Living In The Comfort Zone

Martin Rinard

Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 021139
rinard@csail.mit.edu

Abstract

A comfort zone is a tested region of a system's input space within which it has been observed to behave acceptably. To keep systems operating within their comfort zones, we advocate the interposition of *rectifiers* between systems and their input sources. Rectifiers are designed to transform inputs to ensure that they are within the comfort zone before they are presented to the system. Rectifiers enforce a highly constrained input format and, if necessary, discard information to force inputs to conform to this format. Potential benefits of this approach include the elimination of errors and vulnerabilities, the excision of undesirable excess functionality from large, complex systems, and a simplification of the computing environment.

We have developed a rectifier for email messages and used this rectifier to force messages into a specific constrained form. Our results show that this rectifier can successfully produce messages that keep the Pine email client strictly within code previously confirmed (during a small testing and training session) to function acceptably. Our results also show that the rectifier completely eliminates a security vulnerability in the Pine email client. And finally, the rectifier is able to accomplish these goals while still preserving an acceptable amount of information from the original messages.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications;
D.2.3 [*Software Engineering*]: Coding Tools and Techniques;
D.2.5 [*Software Engineering*]: Testing and Debugging;
D.3.3 [*Programming Languages*]: Language Constructs and Features

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

General Terms Design, Languages, Reliability, Security

Keywords Comfort Zone, Acceptability Properties, Repair, Monitoring, Rectification

1. Introduction

By the time a typical software system is deployed, its developers have tested its behavior on a range of inputs and verified that the behavior is acceptable. And in fact, the deployed system usually works well on inputs that are similar to those in the test suite. To exercise an error or vulnerability, one must typically find an anomalous input with some unusual feature that manages to take the system into a poorly tested region of its execution space. Conceptually, the system has a *comfort zone* — a collection of inputs that is similar to those that it has seen before and for which it is almost certain to deliver expected and acceptable behavior.

Ideally, one would never run a system on an input outside its comfort zone. Indeed, the goal of much of the effort that goes into testing and debugging is to maximize the size of the comfort zone by 1) exercising as much of the functionality of the system as possible and 2) eliminating as many unacceptable errors within the exercised part of the system as possible. In this way the developers hope to minimize the likelihood of the system encountering an input outside its comfort zone that causes it to behave unacceptably.

Despite the tremendous effort devoted to testing and debugging, however, it is almost always possible to find inputs that cause the system to behave in unanticipated or unacceptable ways. This problem can become a source of frustration for users who inadvertently stumble across such inputs and are consequently unable to obtain their desired results. The problem can be especially severe for systems (such as web browsers, servers, and email clients) that are placed in the position of having to process arbitrary inputs from potentially untrusted sources — many successful security attacks involve unusual inputs that exercise unanticipated functionality or outright errors hidden within an otherwise successfully operating system [18].

1.1 Our Approach

This paper presents a complementary approach to eliminating unanticipated and unacceptable system behavior. Instead of attempting to make the system behave acceptably on all possible inputs, we instead propose to configure the environment surrounding the system with the goal of ensuring that the system only executes within an appropriately chosen comfort zone. The basic idea is to interpose a *rectifier* (a set of transformations) between the input sources and the software system. Together, these transformations are designed to move all inputs that were originally outside the comfort zone into the comfort zone. Ideally, this approach would eliminate the anomalies that come from processing problematic inputs while preserving the desired behavior of the system.

There are two potential issues associated with this approach. First, the rectifier may destroy information in the original input, with this information destruction causing the system to produce an inappropriate result. Second, the rectifier may fail to completely remove some problematic features of the input, leaving the input presented to the system outside the comfort zone.

We address these issues as follows. In some cases the problematic features of the input may occur in the way the information is formatted or presented in the input rather than in the content of the information itself. In these cases appropriately structured transformations may be able to leave the information completely intact (although the presentation of the information to the user may change). In other cases the problematic features may occur in information that is tangential to the primary purpose of the system. In these cases appropriately structured transformations may be able to preserve the most important information, leaving the system able to deliver most of its desired functionality. And in all cases, we propose to generate reports that users can examine to determine the potential consequences of applying the transformations.

Rectifiers that may sometimes fail to move an input into the comfort zone can leave systems vulnerable to errors or exploits. We propose to ameliorate this problem in two ways. First, we advocate the use of aggressive rectifiers that produce a very narrow, tightly constrained class of inputs. The goal is to enforce broad, sweeping restrictions that rule out all potentially anomalous inputs. Second, we propose to check all inputs for membership in the comfort zone as the program executes. Specifically, we propose to train the system on a set of validated inputs and record the blocks of code that execute during this training process. Any significant deviation from from this set of previously executed blocks is taken as an indication that the input is outside the comfort zone. The appropriate action to take depends on the context in which the system is used — we are able to take any action from terminating the system as soon as it attempts to execute a new block of code to simply recording the new blocks that execute as the system processes the input.

We anticipate that both the transformations and the overall success of the proposed approach will depend on the specific system and the context in which the system is deployed. To explore how well this approach works, we have built a rectifier for the Pine email client [13] and performed a series of experiments that explore how well the rectifier works. Our results show that it is possible, for this application, to build a rectifier that can effectively move even problematic inputs (such as attacks that would otherwise exploit security vulnerabilities) into the comfort zone of the system for safe processing. The transformations are simple to implement and in most cases preserve virtually all of the important information in the original email messages. And when the rectifier does remove information, it generates informative rectification logs that users can quickly examine to determine the changes that the rectifier made.

