Automatically Identifying Critical Behaviors in Programs

by

Michael Carbin

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

Masters of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author .....
Department of Electrical Engineering and Computer Science
September 4, 2009

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

Accepted by .....
Terry P. Orlando
Chairman, Departmental Committee on Graduate Students

Automatically Identifying Critical Behaviors in Programs

by

Michael Carbin

Submitted to the Department of Electrical Engineering and Computer Science
on September 4, 2009, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science and Engineering

Abstract

The large size of modern software systems has led to an increase in the complexity of the interaction between a system's code, its input, and its output. I propose the following classifications for the regions of a system's input:

- *Critical control*: data that influences the internal operation and output of the system.
- *Critical payload*: data that heavily contributes to the output of the program but does not substantially influence the internal operation of the program.
- *Benign control*: data that influences the internal operation of the system, but does not contribute to the output of the system.
- *Benign payload*: data that neither contributes to the output nor substantially influences the internal operation of the program.

In this thesis, I present Chaos, a system designed to automatically infer these classifications for a program's inputs and code. Chaos monitors the execution trace and dynamic taint trace of an application over a suite of inputs to determine how regions of the programs's code and input influence its behavior and output. This thesis demonstrates the accuracy of Chaos's classifications for a set of imaging applications and their support libraries.

These automatically inferred classifications are relevant to a variety of software engineering tasks, including program understanding, maintenance, debugging, testing, and defect correction triage.

Thesis Supervisor: Martin C. Rinard
Title: Professor

In memory of

Rowena Joyce Carbin

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A program's input often contains two kinds of data: *control*, which influences aspects such as the invoked functionality or interpretation of subsequent parts of the input, and *payload*, which contains the data that the program directly processes to generate the output. In data container formats (e.g. image, audio, and video), control data specifies meta-properties such as data layout, the type of compression algorithm, and the dimensions of the output. Payload data, on the other hand, often manifests itself as data processing parameters or the raw data for processing.

A program typically defines many precise invariants about the content of control data. As the program processes these inputs, violations of these invariants can lead to large, discontinuous changes in the behavior of the program. For instance, an invalid piece of control data may cause the program to invoke incorrect functionality. On the other hand, programs typically define few invariants about the content of payload data. Because this data represents the raw, unprocessed output (e.g. an image's contents), a program often has a similar behavior over the domain of possible payload data values.

The difference between a program's behavioral sensitivity on control data versus payload data suggests that some data is more *critical* to the behavior and output of the program than other data. More specifically, a corruption to a critical piece of data can make a program crash, invoke the wrong functionality, or produce a significantly distorted output. On the other hand, a corruption to a *benign* piece of data can have

a relatively benign effect on the behavior and output of the program. For example, a benign piece of data may affect a relatively small portion of the output, such as a single pixel in an image. Therefore, a corruption of such data may lead to a small amount of corruption in the output.

Although a program's behavioral sensitivity to a piece data may identify the data as critical, it does not entirely identify every piece of critical data in a program's input. For example, it may be the case that a large behavioral change represents logging or other functionality that has a benign effect on the program's output. Additionally, certain regions of payload data may have a larger contribution to the program's output than other payload regions. For instance, a paletted image is an image in which each pixel value is encoded as an index into a color palette table. Each entry in the color palette table specifies an appropriate color value. In such images, a corruption of the index for a single pixel will have a relatively benign effect on the output because only that single pixel value will be corrupted. However, a corruption of the color palette table will be critical because every pixel in the image makes a reference to this table.

These examples suggest that a particular piece of data can be separately identified as either control or payload and also as critical or benign. This defines the following ontology for a region of the input to a program:

- **Critical control**: data linked to behavioral changes that produce large changes in the output. Structural information, such as embedded field sizes, are pieces of critical control data because the program will be unable to continue processing the input if the structural information is incorrect.

- **Critical payload**: data that heavily contributes to the output of the program but does not substantially influence the behavior of the program. The color palette table of a paletted image critically determines the quality of the final, outputted image.

- **Benign control**: data linked to behavioral changes that produce little to no changes in the output. Fields that designate the existence of irrelevant functionality (i.e. embedded comments that indicate which software package created

the file) may change the observed behavior of the program, but these behavioral changes have a benign effect on the program's output

- **Benign payload**: data that neither contributes to the output nor substantially influences the behavior of the program. For uncompressed data, such as raw images or plain text, localized changes to the input will produce localized changes in the output. Further, such changes will not alter the behavior of the program.

These data classifications are an artifact of the way a program uses a piece of data. Because the behavior of a program is given by its source, these data classifications correspond to pieces of code in the program.

- **Initialization and Cleanup:** Most applications have pieces of code that deal with no part of the input. This code often handles initialization and cleanup of global data structures.

- **Parsing, Verification, Sanitation:** Code associated with input parsing, verification, and sanitation will touch all types of input. This code will critically determine the behavior and output of the application because the application will reject inputs that violate structural or semantic constraints. Sanitation code may be benign because it may implement techniques to cleanse an input that violates structural or semantic constraints.

- **Feature Sorting:** Many binary container formats consist of a sequence of commands. Each command is minimally specified by a type and command length; the rest of the command specifies command-specifc data. An application for such a format will first read a command and then use the command's type to determine which piece of the functionality to invoke. The code that directs commands to sub-components touches control data. This code is critical because an error in these regions of code may cause the program to invoke the wrong functionality. The code that implements each sub-component may be either

critical or benign. For example, some sub-components may be benign because they implement functionality that has no effect on the output.

- **Payload processing:** Code that implements compression and decoding schemes often touches the entirety of the payload data that the program uses to generate an output. This type of code is critical to the output quality of the application because it is responsible for generating the entire output. However, a program may also contain benign pieces of code that generate only small portions of the entire input.

These code distinctions give software developers a better understanding of their system. Errors in control code and payload code typically manifest themselves as distinct failures [28]. Errors in control code typically cause the program to diverge dramatically from its desired behavior. Possible symptoms include the inability to process the input at all or the invocation of the incorrect functionality. Errors in payload code tend to be more subtle, typically an anomaly in the content as it is presented to the user. Understanding this distinction can help developers more accurately diagnose and eliminate program defects.

Developers often find themselves with more defects than they can reasonably attempt to correct. Because, in general, defects in benign code tend to have less drastic consequences on the overall execution than do defects in critical code, understanding the differences between these two kinds of code and the roles they play in the system can help developers make better decisions about which defects to fix first.

This thesis presents Chaos. Chaos gives developers an automated tool to infer the control versus payload and critical versus benign distinctions of a program's code and input. Given a program and an input, Chaos produces the following:

- **Input Specification:** It produces a chunking of bytes in the input to semantically grouped fields.

- **Input Classification:** It produces, for each field of the input specification, a classification of whether the field is control or payload and critical or benign.

- **Code Classification:** It produces, for each executed block of source code, a classification of whether the block is control or payload, critical or benign, or a mixture of classifications.

Chaos first uses automated input specification inference to produce a mapping from input bytes, to semantic fields. Chaos then uses automated, directed fuzzing to explore the behavior of the program. By observing how drastically the behavior of the system changes as a result of a change to an input field, Chaos determines if the field is control or payload. By analyzing how much the program's output depends on each field, Chaos determines if a field is critical or benign. After Chaos classifies each field in the input, it uses dynamic taint tracing to identify which code blocks reference which fields. Chaos produces a classification for each block by aggregating the classifications for the fields each block references.

This thesis makes the following contributions:

- **Concept:** This thesis proposes and explores the concept that control versus payload and critical versus benign distinctions for data and code represent a productive classification in current software development practices.

- **Technique:** This thesis presents Chaos, a novel system designed to automatically infer the control versus payload and the critical versus benign distinctions for a program's input and code. Chaos uses execution monitoring, dynamic taint tracing, automatic input specification inference, and behavioral distance measurement to produce its classifications.

- **Results:** This thesis gives a detailed analysis of the quality Chaos's classifications by discussing how they correspond to the design and implementation of real programs.

Chapter 2 gives an overview of the components in Chaos's design. Chapter 4 describes, in detail, the design and implementation of each of Chaos's components. Chapter 5 provides an experimental evaluation of the quality and applicability of Chaos's classifications. Chapter 6 discusses the related work.

Chapter 2

Motivation

This chapter uses an example application and file format to motivate the utility and design of Chaos.

Figure 2-1 gives an abbreviated specification of the Portable Network Graphics (PNG) file format [1]. PNG is a popular image format that many image viewers, web browsers, and image manipulation tools support.

Every PNG file begins with a fixed sized header and continues with an arbitrary number of variable-length data chunks. A predetermined, fixed-format chunk ends the list of chunks. I have omitted this chunk from Figure 2-1 for brevity. The file header contains the data required to load a basic PNG file. The first seven bytes of the header contain a magic constant that identifies the file as a PNG image. The rest of the header contains the image's height, width, and encoding type.

Each additional chunk specifies new information about the contents of the image. At the very least, each chunk provides its length, a tag indicating the type of the chunk, and a checksum that ensures the integrity of the chunk's data. These fields allow an application to navigate an entire file without knowing the exact layout of every chunk. The structure of the rest of the chunk is specific to the data that the chunk contains.

The `time` chunk provides a timestamp for the creation date of the image. The `gamma` chunk indicates to the image renderer that gamma correction needs to be applied to the image. Gamma correction transforms the luminance of each pixel

in an image in accordance with a given gamma parameter. The final `data` chunk contains the compressed picture content of the image.

The official reference implementation for loading PNG images, libpng, encompasses some 25KLOC. There are a number of situations in which a libpng developer may want to better understand how the libpng system behaves. For instance, a new developer, charged with implementing a new feature, may want to understand how the existing behaviors of a 25KLOC system interact with the desired behavior required to implement the new feature. In this scenario, there are a number of properties of the system and its input that could help the developer understand how to implement the new feature.

- **Specification:** Given a binary input, the developer may want to know which bytes of the input belong to the same semantic field. For instance, the developer may want to know that bytes 30-34 represent the `length` field of the `time` chunk.

- **Input Localization:** Given an input specification, the developer may want to know which functions manipulate which fields of the specification. For example, the developer may want to know that the `png_set_time` function processes the value of the `time` chunk.

- **Criticality:** Given an input specification, the developer may want to know which fields determine the behavior and output of the application. For example, the developer may want to know that the checksum embedded in the header chunk must match the checksum of the header chunk data that the `png_crc32` function computes. If the checksums do not match, the library will not attempt to load the rest of the input. Therefore, the checksum embedded in the header is critical to the behavior and output of the application. On the other hand, if the checksum embedded in the `gamma` does not match its computed checksum, then the library will still attempt to load the rest of the input. Therefore, the checksum of the gamma chunk has a benign effect on the behavior and output of the application.

```
struct png {
    struct header {
        char magic[] = {0x89, 0x50, ...};
        int width;
        int height;
        byte interlacing;
        int crc;
    }
    struct time {
        int length;
        char tag[] = "tIME";
        int value;
        int crc;
    }
    struct gamma {
        int length;
        char tag[] = "gAMA";
        int gamma;
        int crc;
    }
    struct data {
       int length;
       char tag[] = "IDAT";
       byte data[length - 8];
       int crc;
    }
}
```

Figure 2-1: An abbreviated specification of the Portable Network Graphics file format

Chaos provides developers with a set of components that will automatically infer these properties for an application and its input.

