ENHANCING AUTOMATED FAULT DISCOVERY AND ANALYSIS

By

Jared David DeMott

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science

2012

ABSTRACT

ENHANCING AUTOMATED FAULT DISCOVERY AND ANALYSIS

By

Jared David DeMott

Creating quality software is difficult. Likewise, offensive researchers look to penetrate quality software. Both parties benefit from a scalable bug hunting framework. Once bugs are found, an equally expensive task is debugging. To debug faults, analysts must identify statements involved in the failures and select suspicious code regions that might contain the fault. Traditionally, this tedious task is performed manually. An automated technique to locate the true source of the failure is called *fault localization*.

The thesis of this research is that an automated process to find software bugs and quickly localize the root cause of the failure is possible by improving upon existing techniques. This research is most interested in bugs that lead to security vulnerabilities. These bugs are high value to offensive researchers, and to the typical software test engineer. In particular, memory corruption bugs characterized via an application crash is the subset of all bugs focused on in this work.

Existing distributed testing frameworks do not integrate with fault localization tools. Also, existing fault localization tools fail to localize certain difficult bugs. The overall goal of this research is to: (1) Build a dynamic testing framework powerful enough to find new bugs in commercial software. (2) Integrate an existing fault localization technique into the framework that can operate on code without the requirement of having the source code or pre-generated test

cases. (3) Create a novel fault localization algorithm that better operates on difficult to localize flaws. (4) Test the improvement on benchmark and real-world code.

Those objectives were achieved and empirical studies were conducted to verify the goals of this research. The constructed distributed bug hunting and analysis platform is called *ClusterFuzz*. The enhanced fault localization process is called *Execution Mining*. Test results show the novel fault localization algorithm to be an important improvement, and to be more effective than prior approaches. This research also achieved ancillary goals: visualizing fault localization in a new environment; assembly basic blocks for fully compiled code. A pipeline approach to finding and categorizing bugs paves the way for future work in the areas of automated vulnerability discovery, triage, and exploitation.

ACKNOWLEDGEMENTS

I would like to thank my advisors Dr. Enbody and Dr. Punch for their guidance, encouragement, and assistance. I would like to thank my company, the Crucial Security division at Harris Corporation, for invaluable support. I would like to thank various coworkers, acquaintances, and friends that have encouraged me with personal and professional support; particularly, Dr.'s Pauli and Engebretson of Dakota State University. A special thank you goes to my family, whom without their support this would not have been possible – especially my wife, Michelle. Finally, thank you to God the Father, Son, and Spirit for working in my life and the lives of others to make this degree possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF CODE LISTINGS

CHAPTER 1 INTRODUCTION

A study conducted by NIST in 2002 reports that software bugs[1] cost the U.S. economy $59.5 billion annually [Tassey]. More than a third of this cost could be avoided if better software testing was available. Software can never be completely tested [Lucas 1961]. However, a practical process for continuous improvement is [Lewis 2000]. The improvement process calls for novel or enhanced tools and techniques, which can be practically applied. Distributed dynamic testing frameworks are one practical way researches have been able to scale the automated discovery of software coding mistakes [Gallagher 2009, Nagy 2009, Randolph 2009, Molnar and Opstad 2010, Eddington et al. 2011]. The first portion of this research provides a new distributed framework, which enhances the ability to locate software mistakes.

Debugging located faults is a large portion of the overall cost, as part of the repair of identified bugs [Vessey 1985]. Debugging is traditionally a labor intensive activity where testers or developers seek to identify the erroneous code for which a given input is generating an error. Consequently, researchers seek techniques that automatically help locate the root of bugs for the humans responsible for maintaining the code.

Opposite to software repair, is software exploitation. The tools and techniques to find and understand code mistakes are similar to testing and repair, except that once errors are understood they are weaponized into working software exploits. Countries such as China, Russian, and the

---

[1] A software mistake or defect may be referred to throughout this document as an error, bug, or fault.

United States are actively seeking to create cyber weapons [Clark and Knake 2010, Goel 2011]. The Stuxnet worm is a prime example of a cyber-weapon that leveraged memory corruption bugs [Greengard 2010] in the Microsoft Windows operating system. Therefore, cyber attackers also seek a system that can automatically find and understand security relevant bugs.

Given test cases [Voas and Miller 1992], dynamic code coverage[2] information can be used to aid automatic, root-cause discovery of coding mistakes. This process is known as *fault localization* (FL), and it is an active research area. Many papers on debugging and fault-localization have been published in academic conferences and journals [Pan et al. 1992, Agrawal et al. 1995, Liblet et al. 2003, Renieris and Reiss 2003, Cleve and Zeller 2005, Gupta et al. 2005, Jones and Harrold 2005, Liu et al. 2005, Zhang et al. 2007, Hsu et al. 2008, Jeffrey et al. 2008, Cheng et al. 2009, Santelices et al. 2009, and Artzi et al. 2010]. Despite their work, FL is difficult and needs to be improved upon to see higher utilization [Wong and Debroy 2009].

In particular, existing coverage-based fault localization (CBFL) algorithms fail on certain bug types: Data-only bugs and noisy code bugs. Data-only bugs are those for which measured code coverage does not differ between test cases. Thus, CBFL cannot reliably determine the location of the fault. Noisy code bugs are those which are harder to localize because of noisy coverage information. Noise in this context indicates sporadic and non-relevant code blocks that show up in failed test cases, but not in passing tests. CBFL algorithms keys in on those

---

[2] Code coverage (CC) is a measure commonly used in software testing. It describes the degree to which the source code of a program has been tested. In this case, CC refers to the specific blocks that were executed for a given test.

differences. But if there are many differences, and they do not relate to the bug(s), the CBLF algorithm is led astray and many false locations get reported to the human analyst. The second portion of this research provides an improvement to fault location.

Thesis Statement

The thesis of this research is that popular coverage-based fault localization (CBFL) techniques can be enhanced. The enhancement is done by providing better information about which code blocks are operating on important information. Important information is defined as select portions of untrusted input. The input is tracked throughout a programs execution; this is called tainting. Code blocks that operate on tainted data score in our new CBFL as more suspicious than those that do not. The novel CBFL is also enhanced by pairing it with a distributed testing framework to provide a pipeline of prioritized software bugs.

Contribution

This research is most interested in bugs that lead to security vulnerabilities. These bugs have value to offensive researchers, and to the typical software test engineer. In particular, memory corruption bugs characterized by an application crash are the subset of all bugs focused on in this dissertation. This research also shows that data-only and noisy bugs are a significant portion of the memory corruption subset.

This research provides the following contributions:

1. A powerful distributed fuzzing framework capable of more quickly finding and triaging bugs in large real-world applications. For example, zero-day vulnerabilities were found in Java. A zero-day is a security researcher's term for a new bug, not known to the

vendor, and capable of being translated into a new attack for which no current defense has an exact protection.

2. The combination of fault localization with the distributed fuzzer. To our knowledge we are the first to do this. Furthermore, our system does not depend on source code or pre-generated test cases.
3. Two novel fault localization algorithms:
    a. The first FL improvement cuts through basic noise associated with gathering code coverage information to provide better results.
    b. The more impressive new FL algorithm pairs select input data tracking with coverage-based fault localization. The results show its superior effectiveness on a number of benchmark applications.
4. The test results of the new FL algorithms on two challenging and real-world problems:
    a. The first is a data-only bug in VLC TiVo.
    b. The second is a bug in the Firefox PNG code, which resides in a large and noisy module. In both cases, the new approach significantly outperformed other coverage-based fault localization algorithms.
5. An analysis of the significance of the data-only and noisy subset. We show that our primary novel FL techniques works well for code and data bugs. However, an additional goal was to understand how significant data-flow bugs are, as compared to the relative size of control-flow bugs. We reason and empirically show that data bugs are a significant portion of the studied faults, giving further credence to our work.

CHAPTER 2 SOFTWARE SECURITY TESTING BACKGROUND

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test [Myers 1979]. Security testing is a process to determine that an information system protects data and maintains functionality as intended [Mercedes and Winograd 2008]. The most common subset of security requirements or security goals consists of the following: confidentiality, integrity, and availability (CIA) [Gasser 1988]. To meet those challenges, Microsoft has developed a well-known secure development lifecycle (SDL) [Microsoft 2011].

