves that the learned constraints nullify the root causes of all of the vulnerabilities so that no input, after rectification, can exploit the vulnerabilities.

4.3 Data Loss

We next compute a quantitative measure of the rectification effect on data loss. For each format, we first apply the rectifier to the test inputs. We report the mean data

Input	Test	Rectified	Max enforced	Mean enforced	P_{loss}
SWF	3620	57 (1.57%)	86	8.61	N/A
PNG	1497	0 (0%)	0	0	0%
JPEG	3024	42 (1.39%)	8	2.55	0.08%
TIFF	872	11 (1.26%)	2	1.36	0.50%
WAV	5488	11 (0.20%)	2	1.09	0%

Table 4.3: The statistic of input rectification on test inputs of each input format.

loss percentage of all test inputs for each format. We use the following formula to compute the data loss percentage of a rectified input i:

$$P^i_{loss} = \frac{D^i_{loss}}{D^i_{tot}}$$

D^i_{tot} measures the amount of desirable data before rectification and D^i_{loss} measures the amount of desirable data lost in the rectification process. For JPEG, TIFF and PNG files, D^i_{tot} is the number of pixels in the image and D^i_{loss} is the number of pixels that change after rectification. For WAV files, D^i_{tot} is the number of frames in the sound file and D^i_{loss} is the number of frames that change after rectification. Because SWF files typically contain interactive content such as animations and dynamic objects that respond to user inputs, we did not attempt to develop a corresponding metric. We instead rely solely on our human evaluation in Section 5 for SWF files.

Result Interpretation

Table 4.3 presents rectification results from the test inputs of each input format. The "Input" column presents the input file format. The "Test" column presents the number of test inputs. The "Rectified" column has entries of the form X(Y), where X is the number of test inputs that the rectifier modified and Y is the corresponding percentage of modified test inputs. The "Max enforced" column presents the maximum number of constraints that SOAP enforced over all rectified test inputs of each format. The "Mean enforced" column presents the mean number of constraints that SOAP enforced over all rectified test inputs of each format. Finally, the "P_{loss}" column presents the mean data loss percentage over all test inputs of each format.

First, note that more than 98% of the test inputs satisfy all constraints and are therefore left unchanged by the rectifier. Note also that both PNG and WAV have zero desirable data loss — PNG because the rectifier did not modify any test inputs, WAV because the modifications did not affect the desirable data. For JPEG and TIFF, the mean desirable data loss is less than 0.5%.

One of the reasons that the desirable data loss numbers are so small is that rectifications often change fields (such as the name of the author of the data file or the software package that created the data file) that do not affect the output presented to the user. The application must nevertheless parse and process these fields to obtain the desirable data in the input file. Because these fields are often viewed as tangential to the primary purpose of the application, the code that processes them may be less extensively tested and therefore more vulnerable to exploitation.

Also note that the SOAP rectification algorithm often needed to iteratively enforce correlated constraints on nested fields. For the SWF format, SOAP enforced as many as 86 constraints on some SWF inputs. This is because the SWF format contains deeply nested fields, so that enforcing a violated constraint will cause violations of other correlated constraints at different levels.

4.4 Size of Training Input Set

We next investigate how the size of the training input set affects the rectification result. Intuitively, we expect that using fewer training inputs will produce more restrictive constraints which, in turn, will cause more data loss in the rectification. For each format, we incrementally increase the size of the training input set and record the data loss percentage on the test inputs. At each step, we increase the number of training inputs by 200. Figure 4-1 presents curves which plot the mean data loss percentage of the different input formats as a function of the size of the training input set.

As expected, the curves initially drop rapidly, then approach a limit as the training set sizes become large. Note that the PNG and WAV curves converge more rapidly than the TIFF and JPEG curves. We attribute this to the fact that the PNG and WAV formats are simpler than the TIFF and JPEG formats (see Table 4.2 for the number of semantically distinct fields).