1.2 Usage Scenarios

We have presented input rectification as a way to avoid unanticipated errors or vulnerabilities in software systems. It is clear, however, that this technique can be productively applied to solve a variety of other problems that arise in using software systems:

- **Functionality Elimination:** Many large systems contain significantly more functionality than any single user or organization will ever use. In some cases this excess functionality may be undesirable — consider, for example, a feature in Microsoft Word that executes scripts when reading in Word files. When Word is used as a document editor in the Outlook email client to prepare forwarded messages, this feature causes Word to automatically execute any scripts that may have been present in the original email sent to the user (a clear security violation) [7]. Note that script execution is an intended feature in Word and arguably made sense in the context in which Word was originally deployed. An appropriately designed rectifier can eliminate this kind of undesirable functionality (for example, by simply eliminating all scripts from Word documents), making it possible to safely use the original system in a new context in spite of the undesirable functionality.

  Systems tend to accrete functionality over time, to the point that too much functionality can become as much or more of a problem as too little functionality. Identifying a narrow comfort zone and using input rectification to enforce that comfort zone can make it possible to effectively use a single existing functionality-laden system in many different contexts and for many different purposes without incurring the risks and complications associated with the excess functionality normally present. This can be a significant improvement over the alternative (developing multiple systems, each tailored for use in its own context).

We note that in many cases nobody understands all of the functionality that a given system may offer. We therefore advocate the use of aggressive rectifiers that look for input patterns that they recognize as safe (and block anything else) rather than less aggressive rectifiers that look for specific input patterns to block (and pass through anything else).

- **Incorrect Inputs:** Most systems contain input correctness checks designed to verify that the input is in a form that the system can successfully process. If an input fails a correctness check, the system typically refuses to process the input. In some cases, however, a user may need the system to process the information in the input even if the input itself is malformed (and therefore fails the correctness check). This can occur if the system that produced the input has an error or is simply somewhat incompatible with the system that should process the input. In this case an appropriately designed rectifier may be able to produce an input that the first system will accept and process.
- **Test-Suite Directed Development:** Much of the code in many systems is present only to handle cases that occur infrequently. This code increases the size, development cost, and complexity of the final system. A sufficiently aggressive rectifier may be able to substantially reduce the number of input cases that the system must process (and therefore eliminate the need for the code that would otherwise be required to handle these cases). In fact, it may be desirable to refuse to code *any* case until there is an input in the test suite that exercises that code — after all, any input that exercises that code will be outside the comfort zone. The result could be a lazy code development process that does not implement any case until there is an input in the test suite that hits that case. And if a new input in the test suite does happen to hit an unimplemented case, the developer would have two options: code up the case, or modify the rectifier to eliminate the features of the input that caused it to hit the unimplemented case.
- **Reduced Testing Burden:** Systems with many features require lots of testing. Eliminating functionality can make it possible to reduce the amount of testing required to develop confidence in the system. For sufficiently narrow comfort zones, it may even be possible for users to perform enough tests to become confident that the system will work for them even though their tests do not come close to exploring the full functionality present in the system.
- **Eliminating Input Correctness Checks:** In practice, input correctness checks can be a problematic source of errors. Because the code that implements these checks is not usually exercised during the normal operation of the system, it may receive less attention during development and testing (with the result that it may contain more errors when the system moves into production). Because the developer can make no assumptions whatsoever about the input, he or she may overlook cases, leaving the system open to exploitation by inputs that target these overlooked cases.

  An appropriately designed rectifier can completely eliminate all of these problems by always producing correctly formed inputs. Moreover, such a rectifier would make it possible to completely eliminate the error checking code altogether. The resulting benefits can include a dramatic simplification of the input processing code and the elimination of errors and vulnerabilities.
- **New Input, Old System:** New versions of systems often produce data with enhanced features that old versions are unable to correctly process. In this case rectifying data produced by the new version may produce an input that the old version can successfully process. Possible motivations for continuing to use the old version include reluctance to abandon a version that is working well, an aversion to paying upgrade costs, or errors in the new version that prevent it from working as well as the old version.
- **Old Input, New System:** The dual problem arises when a new version may be unable to process inputs produced by the old version. In this case the rectifier may be able to produce inputs that the new version can successfully process.

1.3 Why Our Approach Makes Sense

The basic concept behind our approach is to change the environment to work around limitations or inappropriate features of the software system. In theory, this may make little sense. After all, software should be one of the most flexible technologies ever developed — the medium itself is extraordinarily malleable, and changes can be implemented and disseminated at little cost. It may seem that, if there is a problem, it makes more sense to change the software to deal with the environment rather than the other way around.

In practice, however, most deployed software systems are extremely difficult to work with. Even if the source code is available, the sheer size and complexity of most systems make it virtually impossible for anyone except the developers and maintainers to update the system. Moreover, deployed systems vary widely in quality and are often known to contain errors or security vulnerabilities. Nevertheless, users are locked in — if an error or vulnerability shows up that compromises the ability of the system to acceptably satisfy the needs of some of its users, the users typically have little choice except to wait for the developers to update the software and release a new version. This situation can be especially problematic if it takes a long time to develop and release the updates or if the software is no longer maintained (in which case the update may never come through).

Our proposed technique, using rectification to move inputs into the comfort zone of the system, can enable users to make productive use of flawed systems by adapting the environment to eliminate problems that would otherwise make the system difficult or even impossible to work with. As the world becomes ever more full of large and imperfect systems, we believe that these kinds of techniques will play an increasingly important role in keeping our software infrastructure functioning acceptably.