Secure Development Lifecycle

Creating robust and trustworthy applications is difficult. Figure 1 shows Microsoft's SDL process. Each step of Figure 1 is critical. Quality training raises the security IQ of an organization, and makes security part of the culture. In the requirements stage, threat modeling insures appropriate security measures are implemented on a per application basis. Auditing the design diagrams is the first security and quality control measure. During implementation, programmers should use best-practices and up-to-date build tools. Static analysis upon code check-in provides programmers with instant feedback and accountability. Dynamic, or runtime testing, is first used in the verification phase. This is the ideal time for development organizations to apply our research. Fuzz testing has become a popular dynamic testing technique [Miller et al. 1990]. Before release, a final security review is conducted for most applications. Applications meeting certain threat thresholds receive a manual code review. Finally, plans to maintain, patch, and respond to post-release events is put in place. That is required for bugs naturally found, and because hackers, or legitimate offensive professionals, are

known to implement bug hunting frameworks (such as the one presented in this dissertation) to uncover coding errors after software has been fielded.

Figure 1: The Microsoft Security Development Lifecycle

For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.

Static, Dynamic, and Manual Auditing

Static testing tools are best integrated into the development process. These tools are configurable and can be used to enforce custom rules, such as denying the use of dangerous API functions. Static tools are known to the raise the overall security posture, or baseline security, of a code base. However, static tools often have false negatives and positives. Therefore, dynamic testing is required once the code base is stable. Fuzzing is discussed in the next section. Manual review for high threat applications is desirable, because each type of testing tends to catch errors missed by other types. Manual review is expensive because humans are not as efficient as computers. But highly skilled humans often catch obvious and nonobvious coding mistakes missed by automated techniques.

Fuzzing Background

Fuzzing is a testing technique known for its ability to uncover memory corruption bugs, which tend to have security implications [Takanen et al. 2008]. Hackers and security researchers have used fuzzing for years to find bugs in widely available commercial (closed source) and open source software. Because of its effectiveness, fuzzing is included in many development environments, e.g. Microsoft incorporates fuzzing into their Secure Development Lifecycle [Microsoft 2011].

Fuzzing Basics

Fuzzing can be defined as a: Runtime testing technique where semi-invalid input is delivered while the system under test (SUT) is monitored, often with a debugger, for common exceptions such as memory access violations.

Figure 2 shows the fuzzing methodology from a high level [Oehlert 2005]. In Figure 2, data is created or mutated, and given to the SUT which is monitored. If the SUT crashes, the current context information from the program along with the input data is saved. Fuzzing continues until a stopping point. Random based fuzzers often have no defined stopping point.

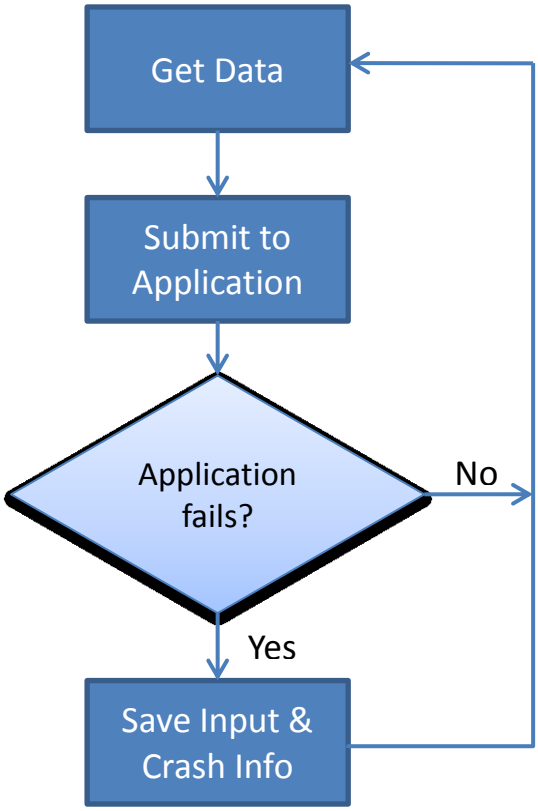

Figure 2: High Level Fuzzing Diagram

Fuzzing can create totally random input. Barton Miller et al. did this over twenty years ago against UNIX utilities, and for graphical user interfaces [Miller et al. 1990, Miller et al. 1995, Miller et al. 2006]. They had much success. They found post-release bugs missed by the

manufactures. However, much of today's fuzzing is less random. Data is either constructed based on samples (mutation fuzzing), or from the protocol specification (intelligent fuzzing).

Fuzzing is mostly concerned with software faults that tend to have security implications. Crash bugs caused by memory access violations are the most common type of bug discovered. These bugs can be easily caught in a debugger, and can often be turned into either a DoS (Denial of Service) or an access tool (software exploit). Because of this, fuzzing would not be an ideal tool for all testing, since there is typically no fuzzing oracle to catch non-crashing bugs. A non-crashing oracle could be constructed as some have done for web application fuzzing. Web fuzzing is discussed in Chapters 9 and 10 of a recent fuzzing book [Sutton et al. 2007]. Finally, there is no completeness guarantee with fuzzing. It is a heuristic technique that has been empirically proven successful.

Mutation Fuzzing

Mutation fuzzing uses data samples as a starting point. Slight mutations are made with each iteration, before the data is delivered to the application. In this way the notion of "semi-valid" data is maintained. That is, to achieve sufficient protocol penetration, most of the input needs to pass the applications basic data sanity checks.

File fuzzing is the most popular type of mutation fuzzing in use today. Samples for client-side applications such as Adobe Reader, Microsoft Word, etc. are collected and used to fuzz the application. Much success has been reported with mutation fuzzing [Sutton 2006, Miller 2010].

Fuzzing with many different samples has been shown to be more effective than using a single sample. Intelligently selecting samples for mutation has been shown to be even more effective and efficient than simply using lots of samples [Molnar and Opstad 2010]. The method for

choosing samples wisely is sometimes referred to as MinSet operation. MinSet refers to finding the minimum set of inputs that cover that maximum amount of code. This increases effectiveness, while decreasing runtime. MinSet is covered later in Chapter 3 as part of Algorithm 1.

Intelligent Fuzzing

Intelligent fuzzing means building a data model that describes the input data. That way, each data type can be mutated in a way that makes sense for the current data element. For example, it is common for binary protocols to expect data in what is called the Type/Length/Variable (TLV) format. To fuzz such a protocol, the data model must specify the number of bytes that is included in each TLV element. Then, when the fuzzer mutates an element, TLV integrity will be maintained. E.g. the length variable will be correctly updated with a new value, if the variable data is made longer or shorter than it originally was. Intelligent frameworks should also be able to correctly handle other complicated data elements such as checksums, hashes, and more.

Two common intelligent fuzzing frameworks are the *Peach Fuzzing framework* [Peach 2011], and *Sulley Fuzzing Framework* [Amini and Portnoy 2010]. Sulley is geared toward network fuzzing. Sully data models are constructed directly in the Python language that Sulley is written in. Peach is a more robust tool, but developing the Peach data models in XML (as required) is considered more difficult. One notable improvement over Sulley is that Peach can leverage a crash sorting plug-in by Microsoft called *!exploitable* (pronounced "bang exploitable") [Exploitable 2011]. !exploitable is discussed further in Chapter 3. Peach also can perform file fuzzing without framework modifications, unlike Sulley.

Researchers have reported success with intelligent fuzzing [Sutton and Greene 2005, Mulliner and Miller 2009]. While intelligent fuzzing is able to handle complex elements like data checksums, some researchers believe that with enough samples and iterations intelligent-like results can be achieved via mutation fuzzing, in sufficiently complex software, without the effort of constructing a data model [Miller 2010, Molnar and Opstad 2010]. Regardless, experts agree that different techniques find different bugs [Molnar and Opstad 2010]. A common recommendation is to use multiple fuzzing approaches, since no one testing technique captures all bugs.

Feedback Fuzzing

Fuzzers that adapt, or create dynamic inputs, based on information from the target are called feedback fuzzers. Evolutionary fuzzing is one such example. Evolutionary fuzzers operate based on a fitness function, such as covered code. As the fuzzer runs, new inputs are breed and delivered using genetic algorithms [McMinn and Holcombe 2006].