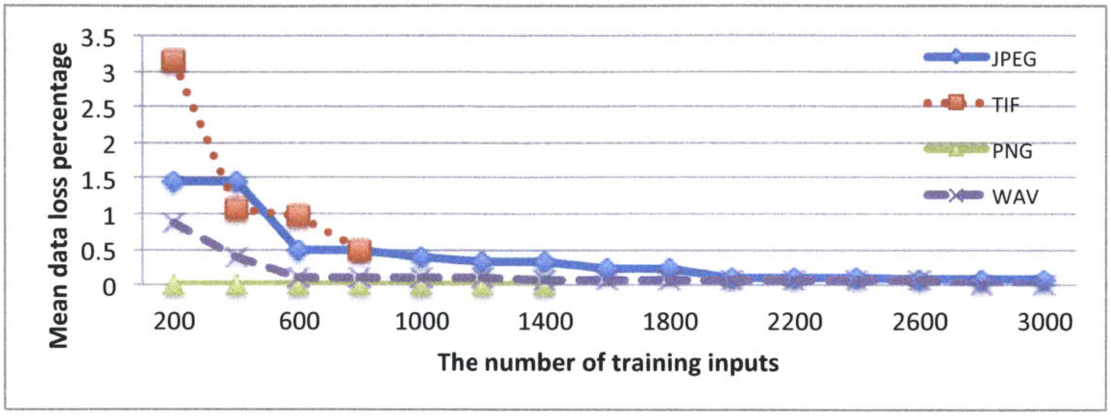


Figure 4-1: The mean data loss percentage curves under different sizes of training input sets for JPEG, TIFF, WAV and PNG (see Section 4.4). X-axis indicates the size of training input sets. Y-axis indicates the mean data loss percentage.

Input	Mean	Parse	Rectification	Per field
SWF	531ms	443ms	88ms	0.096ms
PNG	23ms	19ms	4ms	0.075ms
JPEG	24ms	21ms	3ms	0.080ms
TIFF	31ms	26ms	5ms	0.093ms
WAV	1.5ms	1.3ms	0.2ms	0.088ms

Table 4.4: The running time of SOAP on test inputs of each format.

4.5 Overhead

We next evaluate the rectification overhead that SOAP introduces. All times are measured on an Intel 3.33GHz 6-core machine with SOAP running on only one core.

Table 4.4 presents the mean running time of the SOAP rectifier for processing the test inputs of each file format. The "Input" column presents the input file format. The "Mean" column presents the mean running time for each test input including both parsing and rectification. The "Parse" column presents the mean parsing time for each input. The "Rectification" column presents the mean rectification time for each input. The "Per field" column presents the mean running time divided by the number of input fields.

The results show that the majority of the execution time is incurred in the Hachoir parsing library, with the execution time per field roughly constant across the input formats (so SWF files take longer to parse because they have more fields than other kinds of files). We expect that users will find these rectification overheads negligible during interactive use.

Chapter 5

Mechanical Turk-Based Evaluation

Amazon Mechanical Turk [1] is a Web-based crowdsourcing labor market. Requesters post Human Intelligence Tasks (HITs); workers solve those HITs in return for a small payment.