1.4 Large Systems

The primary focus of this paper is on using rectifiers at system boundaries to move external inputs into the comfort zone of a single system. But is of course also possible to use rectifiers within a large system to mediate interactions between the components of the system. In this scenario, each rectifier would interpose itself at some abstraction layer, then rectify information passing through that abstraction layer.

A key question is the abstraction layer and the interposition mechanism. If the components interact by reading and writing files, the rectifier can simply read the original input file, then write the rectified input file. The component then reads the rectified input file instead of the original input file. If components interact via network connections, the rectifier can be structured as an intermediate process that splits the original direct connection into two connections, one of which carries the unrectified input into the rectifier, the other of which carries the rectified input out of the rectifier to the component that reads it.

Other interactions can require more involved interposition mechanisms. If components interact via procedure or system calls, for example, there are a variety of mechanisms that one can use to intercept the calls, especially in the presence of interface structuring techniques such as the Windows Import Address Table [11].

Rectification within a system may be especially appropriate if the system contains poorly understood or overly general components whose original interfaces support much more functionality than the system needs. It may also be a useful way to eliminate poorly documented or poorly understood parts of a component's interface.

1.5 Simplicity in a Complex World

Our existing computing environment is the result of decades of system evolution. The retention of features from the past combined with the systematic and purposeful addition of new features has produced an environment of unprecedented complexity. Indeed, this complexity has reached the point where it can hamper the ability of users to work productively within the environment. Complex environments also support complex ecosystems; our computing environment is unfortunately now so complex that it supports a thriving community of abusers who participate in the environment primarily to exploit other users.

We believe that our approach can help simplify this complex environment. Tightly focused comfort zones can eliminate many of the features that make modern environments so complex. We have so far focused on how this kind of simplicity can help protect users against errors and vulnerabilities. But this simplicity can also reduce the cognitive load of using the system and eliminate many of the distracting and extraneous details that prevent users from focusing fully on the task at hand. In the end, this may be the most important potential benefit that our approach may have to offer.

One of the most important impediments to achieving simplicity is the well-known fact that most users will almost always choose the system that promises the most features, even if they will never use most of the features and even if the excess features make the system more difficult to use. Indeed, this phenomenon may be largely responsible for motivating organizations to relentless add new features as their systems evolve over successive releases. Ideally, our approach may be able to help users obtain the best of both worlds — the burst of elation that comes from acquiring a system that has every feature one could ever possibly desire combined with the quiet satisfaction of using a simple system that enables one to work smoothly and efficiently with little excess stress and clutter.

1.6 Paper Structure

The remainder of the paper is structured as follows. In Section 2 we present the concepts of our approach in more detail. In Section 3 we discuss our experience developing a rectifier for the Pine email client and using the rectifier to place email messages within the Pine comfort zone. Section 4 presents experimental results that characterize the effect of using this rectifier on a corpus of email messages. We discuss related work in Section 5 and conclude in Section 6.

2. Conceptual Framework

We make the definition of a comfort zone precise as follows. A comfort zone is a property of a system and a training suite. The *exercised region* of the program consists of all blocks of code that execute when the system processes inputs in the training suite. The *comfort zone* is then the set of all inputs that cause the system to remain within the exercised region when it processes the input.

Note that it is usually important to verify that the system behaves in an expected, acceptable way for all of the inputs in the training suite (otherwise the comfort zone will include inputs that can cause the system to behave in unexpected or unacceptable ways, which is usually not the intended effect). Also note that the same system may have different comfort zones for different training suites — indeed, as we discuss in Section 1.2, different comfort zones may be appropriate in different contexts.

2.1 Feature Selection

As stated, the definition of comfort zone can be difficult to work with — determining whether a given input is in the comfort zone or not requires reasoning about the behavior of the program on that input. We therefore advocate designing rectifiers with the aid of a set of *features*. Each feature is simply a property of the input. When designing a rectifier for text inputs, for example, one might choose features such as the presence or absence of certain characters in the inputs, the lengths of various fields or lines, or the overall size of the input. The features should be chosen to enable comfort zone membership recognition — it should be possible to determine, with high likelihood, if a given input is in the comfort zone by simply looking at the input features rather than at the input itself.

2.2 Constraints

Given a set of features, the next step is to select a set of constraints that involve the features. For example, a constraint might require certain characters to be absent or present in the input. Or a constraint could require the length of a given line to be less than a certain number of characters. The constraints should be chosen so that if an input satisfies the constraints, it is highly likely to be within the comfort zone.

2.3 Constraint Enforcement

The next step is to develop transformations that, together, enforce all of the constraints. There is typically one transformation for each constraint; the specific algorithm that the transformation employs typically depends on the constraint. If a constraint states that certain characters must not be present in the input, for example, the corresponding transformation could simply delete all such characters from the input. Or if a constraint requires all lines to be less than a certain number of characters long, the transformation could either truncate overly long lines or insert line breaks to bring all lines within the stated limit.

One obvious goal is that the transformations should attempt to preserve as much information from the original input as possible. In particular, it would be desirable for the transformations to leave inputs intact if they do not violate any of the constraints. We anticipate that the precise transformation algorithms will vary depending on the kind of constraints they are intended to enforce and the context in which they will be used.

Finally, the transformations should generate a record of the changes they make. The rectifier will later combine these records to make a rectification report for each input. Users can access this report to obtain an understanding of the effect of applying the transformation.

