MIT-CSAIL-TR-2011-044 October 3, 2011

Automatic Input Rectification

Fan Long, Vijay Ganesh, Micheal Carbin, Stelios Sidiroglou, and Martin Rinard

Automatic Input Rectification

Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard
MIT CSAIL
{*fanl, vganesh, mcarbin, stelios, rinard*}*@csail.mit.edu*

***Abstract*—We present a novel technique, *automatic input rectification*, and a prototype implementation called 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 is 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 (namely, 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.

I. 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 standard 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 atypical malicious inputs, thereby denying the user access to useful inputs.

A. 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 behavior for benign atypical 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* [32] 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 b) preserving the ability of the user to access desirable data in atypical inputs (either benign or malicious).

We present SOAP (Sanitization Of Anomalous inPuts), an automatic input rectification system designed to prevent *overflow vulnerabilities* and other memory addressing errors. SOAP first learns a set of constraints over typical inputs that characterize a comfort zone for the application that processes those inputs. It can then take the constraints and automatically generate 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.

B. 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 yet 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 videos and web pages with embedded malicious content. Rectification may eliminate the exploits while preserving much of the 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:** We note that rectification can offer similar advantages for atypical inputs that may not contain security exploits but nevertheless expose an error in the application that prevents the application from successfully processing the input.

C. *The Input Rectification Technique*

SOAP operates on the parse tree of an input, which divides the input into a collection of 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) 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) field length indicator constraints that capture relationships between the values of integer fields and the lengths of 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. SOAP uses dynamic taint analysis [10], [28], [17] to identify those input fields that are related to critical operations during the execution of the application such as 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 rectifier automatically modifies input fields iteratively until all of the constraints are satisfied.

D. *Key Questions*

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 non-malicious or 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 the rectifier has modified to conform to the constraints?
- **Security:** Does the rectifier effectively protect the application against inputs that exploit errors and security 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, JPG), wave sound (WAV) and Shockwave flash video (SWF). We evaluate the effectiveness of our rectifier by performing the following experiments:

- **Input Acquisition:** For each application, we acquire a set of inputs from the Internet.
- **Benign Input Acquisition:** 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 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 in the applications.
- **Learning:** We use the training set to automatically learn the set of constraints that characterize the comfort zone of the application.
- **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*. For our set of applications, the percentage of atypical benign inputs ranges from 0% to 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 average 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 manually identify the root cause of the vulnerability that the malicious input exploited. We then examined the set of learned constraints and verified that if an input satisfies the constraints, then it will not be able to exploit the vulnerabilities.

E. *Understanding Rectification Effects*

We examined the original and rectified images or videos for all test input files that the rectifier modified. All of these files are available at:

https://sites.google.com/site/inputrectification/home

For the majority of rectified inputs (83 out of 110 inputs), the original and rectified images or videos appear identical. The average Mechanical Turk rating for such images or videos was between 2.5 and 3.0. We attribute this phenomenon 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 parses them may be less extensively tested and therefore more likely to contain errors.

For some of the rectified image inputs (8 of 53 image inputs), the rectifier truncates part of the image, leaving a strip along the bottom of the picture. For the remaining inputs (16 of 110), the rectifier changes fields that control various aspects of core application functionality (for example, the color of the image, the alignment between pixels and the image size, or interactive aspects of videos). The average Mechanical Turk rating for such images or videos varied depending on the severity of the effect. For some of these 16 inputs the delivered data/functionality remained essentially intact; in all cases the application was able to successfully process the rectified inputs without error to present the remaining data to the user.

F. Contributions

We make the following contributions:

- **Basic Concept:** We propose a novel technique for dealing with anomalous and potentially malicious inputs, namely, automatic input rectification, and an 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.
- **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.
- **Experimental Results:** We use Amazon Mechanical Turk [1] to evaluate the subjective perceptual quality of the outputs for rectified inputs. Our results indicate that Mechanical Turk workers perceive rectified images and videos to be, in most cases, close or even identical to the original images and videos.
  These results are consistent with our own qualitative and quantitative evaluation of the differences between the original and rectified images and videos.
- **Explanation:** We explain (Sections I-E and V) why rectification often preserves much or even all of the desirable data in rectified files.

For our set of benchmark applications, rectification can preserve much, and in many cases all, of the desirable data in the original input file. We note that our own qualitative analysis of the differences between original and rectified images and videos correlates closely with the evaluation of Mechanical Turk workers (see Section V), and with the quantitative data loss analysis we present in the experimental section (see Section IV).