One of the first published approaches to evolutionary fuzzing came from our work at Michigan State University[3] [DeMott et al. 2007]. The approach was a grey-box technique (assumes no access to source code, instead using analyzed runtime information). Using a python

---

[3] First place research poster award was given at MSU for this work in 2007.

http://www.cse.msu.edu/?Pg=50&Col=3&Inc=1&Nid=115

debugger to measure covered code blocks, new inputs were generated and delivered that achieved generationally better code coverage to find bugs.

Whitebox Fuzzing

Researchers at Microsoft created a tool called SAGE (Scalable, Automated, Guided Execution), an application of whitebox file fuzzing for x86 Windows programs [Godefroid et al. 2008]. SAGE works by starting with an initial input. This input is then symbolically executed by the program. Information about how the data was used is stored. Constraint (branch condition) information is an important part of the recorded data. Each constraint is negated one at a time until the entire system is solved. This results in a new input to the program which has a different execution path. This is repeated for each constraint in the program. In theory, this technique should provide 100% code coverage for the entire attack surface. In practice, this is not always the case as will be discussed later in this section.

The following is a contrived function, for which SAGE can quickly get complete code coverage, where mutation fuzzers would struggle:

```
void tricky_function ( char input[4] )
{
    int cnt = 0;
    if(input[0] == 'b') cnt++;
    if(input[1]=='a') cnt++;
    if(input[2]=='d') cnt++;
    if(input[3]=='!') cnt++;

    if(cnt>=3) abort();          //This line is the error.
}
```

The above code illustrates a point: by using random inputs, the probability of finding the error is approximately 2^(-30). Therefore, mutation fuzzers would require a sample containing the input "bad!" to follow this code path. Obtaining such a sample may or may not be likely, depending on available samples for this hypothetical protocol.

To continue the example, we walk through how SAGE would generate inputs for each constraint. Suppose the initial input is "root". The resulting constraints are:

{input[0] != 'b', input[1] !='a', input[2]!='d', input[3]!='!'}.

SAGE will systematically negate each of the constraints to get new inputs. Eventually, the following set of constraints will be generated:

{input[0] == 'b', input[1] =='a', input[2]=='d', input[3]=='!'}.

The final solution gives the input "bad!", which finds the bug.

The SAGE technique does have limitations. The most obvious is that there are a very large number of paths in real-world programs. This is called the path explosion problem. Path explosion can be partially mitigated by generating inputs on a per-function basis and then tying the information together. Other factors might prevent SAGE from being able to solve the constraints in a reasonable amount of time. This happens when a difficult to solve condition occurs. Checking cryptologic variables is a good example. Yet another problem arises because symbolic execution may be imprecise due to interactions with system calls and pointer aliasing problems. Finally, the ability of SAGE to quickly generate good inputs relies heavily on the quality of the initial input, much like mutation based fuzzing.

Despite these limitations, SAGE claims to have an impressive history of finding real vulnerabilities in real products. For example, Microsoft states that SAGE was able to uncover

the ANI format animated cursor bug [SDL 2007]. This vulnerability specifically arises when an input is used with at least two ANIH records, and the first one is of the correct size. Microsoft had previously fuzzed this code, but all of their inputs only had one ANIH record. Therefore, they did not find this particular bug (before it was released in the wild). However, given an input with only one ANIH record, SAGE generated an input with multiple ANIH records and quickly discovered this bug. The relevant code contained 341 branch constraints and SAGE required about 8 hours to uncover the bug.

Unfortunately, at the present time, SAGE is not available outside of Microsoft Research. Also, for our work we assume no access to source code information. Assuming no source is a valid assumption for blackbox testing, legacy testing (where source has been lost), and most prominently for offensive research when source code access is not permitted.

Distributed Fuzzing

The state-of-the-art in fuzzing is now focused on distribution or parallelization. Consider a fuzzing engine that generates 800,000 tests, each of which will take 10 seconds to complete. It would require roughly 92 days of runtime on one machine. If 92 machines are used, the same tests finish in one day. An expanded platform based on these notions, created in this research work and called *ClusterFuzz* is later discussed.

Microsoft and Adobe have both made reference to in-house distributed fuzzing tools [Gallagher 2009, Randolph 2009]. At Microsoft, Gallagher speaks about how the Office test

team turned desktop and lab machines into a botnet[4] for fuzzing during downtime. At Adobe, Randolph mentions that the Peach fuzzer [Peach 2011] is used in a distributed fashion, though they give few details on how Peach was extended to start jobs or collect results on virtual machines (VMs). Molnar shows how he uses Amazon Elastic Cloud Computing to create a distributed fuzzer [Molnar and Opstad 2010]. Molnar urges individuals to download and run tests on his pre-built Linux VM. Results are then sent to his website: www.metafuzz.com. Nagy spoke about a *FuzzPark* he created in Ruby to fuzz Microsoft Word. Nagy discovered many crashes, and shared those with Microsoft [Nagy 2009].

---

[4] Botnet is a jargon term for a collection of software agents, or robots, that run autonomously and automatically. The term is most commonly associated with malicious software, but it can also refer to a network of computers using distributed computing software.

CHAPTER 3 ENHANCING VULNERABILITY DISCOVERY

Fuzzing is important to us because it provides a way to find new bugs and apply fault localization to closed-source, i.e. commercial code. However, to be effective as a front-end to fault localization, fuzzing must generate and examine a large number of cases requiring significant compute time. To mitigate this problem, a distributed fuzzer was created and combined with a backend to categorize the many potential bugs such an environment will generate. This framework is described next.

Distributed Fuzzing and Analysis Framework

The distributed fuzzing platform that was constructed to feed the fault localization process is called *ClusterFuzz* (CF). ClusterFuzz is a distributed computing framework that facilitates fuzzing of larger data input sets. It can be used as a stand-alone fuzzer or as a front end to a fault-localization tool. ClusterFuzz does fuzzing in parallel, making fuzzing more efficient, and does not require source code or a test set to generate tests. The speed-up is linear as resources are added to CF. E.g. A fuzzing run that would have taken 200 days on 1 computer, can be done in 1 day on 200 virtual machines in CF.

Once an application has been fuzzed and bug results have been generated, ClusterFuzz proceeds to do further evaluation of the results. First, it examines the generated bugs and attempts to cluster them based on the similarity. Second, it attempts to rate the severity of the observed bugs. Figure 3 shows the high level design for ClusterFuzz. Each of the major sections from the diagram is briefly described below.