2.4 Rectification

The next step is to bundle all of the transformations together to obtain a rectifier. The rectifier applies the transformations in turn to each input to ensure that the input satisfies all of the constraints. It also combines the modification records to generate the rectification report. The rectifier is then inserted between the input generators and the system so that it rectifies all inputs before the input processes them.

2.5 Monitoring

We expect that in most cases the constraints will mostly, but not completely, characterize the comfort zone. We therefore monitor the execution of the system as it processes each input to find any violations of the exercised region. The response to such violations varies depending on the context. In some cases it may make sense to query the user to see if the behavior of the program was acceptable in spite of the violation. If so, it may make sense to add the newly executed blocks of code to the executed region. In other cases users may be willing to live with small violations of the executed region (as measured by the number of newly executed blocks). In yet other cases it may make sense to terminate the program before it can execute any block outside the executed region. Our implemented system supports all of these options.

2.6 Practical Considerations

In principle, it should be possible to use any effective profiling and monitoring system to find and enforce the comfort zone. In practice there are a variety of issues that can complicate these two tasks. These issues may include the difficulty of monitoring code in dynamically-linked libraries, poor performance of the monitoring system, and failure to handle all of the features present in large systems. Fortunately, there are a number of robust code instrumentation systems available that can effectively address these issues[17, 6, 8].

We used the DynamoRIO [17, 2] system for our experiments. Instead of executing the binary directly on the processor, DynamoRIO executes all code out of a cache of recently executed code blocks. When DynamoRIO encounters a branch to a block of code that is not in the cache, it fetches the block from the address space of the executing program and presents it to the monitoring system. The monitoring system can then take a variety of actions, including changing the instructions in the code block before the code block is inserted into the cache and executes.

The DynamoRIO implementation has been engineered over the course of several years, to the point that it is now a robust system that fully supports Windows x86 executables. It imposes relatively little overhead and monitors all of the code at the application level including dynamically linked libraries.

As the system processes the inputs in the training suite, our comfort zone discovery system records all of the executed code blocks. Together, these code blocks comprise the executed region that defines the comfort zone of the system. During production, our monitoring system checks every block inserted into the code cache to verify that it is part of the executed region. If it is not, our system can either terminate the system before the new block executes or simply record the new block and let the application continue. DynamoRIO supports all of these actions.

2.7 Empirical Preconditions

One of the concepts implicit in our our approach is that each implemented system has an *empirical precondition* that its inputs must satisfy for the system to execute correctly. Unlike standard preconditions, which are typically identified during the design phase to provide the developer with a specification of the properties that he or she can assume that inputs will satisfy, the empirical precondition is defined by the behavior of the system. Specifically, an input satisfies the empirical precondition if the program produces the correct (or, in some contexts, an acceptable) output when it processes the input.

Note that the presence of errors in the system can cause the empirical precondition to be stronger than the standard precondition — that is, there are inputs that the implemented system 1) was designed to process correctly, but 2) fails to process correctly in practice because of errors in the implementation.[1] It is usually the case (in part because of such errors) that nobody can completely predict the entire behavioral range of a large software system. This fact means that the empirical precondition is almost always only partially known to the users and developers of the program. It is also only partially known to the developer of the rectifier. The question then becomes how does one obtain an effective rectifier with only partial knowledge of the empirical precondition?

Our conceptual framework provides one answer to this question. Specifically, use features and constraints to obtain a (hopefully) conservative approximation to the (only partially known) empirical precondition. The structure present in this approach helps the developer of the rectifier identify properties that characterize the kinds of common, expected inputs that the system will have repeatedly encountered (and therefore have been made to successfully process) in the past. Together, these properties constitute a *rectifier precondition* that the rectifier forces all inputs to satisfy. Given that the rectifier developer has only partial knowledge of the empirical precondition, the goal is usually to obtain a conservative rectifier precondition that is, in practice, stronger than the empirical precondition. A strong rectifier precondition minimizes the undesirable possibility of the system encountering an input that satisfies the rectifier precondition but not the empirical precondition.

[1] Of course, it is also possible for the implemented system to exceed its specification and process some inputs correctly even though these inputs do not satisfy the standard precondition. This can happen if the implementor decided to build a more general system than required, if the system is built out of more general components than are absolutely necessary to provide the desired functionality, if the presence of ambiguities or errors in the specification caused the developer to conservatively work from a weaker standard precondition, if the developer misunderstood the requirements, or if the program simply happens to produce a correct output (for whatever reason) for some inputs that do not satisfy the standard precondition. In general, we would expect the empirical precondition and the standard precondition to be incomparable. That is, we would expect some inputs to satisfy the standard precondition but not the empirical precondition and vice-versa.

3. Pine Email Client

To illustrate how to apply our approach in practice, we present our experience applying our approach to the Pine [13] email client. Our goal is to use Pine as a simple email reader for text messages. We explicitly wish to eliminate functionality associated with processing more complex types of messages (such as messages with attachments of various types). Pine provides a variety of views for email messages. Our goal is to support list views (a view that lists the messages in a folder, with one line for each message) and message views (a view that displays the contents of a single message on the screen).

3.1 Pine Comfort Zone

The first step is to obtain a training suite of representative messages that we can use to establish a comfort zone for Pine. Our training suite consists of 450 email messages sent to the author between July 1997 and February 1998. We configure Pine to process each message in the training suite in both the list and message views. We then record, for each view, all of the blocks of code that Pine executes as it processes each message.