We organize the rest of the paper as follows. Section II gives an overview of SOAP with a motivating example. We describe the technical design of SOAP in Section III. We present quantitative evaluation of SOAP in Section IV and subjective human evaluation of SOAP in Section V. Section VI discusses related work. We finally conclude in Section VII.

```
1   //Dillo's libpng callback
2   static void
3   Png_datainfo_callback(png_structp png_ptr, ...)
4   {
5     DilloPng *png;
6     ...
7     png = png_get_progressive_ptr(png_ptr);
8     ...
9     /* check max image size */
10    if (abs(png→width*png→height) >
11      IMAGE_MAX_W * IMAGE_MAX_H) {
12      ...
13      Png_error_handling(png_ptr, "Aborting...");
14      ...
15    }
16    ...
17    png→rowbytes = png_get_rowbytes(png_ptr, info_ptr);
18    ...
19    png→image_data = (uchar_t *) dMalloc(
20      png→rowbytes * png→height);
21    ...
22  }
```

Figure 1. The code snippet of Dillo libpng callback (png.c). Highlighted code is the root cause of the overflow bug.

II. Example and Overview

Figure 1 presents the source code from Dillo 2.1, a lightweight open source web browser. Dillo uses libpng to process PNG files. The libpng callback function *Png_datainfo_callback()* shown in Figure 1 is called when Dillo starts to load a PNG file. The function contains an integer overflow bug 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 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, their bound 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. To nullify the above Dillo error, SOAP performs following steps:

- **Understand Input Format:** SOAP first parses a PNG image file into a collection of input fields shown as Figure 3, so that SOAP knows which input bytes in the PNG image file correspond to the image width and height in the above example.
- **Identify Critical Fields:** SOAP monitors the execution of Dillo to determine that values in the image width and height fields flow into the variables *png→width* and *png→height*. These two variables influence a memory allocation statement at lines 19-20. Thus SOAP marks width and height in PNG images as critical fields, which can potentially cause dangerous overflow.
- **Infer Constraints:** SOAP next infers constraints over the critical fields. Specifically, SOAP processes the benign training PNG images to use the maximum image width and height values that appear in these inputs as their upper bounds. SOAP also infers correlated relations where an integer field indicates the length of other data fields.

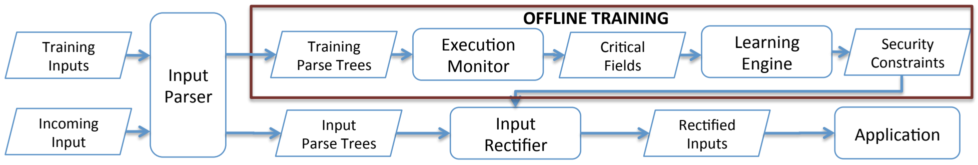


Figure 2. The architecture of automatic input rectification system.

Figure 4 presents examples of constraints for PNG images.

- **Rectify Atypical Inputs:** When it encounters an atypical input whose width or length fields are larger than the inferred bound, SOAP enforces the bound by changing the field to the inferred bound. Note that such changes may, in turn, cause other 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 until all constraints are satisfied.

Both critical field identification and constraint inference are done offline. Once SOAP generates safety constraints for the PNG format, it can automatically rectify new incoming PNG images.

III. 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 2). 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 processing by the remaining components.
- **Execution Monitor:** The execution monitor uses taint tracing to analyze the execution traces of an application. It ***identifies critical input fields*** that influence sensitive operations including memory allocations and memory writes.
- **Learning Engine:** The learning engine starts with a set of benign training inputs. It ***infers safety constraints*** based on the values of the fields 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 rectification algorithm modifies the input iteratively until it satisfies all constraints.

A. Input Parser

As shown in Figure 2, 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. Each leaf field has a type, which can be integer, string or raw bytes. The specification contains low-level rules that SOAP uses to parse values in the input file. These rules describe, for example, how the input file encodes these values.

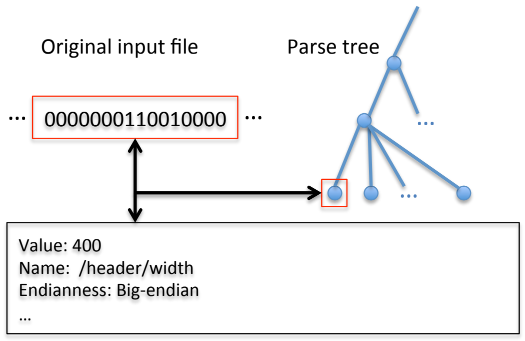


Figure 3. An example of syntax parse tree.

Figure 3 presents an example of a 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 input rectifier uses this information in the descriptor when modifying the field.

B. Execution Monitor

The execution monitor is responsible for identifying the critical input fields that are involved in the learned constraints. 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.