- *Input Specification Generator:* determines the structure of semantic fields in a binary input.
- *Field Classifier:* classifies each field in the specification as control (i.e. the field influences the behavior of the application) or payload (i.e. the field does not influence the behavior of the application). The Field Classifier also classifies each field as critical (i.e. the field substantially influences the behavior or output of the application) or benign (i.e. the field does not substantially influence the behavior or output of the application).
- *Code Classifier:* determines, for each basic block in the program's code, which fields in the input that the block references. The Code Classifier also classifies each basic block as either control or payload and either critical or benign.

Chapter 3

System Design

Figure 3-1 depicts Chaos's high-level design. Chaos is composed of an Execution Monitor that monitors the behavior the application, an Input Specification Generator that produces a structural specification of the input, a Fuzzer that explores the behavior of the application as it responds to changes to its input, a Field Classifier that produces classifications for the application's input fields, and a Code classifier that produces classifications for the application's code.

3.1 Execution Monitor

Figure 3-2 depicts the inputs and outputs of the Execution Monitor. The Execution Monitor takes an application and a test input. The Execution Monitor runs the application on the input and records the application's *execution trace* and *dynamic taint trace.*

- **Execution Trace.** The execution trace of an application is the sequence of function call and branch instructions executed during the run of the program. Each entry in the execution trace contains a reference to the executed function call or branch instruction and reference to the function or basic block to which control flow transfered.

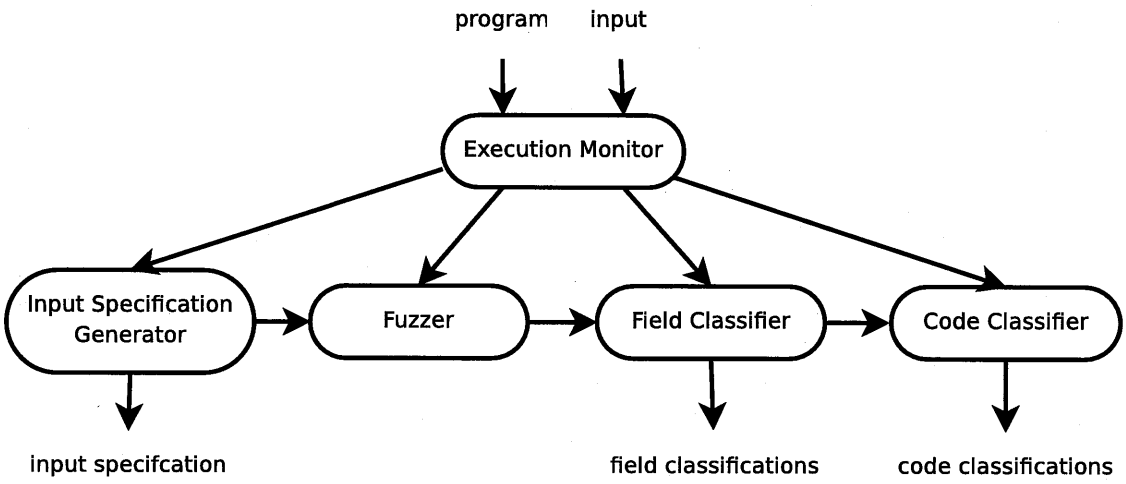

Figure 3-1: System design

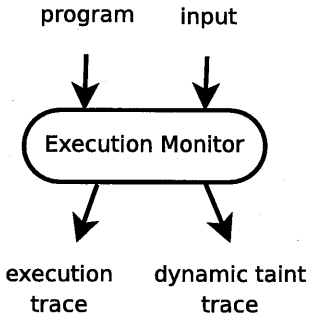

Figure 3-2: Execution Monitor

- **Dynamic Taint Trace.** The dynamic taint trace of an application is the sequence of executed instructions that were affected by the input. Each entry in the dynamic taint trace contains a reference to the executed instruction and the list of input bytes that affected the instruction.

3.2 Input Specification Generator

Figure 3-3 depicts the inputs and output of the Input Specification Generator. The Input Specification Generator groups consecutive bytes of the application's input into fields. The Input Specification Generator first uses the dynamic taint trace to

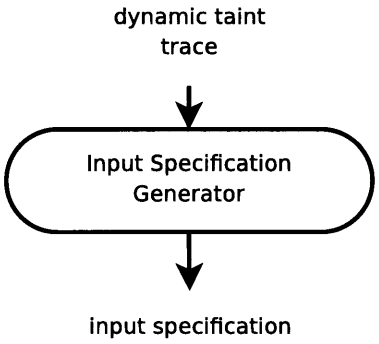

Figure 3-3: Input Specification Generator

determine which instructions each byte of the input affects. The Input Specification Generator then groups fields according to the heuristic that bytes that are adjacent in the input and affect the same operations in the application are in the same field.

3.3 Fuzzer

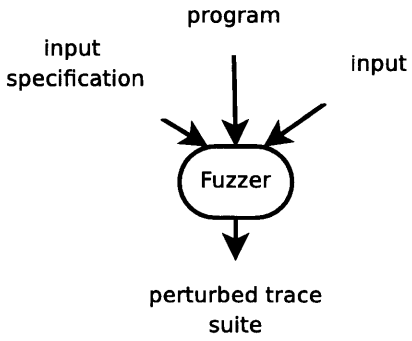

Figure 3-4: Fuzzer

Figure 3-4 depicts the inputs and outputs of the Fuzzer. The Fuzzer provides Chaos with a means to explore the behaviors of the application on a suite of inputs that are similar to the original input. The Fuzzer generates a new, randomly fuzzed input for each field in the input specification. More specifically, each perturbed input is the same as the original input except for the value of a single field. The Fuzzer

uses the Execution Monitor to record the execution trace of the application as it runs over each perturbed input; this produces a suite of perturbed execution traces.

3.4 Field Classifier:

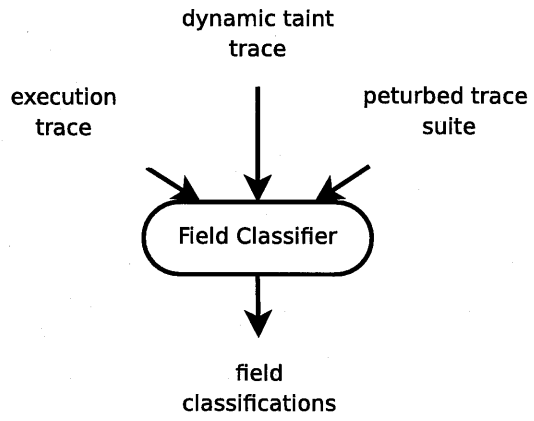

Figure 3-5: Field Classifier

Figure 3-5 depicts the inputs and outputs of the Field Classifier. The Field Classifier classifies each field of the input specification as either control or payload and either critical or benign. The Field Classifier uses *behavioral distance* and *output dependence* to compute its classification for a field.

- **Behavioral Distance:** The behavioral distance describes how similar one execution is to another. A behavioral distance of 0 means that both runs of the application executed the exact same functionality. A behavioral distance of 1 means that each run of the application executed entirely different pieces of functionality. The Field Classifier compares the execution of the program on the unperturbed input to each perturbed execution by computing the behavioral distance between their two execution traces. Because the Fuzzer creates each perturbed execution by changing a single field of the input, the Field Classifier assigns the behavioral distance of a perturbed execution to the field associated with that execution.

The Field Classifier clusters all the behavioral distances observed among the fields into two clusters. The first cluster consists of all fields that have a behavioral distance close to 0 (i.e. close to the original execution); these fields are given a payload classification. The fields in the second cluster are those with a large behavioral distance; these fields are labeled as critical control. As discussed in Section 5.4, Chaos labels all control fields as critical.

- **Output Dependence:** The Field Classifier inspects the dynamic taint trace of the original, unperturbed execution to compute a field's output dependence. The output dependence of a field is the percentage of the operations involved in the computation of the output that are affected by the field.

  The Field Classifier uses output dependence to determine if a payload field is critical or benign. If a field has an output dependence greater than .1 (i.e. it affects 10% or more of the operations involved in computing the output), then the Field Classifier classifies the field as critical. Otherwise, the Field Classifier classifies the field as benign.

3.5 Code Classifier

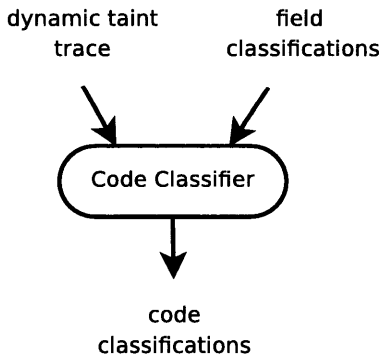

Figure 3-6: Code Classifier

Figure 3-6 depicts the inputs and outputs of the Code Classifier. The Code Classifier uses the Field Classifier's results to compute a classification for each executed

block in the application. The Code Classifier inspects the dynamic taint trace from the execution of the application on the unperturbed input to determine which basic blocks reference which fields of the input. For each basic block, the Code Classifier collects the classification for each field that the basic block references. The Code Classifier then assigns the basic block the classification that occurs most frequently among its fields. Basic blocks that do not have a single, dominant classification are reported as *mixed usage*.

Chapter 4

Implementation

As discussed in Chapter 3, Chaos consists of an Execution Monitor, an Input Specification Generator, a Fuzzer, a Field Classifier, and a Code classifier. This chapter discusses the design and implementation of each component.

4.1 Execution Monitor

As described in Section 3.1, the Execution Monitor records an application's execution trace and dynamic taint trace while the application runs on an input. Chaos uses the execution trace to compute the behavioral distance between two executions of the application on two similar inputs. Chaos uses the dynamic taint trace to generate the input specification, determine output dependence, and to classify basic blocks in the application's source. Each trace can be viewed as a sequence of events where each event corresponds to an executed instruction. Section 4.1.1 discusses the individual events in the execution trace and Section 4.1.2 discusses the individual events in the dynamic taint trace. Sections 4.1.3 and 4.1.4 discuss the Execution Monitor's implementation.