The goal is to use our training suite to obtain complete statement coverage for an acceptable comfort zone (in other words, our goal is to execute every instruction required to successfully process simple text email messages). We note in passing that obtaining complete statement coverage for any sizable software system is considered to be a difficult task and that a training set of our size would usually be woefully inadequate for this task. The difference is that we have no intention of supporting the full functionality of the system. Indeed, our goal is instead to use the rectifier to narrow down the required functionality as much as possible. Our experimental results show that this approach can make it possible to obtain full code coverage of an acceptable part of the system with a relatively small training suite — indeed, a training suite so small that almost any organization considering using Pine as an email reader can easily obtain such a training suite and verify that Pine acceptably processes the messages in the suite.

3.2 Pine Rectifier

The Pine rectifier is designed to take arbitrary email messages and transform them into a simple text message format. Pine (like many email clients) is designed to process messages stored in mbox format. In this format each message starts with a line of the form `From` and ends with a blank line. The message itself may have a header with various fields such as `From:`, `To:`, etc. A blank line separates the header from the message body. Here is a sample message stored in mbox format [20]. Note that the `From:` field is distinct from the `From` line that indicates the start of the email message.

```
From "Rectifier" Wed Dec 20 20:31:22 2006
To: me@mit.edu
From: someone@csail.mit.edu
Subject: Example
Date: Thu, 28 Aug 1997 15:15:27 -0400

This is an example of a simple text message.
```

Our Pine rectifier is designed to produce messages that closely follow the format of this example.

3.2.1 Message Features

Our rectifier works with the following set of features:

- **From Line:** The contents of the `From` line.
- **Header Fields:** The set of fields in the header, as identified by the names of the fields as found in the header (for example, `From:`, `To:`, etc.).
- **Header Field Contents:** The contents of the `To:`, `From:`, `Subject:`, and `Date:` header fields.
- **Line Lengths:** The lengths of the lines in the message.
- **Body Lines:** The number of lines in the body of the message.
- **Characters:** The characters that appear in the body of the message.

3.2.2 Constraints

The rectifier enforces the following set of constraints:

- **From Line:** The From line must be: `From "Rectifier" Wed Dec 20 20:31:22 2006`.
- **Header Fields:** The message contains a `To:`, a `From:`, a `Subject:`, and a `Date:` field, in that order. It contains no other fields.
- **To: Field:** The `To:` field consists of a single line containing a sequence of comma-separated email addresses. Each email address must be in a form similar to the address `me@mit.edu`. Specifically, each email address must match the following Perl[10] match expression:
  /(\w[-.\w]+\@[-.\w]+\.\w{2,3})\W/
- **From: Field:** The `From:` field consists of a single email address in the same form as the email addresses in the `To:` field.
- **Subject: Field:** The `Subject:` field may contain only "reasonable" characters. Specifically, the subject field may contain only the characters identified by the following Perl character expression:
  \n !"#&'()*+,\-.\/0-9:;<>?@A-Za-z[]_~
- **Date Field:** The `Date:` field consists of a single line in a form similar to `Thu, 28 Aug 1997 15:15:27 -0400`. Specifically, the date consists of a three-character day, a one or two digit day of the month, a three-character month, a four-digit year, a time specifier consisting of three two-digit time components separated by colons, and a five-character time zone specifier.
- **Line Lengths:** No line in the message exceeds 80 characters in length.
- **Body Lines:** The body of the message contains no more than 10,000 lines.
- **Characters:** The message body contains only "reasonable" characters. Specifically, the message body may contain only the characters identified by the following Perl character expression:
  \n\t !"#$%&'()*+,\-.\/0-9:;<=>?@A-Z[\\]^_`a-z{|}~

3.2.3 Constraint Enforcement

The rectifier processes each message to enforce the constraints as follows. It is structured as a pipeline of two transformers, the attachment transformer and the content transformer. The attachment transformer processes the message to eliminate all attachments. It expands all plain text attachments into the body of the message and deletes all other attachments, leaving behind a line in the body of the message indicating that the attachment was deleted. Both the attachment transformer and the content transformer are implemented in Perl. The attachment transformer consists of 94 lines of Perl code; the content transformer consists of 151 lines of Perl code. We note that these transformers are small enough for even organizations with few resources to develop and/or inspect to gain trust in the rectifier.

As the content transformer processes the message header, it looks for the `To:`, `From:`, `Subject:`, and `Date:` fields. When it encounters the line containing the `To:` field, it looks for consecutive, non-overlapping substrings within the line that match the email address pattern described above in Section 3.2.2. It then stores a comma-separated list of these substrings, omitting any substrings that would cause the produced `To:` field to exceed 80 characters in length. Similarly, when it encounters the line containing the `From:` field, it looks for the first substring within the line that matches the email address pattern described above in Section 3.2.2. It then stores the substring.

When the content transformer encounters the `Date:` field, it looks for a substring that matches the date pattern described above in Section 3.2.2. It then stores the substring. Finally, when it encounters the line containing the `Subject:` field, it removes any "unreasonable" characters (as defined above in Section 3.2.2) and, if necessary, truncates the line at 80 characters. It then stores the line away.

When the content transformer reaches the end of the message header, it prints out, in order, the `From` line described above in Section 3.2.2, a `To:` field containing the stored list of email addresses, a `From:` field containing the stored email addresses, a `Subject:` field containing the stored subject string, and a `Date:` field containing the stored date. If the rectifier fails to find a field as it processes the header, or if a field fails to contain any of the substrings that the transformer looks for, the transformer simply prints out a default field that satisfies the field constraints. Note that this approach results in the deletion of any line in the header that is not in one of the fields that the transformer looks for.