The SOAP execution monitor uses dynamic taint analysis [10], [28] 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. 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 inputs that 1) minimize the execution time required to find the integer fields and 2) together cover all of the integer fields that may appear in the input files.

```
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 4. A subset of constraints generated by SOAP for PNG image files.

C. Learning Engine

The learning engine works with the parse trees of the training inputs and the specification of critical fields as identified by the execution monitor. It uses this information to infer safety constraints over critical fields (see offline training box in Figure 2).

Safety Constraints: Overflow bugs are typically exploited by large data fields, extreme values, negative entries or inconsistencies of multiple fields. SOAP infers two types of safety constraints, bound constraints and length indicator constraints. Bound constraints are associated with individual fields, which bound integer values and sizes of data fields in incoming inputs. Length indicator constraints (i.e., an integer field that indicates the actual length of a data field) are correlated constraints associated with multiple fields.

Figure 4 presents several examples of constraints that SOAP infers for PNG image files. Specifically, SOAP infers upper bounds of integer fields (line 1), non-negativity of integer fields (line 2), upper bounds of lengths of data fields (line 3), and length indicator constraints between values and lengths of parse tree fields (lines 4-5 in Figure 4).

These 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 our target vulnerabilities in the protected programs.

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

Inferring Bound Constraints: SOAP infers three kinds of bound constraints: upper bounds of lengths of data fields, upper bounds of integer fields, and whether integer fields are non-negative. SOAP sets the maximum length of a data field that appeared in training inputs as the upper bound of its length. SOAP sets the maximum value of an integer field in training inputs as the upper bound of its value. SOAP also sets an integer field to be non-negative if it is never negative in all training inputs. SOAP infers all these constraints with a single traversal of the parse tree of each training input.

Inferring Length Indicators: Inferring length indicator constraints is challenging, because these constraints can have various forms. For example, an integer field may either indicate the length of the next data field or the total length of a data chunk composed of multiple fields, depending on the input specification.

```
1  // Initialization
2  FOR f IN all integer fields:
3    R[f] ← ∅
4    p ← f.parent
5    FOR S IN subsets of consecutive children of p:
6      FOR scale IN {1, 8}:
7        relation.fields ← S
8        relation.scale ← scale
9        R[f].add(relation)
10     END FOR
11   END FOR
12 END FOR
13
14 // Checking relations against each parse tree
15 FOR parse_tree IN training_input_set:
16   FOR f IN parse_tree.integer_fields:
17     FOR relation IN R[f]:
18       sum ← 0
19       FOR data_field IN relation.fields:
20         sum ← sum + sizeinbit(data_field)
21       END FOR
22       IF f.value * relation.scale ≠ sum:
23         R[f].erase(relation)
24       END IF
25     END FOR
26   END FOR
27 END FOR
```

Figure 5. Inference algorithm for correlated constraints.

Figure 5 shows the pseudo-code of the inference algorithm for length indicator constraints in SOAP. This algorithm first assumes that all possible length indicator constraints are true. When processing each training input, the algorithm eliminates constraints that do not hold in this input. It can be extended to infer other kinds of correlated constraints.

SOAP infers a length indicator field f which is associated with the total length of consecutive children of the parent field of f. For instance, lines 4-5 in Figure 4 present a length indicator constraint. 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 “`/text/size`” is the total length of “`/text/keyword`” and “`/text/text`”, which are two consecutive children of “`/text`”.

The pseudo-code in Figure 5 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.

D. Input Rectifier

Given safety constraints generated by the learning engine and a new input, the input rectifier rectifies the input if it violates safety constraints (see Figure 2). The main challenge

[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.