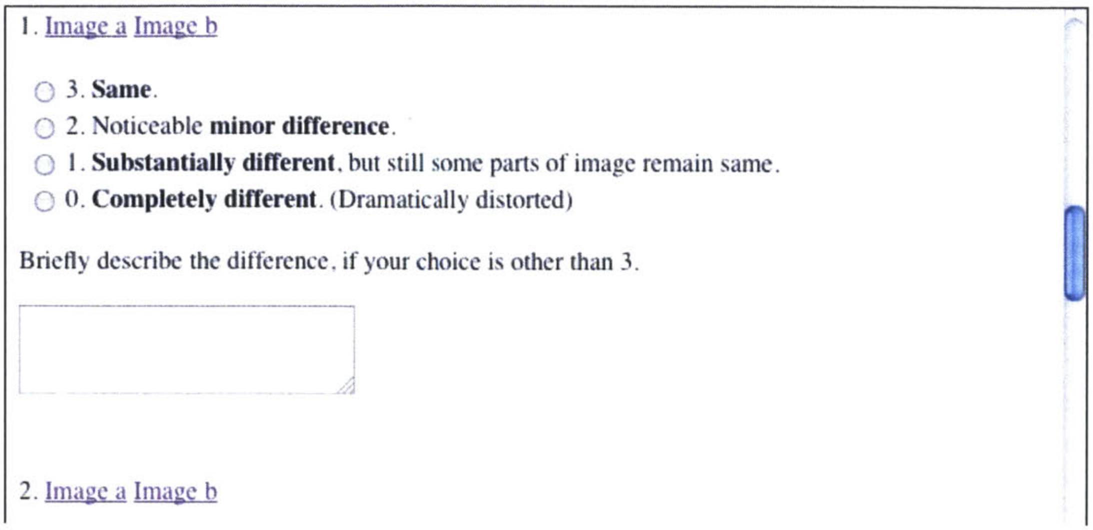
5.1 Experiment Design

We organized the experiment as follows:

- **Input Files:** We collected all of the TIFF, JPEG, and SWF test input files that the rectifier modified (we exclude PNG and WAV files because the original and rectified files have no differences that are visible to the user).

- **HIT Organization:** Together, the TIFF and JPEG files comprise the image files. The SWF files comprise a separate pool of video files. We partition the image files into groups, with four files per group. There is one HIT for each group; the HIT presents the original and rectified versions of the files in the group to the worker for rating. The HIT also contains a control pair. With probability 0.5 the control pair consists of identical images; with probability 0.5 the control pair consists of two completely different images. We similarly create HITs for the videos.

- **HIT Copies:** We post 100 copies of each HIT on Mechanical Turk. Each Mechanical Turk worker rates each pair in the HIT on a scale from 0 to 3. A score of 3 indicates no visible difference between the images (or videos), 2 indicates only minor visible differences, 1 indicates a substantial visible difference, and 0 indicates that the two images (or videos) appear completely different.



1. Image a Image b

○ 3. **Same**.
○ 2. Noticeable **minor difference**.
○ 1. **Substantially different**, but still some parts of image remain same.
○ 0. **Completely different**. (Dramatically distorted)

Briefly describe the difference, if your choice is other than 3.

2. Image a Image b

Figure 5-1: The user interface of an image comparison HIT.

Figure 5-1 presents the user interface of an image comparison HIT. The worker accesses images through the links. The worker rates the difference of each pair of images by choosing one of the four options. The worker also describes the difference in the text box.

5.2 Mechanisms for Validity

As with all marketplaces that involve the exchange of currency, Amazon's Mechanical Turk contains misbehaving users. For example, some workers attempt to game the system by using automated bots to perform the HITs or simply by providing arbitrary answers to HITs without attempting to perform the evaluation [25]. We used three mechanisms to recognize and discard results from such workers:

- **Approval Rating:** Amazon rates each Mechanical Turk worker and provides this information to HIT requestors. This rating indicates what percentage of that worker's previously performed HITs were accepted by other requestors as valid. We required that prospective Mechanical Turk workers have an acceptance rate of at least 95%. Using an approval rating filter provides an initial quality filter but cannot guarantee future worker performance.

- **Control Pairs:** Each HIT contains five pairs, one of which was a control pair. Half of the control pairs contained identical images or videos, while the

Format	Undetectable diff.	Minor diff.	Substantial diff.	Complete diff.
SWF	43 (1.19%)	7 (0.19%)	7 (0.19%)	0
JPEG	37 (1.22%)	3 (0.10%)	1 (0.03%)	1 (0.03%)
TIFF	3 (0.34%)	5 (0.57%)	2 (0.23%)	1 (0.11%)

Table 5.1: Numerical results of Mechanical Turk experiment.

other half contained completely different images or videos (one of the images or videos was simply null). If a worker did not correctly evaluate the control pair, we discarded the results from that worker. This technique can effectively detect bots and misbehaving workers but requires control pairs that are difficult to misinterpret.