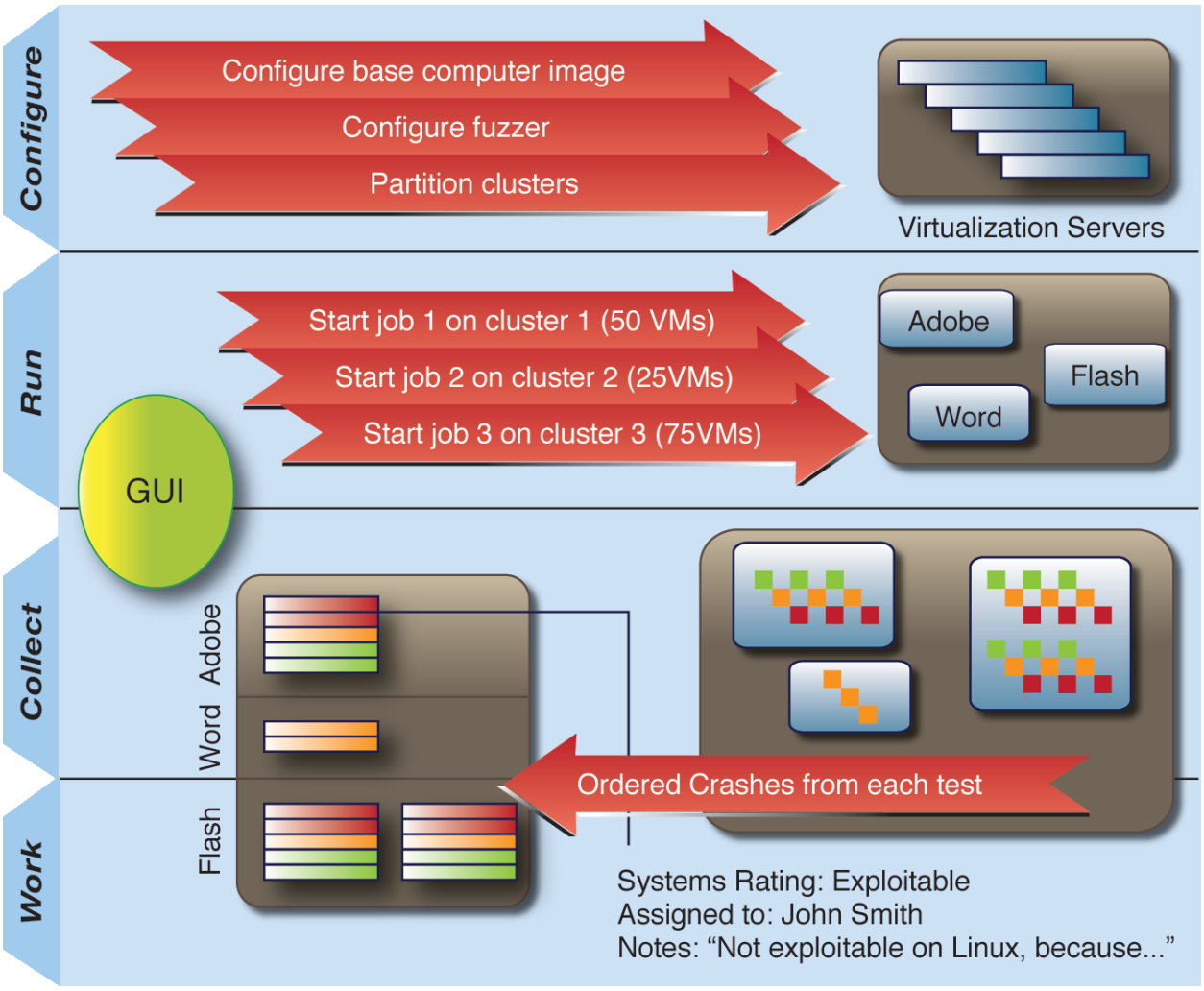

Figure 3: ClusterFuzz

Configure. The first thing the bug hunter does is choose a target to fuzz. That target, a.k.a. the system under test, must then be properly installed in a base virtual machine (VM). Windows XP was used for the base VM operating system (OS) as it is easier to configure and requires fewer resources than Windows Vista or 7. Windows in general was chosen both because the Peach fuzzing tool [Peach 2011], which is used in this work, operates best under Windows, and because many interesting targets to fuzz, such as Microsoft's Internet Explorer, reside only on

Windows. However, it is possible to use a different base OS and a different fuzzer with ClusterFuzz. Next, the test configuration must be constructed specifically for the SUT. The test configuration specifies all the details surrounding the setup of the SUT: input types, execution requirements, etc. The test configuration includes the data model (DM), which describes the protocol required for intelligent fuzzing (if desired). To aid the user, a graphical user interface (GUI) was created, which includes an option to automatically create a simple mutation DM, thus bypassing the manual work of protocol description. The base DM often yields reasonable results because the input samples are intelligently gathered (using Auto-MinSet described below). A common practice is to run the base DM initially, and then later design a custom DM for the particular application if budget allows.

Run. The present setup has six VMware ESX 4.0 virtualization servers. Based on current hardware, each of the servers can handle approximately 25 virtual machines cloned from the base image. Therefore, 150 VMs are running on hardware that cost $30,000 USD when procured (October 2009). Each virtual machine has generous amounts of RAM and CPU cycles. These VMs may be used for a variety of purposes such as:

- Auto-MinSet is a technique created as part of this research to find good input samples from the Internet for a given file format protocol. Auto-MinSet uses code coverage to find an optimal test set from samples, which were automatically searched for and downloaded. The method for choosing samples wisely is referred to as MinSet, since the goal is to select the minimum set of files that obtains that maximum amount of code coverage. The MinSet subroutine in Algorithm 1 provides the details. Briefly described here, MinSet:

1. Gathers a list of inputs and locates the input that covers the most code.
2. Next, the covered regions of each input are tested against the pool of regions covered thus far. If new regions are covered, the input is deemed worthy to be added to the minimum set of desired inputs. Otherwise, that input is discarded.

- Auto-Retest, which checks the reliability of discovered bugs.
- Execution Mining (described later).
- Fuzzing.

Collect. Once the fuzzing sessions have completed, the ordered (from !exploitable, described in the next section) bug results are collected. The CF GUI provides a convenient way for analysts to view each bug and create notes as they work through the pipeline of results. CF is described again in Algorithm 1:

Algorithm 1. ClusterFuzz: Bug Hunting and Analysis System

Input: A test configuration controls the fuzzing of each system under test, and is also typically used to automatically fetch good inputs. Each data sample in the collected set of *Inputs* is fuzzed to create a broad set of test inputs. Fuzzed permutations of an input *i* are denoted as *i´*. The most common EXCEPTION sought is a memory access violation.

Output: Reliable, severe, and unique bugs (defined shortly) are sorted (in bins) in the set *Results*, and ready to be later used by Execution Mining

---

Inputs=DownloadSamples(*Test_Configuration*);

Inputs=MinSet(*Test_Configuration*);

for each input *i* in *Inputs* **do**

for each fuzzed permutation *i´* of *i* **do**

```
            result, exception_data = Test(Test_Configuration, i´);
            if(result) == EXCEPTION
            then
                logs=Log(exception_data, i´, i)
            end
        end
    end

    Results = Combine_Bugs_from_VMs (logs);
    Results = Order_Within_Bins(Results);
    Results = AutoRetest(Results);

Subroutine MinSet(Inputs):
  largest, coverage = set();
  CC = dictionary{ input: BasicBlocks}
  for each input i in Inputs do
    CC[i] = Get_Set_of_BBs_Executed(i);
  if length(largest) < length(CC[i])
    then
        largest = CC[i];
    end
  end
```

minset = [ ];

coverage = *largest*;

for each input *i* in *Inputs* **do**

if any of *CC[i]* not in *coverage*

then

minset += *i*;

coverage.update(CC[i])

end

end

Return *minset*

Initial Results for the Fuzzer Component

To demonstrate the robustness of ClusterFuzz, it was tested on a variety of commercial, closed-source software. The following are applications and data formats that have been fuzzed using ClusterFuzz:

- Client-side applications
  - Browsers
    - Internet Explorer, Chrome, Firefox, Safari, Opera
  - Office Applications
    - Writer (Open Office), Word (Microsoft Office), Adobe Reader, Adobe Flash Player, Picture Manager (Microsoft Office)
  - Other
    - iTunes, QuickTime, Java, VLC Media Player, Windows Media Player, RealPlayer

- File formats:
  - Images:
    - JPG, BMP, PNG, TIFF, GIF
  - Video:
    - AVI, MOV
  - Office:
    - DOC, DOCX, XLS, XLSX, ODT
  - Adobe:
    - PDF, SWF

Statistics were gathered over one month of running CF on the previously listed applications, providing many of the file formats as input. Not every combination produced a fault, but when faults were noted they were recorded as collective results, which are as follows:

- 141,780 faults total
  - faults/day: 4726
  - faults/hour: 197
- 828 total unique fault bins
  - 17 "Exploitable" bins
  - 6 "Probably exploitable" bins
  - 0 "Probably Not exploitable" bins
  - 805 "Unknown" bins
  - Unique fault bins/day: 28
  - Unique bins "probably exploitable" or "exploitable" /day: 0**.9**

Many observed faults are actually manifestations of the same fault, which is why the word "bin" is used above to show each grouping of bugs. Classifying faults is important to reduce downstream effort and a tool exists to do just that: !exploitable [Exploitable 2011].

The !exploitable tool is a Windows debugging (Windbg) extension that provides automated crash analysis, security risk assessment and guidance. The tool uses a hash to uniquely identify a crash and then assigns one of the following exploitability ratings to the crash: Exploitable, Probably Exploitable, Probably Not Exploitable, or Unknown. In an attempt to identify the uniqueness of each bug, two hash types are provided: major and minor. The difference is that the minor hash hashes more of the stack information, and therefore attempts to provide a finer amount of detail when sorting bugs.

The data shown above helps illustrate how !exploitable works, and the importance of filtering in fuzzing. While there may be many crashes, often there are many fewer unique and security-critical bugs. These high quality bugs are what developers and bug exploiters usually want to focus on. In this research, high quality (HQ) is defined as: reliable (repeatable when retested) and severe (a rating of Probably Exploitable or higher).