4.1.1 Execution Trace

There are a number of ways in which one can represent an application's execution trace. For example, one can describe the execution trace as a set or a sequence of executed modules, functions, basic blocks, or branches. I have chosen to represent the execution trace by the sequence of executed functions and branches. The execution trace consists of the following events:

- *Function call:* A function call event corresponds to the execution of a function call instruction. The event records the function invoked by the instruction.
- *Branch:* A branch event corresponds to the execution of a branch instruction. The event records the basic block to which the instruction jumped.

These two different events allow Chaos to inspect the both the coarse (function call) and fine (branch) behaviors of an application.

4.1.2 Dynamic Taint Trace

Most dynamic taint tracing implementations only track enough information to determine whether or whether not an external source of input taints an operation in the execution of a program [24, 29, 10, 27, 17]. These taint tracing implementations assign a single boolean value to each operation in the program. Chaos, on the other hand, requires more precise information. Chaos's Input Specification Generator and Code Classifier need to be able determine which inputs and which offsets of those inputs affect an operation. To support this, the Execution Monitor associates the output of each operation in the program with a *taint item.* A taint item identifies a collection of inputs and offsets into those inputs. During the execution of an application, the Execution Monitor can associate an operation's output with one of the following taint items:

- *Taint Object:* a source of input to the application. A taint object is given by a string that names the origin of the object. In the case of a file, this string is the

name of the file. In general, a taint object can be a file, a socket, or a memory buffer.

- *Taint Tag:* an offset into a taint object. A taint tag is given by a reference to a taint object and an integer that indicates the offset into the taint object.
- *Taint Range:* a range of offsets into a taint object. A taint range is given by a reference to a taint object and two integers that indicate the beginning and the end of the range of offsets. A single taint range more efficiently represents a large collection of contiguous offsets than a collection of taint tags.
- *Taint Set:* a union of taint objects, tags, ranges, or other taint sets. A taint set is given by a list of references to its constituent taint items.

Each event in the dynamic taint trace consists of a taint item and a reference to the operation in the program for which the Execution Monitor generated the taint item. Section 4.1.3 describes how the Execution Monitor creates and manages taint items for each operation in the program.

4.1.3 Instrumentor

Without hardware support for execution monitoring or dynamic taint tracing, the Execution Monitor must instrument the source or binary of the application with additional code that generates the execution trace and dynamic trace.

I implemented a static source code instrumentor built on top of the Low Level Virtual Machine (LLVM) compiler infrastructure [18]. LLVM provides a language and platform independent Intermediate Representation (IR), an introspection library, and an extensible compiler that makes it possible to easily manipulate and generate native code for applications compiled to the LLVM IR. Though Chaos's workflow currently relies on having the source of an application, the instrumentor could target any language, including assembly language, for which an LLVM IR translator exists. The LLVM project currently provides translators for x86 assembly, C, C++, Microsoft Intermediate Language (MSIL), Java bytecode, Fortran, and Ada.

The instrumentor modifies the source of the application to produce a new, instrumented application. The instrumentor instruments function call, branch, load, and store instructions with code to emit events to the execution trace and to manage taint items. For all other types of operations in the program, the instrumentor uses a static taint analysis to determine which operations should be instrumented with code to emit an event to the dynamic taint trace.

- **Function Call.** The instrumentor instruments each function call instruction in the application with code that records a function call event to the execution trace. The instrumentation code also transfers taint items for the caller's arguments to the callee's parameters and transfers the taint item for the callee's return value to the caller.

- **Branch.** The instrumentor instruments each branch instruction in the application with code that records a branch event to the execution.

- **Load.** The instrumentor instruments each load instruction in the application with a read barrier. The read barrier transfers the taint item for the contents of the load's source memory address to the load's destination operand.

- **Store.** The instrumentor instruments each store instruction in the application with a write barrier. The write barrier transfers the taint item for the store's source operand to the contents of store's destination memory address.

Static Taint Analysis

To record the dynamic the taint trace, the instrumentor must insert instrumentation code at each operation in the program that may be tainted by the input. At runtime, the instrumentation code takes the union of the operation's inputs and creates a new taint set for the operation's output.

In a naive implementation, the instrumentor could instrument every operation in the program. However, this approach would lead to unnecessary overhead for operations that are never tainted by the input. For example, programs compiled without

optimization may store the results of many constant operations; these operations are never tainted by the input and, therefore, do not need to be instrumented. This technique would also lead to unnecessary overhead for operations that are tainted by the same regions of input. Many of the operations in a single function can be tainted by the same regions of input. A naive instrumentor would create new taint sets for each of these operation when, in fact, each operation could share a single taint set.

For each function, the instrumentor uses a static analysis to identify the set of *local taint sources* that taint each operation in the function. A function's local taint sources consist of the function's parameters, return values from any function calls, and values loaded from pointers to memory. The instrumentor uses these sets as follows:

- If an operation is not tainted by a local taint source, then the instrumentor does instrument the operation.
- If two operations are tainted by the same local taint sources, then the instrumentor will reuse one operation's taint set as the taint set for the other operation.

The taint analysis calculates the set of local taint sources that affect an operation with an intra-procedural dataflow analysis. The taint analysis does not propagate taint through control flow or pointer arithmetic operations. Therefore, the taint analysis does not track control or indirect data dependencies. Section 5.4 discusses the effects of ignoring control and indirect dependencies on Chaos's classifications.

4.1.4 Runtime

In addition to an instrumentor, the Execution Monitor also contains a runtime. The runtime consolidates the majority of the logic required to access and manage the execution trace and the dynamic taint trace. Each fragment of instrumentation code typically contains at least one call into the runtime. The runtime is instrumentor-oblivious and could be used to support instrumentors built upon other instrumentation frameworks, such Valgrind [23], Pin [21], and DynamoRIO [5]. The runtime provides the Execution Monitor with the following components:

- **Trace Manager.** The Execution Monitor uses a trace manager to insert events into the execution trace and the dynamic taint trace.
- **Shadow Registers.** The Execution Monitor uses shadow registers to store taint items for the machine registers of the application.
- **Shadow Memory.** The Execution Manager uses a shadow memory to store taint items for each memory address in the application.
- **Shadow Stack.** The Execution Monitor uses a shadow stack to maintain taint items for each frame in the application's call stack.
- **Shadow File System.** The Execution Monitor uses the shadow file system to automatically identify the external inputs to the application.
- **Library Shim.** The Execution Monitor uses a library shim to intercept calls to C standard library functions.

The following sections discuss the design, implementation, and use of each of the runtime's components.

Trace Manager

At its core, the trace manager maintains a large buffer of events. The trace manager provides an API that allows the Execution Monitor to add function call, branch, and dynamic taint events to the buffer. As the Execution Monitor adds events to the buffer, the trace manager first serializes and compacts the buffer. The trace manager then directs the buffer either to disk or to one of Chaos's analysis components (e.g. the Input Specification Generator). Section 4.1.5 discusses how Chaos directs the execution trace and the dynamic taint trace to its analysis components.

Shadow Registers and Shadow memory

For each operation the program, the Execution Monitor must track the taint items for the operation's inputs and outputs. The Execution Monitor uses shadow registers

to hold taint items for inputs and outputs that reside in machine registers. The Execution Monitor uses shadow memory to hold taint items for inputs and outputs that reside in memory.

Shadow Stack

The Execution Monitor must be able to mimic the call stack behavior of an application. More specifically, applications compiled for LLVM may use the call stack to preserve registers, pass arguments, and receive a return value when the application makes a function call. Similarly, the Execution Monitor uses a shadow stack to preserve taint items for the machine registers, pass taint items for the arguments to a function call, and receive the taint item for the return value of a function call.

Shadow File system

The Execution Monitor uses the shadow file system to automatically identify the application's external inputs. The shadow file system provides a mechanism to intercept calls to the following UNIX file system calls:

- `open`: serves an application's request to open a file; `open` takes a filename as a parameter and returns a unique handle as a result. On call to `open` the Execution, Monitor allocates a taint object that identifies the opened file as a source of external input to the application.

- `read`: serves an application's request to read an opened file; `read` takes a handle to a file, an integer that indicates the number of bytes to read from the file, and a pointer to a memory buffer in which to store the bytes from the file. On a call to `read`, the Execution Monitor first determines which taint object the handle represents. The Execution Monitor then inserts a sequence of taint tags into the shadow memory at the address of the memory buffer; the sequence of taint tags corresponds to the sequence of offsets for the bytes that were read from the file.

- `write`: serves an application's request to write to an opened file; `write` takes a handle to a file, an integer that indicates the number of bytes to write to the file, and a pointer to a memory buffer that gives the source of the bytes to write to the file. On a call to `write`, the Execution Monitor reads the taint items from the shadow memory location of the memory buffer and records that the taint items were outputted by the application.

- `mmap`: serves an application's request to map a file to a region of memory; `mmap` takes a handle to a file, an integer that indicates the number of bytes to map, and an address at which to map the file. `mmap` can also generate a mapping address if one is not passed as a parameter. On a call to `mmap`, the Execution Monitor inserts a sequence of taint tags into the shadow memory at the mapping address; the sequence of taint tags corresponds to the sequence of file offsets that were mapped.

Library Shiming

To gain a complete representation of the behavior of the application, the Execution Monitor must be able to observe the entire execution trace and dynamic taint trace of an application. Though the Execution Monitor instruments every function in the application's source, an application may invoke functions in a support library for which the source is not available. In the case of the C standard library, the runtime provides the Execution Monitor with a suite of functions that emulate the execution trace and dynamic taint trace of several functions. For example, the runtime provides emulations for `memcpy` and `memset`.

4.1.5 Architecture

Recording the execution trace and dynamic taint trace of an application imposes a significant overhead on the execution of an application. For one of the benchmarks presented Chapter 5, a complete run of Chaos, which includes the Fuzzer, required 500 monitored runs of the application. The execution trace and dynamic trace of an

short-running application (i.e. one to five seconds) can be tens of megabytes in size. Therefore, a complete run of Chaos can generate multiple gigabytes of data.

Writing these execution traces and dynamic taint traces to disk would consume a large amount of disk space and, more importantly, bottleneck the execution of the application to the performance of the computer's I/O system. Instead, the Execution Monitor implements a modular, streaming architecture. The Execution Monitor's architecture specifies that any consumer of an execution trace or a dynamic taint trace be given as a module to the system. During the execution of an application, the Execution Monitor will then dynamically stream execution trace events and dynamic taint trace events to the given modules. This architecture provides two performance improvements :

- I/O performance: in this architecture, the Execution Monitor need not write traces to disk because these traces can be kept in memory.
- Concurrency: On a multi-core machine, this architecture reduces the overhead of execution monitoring and analysis by allowing analysis modules to run concurrently with the application. Chaos also uses this architecture to concurrently monitor and analyze multiple inputs.

The following sections describe each of Chaos's analysis components. Each component is a module in the Execution Monitor's architecture.