- **Descriptions:** For each HIT, we require workers to provide a short description of the perceived differences (if any) between image or video pairs. By forcing users to provide a textual description, we help users transition from performing motor control actions (e.g., clicking on images) to cognitive executive functions. This technique helps improve the performance of legitimate workers and enables the detection of misbehaving users by monitoring for empty or nonsensical descriptions.

Whenever we discarded a result from a worker , we reposted a copy of the HIT to ensure that we obtained results for all 100 copies of each HIT.

5.3 Experiment Result

For each HIT h, we computed the mean scores over all the scores given by the workers assigned to h. We then used the mean scores to classify the files in h into four categories: undetectable difference (score in $[2.5, 3]$), minor difference (score in $[1.5, 2.5)$), substantial difference (score in $[0.5, 1.5)$), and complete difference (score in $[0, 0.5)$).

Table 5.1 presents, for each combination of input file format and classification, an entry of the form X(Y), where X is the number of files in that classification and Y is the corresponding percentage out of all test inputs. Figure 5-2 presents graphical ratio results for each categories.

Note that, out of 110 rectified inputs, only two exhibit a complete difference after rectification. Only 12 exhibit more than a minor difference. This shows that SOAP can often preserve most or even all desirable data in the test inputs for the user.

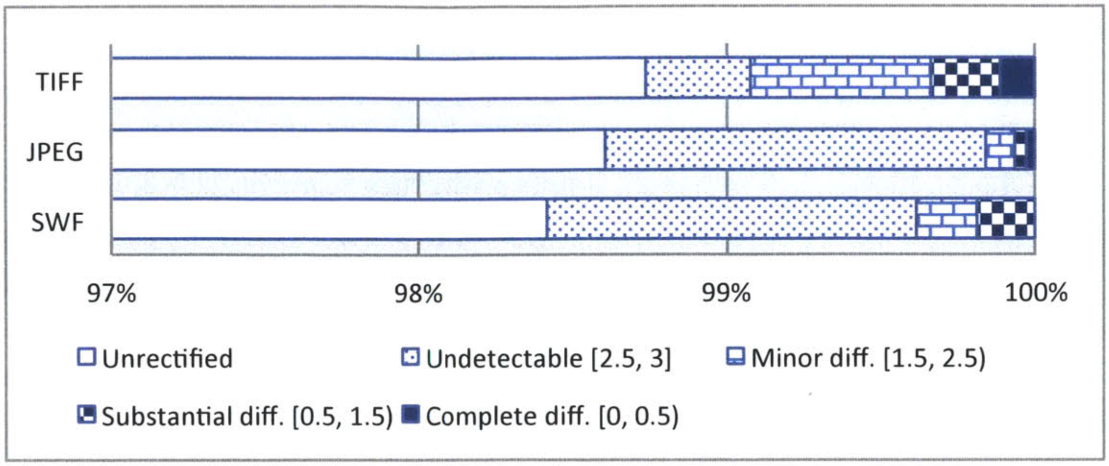


Figure 5-2: Graphical results of Mechanical Turk experiment.

To compare the Mechanical Turk results with the quantitative data loss percentage results on image files (see Section 4.3), we compute the correlation coefficient between these two sets of data. The correlation coefficient is -0.84, which indicates that they are significantly correlated ($p < 0.01$). For complex rectification effects, we find that Mechanical Turk workers can provide a more intuitive evaluation than the than quantitative data loss percentage provides. For example, only the image color in Figure 1-3 changes (Mechanical Turk score 1.42), but the quantitative data loss percentage reports simply that all pixels change.

5.4 Rectification Effect Explanation

JPEG

When we compare the original and rectified JPEG files, we observe essentially three outcomes: 1) The rectification changes fields that do not affect the image presented to the user — the original and rectified images appear identical (37 out of 42 inputs with Turk scores in $[2.5, 3.0]$). 2) The rectification truncates part of the picture, removing a strip along the bottom of the picture (3 out of 42 inputs with Turk scores in $[2.0, 2.3]$, see Figure 1-1). 3) The rectification changes the metadata fields of the picture, the pixels wrap around, and the rectified image may have similar colors as the original but with the detail destroyed by the pixel wrap (2 out of 42 inputs with Turk scores in $[0, 1)$, see Figure 1-2).