```
1  REPEAT
2    violated $\leftarrow$ false;
3    FOR f IN input.integer_fields:
4      // Checking against upper bounds
5      IF f.value $>$ upbound[f]:
6        f.value $\leftarrow$ upbound[f]
7        violated $\leftarrow$ true
8      END IF
9      // Checking against non-negativeness
10     IF f.value $< 0$ and f $\notin$ may_neg:
11       f.value $\leftarrow 0$
12       violated $\leftarrow$ true
13     END IF
14   END FOR
15
16   FOR f IN input.data_fields:
17     // Checking against length upper bounds
18     IF f.size $>$ upbound[data_field.name]:
19       truncate f to size upbound[data_field.name];
20       violated $\leftarrow$ true
21     END IF
22   END FOR
23
24   FOR f IN input.integer_fields:
25     FOR rel IN R[f]:
26       // Checking against length indicator constraints
27       IF f.value * rel.scale $>$ sizeinbit(rel.fields):
28         f.value $\leftarrow$ sizeinbit(rel.fields) / rel.scale
29         violated $\leftarrow$ true
30       ELSE IF f.value * rel.scale $<$ sizeinbit(rel.fields):
31         truncate rel.fields to size f.value * rel.scale
32         violated $\leftarrow$ true
33       END IF
34     END FOR
35   END FOR
36 UNTIL violated = false
```

Figure 6. The rectification algorithm in SOAP.

in designing the input rectifier is enforcing safety constraints while preserving as much useful data as possible.

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 finds a single violated constraint and applies a minimum modification or truncation to satisfy the violated constraint. It repeats this process until there are no more violated constraints.

Figure 6 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 have negative values. R stores length indicator constraints (see Section III-C).

The main loop iteratively checks the input against learned constraints. At each iteration, it selects and rectifies the violated constraints. The main loop exits when the input no longer violates any safety constraints.

- **Upper bounds of integer fields:** At lines 4-8, the algorithm rectifies a violated upper bound constraint of an integer field by changing the value of the field back to its learned upper bound.
- **Non-negativities of integer fields:** At lines 9-13, the algorithm changes the value of an integer field to 0, if the input violates the non-negative constraint of the integer field.
- **Length upper bounds of data fields:** At lines 16-22, the algorithm truncates a data field to its length upper bound, if the input violates the length upper bound constraint of the data field.
- **Length indicator constraints:** At lines 27-29, the algorithm changes the value of the length indicator field to the actual length of the data field, if the value is greater than the actual length. At lines 30-32, the algorithm truncates the data fields to the length indicated by the corresponding integer field, if the data is longer than the indicated length. Note that the length indicator constraints may be violated due to previous fixes for other constraints. Our algorithm cannot increase the value of the length indicator field or increase the length of the data field here, which will roll back previous fixes.

Note that, because the absolute value of at least one integer field or data field length always decreases at each iteration, this algorithm will always terminate. Note also that, because the algorithm truncates a minimum amount of data, the algorithm attempts to minimize the total amount of discarded data.

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. It is also possible to use an more automatic checksum repair technique [38].

E. Implementation

The SOAP learning engine and input rectifier are implemented in Python. The execution monitor is implemented in C based on Valgrind [27], 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 that allow to support other input formats.

IV. 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 inputs affect SOAP's ability to preserve desirable data?

Applications and Errors: We use SOAP to rectify inputs for five applications: Swfdec 0.5.5 (a shockwave player) [7], Dillo 2.1 (a lightweight 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].

Figure 7 presents a description of each error in each application. In sum, all of these applications consume inputs that (if specifically crafted) may cause the applications to incorrectly allocate memory or perform an invalid memory access. The input file formats for these errors are the SWF Shockwave Flash

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

Figure 7. Six errors used in our experiment. SOAP successfully nullifies all of these errors (see Section IV-A). “Source” is the source where we collect this bug. “Fault” and “Format” present the fault type and the format of malicious inputs that can trigger this error. “Position” indicates the source code file and/or line positions that are related to the root cause. “Related constraints” presents constraints generated by SOAP that can help to avoid this bug.

	Rectification Statistics						Running Time			
Inp.	App.	Train	Test	Field (Distinct)	Rectified	Avg. P_{loss}	Mean	Parse	Rect.	Per field