As the content transformer processes the body of the message, it removes any "unreasonable" characters (as defined above in Section 3.2.2). It also inserts line breaks as necessary to keep all of the lines in the body less than 80 characters in length. Finally, it keeps track of the number of lines in the body, truncating the message when the number of lines reaches 10,000.

3.2.4 Report Generation

For each message, the attachment transformer produces a report indicating how many attachments it found, how many attachments it deleted, and (implicitly) how many plain text attachments it expanded into the body of the message. The content transformer produces a report indicating which header lines it removed, which header lines it changed, how many body characters it removed, how many newlines it inserted into the body, and how many lines it dropped if it truncated the body. The user can examine this report and if desired, use an appropriate tool (such as a text editor or the Unix `cat` command) to examine the original message (the transformers preserve this message) for more details. The user can also use a separate tool to extract any attachments or even (if he or she is feeling lucky and is sufficiently motivated) use an email reader to view the original message.

4. Experimental Results

To evaluate the effectiveness of our rectifier in moving messages into the comfort zone while preserving an acceptable amount of information from the original messages, we applied our rectifier to several test collections of messages. We used Pine on our training suite of 450 messages to establish a comfort zone for both list views and message views. We then applied Pine to the messages in our test collections both before and after rectification, testing both the list view and the message view for each message. We ran Pine under the control of our monitoring system, counting the number of executed blocks not in the exercised region (instead of terminating Pine as soon as it attempted to execute the first such block). Here are the test collections we used:

- **Normal:** A set of 40 "normal" messages sent to the author in January and February of 1998. These messages were the next 40 messages received after the 450 messages in the training suite.
- **Unusual:** A set of 40 "unusual" messages received by the author between January and April of 2002. We selected this set by first running Pine on 6497 messages received between 1997 to 2001 (without rectification), recording the executed blocks of code, then selecting messages from among 2167 messages received between January and April of 2002 that caused Pine to execute new blocks of code. The goal was to obtain a set of outlier messages that exercise rarely executed blocks of code.
- **Exploit:** A message that exploits a vulnerability in certain versions of Pine. Pine version 4.44 has a string overflow vulnerability associated with a failure to allocate enough memory to hold a parsed string corresponding to certain `From:` fields in the list view [12]. We obtained Pine version 4.64, reinserted the vulnerability from version 4.44 (this vulnerability was removed between version 4.44 and version 4.64), and used this version of Pine in all of our experiments.

  We obtained a message that exploited this vulnerability and verified that it caused Pine to crash when it attempted to create a list view including this message. We verified that if someone sends this message to a user running a version of Pine with the vulnerability, it will cause Pine to crash as soon as the user attempts to view the list of new messages. While we did not verify that it is possible to exploit this vulnerability to execute arbitrary injected code, it is typically possible to exploit this kind of vulnerability for this purpose.

4.1 Normal Message Set

None of the unrectified versions of the normal messages are in Pine's comfort zone. The new block counts for the list view range from a high of 130 new blocks to a low of 68 new blocks with a median of 93 new blocks. The new block counts for the message view range from a high of 1233 (this is a significant outlier; the next highest new block count is 212) to a low of 68 with a median of 100 new blocks. We attribute this result to the fact that the content transformer discards information from the header of every message in the normal message set. Common kinds of discarded information include the name of the sender (headers often contain this information in addition to the email address of the sender) and various header lines that summarize various aspects of the message content and history (for example, if the message was sent as a reply to some other message). The rectifier had made almost no changes to the message bodies — with one exception, it left all of the message bodies unchanged. For that message body it expanded a text attachment and removed an HTML attachment.

After rectification, all of the normal messages are in the comfort zone for both views — Pine can successfully process all of the messages without executing any new blocks at all.

4.2 Unusual Message Set

None of the unrectified versions of the unusual messages are in Pine's comfort zone. The new block counts for the list view range from a high of 296 new blocks to a low of 108 new blocks with a median of 142 new blocks. The new block counts for the message view range from a high of 338 to a low of 129 with a median of 176 new blocks. Note that the new block counts are generally higher than for the normal message set, with the exception of the outlier high new block count of 1233 for the normal message view. Upon inspection, it is clear that the unusual messages tend to have unusual features such as strange characters in the message headers, HTML message bodies, complicated attachments, and long provenance information. The unusual nature of the messages is reflected in the increased new block counts in comparison with the messages in the normal message set.

After rectification, all but one of the 40 messages is in the comfort zone for the list view (this message causes Pine to execute 7 new blocks); all but three messages are in the comfort zone for the message view (these messages cause Pine to execute 15, 15, and 7 new blocks). The two messages that cause the Pine message view to execute 15 new blocks appear to be two versions of the same message. This message has a fairly unusual HTML body. The single message that caused both the list view and the message view to execute 7 new blocks also has an unusual HTML body. Potential options for moving these messages into the comfort zone include adding more messages with HTML bodies to the test collection of messages or developing a rectifier that eliminates all but the most basic HTML commands. We would recommend the second option since it reduces the size of the exercised region while preserving much of the important information in the original messages.