TIFF

For TIFF files, we observed essentially four outcomes: 1) The rectification changes fields that do not affect the image presented to the user — the original and rectified images appear identical (3 out of 11 inputs with Turk scores in $[2.5, 3.0]$). 2) The rectification truncates the image, removing a strip along the bottom of the picture (5 out of 11 inputs with Turk scores in $[1.0, 2.5]$). 3) The rectification changes the color palette fields so that only the image color changes (2 out of 11 inputs with Turk scores in $[1.5, 2.0]$, see Figure 1-3). 4) The rectification changes metadata fields and all data is lost (1 out of 11 inputs with Turk score 0.2).

SWF

For SWF files, we observed essentially three outcomes: 1) The rectification changes fields that do not affect the video (43 out of 57 inputs with Turk scores in $[2.5, 3.0]$). 2) The rectification changes fields that only affect a single visual object in the flash video such as an embedded image or the background sound, leaving the SWF functionality largely or partially intact (3 out of 57 inputs with Turk scores in $[1.5, 2.5]$). 3) The rectification changes fields that affect the program logic of the flash video so that the rectified flash fails to respond to interactive events from users (11 out of 57 inputs with Turk scores in $[0.5, 2.6]$ depending on how important the affected events are to the users).

Chapter 6

Related Work

Input Rectification

Applying input rectification to improve software reliability and availability was first introduced by Rinard [37], who presented the implementation of a manually crafted input rectifier for the Pine email client. SOAP improves upon the basic concept by automating the fundamental components of the approach: learning and rectification.

Data Diversity

Ammann and Knight [9] propose to improve software reliability using data diversity. Given an input that triggers an error, the goal is to retry with a reexpressed input that avoids the error but generates an equivalent result. Input rectification, in contrast, may change the input (and therefore change the output). The freedom to change the input semantics enables input rectification to nullify a broader class of errors in a broader class of applications (specifically, applications for which equivalent inputs may not be available).

Anomaly Detection

Anomaly detection research has produced a variety of techniques for detecting malicious inputs [38, 26, 39, 29, 22, 34, 40]. Web-based anomaly detection [38, 26] uses input features (e.g. request length and character distributions) from attack-free HTTP traffic to model normal behaviors. HTTP requests that contain features that violate the model are flagged as anomalous and dropped. Similarly, Valeur et al [39] propose a learning-based approach for detecting SQL-injection attacks. Wang et al [40] propose a technique that detects network-based intrusions by examining the character distribution in payloads. Perdisci et al [34] propose a clustering-based anomaly detection technique that learns features from malicious traces (as opposed to benign traces). SOAP differs from anomaly detection techniques in its aim to rectify inputs and to preserve desirable data in inputs, while anomaly detection techniques simply recognize and drop potentially malicious inputs.

Signature Generation

Vigilante [15], Bouncer [14], PacketVaccine [42], VSEF [31], and ShieldGen [17] generate vulnerability signatures from known exploits. SOAP differs from such systems in its ability to nullify unknown vulnerabilities and to enable users to access desirable data in potentially malicious inputs (rather than discarding such inputs).

Critical Input, Code, and Data Inference

Snap [12] can automatically learn which input fields, code, and program data are critical to the acceptability of the output that a given application produces. Other fields, code, and data can sustain significant perturbations without changing the acceptability of the output. SOAP could use this criticality information to minimize or even eliminate changes to critical input fields in the rectification process.

Directed Fuzzing

SOAP uses taint analysis to track input fields that may trigger overflow errors. BuzzFuzz also uses taint tracing to track disparate input bytes that simultaneously reach security critical operations [21]. BuzzFuzz uses this information to perform directed fuzzing on inputs that have complex structures. Like BuzzFuzz, SOAP learns which bytes reach security critical operations. Unlike BuzzFuzz, SOAP also learns and enforces safety constraints over these bytes.