SWF	Swfdec	3620	3620	5550.2 (98.17)	57 (1.57%)	N/A	531ms	443ms	88ms	0.096ms
PNG	Dillo	1496	1497	306.8 (32.3)	0 (0%)	0%	23ms	19ms	4ms	0.075ms
JPEG	IMK, Picasa	3025	3024	298.2 (75.5)	42 (1.39%)	0.08%	24ms	21ms	3ms	0.080ms
TIFF	IMK, Picasa	870	872	333.5 (84.5)	11 (1.26%)	0.50%	31ms	26ms	5ms	0.093ms
WAV	VLC	5488	5488	17.1 (16.8)	11 (0.20%)	0%	1.5ms	1.3ms	0.2ms	0.088ms

Figure 8. The benchmarks and numerical results of our experiments. “Inp.” and “App.” columns show input file formats and applications. “Train” and “Test” indicate the number of inputs used for training and testing respectively in our experiment. “Field (Distinct)” column is in the form of X(Y), where X indicates the average number of fields in one test input of each format and Y indicates the average number of semantically distinct fields (i.e. fields that have different names in their descriptors) in one test input. The ”Rectified” column contains 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 “Avg. P_{loss}” column presents the average data loss percentage of all test inputs of each format (see Section IV-B). “Mean” is the average running time for one test input including both parsing and rectification. “Parse” indicates the average parsing time for one input. “Rect.” indicates the average rectification time for one input. “Per field” indicates the average running time amortized to a single field for each format. “IMK” is an abbreviation of ImageMagick.

format; the PNG, JPG, and TIF image formats; and the WAV sound format.

Malicious inputs: We obtained six input files, each of which targets a distinct error (see Figure 7) in at least one of these applications [17], [38], [2]. We obtained three of these inputs from the CVE database [2], two from the example inputs of the Buzzfuzz project [17], and one from the example inputs of the TaintScope project [38].

Benign inputs: We implemented a web crawler to collect input files for each format (see Figure 8 for the number of collected inputs for each input format). 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 processed these inputs. For each format, we randomly partitioned these inputs into two sets, the training set and the test set (see Figure 8).

A. Nullifying Vulnerabilities

We next evaluate the effectiveness of SOAP in nullifying six vulnerabilities in the benchmark applications (see Figure 7). We first applied the trained SOAP rectifier to the obtained malicious inputs. The rectifier detected that all of these inputs violated at least one safety constraint. It rectified all violated constraints to produce six corresponding rectified inputs. We verified that the applications processed the rectified inputs without error and none of the rectified inputs exploited the vulnerabilities. We next discuss the interactions between the inputs and the root cause of each vulnerability.

Flash video: The root cause of the X11 crash error in Swfdec is a failure to check for large Swfdec 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 that */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 bug 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 sufficiently large specified width and height values, this calculation will overflow and Swfdec will allocate a buffer significantly smaller than the required size. When it enforces the learned safety constraints, SOAP nullifies the error by limiting the size of the embedded image. No rectified input will be able to exploit this error.

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 Section II). The same malicious JPEG and TIFF images can exploit vulnerabilities in both ImageMagick and Picasa Photo Viewer. 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 [38]. By enforcing the safety constraints, SOAP limits the size of input images and nullifies these vulnerabilities.

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 the overflow vulnerability. By enforcing the constraint */format/size* $\leq$ *150*, SOAP limits the size of the format chunk in WAV file and nullifies this vulnerability.

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

B. Data Loss

We next compute a quantitative measure of the effect of rectification on data loss. For each input format, we first apply the SOAP rectifier to the test inputs. We report the average data loss percentage of all test inputs for each format. We use the following formula to compute the data loss percentage of each rectified input:

$$P_{loss} = \frac{D_{loss_i}}{D_{tot_i}}$$

D_{tot_i} measures the amount of desirable data before rectification and D_{loss_i} measures the amount of desirable data lost in the rectification process. For JPG, TIFF and PNG files, D_{tot_i} is the number of pixels in the image and D_{loss_i} is the number of pixels that change after rectification. For WAV files, D_{tot_i} is the number of frames in the sound file and D_{loss_i} 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 for these files.

Result Interpretation: Figure 8 presents rectification results of the test inputs of each input format. First, note that the vast majority of the test inputs satisfy all of the learned 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 average desirable data loss is less than 0.5%.