On top of the classification provided by !exploitable, CF also collects and stores relevant crash information such as the registers used and nearby disassembly code in the database. This information allows researchers to later search for particular register patterns as they become widely known. For example, when the Intel registers ECX and EIP contain the exact same value that is a condition that may indicate the presence of an exploitable C++ structured exception handler (SEH) overwrite.

The !exploitable output provides a rough sorting mechanism according to type and severity. However, by looking back at Figure 3, notice that each unique grouping of bugs must still be analyzed in the final section marked “work.” Further information about the crash such as the location of the actual erroneous code block (fault localization) and other metadata (visualizing

blocks covered, etc) can expedite the analysis process. The next section discusses discovered bugs and visualization.

New Vulnerabilities Discovered

To show that the system finds new real-world bugs (sometimes called zero-day vulnerabilities), multiple previously undisclosed flaws were identified in the latest java version (1.6.0_25) when fuzzing the Java picture parsing routines by using sample JPEG pictures. The flaws reside in the color management module (CMM.dll) of the Windows Java package, specifically in the *SpTagToPublic* function. Code Listing 1 shows the area of code where one of the bugs resides. Lines 11-16 contain the bug. *Buf* is incremented for certain character inputs. Later in lines 18-25 *Buf* is used in a memory copy, and data outside the bounds of *Buf* may be inserted into the destination buffer. It is possible to use this type of vulnerability to leak internal program information. Leaking the location of a module is a recent approach used by attackers as part of a multi-part technique to bypass modern protections (non-executable data memory and address space randomization).

CODE LISTING 1. *SpTagToPublic* Function

```
1. #define IS_VALID_SIGNATURE_CHAR(c) ((0x30 <= ((unsigned char)(c)) && ((unsigned
   char)(c)) <= 0x39) ||  (0x41 <= ((unsigned char)(c)) && ((unsigned char)(c)) <= 0x5a) ||
   (0x61 <= ((unsigned char)(c)) && ((unsigned char)(c)) <= 0x7a))
   /*DESCRIPTION  -- Convert attribute value from more compact internal form. */
2. SpStatus_t KSPAPI SpTagToPublic ( SpTagId_t TagId, KpUInt32_t TagDataSize,
   SpTagValue_t FAR *Value, …)
```

```
    {
3.      char        KPHUGE *Buf;
4.      char        KPHUGE *BufSave;
5.      KpUInt32_t  Index, Limit;
6.      SpData_t    FAR *Data;
7.      SpSig_t     TypeSig;
8.      KpUInt32_t          trimCount = 0;
    /* set tag id and type in callers structure */
9.      Value->TagId = TagId;
    ...
10.     Buf = TagData;
    /* trim tag signature if needed */
11.     while (!IS_VALID_SIGNATURE_CHAR(*Buf)) {
12.         if (++trimCount > TagDataSize) {
13.             return SpStatBadTagData;
14.         }
15.         Buf ++;
16.     }
    ...
17. switch (Value->TagType) {
    ...
```

```
18.    case Sp_AT_Unknown:
19.        Value->Data.Binary.Size = TagDataSize;
20.        BufSave = (char KPHUGE *) SpMalloc (TagDataSize);
21.        if (NULL == BufSave)
22.            return SpStatMemory;
23.        KpMemCpy (BufSave, (void *)(Buf - 8), TagDataSize);
24.        Value->Data.Binary.Values = BufSave;
25.        return SpStatSuccess;
```

The coloring and fault localization (described in Chapter 5) are helpful since the function where the bug exists is complex. Figure 4 shows the visualization for Code Listing 1.

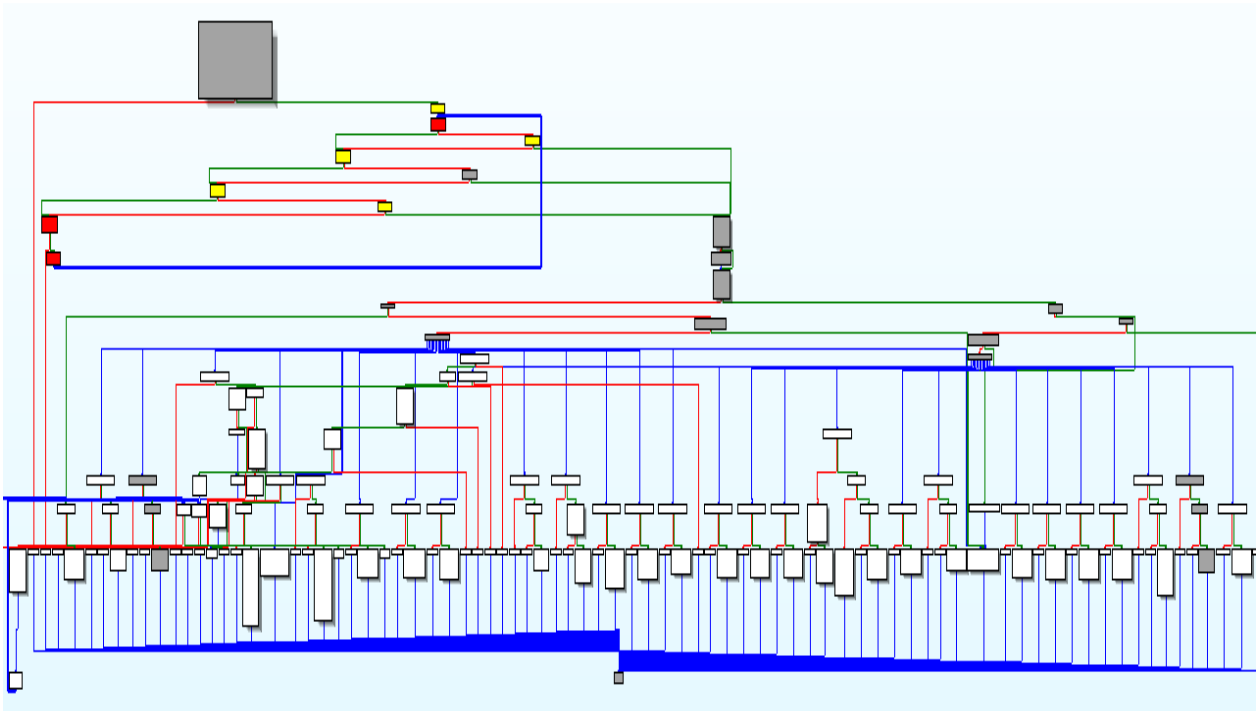

Figure 4: Visualization of Java Bug

The small red and yellow blocks (upper left) indicate the most suspicious blocks (lines 11-16). The grey blocks indicate basic blocks executed by good and bad traces. The white blocks were not executed. These colorations aid manual analysis for challenging bugs. Visualization is later described in detail. This illustration is provided to ignite interest in the chapters to come.

CHAPTER 4 FAULT LOCALIZATION BACKGROUND

Fault localization (FL) techniques normally make use of runtime trace information collected from both passing and failing executions[5]. FL generates an ordered set of code regions that are predicted to contain a bug. The method for generating the order is an important part of the technique and is critical to the technique's success. For the FL technique to be considered helpful, the location of the actual bug should be presented within the top X suspicious code regions. If this is not the case, FL is not saving enough debugging time for the programmer inspecting the FL results. For example, Cheng et al. [2009] assume that the tester will tire if more than the top 5 locations must be examined. Based on that work, our measure of success also requires the actual bug to occur within the top 5 rated FL candidates[6].

Each of the following sections provides an overview of prior relevant approaches. Each description will focus on two important elements: first, the method for computing the initial set of suspicious statements, and then the method for ordering the rest of the statements.

Set Union and Set Intersection

Set union and set intersection use code coverage information for fault localization. This dissertation refers to FL techniques that operate based on comparing covered code regions (line,

---

[5] As a measure of effectiveness in fault localization, the percent of the program that does not have to be examined to find the bug after localization is one common measure.

[6] The top 10 suspicious locations are colored in the visualization in the event an analyst wishes to keep looking beyond the top 5.

basic block, edges, or procedural) among passing and failing traces as a *coverage-based* technique. *Path-based*, *control-flow-based*, *program spectrum-based, and statistics-based* are similar terms that appear in other FL documents. Agrawal et al. [1995] used a technique that computes the set difference of the statements covered by two test cases, one passing and one failing. An initial set of suspicious statements is obtained by subtracting the statements executed in the passing trace from the statements in the failing trace. They called it a *dice* of two *slices*. Pan et al. [1992] demonstrated a set of dynamic slice heuristics that use set algebra for similar purposes. In addition to using the information listed above, Agrawal et al. also use analysis information to compute dynamic slices of the program and test cases from a particular program point.