4.2 Input Specification Generator

The Input Specification Generator infers a specification of the semantic fields of the input. This input specification provides the Fuzzer with an effective mechanism to reduce the search space of possible inputs. More precisely, the input specification enables the Fuzzer to generate inputs by perturbing multiple bytes at a time (as opposed to perturbing a single byte at a time). This is not possible without a specification because perturbing bytes that belong to multiple semantic fields will cause Chaos to report inaccurate classifications.

The Input Specification Generator uses the heuristic that if two adjacent bytes appear together in operations of the program, then the two bytes should be grouped into the same field.

Let O_i denote the set of operations affected by byte i of the input. The Input Specification determines this set by inspecting the dynamic taint trace generated by the Execution Monitor. For two adjacent bytes in the input, i and j, let P_{ij}, the probability that byte i and byte j belong to the same field, be given by the Hamming distance between their sets of affected operations:

$$P_{ij} = \frac{|O_i \Delta O_j|}{|O_i \cup O_j|} \tag{4.1}$$

This heuristic derives from the observation that bytes from the same field will affect the same operations in the execution of the program. Heuristics based on reference locality fail to capture fields that are not accessed sequentially. For instance, applications with large, compressed data fields in their inputs have highly non-sequential access patterns; these applications intersperse accesses to the field with accesses to compression tables. On the other hand, this heuristic does not accurately infer fields with bytes that do not affect the same operations. For example, this heuristic will not infer that a null-terminated string is a single field. However, for the benchmarks and results presented in Chapter 5, this shortcoming is not a problem in practice.

4.3 Fuzzer

The Field Classifier's ability to determine control and payload distinctions depends on the Fuzzer's ability to systematically explore the behavior of the application. As discussed in Chapter 2, Chaos first establishes a baseline for the behavior of an application by monitoring the behavior of the application as it executes on the original input. Chaos then explores the behavior of an application by using a Fuzzer to create a suite of similar inputs. Chaos then observes the behavior of the application for each of the similar inputs. This approach relies on the Fuzzer's ability to generate perturbed inputs that exercise different behaviors in the application.

The Fuzzer uses the Input Specification Generator's specification to reduce the search space of perturbed inputs. For each field in the specification, the Fuzzer creates a new perturbed input by selecting a new value for the field. The Fuzzer samples values for the field randomly. This technique is know as *random fuzzing.*

Random fuzzing may not find values that trigger new behaviors. As discussed in Section 5.4, random fuzzing will not likely produce a behavior if the behavior is triggered by only a small number of values in the entire domain of values for a field. To address this issue, the Fuzzer first selects an *extremal value* for a field. An extremal value is a value with all its bits set to zero or one. Extremal values have been shown to push applications towards outlier behaviors [13].

4.4 Field Classifier

As described in Section 3.4, the Field Classifier classifies each field in the input specification along two dimensions: control or payload and critical or benign. Section 4.4.1 discusses how the Field Classifier classifies each field as either control or payload. Section 4.4.2 describes how the Field Classifier classifies each field as either critical or benign.

4.4.1 Classification of Control and Payload Fields

As described in Chapter 1, Chaos classifies input fields that heavily influence the behavior of the program as control; input fields that do not heavily influence the behavior the program are payload. For each field in the input specification, the Field Classifier uses the behavioral distance between the execution of the application on the original input and the execution of the application on an input with a perturbed value for the field to quantify how much a field influences the behavior of the program.

For a run of the application, i, let B_i denote the set of executed basic blocks. The Field Classifier determines this set by inspecting the branch events of the execution trace. Let D_{ij}, the behavioral distance between runs i and j of the application, be given by the Hamming distance between the two sets of basic blocks:

$$D_{ij} = \frac{|B_i \Delta B_j|}{|B_i \cup B_j|} \tag{4.2}$$

Although Hamming distance makes no considerations for the sequence in which the basic blocks were executed, the results presented in Chapter 5 prove that Hamming distance effectively identifies large changes in the behavior of the application.

After computing the behavioral distance for each field of the input specification, the Field Classifier uses hierarchical agglomerative clustering to cluster the fields based on their behavioral distance.

Hierarchical agglomerative clustering is a well-known clustering technique[30] . The hierarchical agglomerative clustering algorithm first assigns each field to its own cluster. The Field Classifier adds an additional dummy field that has a behavioral distance of 0. The algorithm then iteratively merges clusters that are close together. This iterative process produces a dendrogram, or binary tree, where each node of the tree represents a cluster that was produced during the merging process. Each leaf node represents a cluster containing a single field while the root node represents a cluster containing all the fields. The Field Classifier selects the two children of the root node as its control and payload field clusters. Input fields that are grouped into the same cluster as the dummy field are classified as payload while input fields in the other cluster are classified as control.

4.4.2 Classification for Critical and Benign Fields

As described in Section 3.4, the Field Classifier classifies fields that contribute heavily to the output of the application as critical; fields that do not contribute heavily to the output of the application are benign.

As discussed in Section 5.4, the contents of control fields do not, in general, contribute directly to the operations involved in the computation of the output. Instead, their contribution to the output is captured by the control dependencies of the application. Because the Execution Monitor does not track control dependencies, the Field Classifier cannot directly calculate their contribution to the output. However,

the results presented in Chapter 5 indicate that large changes in the behavior of the program typically do lead to large changes in the output. Therefore, the Field Classifier classifies all control fields critical.

For each payload field in the input specification, the Field Classifier uses output dependence to estimate a field's contribution to the output of an application. The Field Classifier determines output dependence by inspecting the dynamic taint trace. The Field Classifier classifies all payload fields that have a output dependence greater than .1 as critical. Payload fields with a output dependence less than .1 are classified as benign.

4.5 Code Classifier

The Code Classifier classifies each executed basic block in the application. The Code Classifier inspects the dynamic taint trace of the application to determine which basic blocks in the application reference which fields of the input. For each basic block, the Code Classifier collects the field classifications for the fields that the basic block references. Basic blocks that reference fields of a single, dominant classification are given that dominate classification. Blocks that do not have a single, dominate classification are classified as mixed usage.

Chapter 5

Evaluation

As described in Chapter 1, Chaos takes an application and an input to the application and produces the following:

- An input specification that groups the bytes of the flat, binary input into semantic fields.
- For each field in the specification, a classification of whether that field is critical or benign to the behavior and output of the application.
- For each executed block of code in the application, a classification of whether that block is critical or benign to the behavior and output of the application.

This section provides an experimental evaluation of the quality of Chaos's field and code classifications for five benchmarks. To evaluate the quality of these classifications, I performed an experiment to compare the quality of Chaos's automated classifications to the classifications of an application-specific test oracle. For each benchmark, I derived the classifications of the application-specific test oracle as follows:

1. I ran the system over five inputs to produce field and code classifications.
2. For each input, I generated n new perturbed inputs where each input differs at a single byte. I then ran the application on the n perturbed inputs to produce n perturbed outputs.

3. For each perturbed output, I used the application-specific test oracle to determine how much the perturbed output differs from the original, unperturbed output. The test oracle takes the unperturbed output and a perturbed output and produces a metric that describes the distortion between the two outputs. For this experiment, I used a normalized Hamming distance calculation on the bytes of the two outputs. This metric is a value on the interval $[0, 1]$; 0 means that the two outputs are identical and 1 means that the outputs differ at all bytes.

4. I set a distortion threshold of .1 and classified all bytes with a distortion above the threshold to be critical.

To compare the quality of Chaos's field classifications, I check the number of classifications on which Chaos and the application-specific test oracle agree. Section 5.1 discusses the benchmarks used in the experiment, Section 5.2 discusses the classification results for each benchmark's fields, and Section 5.3 discusses the classifications results for each benchmark's code.

5.1 Benchmarks

name	description	size
libjpeg	image file format	35 KLOC
libpng	image file format	25 KLOC
giflib	image file format	5 KLOC
imlib2	image manipulation library	39 KLOC
zlib	compression file format	11 KLOC

Table 5.1: Benchmarks

Table 5.1 lists the benchmarks considered for evaluation. The PNG, JPEG, and GIF benchmarks are 3 widely used image file formats. The zlib benchmark is a widely used compression library that is the basis of PNG's image compression. The imlib2 image manipulation library gives me a common interface with which to exercise the behavior of an example program over each image file format. The example program

benchmark	critical control		critical payload		benign payload	
	correct	missed	correct	missed	correct	missed
png	1458	0	8122	18	451	5
gif	54	5	6897	1412	2149	18
jpeg	695	0	4428	1831	542	27

Table 5.2: Byte classifications for image benchmarks

for this experiment takes an input image and converts it to the bitmap file format (BMP). Using a common front-end across multiple benchmarks allows me to also report Chaos's classifications for the code within the front-end (Section 5.3.4)

5.2 Field Classifications

Table 5.2 presents Chaos's results for determining the critical control, critical payload, and benign payload classifications for each benchmark. As discussed in Section 5.4, Chaos currently does not make a distinction between critical and benign control fields; Chaos marks all control fields as critical. A manual inspection of the inputs available for test revealed that none of the inputs had benign control fields. Therefore, I do not report the results for this classification as it is not applicable.

Under each classification type listed in in Table 5.2, there are two columns. The first column, labeled "correct", presents the number of bytes on which Chaos's classification agreed with that of the test oracle. The second column, labeled "missed", presents the number of bytes on which Chaos's classification did not agree with that of the test oracle.

In general, Chaos performs well on critical control and benign payload classification. Chaos's accuracy for critical payload bytes is significantly lower than its accuracy for critical control and benign payload bytes. As discussed in Section 4.4, Chaos classifies an input field as critical if, according to the dynamic taint trace, it affects more than 10% of the operations in the calculation of the output. However, the accuracy of this classification scheme depends on how the program computes the output. For example, the JPEG benchmark uses bytes of the input in pointer arithmetic (i.e. array accesses). Because Chaos does not track the indirect data

dependencies induced by pointer arithmetic, the output dependence calculation for these bytes underestimates the contribution of these bytes to the output. Therefore, Chaos incorrectly classifies 30% of JPEG's critical payload bytes as benign.

Figures 5-5, 5-6, and 5-7, depict the test oracle distortion, behavioral distance, and output dependence for the PNG benchmark. Figures 5-8, 5-9, and 5-10, and Figures 5-11, 5-12, and 5-13 present the same for the GIF and JPEG benchmarks, respectively. A comparison of the graphs for each benchmark reveals how Chaos's uses behavioral distance and output dependence to model different aspects of the test oracle's observed distortion .

- *Test Oracle Distortion:* The first graph in each series plots test oracle distortion on the y-axis as a function of the position of the corresponding perturbed input byte on the x-axis. In these graphs, each region of the input exhibits one of the following types of distortion:

  1. *Complete Distortion:* the program either produces no output or an output that is different at all bytes. In these regions, test oracle distortion equals one.
  2. *Partial Distortion:* the program produces an output that is different at some, but not all, bytes. In these regions, test oracle distortion is greater than zero but less than one.
  3. *No Distortion:* the program produces exactly the same output. In these regions, test oracle distortion is zero.