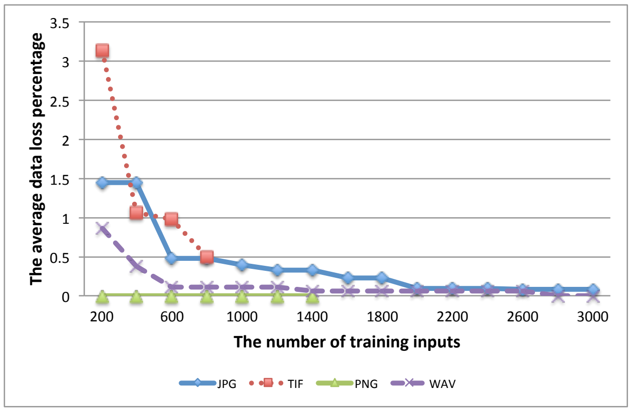


Figure 9. The average data loss percentage curves under different sizes of training input sets for JPEG, TIFF, WAV and PNG (see Section IV-C). X-axis indicates the size of training input sets. Y-axis shows the corresponding average data loss percentage.

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.

C. Size of Training Input Set

We next investigate how the size of the training input set affects the effectiveness of the rectification. Intuitively, we expect that using less training inputs will produce more restrictive constraints which, in turn, 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 size of training input by 200. Figure 9 presents the curves of the average data loss percentage of the test inputs of the different formats as the sizes of the training input sets change.

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 JPG curves. We attribute this phenomenon to the fact that the PNG and WAV formats are simpler than the TIFF and JPG formats (see Figure 8 for the number of semantically distinct fields of each format).

D. Overhead

We next evaluate the overhead introduced by SOAP. Figure 8 presents the average running time of the SOAP rectifier for processing the test inputs of each file format. All times are measured on an Intel 3.33GHz 6-core machine with SOAP running on only one core.

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 file formats (so SWF files take longer to parse because they have significantly more fields than other kinds of files). We believe that users will find these rectification overheads negligible if not imperceptible during interactive use.

V. Mechanical Turk-Based Evaluation

Amazon Mechanical Turk [1] is a Web-base labor market. Requesters post Human Intelligence Tasks (HITs); workers solve those HITs in return for a small payment. We organized the experiment as follows:

- **Input Files:** We collected all of the TIFF, JPG, and SWF test input files that the rectifier modified.[1]
- **HIT Organization:** Together, the TIFF and JPG 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 construct HITs for the video files.
- **HIT Copies:** We publish 100 copies of each HIT on Mechanical Turk. Each copy has a different random order of the pairs. We waited until all of the copies of the HITs were completed by Mechanical Turk workers. Each Mechanical Turk worker rates each pair in the HIT on a scale from 0 to 3. A rating of 3 indicates no visible difference between the images (or videos) in a given HIT, 2 indicates only minor visible differences, 1 indicates a substantial visible difference, and 0 indicates that the two images (or videos) appear completely different.

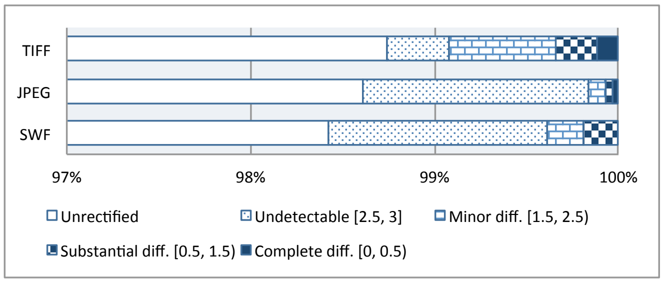
Some Mechanical Turk workers attempt to game the system, for example by using bots to perform the HITs or simply by providing arbitrary answers to HITs without attempting to actually perform the evaluation [21]. We used several mechanisms to recognize and discard results from such workers:

- **Previous Acceptance:** Amazon rates each Mechanical Turk worker, and provides this information to the requestor of HITs. This rating indicates what percentage of that worker's previously performed HITs were accepted by other requestors as valid. We required prospective Mechanical Turk workers have an acceptance rate of at least 95%.
- **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 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.

[1] We exclude PNG and WAV files because the original and rectified files have no user-visible differences.

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

(a) Numeric results



(b) Visualized results

Figure 10. Results of Mechanical Turk experiment. "Unrectified" means those files that are not changed by the rectifier. "Undetectable", "Minor" , "Substantial" and "Complete" correspond respectively to rectified files whose average scores are in $[2.5, 3]$, $[1.5, 2.5)$, $[0.5, 1.5)$ and $[0, 0.5)$.

- **Descriptions:** For each HIT, we asked the worker to provide a short English description of the differences, if any, between the images in each pair. The rationale is that legitimate workers are likely to provide reasonable descriptions, while workers who are attempting to game the system or bots are likely to provide non-sensical descriptions. We manually evaluated these descriptions to discard results from workers with non-sensical descriptions.

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

Results: For each HIT h, we computed the average score over all the scores given by the workers assigned to h. We then classified the rectified file in h into the following categories, corresponding to their average scores, $[2.5, 3]$, $[1.5, 2.5)$, $[0.5, 1.5)$, and $[0, 0.5)$. We counted the first category as the set of rectified files with undetectable changes, and the remaining three categories contain rectified files that have minor differences, substantial differences and compete differences, in that order.