A problem occurs if all statements are executed by both the passing trace and failing trace, resulting in an empty set of candidate statements. However, even if the initial set of candidate statements is empty or the bug is not in the initial set, there are techniques that can be used to identify suspect code. Renieris and Reiss [2003] show a technique that provides an ordering to the entities based on the system dependence graph, or SDG. Their fault localization technique and this SDG-ranking method are described next.

Nearest Neighbor

Renieris and Reiss [2003] addressed the issue of failing code occurring in passing traces using their Nearest-Neighbor FL technique. Rather than removing, by set difference, the statements executed by *all* passed test cases from the set of statements executed by a single failed test case, they selectively choose a *single* best passed test case for the set difference. The important problem is identifying the "best" case. They do so by first identifying a single failed test case

and then selecting an associated passing test case that has similar code coverage to the failed case. Using these two test cases, they subtract the set of statements executed by the passed test case from the set of statements executed by the failed test case. The resulting set of statements is the initial set of statements from which the programmer should start looking for the bug. As before, this process could still result in an empty set. If the bug is not contained in the initial set, they use the SDG-ranking technique discussed next.

SDG-ranking Technique

In the work of Renieris and Reiss [2003] the code entities in the initial suspicious set from FL are called the "blamed" nodes. A breadth-first search is conducted from the blamed nodes along dependency edges in both forward and backward directions. An example in the next paragraph will clear up how this works. All nodes that are at the same distance are grouped into a rank. Given a distance *d*, and a set of nodes at that distance S(*d*), the rank number that is assigned to every node in S(*d*) is the size of every set of nodes at shorter distances plus the size of S(*d*).

For example, suppose an initial list of suspicious code blocks contained two statements. The programmer looks at each and finds that the bug is not in either one. He then inspects all forward and backward control-flow and data-flow dependencies at a distance of 1. Assume that yields an additional 5 nodes. The rank number of the initial set is 2 and the rank number of all the nodes a distance of 1 is 7 (2+5). Using the size of the rank, plus the size of every rank at a smaller distance for the rank number, gives the maximum number of nodes that would have to be examined to find the bug following the order specified by this technique. Since this technique does not use the original runtime information that was part of the fault localization, except as a starting point, its effectiveness is limited.

Cause Transitions

Cleve and Zeller [2005] use a binary search of the memory states of a program between a passing test case and a failing test case. This technique is part of a suite of techniques defined by Zeller et al. called *Delta Debugging* [1999]. Their FL technique called *Cause-Transitions*, defines a method to automate the process of making hypotheses about how state changes will affect output. In this technique, the program under test is halted by a debugger along the trace of both a passing and failing test case. Part of the memory state is swapped between the two traces and then allowed to continue running to termination. The memory that appears to cause the failure is narrowed down using a technique much like a binary search with iterative executions of the program in the debugger. This narrowing of the state is iteratively performed until the smallest state change that causes the original failure can be identified. This technique is repeated at several program points to find the flow of the differing states causing the failure throughout the lifetime of each execution. These program points are then used as the initial set of points from which to search for the bug.

If the bug is not contained in this initial set, they too suggest the SDG-ranking technique. They provide two improvements to the SDG-ranking that can exploit the programmer's understanding of whether particular states are "infected" by a bug or "cause" the bug to be manifested. These improvements are called "exploiting relevance" and "exploiting infections" and are fully defined in Cleve and Zeller's 2005 paper.

SOBER

Liu et al. [2005] present their SOBER technique that performs statistical calculations on passing and failing traces to isolate bugs. They show their technique to be superior to Liblit et al. [2003,

2005] which used predicate probabilities to isolate bugs from collected crashes (failing traces only). SOBER considers a predicate to be relevant if its evaluation pattern in incorrect traces differs significantly from that of correct ones.

SOBER uses profiles of branch executions within each test case to localize branches that are related to bugs. The algorithm begins by modeling evaluations of a predicate *P* as independent Bernoulli trials: each evaluation of *P* returns either True or False. Next, the estimate of the probability of *P* being True, which is called the "evaluation bias," is calculated from the fraction of True observations within each program execution. Let *x* be the random variable carrying the evaluation bias of predicate *P*, there exist two statistical models, f(*x*|Correct) and f(*x*|Incorrect), that govern the evaluation bias observed from correct and incorrect executions. Finally, if f(*x*|Incorrect) significantly differs from f(*x*|Correct), it indicates that the evaluation of *P* in incorrect traces is abnormal, and thus likely related to a bug. This quantification is used to rank all the instrumented predicates, creating a ranked list of suspicious predicates. Programmers can either examine the list from top down, or they can choose to examine only the top-ranked suspicious branches for debugging. One issue with this approach is that only predicates can be considered as suspicious locations.

Failure-Inducing Chops

Since debugging is usually performed by analyzing the statements of the program when it is executed using a specific input, Korel and Laski [1988] proposed the idea of dynamic program slicing. A dynamic slice identifies a subset of executed statements that is expected to contain faulty code. The effectiveness of a given slicing algorithm in fault location is determined by two

factors: h*ow often is the faulty statement present in the slice,* and *how big is the slice, i.e. how many statements are included in the slice?*

There are two kinds of dynamic slices: backward dynamic slices and forward dynamic slices. The backward dynamic slice of a variable at a point in the execution trace includes all those executed statements which affect the value of the variable at that point. In contrast, the forward dynamic slice of a variable at a point in the execution trace includes all those executed statements that are affected by the value of the variable at that point. Backward dynamic slicing has been proposed to guide programmers in the process of debugging [Agrawal et al. 1993] by focusing the attention of the user on a subset of program statements which are expected to contain the faulty code.

Gupta et al. [2005] combined Delta Debugging [Zeller 1999] with dynamic slicing [Agrawal and Horgan 1990] to narrow down the search for buggy code. Prior work had shown backward dynamic slicing to be useful for debugging, but their work was the first to show the application of forward slicing with Delta Debugging for FL. The technique works by identifying a minimal failure-inducing input, using this input to compute a forward slice, and then intersecting the statements in the forward slice with the statements of the backward slice of the erroneous output to compute a failure-inducing chop. The relevant part of the input is the minimal failure inducing input. Gupta et al. compute the intersection of the statements in the forward dynamic slice of the failure-inducing input and the backward dynamic slice of the faulty output to locate the likely faulty code. The statements in the intersection of forward and backward dynamic slices are known as the failure inducing chop. The failure-inducing chops are expected to be much

smaller than backward dynamic slices since they capture only those statements of the dynamic slices that are affected by the minimal failure-inducing input.

Towards Locating Omission Errors

Zhang et al. [2007] noted that execution omission errors, that is code that should have been executed as part of the program but was not, are difficult to find dynamically, since those statements are never covered and thus cannot be operated on as part of the FL algorithm. They introduce the notion of implicit dependences, which are dependences that are normally invisible to dynamic slicing due to omission errors. They designed a dynamic method that forces the execution of the omitted code by switching outcomes of relevant predicates such that those implicit dependences are exposed and become available for dynamic slicing. Their technique consists of reexecuting the program while switching a specific predicate instance, and aligning the original and switched executions. Then dynamic code coverage information can be recollected for fault localization which they do so using dynamic slices similar to their prior work [Zhang et al. 2005], which is similar to the failure chops described in the prior section.

Tarantula

Researchers at Georgia Tech have considered runtime code coverage information to develop a technique they call Tarantula [Jones and Harrold 2005; Hsu et al. 2008; Santelices et al. 2009]. Tarantula, like Renieris and Reiss and others, uses code coverage information to track each line (or other coverage element) of code as it is executed. A pass-fail oracle labels each trace as such. Like other approaches, the intuition behind Tarantula is that code regions in a program that are primarily executed by failing test cases are more likely to be the culprit than those that are

primarily executed by passing test cases. Tarantula allows tolerance for lines that occasionally participated in passing test cases. This tolerance has been shown to make the technique very effective.