  Chaos's uses behavioral distance and output dependence to model complete distortion, partial distortion, and no distortion.

- *Behavioral Distance:* The second graph in each series plots the magnitude in behavioral change observed by Chaos on the y-axis as a function of the position of the corresponding perturbed input byte on the x-axis. These graphs validate the three hypotheses about application behavior that motivated Chaos's design:

1. *The behavioral space of an application is bifurcated between large and small deviations in the behavior of the application.* In each graph, there is a large gap in the behavioral distance between regions of high behavioral distance (greater than .4) and regions of low behavioral distance (less than .4).

2. *Large deviations in the behavior of an application lead to large deviations in the output.* For every region of the input with a behavioral distance greater than .4, the test oracle observes a near-complete distortion of at least .9 (90% of the output's bytes are different). Chaos classifies these regions as critical control.

3. *Behavioral distance does not account for all distortion.* In each graph, there is large range of the input over which the program does not behave differently, but the test oracle still observes partial distortion. Chaos classifies these regions as payload and then uses output dependence to determine if they are critical or benign.

- *Output dependence:* The third graph in each series plots output dependence as a function of each byte in the file. With the exception of the JPEG benchmark, test oracle distortion directly correlates with output dependence in regions of the input where the test oracle observes partial distortion but the application does not behave differently. If Chaos observes that output dependence of an input region exceeds .1, then Chaos classifies that region as critical payload. Otherwise, Chaos classifies the region as benign payload because it neither influences the behavior nor the output of the application.

In general, Chaos uses an accurate and intuitive model of test oracle distortion to classify fields. A critical control classification means that the application's behavior and output critically depend on the value of the field. A critical payload classification means that the field does not influence the behavior of the application but the quality of the output critically depends on the value of the field. A benign payload classification means that neither the application's behavior nor the quality of the output depends on the field.

5.3 Code Classifications

Chaos provides a critical control, critical payload, or benign payload classification for each basic block of an application. To investigate the validity of these classifications for my benchmarks, I performed a manual inspection of each executed function to determine if there was a single classification for each of its blocks. If a function had a mixture of classifications among its blocks, I recorded that function as "mixed usage." In general, Chaos's code classifications are correct and each classification attaches a distinct behavioral role to each function.

- *Critical Control:* functions that maintain the application's data structures or select which features of the application to invoke.
- *Critical Payload:* functions that directly compute the output.
- *Benign Payload:* functions that implement irrelevant functionality; these functions do not affect the final output of the application.
- *Mixed Usage:* functions that are used as utilities in the program. These functions appear in multiple contexts and, in each context, touch a different type of data.

Sections 5.3.1, 5.3.2, 5.3.3, 5.3.4 discuss concrete examples of Chaos's code classifications for each benchmark.

5.3.1 PNG and zlib

Critical Control: Figure 5-1(a) lists the PNG and zlib functions that Chaos classified as critical control. The functions prefixed by `png_handle` and `png_set` each implement a particular feature of the specification. `png_handle_IHDR`, and `png_get_IHDR`, read a PNG file format's initial, mandatory header and allocate the library's initial data structures. The `png_handle_PLTE`, `png_set_PLTE`, and `png_expand_palette` functions allocate and build the palette for a paletted image. For both the header and palette functions, if any data corruption occurs, the function will be unable to

function name
png_memcpy_check
png_handle_IHDR
png_set_IHDR
png_get_IHDR
png_read_image
png_do_expand_palette
png_do_read_transformations
png_do_read_filler
png_handle_PLTE
png_set_PLTE
png_set_tRNS
png_handle_tRNS
png_handle_tRNS
png_read_start_row
png_read_filter_row

(a) Critical Control

function name
inflate_table
inflate_fast
inflate
png_read_row
png_read_finish_row
updatewindow

(b) Critical Payload

function name
png_set_tIME
png_handle_tIME
png_handle_gAMA
png_set_gAMA
png_set_gAMA_fixed
png_read_end
png_set_pHys
png_handle_IEND
png_handle_pHYs

(c) Benign Payload

function name
png_crc_read
png_crc_error
png_get_uint_31
png_read_data
png_default_read_data
png_check_chunk_name
png_read_chunk_header

(d) Mixed Usage

Figure 5-1: PNG and zlib code classifications

continue in processing the input. `png_memcpy_check` uses critical data about the length of various fields to allocate memory buffers. Corruptions to the data that flows to `png_memcpy_check` may violate implicit invariants about the intended size of the allocated buffer and lead to uninitialized reads or out of bounds writes.

Critical Payload: Figure 5-1(b) lists the PNG and zlib functions that Chaos classified as critical payload. These functions are responsible for decompressing the image's main content. `png_read_row` and `png_read_finish_row` direct compressed image data from the input file to zlib's `inflate_table`, `inflate_fast`, and `inflate` functions. These functions touch little control data and spend the majority of their time

function name
DGifGetLine
DGifGetImageDesc

(a) Critical Control

function name
DGifDecompressLine
DGifDecompressInput

(b) Critical Payload

function name
DGifGetWord

(c) Mixed Usage

Figure 5-2: GIF code classifications

funneling and uncompressing raw payload data. These functions are critical because the entirety of the application's output is directed through these functions. Therefore, small errors in these functions can lead to global perturbations in the output.

Benign Payload: Figure 5-1(c) lists the PNG and zlib functions that Chaos classified as benign payload. These functions implement optional functionality in the specification. The example program requires a very small portion of the total functionality offered by the PNG library. Because of this, timestamps (`png_set_tIME` and `png_handle_tIME`) and physical device parameters (`png_set_pHys`) neither change the behavior of the example program nor effect changes in the output. Faults in these regions of code will not compromise the output of the program provided that the program can still execute through the fault. `png_read_end` and `png_handle_IEND` handle the final, delimiting end chunk. Though the PNG specification mandates that a PNG file be delimited by an end chuck, the PNG library will still return an output if reaches the end of the input and does not find an end chunk.

Mixed Usage: Figure 5-1(d) lists the PNG and zlib functions that had a mixture of classifications. These functions are common utility functions. The PNG library computes a checksum for every chunk in the PNG data stream. Therefore, `png_crc_read` and `png_crc_error`, which are responsible for checksum checking, will touch every piece of data in the input. `png_read_data`, `png_default_read_data`, and `png_get_uint_31` are file reading utility functions.

5.3.2 GIF

Figure 5-2 lists the code classifications for the GIF benchmark. Chaos reports a small number of functions for the GIF benchmark because it is a simple format that does not provide the same level of extensibility as PNG and JPEG. GIF's critical control functions are responsible for parsing the image description header and coordinating the movement of uncompressed output to destination buffers. Both of these tasks require manipulating the critical data associated with the image's dimensions and encoding. Like the PNG file format, GIF's decompression functions are critical payload and faults in these areas will lead to corruptions in the entire image. GIF's single mixed usage function is a utility function that reads raw bytes from the input.

5.3.3 JPEG

Chaos's classifications for the JPEG library mirrors the results of its analysis for the PNG library. Figure 5-3 shows that critical control data tends to flow to functions that implement functionality related to either the structure of the input or the initialization of data structures and processing parameters. `allocate_large` and `allocate_sarray` use the image dimensions to allocate buffers for decompression. `jinit_d_main_controller` initializes the state of parsing engine. In several locations in the library, `jdiv_round_up` is used to round off the image's dimensions. JPEG's critical payload functions are related to decoding and performing image transformations. These functions serve a similar role to the decompression functions found in GIF and PNG and can, similarly, globally perturb the output if they encounter a fault. JPEG's mixed usage code is dedicated to navigating the structure of a JPEG file. These functions touch most of the structural information in the file and also touch large regions of benign data.

5.3.4 imlib2

Figure 5-4 lists the few imlib2 functions used by the example program. To load an image, imlib2 dynamically loads a parsing module for the image format. The

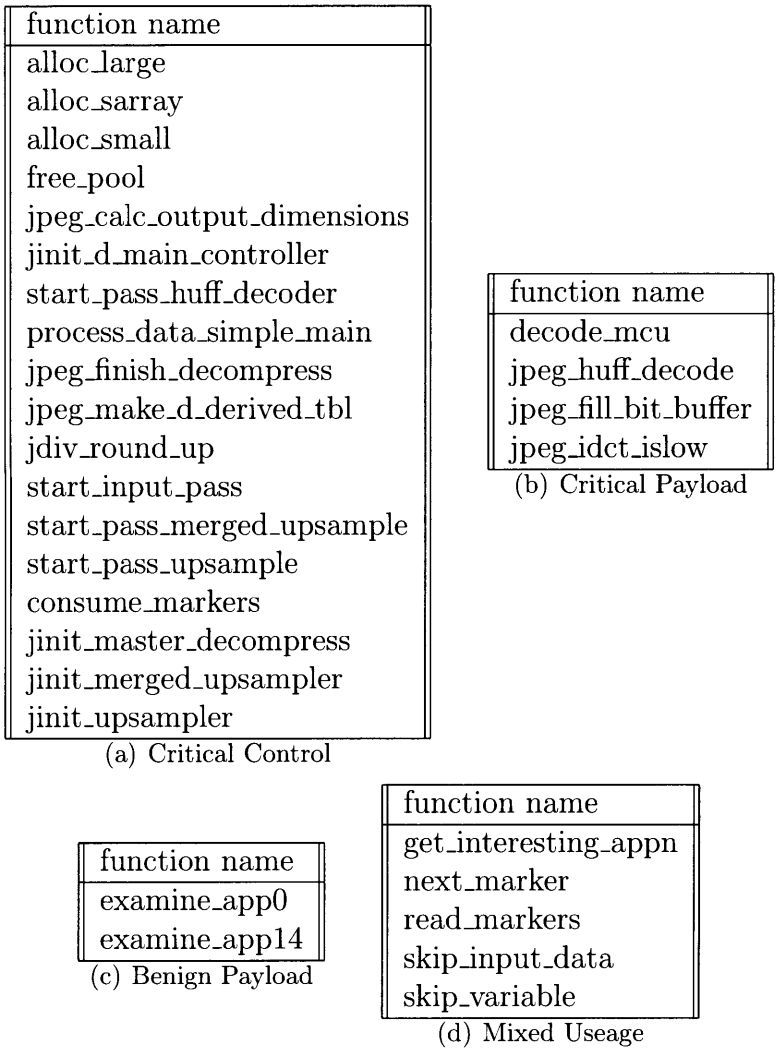

Figure 5-3: JPEG code classifications

module delegates the majority of the image loading work to the format's reference implementation. As a result, the example program exercises only a few of imlib2's functions. `imlib_LoadImage` is the main API function used to load images. It is critical control because it checks certain validity constraints on the image and if these checks fail, the program produces no output. The `load`, `WriteleByte`, `save`, and `imlib_image_query_pixel` functions all reference a combination of critical and payload data. `imlib_image_query_pixel` obtains a mixed use classification because it uses control data to index into the input's critical payload data.

function name
imlib_LoadImage
WriteleLong

(a) Critical Control

function name
load
WriteleByte
save
imlib_image_query_pixel

(b) Mixed Useage

Figure 5-4: Imlib2 code classifications

5.4 Discussion

This section discusses a number of the issues and limitations of my experimental methodology and the quality of Chaos's classifications. The experimental methodology I adopted is designed to present Chaos's classifications about application-specification behavior as objectively as possible. The quality of Chaos's classifications depends on a number of limitations in its technical components. These limitations are due to a need to find a trade-off between engineering effort and classification quality for a number of technical problems that are in active areas of research.