Figure 10(a) presents, for each combination of input file format and Mechanical Turk 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. Note that, out of all of the test inputs, only two exhibit a complete difference after rectification, and only 12 exhibit more than a minor difference.

We compare the Mechanical Turk results with the quantitative data loss percentage results on image files as given in Section IV-B by computing the correlation coefficient between these two sets of data. The correlation coefficient is -0.84, which indicates that they are significantly correlated ($p < 0.01$).

Causes of Rectification Effects: When we compared the original and rectified JPEG files, we observed 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). The average Mechanical Turk rating for such images is between 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, see Figure 11). (3) The rectification changes the metadata fields of the picture, the pixels wrap around, and the rectified image appears to have similar colors as the original but with the detail destroyed by the pixel wrap (2 out of 42 inputs, see Figure 12). The average Mechanical Turk rating for such images is less than 1.

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). The average Mechanical Turk rating for such images is between 2.5-3.0. (2) The rectification truncates part of the picture, removing a strip along the bottom of the picture (5 out of 11 inputs). The average Mechanical Turk rating for such images is between 1.0-2.5, depending on how much of the image was truncated. (3) The rectification changes the color palette fields so that only the color of the image changes (2 out of 11 inputs, see Figure 13). The average Mechanical Turk rating for such images is between 1.5-2.0. (4) The rectification changes metadata fields and all image data is lost (1 out of 11 inputs). The average Mechanical Turk rating for this image is 0.2.

For SWF files, we observed essentially three possible outcomes: (1) The rectification changes fields that do not affect the video (43 out of 57 inputs). For such videos the average Mechanical Turk score is between 2.5-3. (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). For such videos the average Mechanical Turk score is between 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). For such videos the average Mechanical Turk score is between 0.5-2.6, depending on how important the affected events are to the users.

VI. Related Work

Input Sanitization: Applying input sanitization to improve software reliability and availability was first introduced by Rinard [32]. That work describes 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.

Anomaly Detection: SOAP infers safety constraints from training inputs to detect malicious inputs. Detecting malicious inputs has a rich body work in the field of anomaly detection [33], [22], [36], [26], [18], [30], [37].

Web-based anomaly detection [33], [22] uses input features (e.g. request length and character distributions) from attack-free HTTP traffic to model normal behavior. HTTP requests that contain features that violate the model, are flagged as anomalous and dropped. In the same vein, Valeur et al [36] propose a learning-based approach for detecting SQL-injection attacks. Wang et al [37] propose a technique that detects network-based intrusions by examining the character distribution in payloads, bypassing the need to select specific input features. Perdisci et al [30] propose a clustering-based anomaly detection technique that learns features from malicious traces (as opposed to attack-free).

The anomaly detection techniques described above focus on server-side network inputs. SOAP focuses on human-facing client applications that use complex structured inputs such as images or videos. These inputs typically contain correlated (and sometimes lazily parsed) fields that are not handled by the techniques described above. Additionally, SOAP provides the ability to automatically rectify anomalous inputs. Input rectification helps deal with the problem of false positives which typically plagues anomaly detection systems.

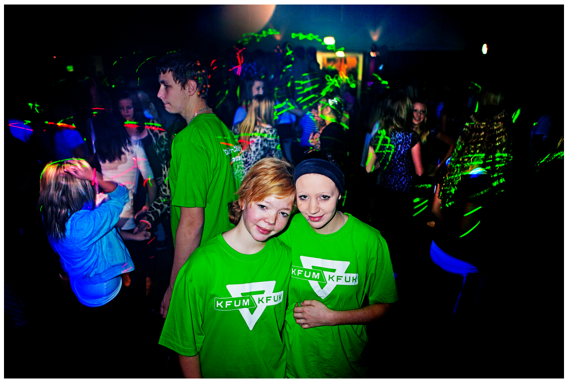
Signature Generation: More closely related to input sanitization are systems that deal with vulnerability signature generation. Systems such as Vigilante [12], Bouncer [11], PacketVaccine [39] and ShieldGen [14]. ShieldGen [14] are closely related to SOAP due to their use of input-format specifications. SOAP improves upon such systems with the ability to detect unknown vulnerabilities and the ability to fix the flagged input.

Critical Field Inference: SOAP uses taint analysis to track those input fields that can possibly trigger overflow. Another tool that uses taint tracing to track disparate input bytes that simultaneously reach security sensitive operations is BuzzFuzz [17]. BuzzFuzz uses this information to perform directed fuzzing on inputs that have complex structures. By contrast, SOAP not only learns the important bytes that reach security sensitive operations but also learns arithmetic constraints over them.