Table 1: Sample Coverage Data

	Trace 1	***Trace 2***	Trace 3	***Trace 4***	Suspiciousness
Line 1	x		x	x	.58
Line 2		x	x	x	***.82***
Line 3	x		x		0
Line 4		x	x		.58
Line 5	x			x	.58

Table 1 provides sample code coverage data to be evaluated by Tarantula using the adapted Ochiai [Abreu et al. 2007] formula, which is described by Equation (1). Traces 2 and 4 failed. Trace 3 passed, but executes lines that failing traces also executed. The likelihood that any one line contains the bug is calculated as follows:

$$suspiciousness(s) = \frac{failed(s)}{\sqrt{totfailed\ x\ (failed(s) + passed(s))}} \qquad (1)$$

In Equation (1), *failed(s)* is the number of failed test cases that executed statement *s* one or more times. *Passed(s)* is the number of passed test cases that executed statement *s* one or more times. *Totfailed* is the total number of failed test cases for the entire test suite. If the denominator is zero, zero is assigned as the result. Using this calculation, Tarantula identifies

Line 2 as the likely bug. The calculation for the line containing the bug from Table 1, using Equation (1), is shown below:

*suspiciousness(*Line2*)* = 2 / sqrt((2 * (2 + 1)))

*suspiciousness(*Line2*)* = .82

Using this suspiciousness score, the covered lines are sorted. The set of lines with the highest score are considered first by the programmer attempting to fix the bug. If after examining the first entity and the bug is not found, the programmer continues down the list from highest to lowest score. Ties must be manually resolved. For a visual cue Tarantula colors code to further help debugging.

Tarantula-based approaches exhibit three weaknesses:

1. Tarantula works well for bugs with differences in covered code, but will likely not identify *data-only* bugs (discussed below).
2. Coverage-based algorithms may become confused in the presence of noise.
3. Previous Tarantula implementations require source code and a test suite of passing and failing traces.

Weaknesses one and two are defined as follows (discussed later as well):

- *Data-only* bugs are ones in which passing and failing traces have no code coverage differences at runtime.
- *Noise* in this article refers to failing traces that contain so many uniquely covered code regions that the typical Tarantula-based approach will end up suggesting many

suspicious locations that are not bugs. Noisy conditions increase the amount of code locations to be considered by the analyst to be prohibitively high before the actual bug is encountered, rendering the FL algorithm ineffective for practical use.

For the class of bugs of interest (crashing memory corruption issues on fully compiled code) all three weaknesses are addressed via this new fuzzing and fault localization system (later described by Algorithms 1 and 2).

Top-K LEAP

Cheng et al. [2009] have used a data mining technique to mine execution graphs to localize bugs. Given a set of graphs of passing and failing traces, they extract the subgraphs that discriminate the most between the program flow of the traces. Their experiments show that their technique performs better than another popular FL approach known as RAPID [Hsu et al. 2008].

Their approach is different from traditional graph mining techniques, which mine a very large set of frequent subgraphs. Since mining many subgraphs is expensive, they formulate subgraph mining as an optimization problem and directly generate the most discriminative subgraph with a graph-mining algorithm called LEAP [Yan et al. 2008]. LEAP is both more efficient and more accurate than previous graph approaches. They subsequently extend LEAP to generate a ranked list of Top-K discriminative subgraphs representing distinct locations that may contain the bugs. Equation (2), Discriminate Subgraph mining FL, shows how subgraphs are pruned via their algorithm.

$$susp(edg) = \frac{failed(edg)}{passed(edg)} > \frac{totalfailed}{totalpassed} \qquad (2)$$

The operators *failed(edg)* and *passed(edg)* report the number of failing program traces that include the edge *edg* and the number of passing program traces that include the edge *edg* respectively. Using this approach, only suspicious edges are retained (nodes with no edges are removed). As an example, assume that 20 of 50 traces were failing traces. Now suppose a particular edge appears in 15 failing traces, but only 5 passing traces. Since $3 > 0.666$ they assume this edge is suspicious.

Test Data Creation

FL algorithms require test suites with many test cases. Artzi et al. [2010] showed how one can automatically build a test suite for use with FL algorithms. Of particular value is that they can generate a sequence of test cases where each subsequent test differs from its predecessor in small, systematic ways. Such an approach works well with coverage-based fault localization, because FL works best when most of the execution trace is the same, allowing the differences to stand out.

They used two similarity metrics, one focused on similar paths, and the other focused on similar data to expand a few test cases into a larger test suite. However, they require at least one failed test case to seed their process, and they require source code (PHP currently) to facilitate their symbolic evaluation. Finding a failed test case in closed-source, commercial code is non-trivial, and symbolic evaluation of large code bases is problematic. However, their technique is elegant and builds an efficient test suite.

Using Input Data for Fault Localization

Selective data flow tracking as part of a code coverage-based fault localization algorithm is novel; it is a contribution of this work. However, using input data to perform FL has been investigated. Researchers Jeffrey et al. [2008] localized bugs by varying inputs. They find inputs similar to failing inputs, such that the bug is no longer triggered. The statements operating on the input are presumed to be the fault, or related to it.

Database Fault Localization

We recently discovered work, which was conducted in parallel to ours without our knowledge. Researchers at IBM [Saha et al. 2011] discovered what we did: fault localization without consideration of data is likely to fail for some bugs. They however did not use FL on typical compiled desktop programs. They focused strictly on database code that conducts business logic for customer transactions. ABAP is the SQL-like language their work can only operate on. They found that through a number of enhancements to a typical slice differencing technique [Agrawal et al. 1995] they could obtain better results. Their primary enhancement operates by tracking a key variable value at each statement location. Thus, if all statements are the same between passing and failing traces, they will still have information that differs. This is similar to one of our enhancements (FL-DFT from Chapter 6), which modifies a code coverage-based FL technique to consider data.

Their approach differs from ours in the following important ways:

1. They do not use a high speed Dynamic Binary Instrumentation (explained in Chapter 6) that can collect trace information for large x86 programs. They use a slower, custom set of single-step debugging scripts to gather data.

2. They do not taint and trace input for closed source applications, but rather depend on the ABAP source code to track code and data.
3. They focus on data-centric code that has very little controlflow difference. They use knowledge of the code semantics to further unwind statements to conduct slicing which includes an important variable for each statement.
4. They do not combine their FL approach to a bug hunting and analysis system

CHAPTER 5 BUILDING IN FAULT LOCALIZATION AND VISULIZATION

Fault localization tools typically require (1) a test suite and (2) the software source code. A test suite will have both input and the ability to recognize output. The output of each test case is labeled, indicating whether that particular result is a "pass" or a "fail", where failure indicates a software-coding mistake. Having identified the existence of a fault, the next step is to localize the error within the code. A typical approach relies on the basic premise that code regions (often lines or basic blocks) that occur more frequently in failing traces[7] are more likely to contain the bug. These suspect regions are identified based on occurrence, and then ranked, creating a final list of suspicious locations. The more similar the passing and failing input test cases are, the better coverage-based algorithms tend to perform, because runtime differences are more easily identified.

This research expands upon this general approach in two important ways:

1. A preexisting test suite is not require to identify faults.
2. Source code is not required to perform fault localization.

These improvements are important because test suites may be incomplete or expensive to generate. In addition, source code may not always be available for legacy code, or for offensive research. Issue 1 is addressed by creating test suites automatically. First, quality inputs are downloaded based on code coverage. Second, the collected set is expanded by systematically

---

[7] A trace is a list of code regions encountered during one execution of the program given some input.

modifying the test inputs, a dynamic testing technique known as fuzzing [Miller et al. 1990]. Issue 2 is addressed because fuzzing can operate on binaries where source is not available. Fuzzing automatically seeks memory corruption vulnerabilities, a critically important subclass of bugs that are the basis for most control-flow hijack exploits. Hackers use fuzzing to uncover bugs and create offensive tools. Commercial developers use fuzzing as a final test of their code—to keep ahead of the hackers.

Once fuzzing has identified an input that generates faulty behavior, the tools in this research generate a smaller test suite (series of similar inputs) that is specific to the discovered bug. This is possible even though the location of the bug is not yet known. As before, fault localization requires that the generated test suite have inputs that generate both passing and failing outputs. A coverage-based algorithm is then applied to localize bugs to basic blocks—a unit of code that does not contain a branch.