4.3 Exploit Message Set

The unrectified exploit message causes Pine to execute 441 new code blocks. After rectification the message is in Pine's comfort zone. The key change is the replacement of the `From:` field in the original message header (which does not comprise a standard email address) with the default `From:` field. The rectifier left the message body unchanged (the message body is not involved in the exploit and exhibits no unusual features). We note that the version of DynamoRIO that we are using will successfully intercept and prevent remote code injection attacks [17, 2]. We also note that any monitor that terminates the execution as soon as the application attempts to execute a new code block will also intercept and prevent such attacks.

An examination of the code that the exploit triggers shows that the rectifier will never produce an input that would trigger the exploit — to trigger the exploit, the `From:` field of the message must contain backslash characters. The rectifier will never produce a `From:` field that has such characters.

4.4 Discussion

From our perspective, our rectifier is fairly aggressive, at least as applied to the header of the message. As it enforces its tightly constrained header format, it will modify virtually all messages encountered in practice. Nevertheless, it still manages to maintain the most important information in the header — the sender's email address (required to reply to messages), the email address to which the message was sent, and the date and subject. It is of course possible to develop a rectifier that preserves more of the structure of the header. The trade-off is that such a rectifier would produce a wider range of headers, which would, in turn, require a larger training suite to obtain an acceptable comfort zone. Our inclination is to minimize the size of the required comfort zone and the size of the corresponding training suite by making the rectifier as aggressive as possible.

The starting point of our research is that one can almost always find inputs that existing systems fail to process acceptably. But for our proposed approach to work, the rectifier itself must be able to correctly process every conceivable input. One may legitimately wonder why it is feasible to construct rectifiers that accomplish this goal when it is apparently so difficult to construct systems that do so.

There are several reasons why we believe it is feasible to build successful rectifiers. First, the rectifier is under no obligation to preserve all of the information in the input. Unlike the system, which has usually been designed to provide as many features as possible and therefore aspires to support a broad range of inputs, the rectifier can focus on extracting only those pieces of information that it needs to generate its tightly focused set of inputs. Second, the rectifier is a stand-alone system with a clear, narrow goal. Unlike the developers of the system, the developer of the rectifier can focus on this goal in isolation, free of the distractions and requirements of operating within a larger development environment. Also unlike the developers of the system, who must typically use a general-purpose language that has been designed to support a wide range of programs, the developer of the rectifier can use a language and development style tailored to support the specific kinds of tasks that the rectifier must perform. We elaborate on these points further below.

One of the keys to building successful rectifiers is to identify a tightly constrained input format (substantially more constrained than most current systems are designed to process) for the rectifier to produce. Ideally this input format would take the form of a template with slots for items from the original input. Once this rigid format is in place, the rectifier can use pattern matching to carefully chose pieces of the original input to place into the slots in the template. Everything else in the original input is simply discarded.

Our content transformer uses this approach to generate a highly constrained class of headers. Because it uses only the most basic functionality, users can have a high degree of confidence that the email client will be able to process the generated headers without problems. Of course, there is a trade-off — most email clients are designed to support a much broader class of headers. With our approach the functionality associated with this broader class is, by design, unavailable.

Pine is, like many existing systems, coded in the C programming language. C is notoriously ill-suited for the kinds of text processing tasks that the rectifier must perform. And, in fact, the version of Pine that we used in our experiments exhibits a classic string overflow error in which the developer failed to allocate a string large enough to hold all potential inputs. Our rectifier, on the other hand, is written in a scripting language (Perl) specifically suited for string processing. Because many of Perl's constructs support the kinds of string-processing tasks that the transformers must perform, it is relatively easy to develop the transformers and, we believe, the transformers tend to have fewer errors than they would if they were developed in a language with less support for this specific task. Moreover, the Perl language model completely eliminates some common kinds of errors (such as string overflow errors) that can easily occur when coding up these kinds of tasks in C. Our experience developing the Pine rectifier supports these points — we found Perl to be well-suited to the task of quickly writing a small (less than 250 lines of code) rectifier. We believe that coding up the rectifier in C would have required substantially more time and effort.

5. Related Work

Email is one of the most prominent interfaces between users and the external world and one of the most widely used attack carriers. The ubiquity of email and its abuse have inspired the development of a range of email filtering technologies, the most widely used of which are spam filters and virus filters. The goal of both of these filtering technologies is to intercept and eliminate problematic email messages before they are presented to the email client.

Standard spam filtering techniques use feature extraction and machine learning to automatically recognize messages that are likely to be spam[1]. Virus filters often use virus signatures to recognize message that may contain viruses [9]. The drawback of this approach is that new viruses may avoid the virus signature databases, which only contain signatures of known viruses. Another approach is to simply remove attachments of file types (such as .scr or .js) that have been known to carry viruses. Yet another approach is to have the email client associate document types (such as Microsoft Office documents) that have been used in the past to carry viruses with non-standard programs that may be less vulnerable to the viruses. So, for example, one might associate Word files with WordPad instead of Word because WordPad does not execute macros in Word files (which may contain viruses that are activated when the macro is executed). We note that this last example shows how excess functionality can become a serious liability for users of the system.

Unlike spam and virus filters, our goal is not to eliminate problematic messages before they reach the email client. Instead, our goal is to use rectification to make every message innocuous so that it can be safely presented to the email client. By monitoring the email client as it processes messages, we can automatically detect comfort zone violations and, if desired, abort the computation before any undesirable functionality can be triggered. Despite these differences, there are also some conceptual similarities between the approaches. One could view the application of monitoring to confine the system to its exercised region as an automated way of obtaining a new system without the harmful excess functionality present in the original system. This could be viewed as similar in spirit to associating certain kinds of document types with alternate programs in the email client, but with the advantage that it does not require the availability of a separate system.