Input Syntax: SOAP requires a parser component to interpret input syntax structure according to its specification. To facilitate parser construction, Binpac [29] describes a declarative language for writing application protocol parsers. Input syntax inference [13], [23], [9], [40] is also an active research topic that is actively studied by security and software engineering community.

Rectification Algorithm: SOAP introduces a rectification algorithm to enforce safety constraints on incoming inputs. A similar approach has been proposed in the context automated data structure repair [15], [20], [16]. The rectification algorithm used in SOAP is inspired by the data structure repair algorithm proposed by Demsky et al [15], which iteratively modifies a data structure to enforce data consistency defined in an abstract model. SOAP improves upon data structure repair by combining input rectification and constraint inference (with a focus on correlated constraints) and a end-to-end automation of the system.

Evaluation with Mechanical Turk: Mechanical Turk is used to evaluate the quality of service of our input rectification technique. By enabling a large-scale, low-cost human computation workforce, Mechanical Turk has become a viable option for many experimental tasks such as training data


(a) The original image

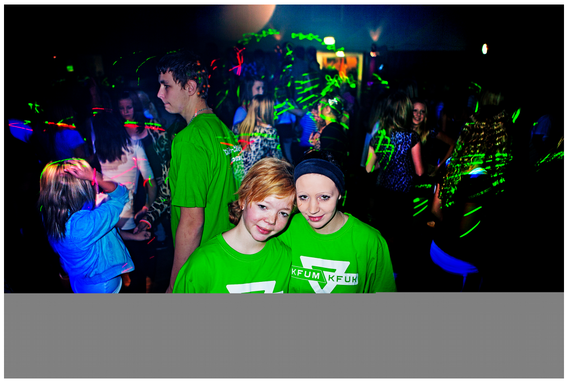
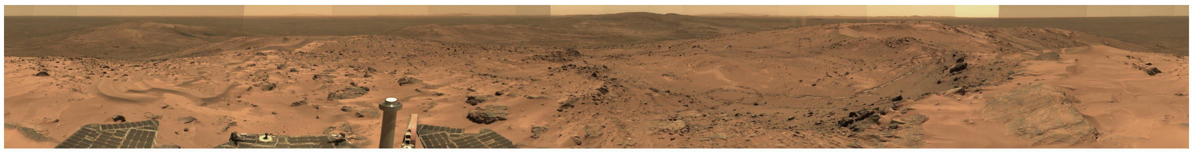

(b) The rectified image

Figure 11. An example image truncated by the rectification.


(a) The original image

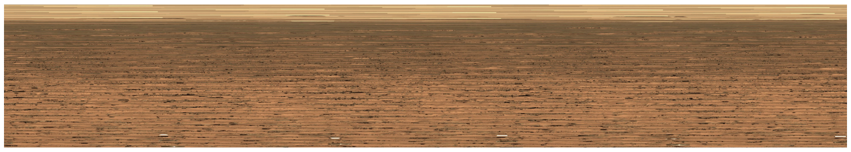
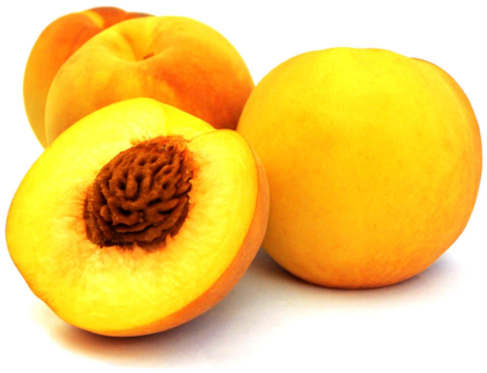

(b) The rectified image

Figure 12. An example image twisted by the rectification


(a) The original image

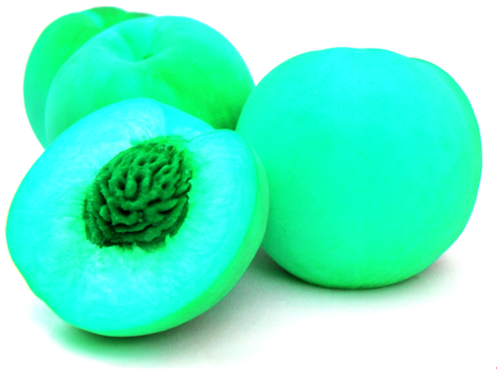

(b) The rectified image

Figure 13. An example image whose color is changed by the rectification.

annotation [35], [24], computation result evaluation [19], [34], [21], and behavior research [25], [31].

VII. Conclusion

Our results indicate that input rectification can effectively nullify errors in applications while preserving much, and in many cases, all, of the desirable data in complex input files.

References

[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] J. Caballero, H. Yin, Z. Liang, and D. 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.

[10] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07. ACM, 2007.

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

[12] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05. ACM, 2005.

[13] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irún-Briz. Tupni: automatic reverse engineering of input formats. In *ACM Conference on Computer and Communications Security*, 2008.

[14] W. Cui, M. Peinado, and H. 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.

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

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

[17] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[18] D. Gao, M. K. Reiter, and D. 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.

[19] J. Heer and M. Bostock. Crowdsourcing graphical perception: using mechanical turk to assess visualization design. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10. ACM, 2010.

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

[21] A. Kittur, E. H. Chi, and B. 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.

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

[23] Z. Lin and X. 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.

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

[25] W. Mason and S. Suri. A Guide to Conducting Behavioral Research on Amazon's Mechanical Turk. *Social Science Research Network Working Paper Series*, 2010.

[26] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous system call detection. *ACM Transactions on Information and System Security*, 9, 2006.

[27] N. Nethercote and J. 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.

[28] J. Newsome and D. 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.

[29] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06. ACM, 2006.

[30] R. Perdisci, W. Lee, and N. 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.

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

[32] M. 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.

[33] W. Robertson, G. Vigna, C. Kruegel, and R. 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.

[34] M. Sanderson, M. L. Paramita, P. Clough, and E. Kanoulas. Do user preferences and evaluation measures line up? In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10. ACM, 2010.

[35] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08. Association for Computational Linguistics, 2008.

[36] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. In *DIMVA 2005*, 2005.

[37] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, 2004.

[38] T. Wang, T. Wei, G. Gu, and W. 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.

[39] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. 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.

[40] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.