5.4.1 Methodology

Multimedia binary formats maintain well-defined specifications that give precise descriptions of file structures, data types, and supported features. While these specifications are concise, their interpretation within a particular application may be ambiguous. As shown in the results for my benchmarks, an application may ignore parts of a specification that have no meaning within the application's context. In the presence of application-specific interpretations of a standardized format, automatically checking the quality of Chaos's distinctions against a known standard would be unreasonable as the result of such an experiment would be subject to the application's interpretation of the standard. Instead, I have adopted a methodology that checks Chaos's classifications against an application-specific test oracle.

5.4.2 Benchmark Selection

The PNG, GIF, and JPEG file formats are interesting benchmarks to study because the reference implementations for these file formats will, in certain circumstances, ignore the inconsistencies of an input and still generate an output. While the quality of this output is entirely dependent on the nature and location of the inconsistency, these applications allow me to investigate a wide range of their behaviors. Some applications, on the other hand, are designed to not tolerate input inconsistencies. For instance, many binary format implementations use checksums to guard against data corruption. If an implementation chooses to not generate an output if the input's checksum is inconsistent, then Chaos will report that every field of the input is critical control. While this classification is correct, these implementations would not allow me to explore critical and benign payload behaviors.

5.4.3 Input Specification Coverage

Due to the nature of dynamic analysis, the experimental results only hold for the set of inputs considered for the experiment. In particular, judging the quality of inference for input-driven approaches requires a notion of input specification coverage and program code coverage. To address this issue, all inputs for the experiments are taken from a custom-built, automated web spider. This spider crawls the internet looking for files of the specified formats. When the spider finds a candidate input, it computes a content hash and a structural hash. The content hash encodes the exact content of the image and allows the spider to quickly filter identical images. The spider computes the structural hash by parsing the input and hashing the input's parse tree. To ensure that no two inputs exercise the exact same path through the system, I chose each input for the experiment to have a different structural hash.

5.4.4 Random Fuzzing

Chaos's ability to identify large changes in the behavior of the application relies on the Fuzzer's ability to generate inputs. One of the main drawbacks of random fuzzing is

its inability to quickly and reliably enumerate the entire input domain. While Chaos uses automatic field identification to reduce the search space of all possible inputs, it is still subject to random fuzzing's inability to efficiently find rare behaviors. For instance, if a system has a behavior that occurs only if a particular 32-bit integer in the input is one of k values, then random fuzzing has a $\frac{k}{2^{32}}$ chance of producing that value. Incorporating a smart concolic tester, such as Klee, SAGE, or SmartFuzz, would allow Chaos to enumerate these rare conditions more quickly [7, 16, 22].

5.4.5 Lack of Output Dependence Distancing

Chaos currently determines a field's output dependence classification from the execution log of the original, unperturbed input. However, this technique does not account for the fact that perturbing a field may induce large changes in the composition of the output. Namely, a change to a field may mean that a different part of the input is used to compute the output. Dynamic taint tracing imposes a significant overhead on top of basic execution monitoring. For a file of length n, if Chaos is configured to take k samples for each field and the input specification generates a poor field specification, then Chaos may need to execute $k \times n$ inputs. Recording and running this many examples is slow with dynamic taint tracing enabled. Therefore, in the interest of performance, Chaos does not run dynamic taint tracing on perturbed inputs. This decision precludes an output dependence distancing algorithm that can better estimate how drastically the output composition changes. For compressed images, such an algorithm would better capture the existence of indirect dependences that are induced by tainted pointer computations.

5.4.6 Stateless Behavioral Distancing

As described in Section 4.4.1, Chaos uses a Hamming distance metric on the set of executed blocks to compute the behavioral distance between two executions. Hamming distance weighs behavioral distance by code size and does not account for the order in which blocks are executed. While Chaos shows that this can be a good ap-

proximation for behavioral distance, a stateful behavioral distancing algorithm that understands the temporal properties of program execution would be more precise.