The pervasive presence of Internet attacks has motivated the development of a variety of tools to filter out such attacks. Firewalls, for example, are often configured to discard packets incoming on ports that attacks have successfully targeted in the past [19]. Conceptually, one can view blocking certain ports as a coarse-grain form of rectification that is designed to completely eliminate access to the services typically present on those ports. Our approach is designed to operate at a much finer granularity to enable access to a simple part of a complex monolithic application implemented as a single binary. Firewalls can also blacklist IP or MAC addresses. While it would be possible to develop rectifiers that perform similar actions based on the identity of the input source, our basic philosophy is to, whenever possible, present the transformed input to the target system. We therefore see rectification as discussed in this paper as orthogonal and complementary to techniques that discard inputs based on the identity of the input source.

Web browsers can often be configured to block or disable certain potentially problematic features such as pop-ups and cookies [5]. Rectifiers can provide similar functionality by eliminating these features as they arrive in the input.

We understand that input format mismatches and other input anomalies occur frequently when people use computers, and that various input conversion programs are often used to make it possible to successfully process otherwise problematic inputs. Our goal is much broader. Specifically, we aim to significantly limit the functionality that the inputs exercise and are willing, within reason, to discard information to accomplish this goal. Moreover, we advocate the pervasive and aggressive use of rectification as a means to simplify the overall computing environment. The benefits of our approach include the ability to use existing complex, functionality-laden systems in simple ways without risking exposure to errors or vulnerabilities that lie outside the exercised region and a reduced cognitive load on the users and maintainers of the computing environment.

Input rectification can be seen as a form of acceptability-oriented computing [14, 15]. Specifically, the set of constraints that the rectifier enforces can be seen as a set of acceptability properties that every input must satisfy to be acceptable. Input rectification is also similar in spirit to data structure repair [4] in that it enforces a set of properties whose violation might otherwise cause problems for the system. One difference is that the goal of data structure repair is to keep a system executing acceptably even though it encounters errors that corrupt its data structures. The goal of input rectification, on the other hand, is to move inputs into the system's sweet spot to avoid encountering errors in the first place.

Common Lisp and ZetaLisp lambda-lists enable a function to treat its argument list as data [16]. My understanding is that this feature was used to develop functions that examined and, if necessary, changed argument lists to values that invoked functions could successfully handle.

6. Conclusion

The complexity of the systems in our current computing environment leads inevitably to errors, security vulnerabilities, and a large cognitive burden on users. Despite these problems, users are locked in — it is usually close to unthinkable to attempt to build a simpler environment (or even significant components of a simpler environment) from scratch. But hidden within almost every large, complex system is a well-tested core that almost always performs acceptably. The identification and enforcement of appropriate comfort zones can extract these cores and eliminate the excess functionality (along with the associated errors and vulnerabilities) currently present in virtually every deployed system.

As always, simplicity requires renunciation. The inevitable price for this reduction in complexity is the elimination of information that would otherwise exercise the discarded functionality. Based on our experience with the Pine email rectifier, we expect that it will be possible, in many cases, to obtain a simple rectifier that successfully moves inputs into the comfort zone while preserving the most important information. The pervasive adoption of such rectifiers may dramatically simplify our computing environment and eliminate frustrating or even dangerous problems while preserving the key benefits that it delivers to its users.

7. Acknowledgements

I would like to thank Sam Larsen for his invaluable help with DynamoRIO. The implemented comfort zone discovery and monitoring system uses a (slightly modified) hash table implementation developed by Christopher Clark [3].

References

[1] Apache SpamAssassin Project. http://www.spamassassin.apache.com.

[2] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusets Institute of Technology, September 2004.

[3] Christoper Clark. Hash table implementation. http://www.cl.cam.ac.uk/~cwc22/hashtable/.

[4] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 2005 International Conference on Software Engineering* , St. Louis, MO, May 2005.

[5] Firefox Options Page. http://www.mozilla.org/support/firefox/options.

[6] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[7] Microsoft word scripting vulnerability. http://www.microsoft.com/technet/security/Bulletin/MS02-021.mspx.

[8] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[9] Norton AntiVirus, marketed by Symantec. http://www.symantec.com.

[10] Perl website. http://www.perl.com.

[11] Matt Pietrek. *Windows 95 Programming Secrets*. John Wiley & Sons, November 1995.

[12] Pine exploit. www.securityfocus.com/bid/6120/discussion.

[13] Pine website. www.washington.edu/pine/.

[14] Martin Rinard. Acceptability-oriented computing. In *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion)*, Anaheim, CA, October 2003.

[15] Martin Rinard, Cristian Cadar, and Huu Hai Nguyen. Exploring the acceptability envelope. In *2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '05 Companion)* , San Diego, CA, October 2005.

[16] G. Steele and R. Gabriel. The evolution of lisp. In *Proceedings of the Second ACM SIGPLAN Conference on the History of Programming Languages*, Cambridge, MA, April 1993.

[17] Gregory Sullivan, Derek Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators (IVME-03)*, San Diego, CA, June 2003.

[18] Wikipedia Buffer Overflow Article. http://en.wikipedia.org/wiki/Buffer_overflow.

[19] Wikipedia Firewall Article. http://en.wikipedia.org/wiki/Firewall_(networking).

[20] Wikipedia Mbox Article. http://en.wikipedia.org/wiki/Mbox.