Automatic Patch

Like SOAP, ClearView [35] enforces learned invariants to eliminate errors and vulnerabilities. Specifically, ClearView learns invariants over registers and memory locations, detects critical invariants that are violated when an adversary attempts to exploit a security vulnerability, then generates and installs patches that eliminate the vulnerability by modifying the program state to enforce the invariants.

Rectification Algorithm

The SOAP rectification algorithm is inspired by automated data structure repair [19, 23, 20], which iteratively modifies a data structure to enforce data consistency defined in an abstract model. It is also possible to use data structure repair to enforce learned data structure consistency properties [18].

Evaluation with Mechanical Turk

By enabling a large-scale, low-cost human workforce, Mechanical Turk has become a viable option for a variety of experimental tasks such as training data annotation [28], computation result evaluation [25], and behavior research [36].

Chapter 7

Conclusion

Standard anomaly detection techniques detect potentially malicious inputs for an application and drop these inputs. Therefore these techniques deny the user to access desirable data in atypical benign inputs and malicious inputs. Automatic input rectification is an alternative technique, which changes atypical inputs back to the application comfort zone.

The combination of quantitative evaluation and human qualitative evaluation indicates that automatic input rectification can effectively nullify errors in applications while preserving much, and in many cases, all, of the desirable data in complex input files.

Bibliography

[1] Amazon mechanical turk. `https://www.mturk.com/mturk/welcome`.

[2] Common vulnerabilities and exposures (CVE). `http://cve.mitre.org/`.

[3] Dillo. `http://www.dillo.org/`.

[4] Hachoir. `http://bitbucket.org/haypo/hachoir/wiki/Home`.

[5] Imagemagick. `http://www.imagemagick.org/script/index.php`.

[6] Picasa. `http://picasa.google.com/`.

[7] Swfdec. `http://swfdec.freedesktop.org/wiki/`.

[8] VLC media player. `http://www.videolan.org/`.

[9] Paul E Ammann and John C Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[10] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10. USENIX Association, 2010.

[11] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

[12] Michael Carbin and Martin C. Rinard. Automatically identifying critical input regions and code in applications. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, 2010.

[13] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07. ACM, 2007.

[14] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07. ACM, 2007.

[15] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05. ACM, 2005.

[16] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irún-Briz. Tupni: automatic reverse engineering of input formats. In *ACM Conference on Computer and Communications Security*, 2008.

[17] Weidong Cui, Marcus Peinado, and Helen J. Wang. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of 2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007.

[18] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 233–243, Portland, ME, USA, July 18–20, 2006.

[19] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05. ACM, 2005.

[20] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08. ACM, 2008.

[21] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[22] Debin Gao, Michael K. Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04. USENIX Association, 2004.

[23] Ishtiaque Hussain and Christoph Csallner. Dynamic symbolic data structure repair. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10. ACM, 2010.

[24] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[25] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with Mechanical Turk. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08. ACM, 2008.

[26] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03. ACM, 2003.

[27] Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering.* ACM, 2008.

[28] M. Marge, S. Banerjee, and A.I. Rudnicky. Using the amazon mechanical turk for transcription of spoken language. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, 2010.

[29] Darren Mutz, Fredrik Valeur, Christopher Kruegel, and Giovanni Vigna. Anomalous system call detection. *ACM Transactions on Information and System Security*, 9, 2006.

[30] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07. ACM, 2007.

[31] James Newsome, David Brumley, and Dawn Xiaodong Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.

[32] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2005.

[33] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06. ACM, 2006.

[34] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10. USENIX Association, 2010.

[35] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[36] David G. Rand. The promise of Mechanical Turk: How online labor markets can help theorists run behavioral experiments. *Journal of Theoretical Biology*, 2011.

[37] Martin C. Rinard. Living in the comfort zone. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07. ACM, 2007.

[38] William Robertson, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the 13 th Symposium on Network and Distributed System Security (NDSS)*, 2006.

[39] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A learning-based approach to the detection of sql attacks. In *DIMVA 2005*, 2005.

[40] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, 2004.

[41] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In

Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10), 2010.

[42] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: black-box exploit detection and signature generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06. ACM, 2006.

[43] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.