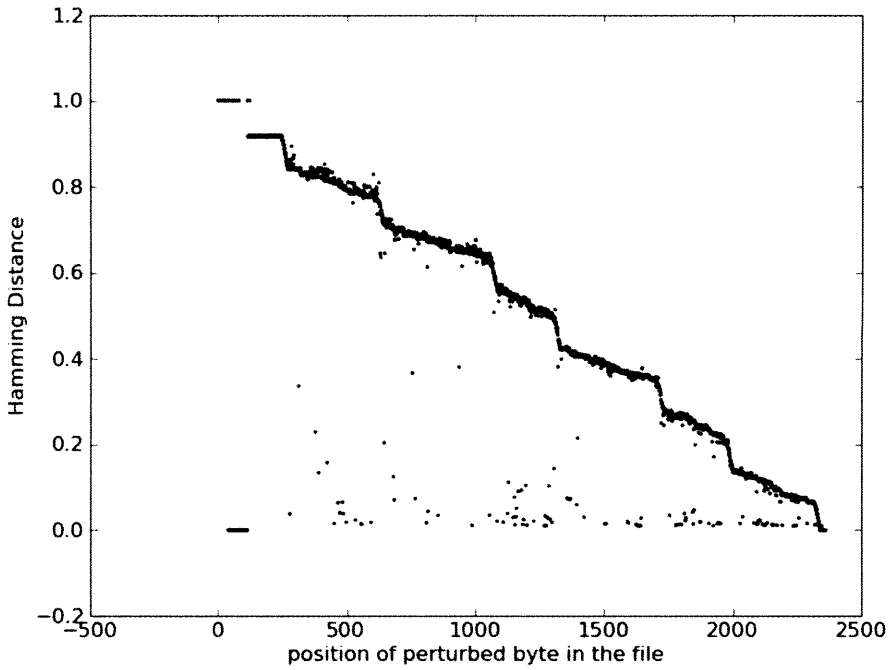

Figure 5-5: PNG Test Oracle Distortion Graph

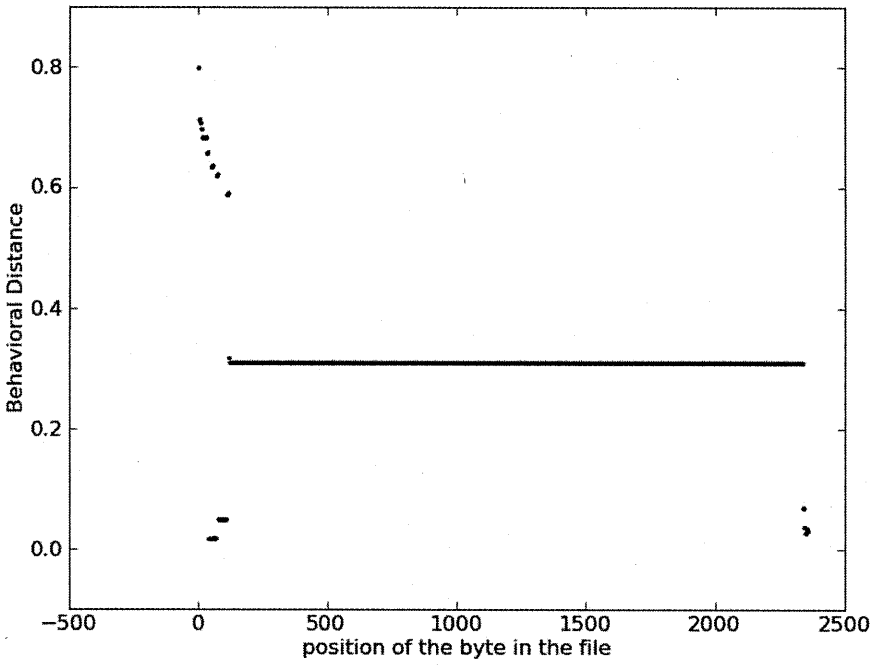

Figure 5-6: PNG Behavioral Distance

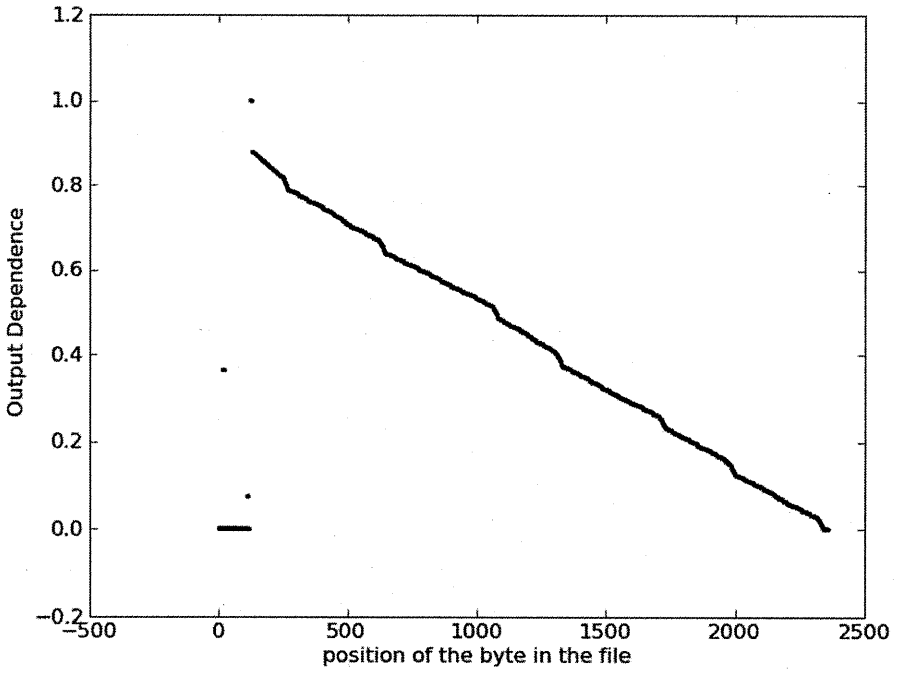

Figure 5-7: PNG Output Dependence Graph

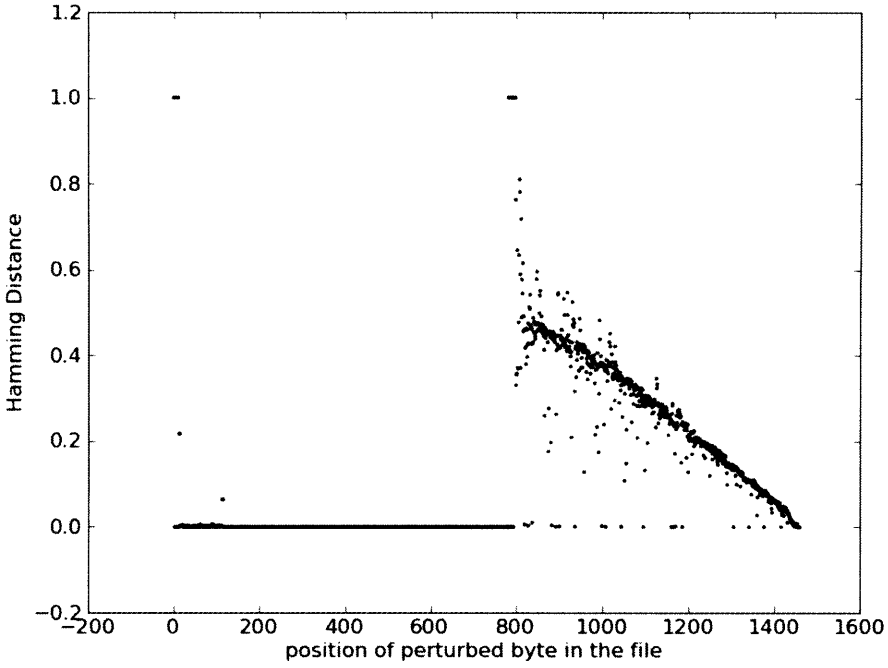

Figure 5-8: GIF Test Oracle Distortion Graph

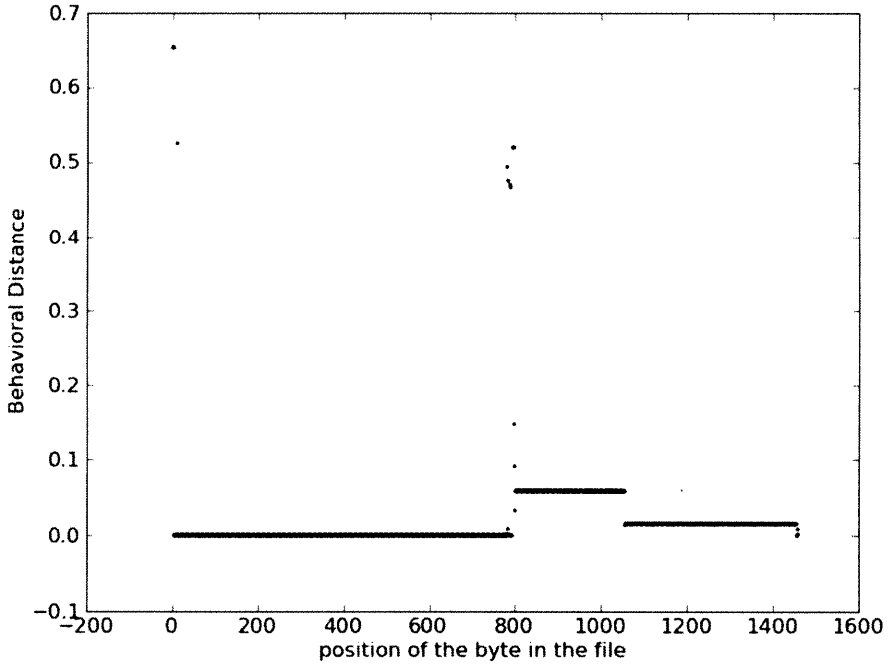

Figure 5-9: GIF Behavioral Distance

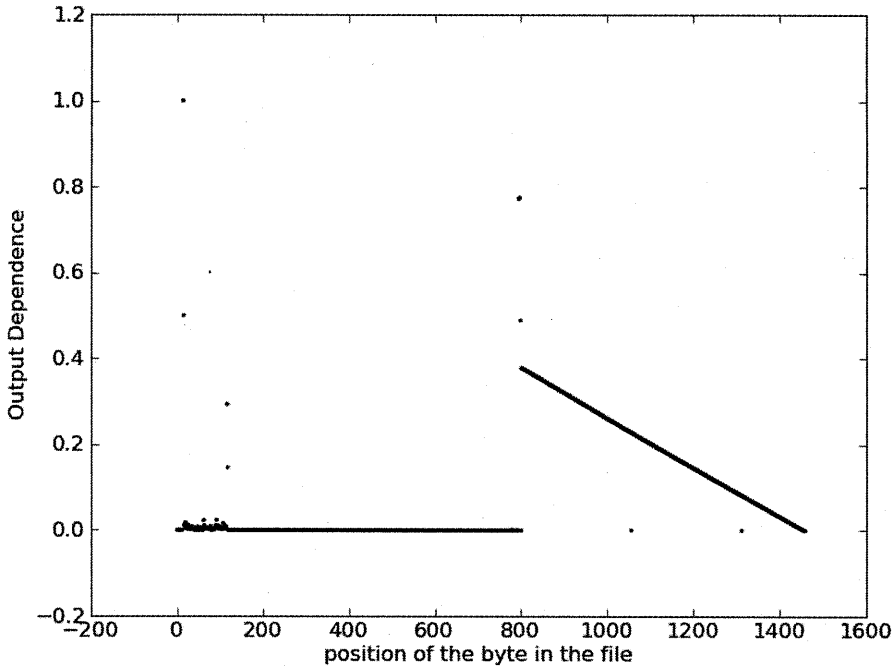

Figure 5-10: GIF Output Dependence Graph

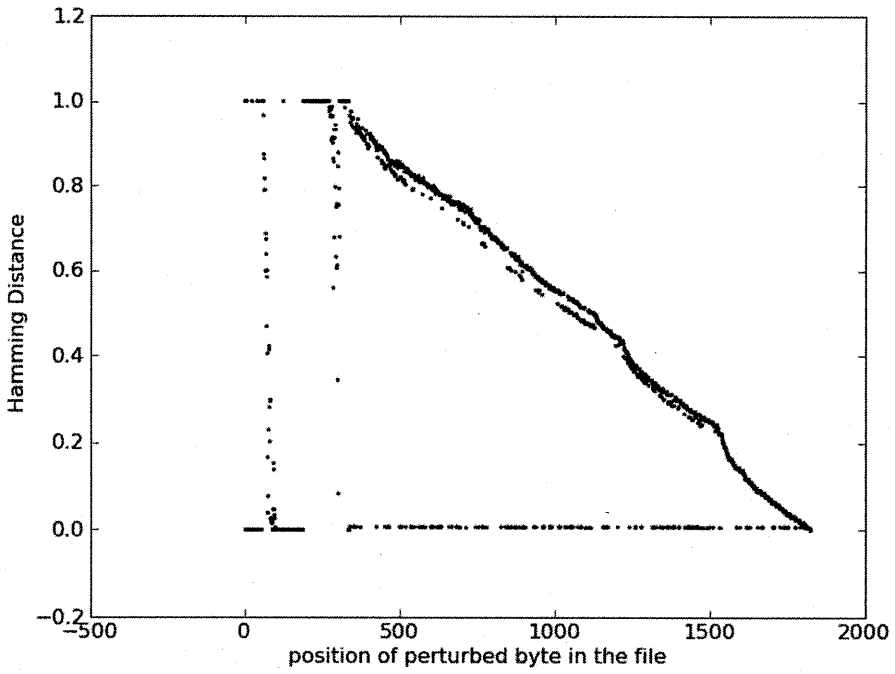

Figure 5-11: JPEG Test Oracle Distortion Graph

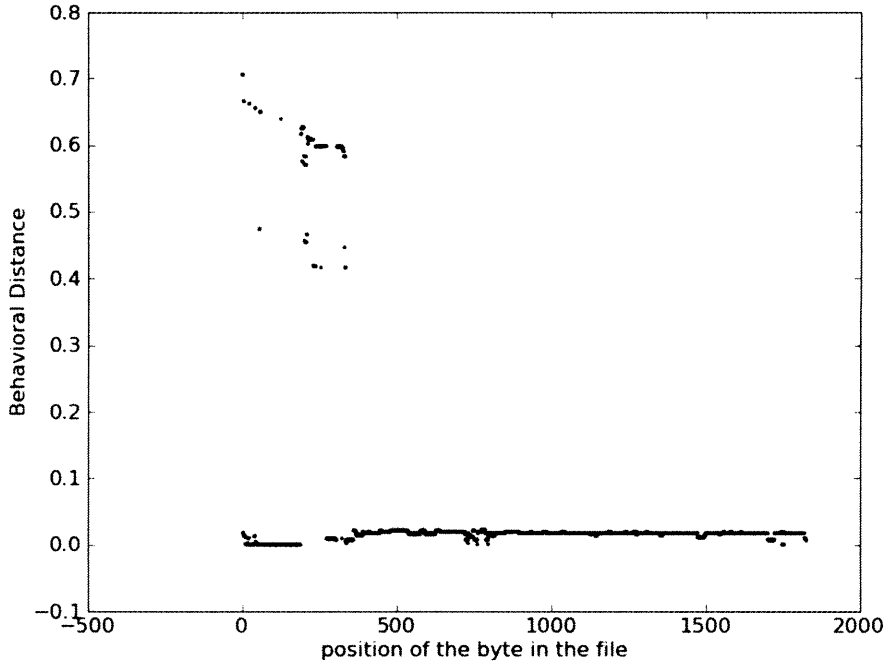

Figure 5-12: JPEG Behavioral Distance

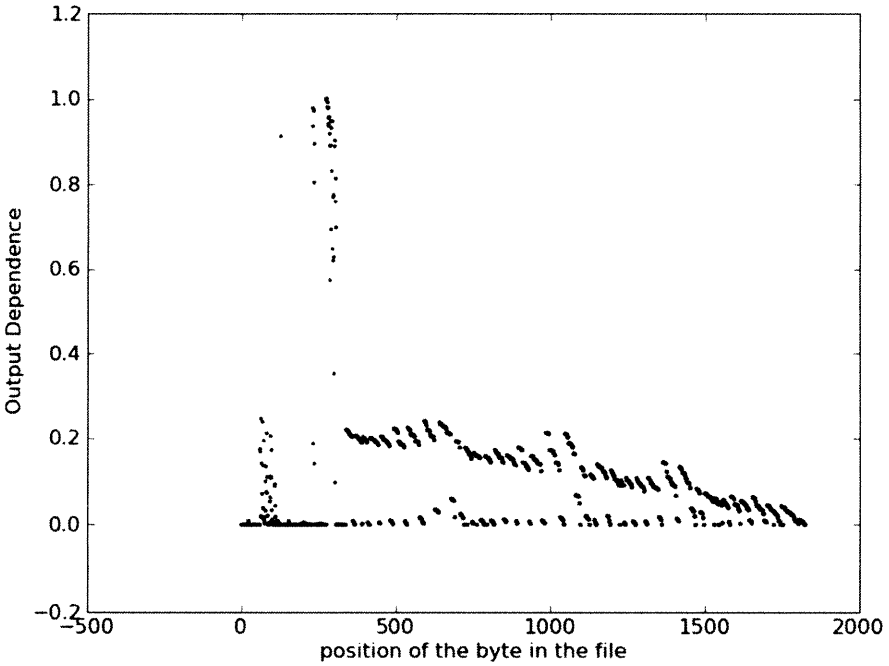

Figure 5-13: JPEG Output Dependence Graph

Chapter 6

Related Work

The distinction between critical and benign regions of an application and its input has long been recognized by researchers. Rinard et al. conducted a study of two applications to determine the applications' *forgiving* and *unforgiving* regions [28]. The forgiving regions of an application are regions of the application's code in which the application can tolerate errors. The unforgiving regions of an application are regions of the application's code in which the application cannot tolerate errors. In their study, the researchers used software fault injection to simulate faults in a video player and an email client. The researchers found that the forgiving regions of the applications corresponded to regions of code that handled the image contents of the video or the text contents of an email. The unforgiving regions of the applications corresponded to regions of code that manipulated the metadata of the video and email. For instance, the researchers observed that a fault inserted into the code that handles the metatdata of a video caused the video player to completely lose track of its place in the input stream.

Pattabiraman et al. have developed Samurai, an implementation of a novel memory model called *critical memory* [26]. Critical memory separates a program's heap into two conceptual heaps: a critical heap and non-critical heap. The critical heap contains objects that are critical to the functionality of the application. A non-critical heap contains objects that are not critical to the functionality of the application. Samurai provides programmers with an explicit API to manually allocate objects in

the critical heap. Samurai then uses object replication and error-correction guard objects allocated in the critical heap from memory corruption.

In their analysis of the utility of the critical memory abstraction, Pattabiraman conducted an experiment in which they compared the failure rate of an application compiled with and without Samurai. In the presence of memory corruptions to critical data, the version of the application built without Samurai suffered failure rates in excess of 50% while the version compiled with Samurai had a failure rate of 0. In the presence of memory corruptions to non-critical data, the application suffered failure rates of approximately 2%. This experiment confirmed their hypothesis that certain pieces of data are more critical to the functionality of the application than others.

Both of these research projects confirm that there exists both critical and non-critical behaviors in an application. Namely, there are regions of code and data on which an application's behavior and output critically depend and there are regions of code and data on which an application's behavior and output does not critically depend. In non-critical regions of data and code, the application can better tolerate memory corruption and programmer error. Chaos compliments the work done by these projects in that it is, to my knowledge, the first system to automatically identify the critical regions of an application's code and inputs.

6.1 Input Specification Inference

A number of researchers in computer systems security are actively working on methods for automatic input specification inference[6, 20, 8, 9]. These approaches are designed to produce input specifications for reverse engineering, testing, and intrusion detection. Like Chaos, these tools use an application's execution trace and dynamic taint trace to infer a specification of an input.

Chaos's generated input specifications are not as detailed as those provided by these tools. These tools are designed to generate expressive specifications that describe nested structures and field alternations. Chaos's specifications are primarily designed to reduce the search space of fuzzed inputs. Although a more expressive

specification could allow Chaos to explore more behaviors of an application, the results of this thesis show that Chaos can already identify critical behaviors.

6.2 Fuzzing

Automated fuzz testing has become an effective means for testing programs. A fuzzing technique can be categorized by the amount of information it needs to know about the program and its input.

Random blackbox techniques require neither access to the source of the program, nor a specification of the structure of the input. A random blackbox tester generates test inputs by either mutating an existing test or generating a test from scratch. Random blackbox testing is known to be an inefficient technique to generate inputs because it will generate many structurally or semantically inconsistent inputs that the application will quickly reject. Random blackbox testing may also generate many tests that exercise the exact same execution path in the program.

Random block-based blackbox and random grammar-based blackbox fuzzing techniques use an input specification to generate inputs. Block-based fuzzers use a specification of the blocks, or fields, of an input to reduce the search space of inputs by fuzzing an input a block at a time[2]. Grammar-based fuzzers use a grammar that gives a specification of the set of legal inputs to an application. While a block-based approach will allow the fuzzer to fuzz multiple bytes, or blocks, of an input at a time, the generated inputs may still have inconsistencies. For example, a block that specifies the length of another field may be inconsistent with the length of the other field if the two fields are not fuzzed together. Grammar-based approaches avoid generating inconsistent inputs by only generating inputs that are, according to the grammar, legal. However, manually developing a grammar for an application may be time consuming.

Whitebox fuzzing techniques generate inputs by symbolically executing an application; this process requires the source or binary of the application. During symbolic execution, a whitebox fuzzer collects the set of conditional predicates, or constraints,

that describe each symbolically executed path in the program. A whitebox fuzzer then solves the set of constraints to produce an input that exercises the path. Whitebox fuzzers can achieve high rates of code coverage if one can afford the large expense of symbolic execution and constraint solving [7, 16, 22, 15].

Other researchers have also proposed hybrid techniques that combine a number of these techniques [13, 25].

Chaos uses random block-based blackbox fuzzing to generate its suite of perturbed inputs. This technique provides an easily implementable way for Chaos to explore the behaviors of an application. Chaos's approach is adequate because Chaos is not designed to find bugs in applications. Oftentimes, bugs are hidden in the subtle, fine-grained behaviors of an application. Chaos, instead, seeks to identify critical control behaviors. As shown in this thesis, these behaviors manifest themselves as large, coarse changes in the behavior of the application. However, Chaos's design is amenable to any sufficiently efficient fuzzing technique. In particular, a whitebox fuzzing technique may allow Chaos to identify fine-grained changes in the behavior of an application.

6.3 Behavioral Distancing

Security researchers have long investigated the notion that the occurrence of an anomalous behavior in a large software system can be indicative of an attack on the system. Researchers have enumerated a number techniques that use execution monitoring (i.e. system call and system call arguments) to build a model of the normal behavior of an application. These models range from simple Hamming distance calculations to stateful, probabilistic models[11, 32, 14, 31, 4].

Software analysis researchers have also used behavioral modeling to find bugs in applications. The CBI project uses a collaborative community to distribute application monitoring over a number of hosts. CBI collects the monitoring information of each host and then uses statistical analysis to correlate bugs with anomalous behaviors in the application[19].

Ammons et al have used the execution traces of library usage to infer specifications about API usage patterns. A specification is given by a probabilistic finite state machine that, when given a candidate trace, can determine the probability that the trace is a valid use of the API. Others have since extended these approaches to infer fine-grained behavioral patterns in large systems [3, 12].

Chaos's goal differs from the mentioned systems in that it is designed to detect large behavioral differences rather than model the likely behaviors of a program. Chaos uses Hamming distance to describe the distance between executions because it provides an easily implementable technique to identify large changes in behavior. In contrast, researchers in these other domains need to model small changes in behaviors because bugs and intrusions are often identified by small changes.

Chapter 7

Conclusion

As discussed in Chapter 1 and shown in Chapter 5, the code and input of a program can be productively classified along two dimensions: control or payload and critical or benign.

Regions of critical control code and data are critical to the behavior and output of the program because errors in these regions can cause the program to invoke incorrect functionality or crash, producing no output. Regions of critical payload code and data are critical to the output of the program because errors in these regions can lead to substantial distortion in the output of the program. In benign control and payload regions, errors may not cause the program to behave differently or produce a substantially different output provided that the program an execute through the error.

Chaos gives developers an automated tool to infer control versus payload and critical versus benign distinctions for a program's code and input. Developers can use Chaos's classifications to more quickly identify the source of errors in the program and, once found, prioritize fixing errors that are potentially more critical to the behavior and output of the application. Although researchers and developers have long understood that some regions of a program and its input are more critical to the behavior and output of an application, to the best of my knowledge, Chaos is the first system designed to automatically identify these regions.

Bibliography

[1] ISO/IEC 15948:2003. *Portable Network Graphics (PNG) Specification (Second Edition)*. ISO, Geneva, Switzerland.

[2] Dave Aitel. The advantages of block-based protocol analysis for security testing. Technical report, 2002.

[3] Glenn Ammons, Rastislav Bodk, and James R. Larus. Mining specifications. In *Symposium on Principles of Programming Language*, volume 37, 2002.

[4] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *IEEE Symposium on Security and Privacy*, 2006.

[5] Derek Bruening and Saman Amarasinghe. Efficient, transparent, and comprehensive runtime code manipulation. 2004.

[6] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *In Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS07*, 2007.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Unassisted and automatic generation of high-coverage tests for complex systems programs. In *In Proceedings of OSDI 2008*, pages 209–224, 2008.

[8] Paolo Milani Comparetti, Wondracek Wondracek, Christopher Kruegel, and Engrin Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, 2009.

[9] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Conference on Computer and Communications Security*, page 11, 2008.

[10] Will Drewry and Tavis Ormandy. Flayer: exposing application internals. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[11] Stephanie Forrest and Thomas A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, 1996.

[12] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. page 10, 2008.

[13] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering*, page 10, 2009.

[14] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral distance measurement using hidden markov models. In *In Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 19–40. Springer, 2006.

[15] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI 2008, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 206–215, Tucson, AZ, USA, June 9–11, 2008.

[16] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. 2008.

[17] Lap Chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. *Computer Security Applications Conference, Annual*, 0:463–472, 2006.

[18] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04)*, 2004.

[19] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Conference on Programming Language Design and Implementation*, volume 40, 2005.

[20] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through conectect-aware monitored execution. In *In 15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[21] Chi-keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[22] David A Molnar and David Wagner. Smartfuzz and metafuzz, 2009.

[23] N Nethercote and J Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *In Proceedings of the 2007 PLDI conference*, 2007.

[24] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[25] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. page 9, 2007.

[26] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai: protecting critical data in unsafe languages. volume 42, page 13, 2008.

[27] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. pages 135–148, Dec. 2006.

[28] Martin Rinard, Cristian Cadar, and Huu Hai Nguyen. Exploring the acceptability envelope. In *Conference on Object Oriented Programming Systems*, 2005.

[29] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. volume 39, 2004.

[30] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition).* 2005.

[31] David Wagner and Drew Dean. Intrusion detection via static analysis. In *In IEEE Symposium on Security and Privacy*, pages 156–168, 2001.

[32] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *In IEEE Symposium on Security and Privacy*, pages 133–145. IEEE Computer Society, 1999.