This work provides a number of contributions, above and beyond those already mentioned. This new approach enhances results by increasing the suspiciousness score of blocks not thought to be noise (explained later). In addition, accuracy is increased by using an input tainting, data-flow tracking algorithm. For data-flow tracking, key portions of the input data are tagged. The program records the basic blocks within the program which operate on tagged data. A score modifier (increased suspiciousness) is then applied to each basic block that uses tainted input data[8]. This chapter will:

---

[8] Input data that is tracked throughout a programs lifespan is considered "tainted input". Tracking means the input will taint other data and registers as it is operated on in compiled code.

1. Describe the novel fault localization approach, and show that the algorithm outperforms previous approaches.
2. Describe the technique that allows the framework to work with closed-source code.
3. Detail the environment (system) in which this research currently operates.

This chapter describes the integration of our FL system to our fuzzing system. This chapter also serves as motivation for the FL enhancements we fully describe in Chapter 6. Chapter 7 discusses the results of those enhancements.

Automatic Fault Localization and Visualization

Now that a set of faults has been discovered, the process to find where the faults occur is conducted. Given that this research targets the class of bugs known as vulnerabilities, a fair question to ask is: doesn't the analyst know where the fault is because the program crashed at that point? Perhaps, for trivial bugs that is true. But not for subtle bugs which are, in fact, the ones most likely to have survived the sophisticated testing of commercial software. A trivial example would be to set a pointer to Null and to reference it at some later point. The program crashes at the later point (Null reference), but the actual fault lies where the pointer assignment was made or even earlier where the Null value was derived. Finding the actual fault is the task of fault localization.

Once a bug is found via fuzzing, the discovered crashing test case is expanded into passing and failing test cases. This is done because FL requires both passing and failing tests. The process for this expansion is explained next.

Execution Mining (EM) is the name for this bug-focused test set creation, fault localization, and visualization process. EM is tied into CF and can automatically localize faults. The Appendix (ClusterFuzz GUI Walkthrough) shows screen shots of an early version of the GUI. Recognition goes to the CF team at Harris Crucial Security, Inc. for their continuing work on CF. Figure 5 shows an overview of the process from which the constructed FL runs.

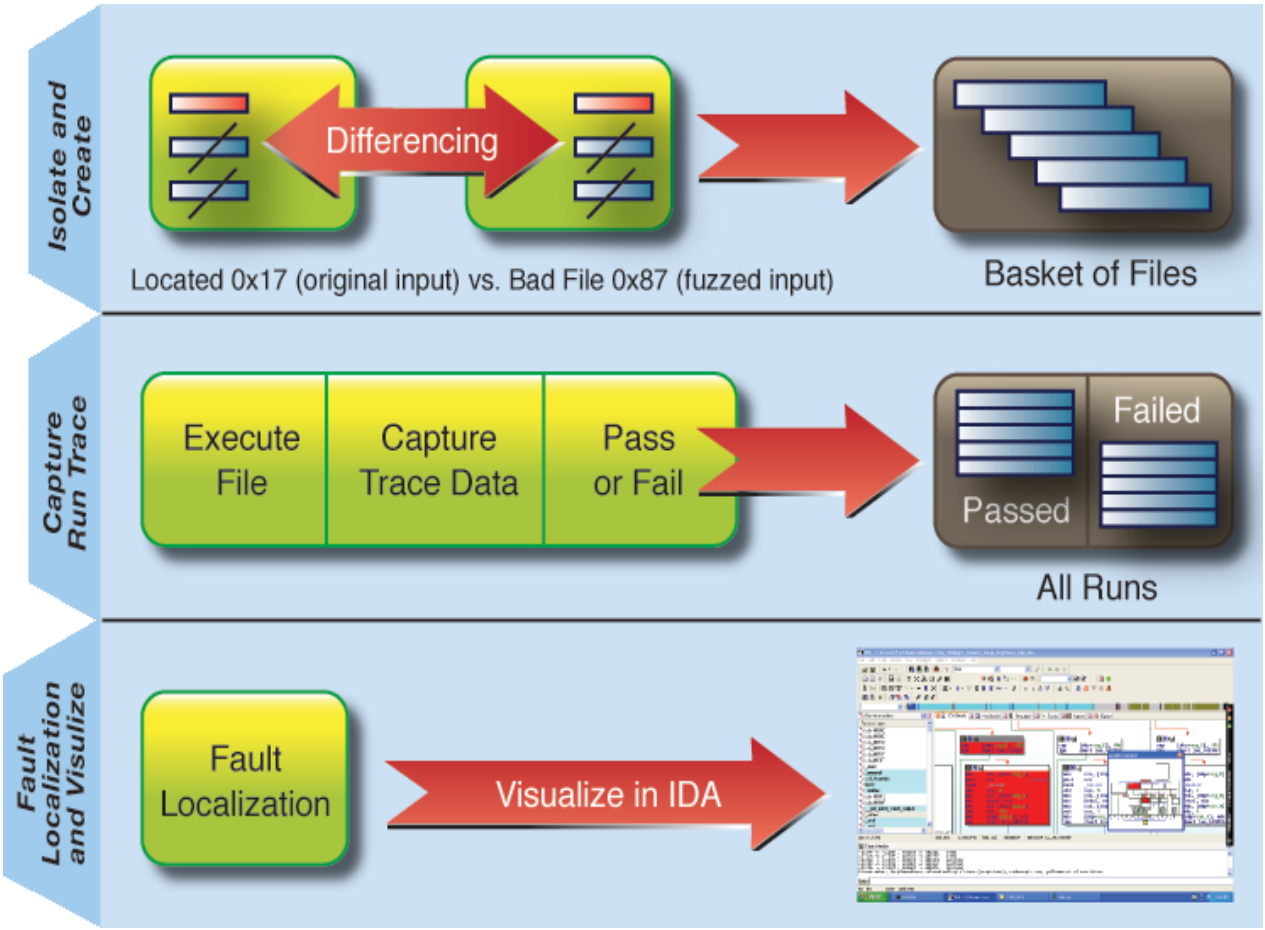

Figure 5: Execution Mining

The first step is to create a set of passing and failing inputs[9]. Following the process shown in Figure 5, the mutated byte(s) is first isolated in the failing input file that caused the application to crash. This research focuses on binary files for client-side applications. The tool changes each difference in the failing input file back to whatever it was originally, until the failing input file no longer crashes the application. It is in this way that the particular data that is responsible for the fault is identified. If the offending data is a single byte, 256 new files can now be created using all possibilities of that byte. The file-and-byte assumptions are reasonable for the type of client-side fuzzing focused on[10].

Now that a set of passing and failing files has been collected, each file can be run through the trace collection software. A basic block hit trace collection is created and tagged as either failed (crashed) or passed (not crashed). After all of the traces have been collected, the fault localization can run. This step is shown at the bottom of Figure 5. Each of the top 10 suspicious

---

[9] Files are operated on in this case, but the technique works for other inputs as well. We later show that our FL algorithm works on STDIN and network inputs, but we have only implemented the automatic test set creation portion of EM for file inputs at this time.

[10] In the author's experience, file fuzzing produces mutated files, which include a bug that can often be localized to one key position in the input. It is possible that multiple-byte bugs could not be isolated with this heuristic, but for the sake of automatic FL that case is ignored for file inputs. Furthermore, rather the bug is exactly isolated via input at this time is not the point. All our system requires to perform FL is passing and failing tests, which we characterize in this case as non-crashing and crashing respectively.

locations is then colored and visualized in IDA pro [Hex-Rays 2011], a popular reverse engineering tool. The IDA pro disassembly screen is centered (redirected) on the most suspicious location, as shown in the bottom of Figure 5. EM is later described in more detail in Algorithm 2.

Example 1: A Simple Bug with Code Coverage Differences

To illustrate EM, a piece of open-source code was modified to contain a bug. The bug occurs early in the code, but the error isn't manifested until later in the code. The open-source code is dlltest, which is part of a suite of Optical Character Recognition [Tesseract OCR 2011] tools. One of the C++ files is called imgbmp.cpp. That file includes code to parse a BMP image file. Minor changes were made, which are highlighted in Code Listing 2. Source code is shown here to explain the bug, but remember that the approach works with the resulting binary and does not require the source code.

On lines 6-7 local variables were added; not the bug. Lines 12-13 are the inserted bug, which doesn't immediately result in an error or crash. On line 21, the bug is activated when the *xsize* data (from the BMP) is greater than 100.

CODE LISTING 2. Code with a Bug because of Path Differences

```
1.  inT8 read_bmp_image(        //read header
        int fd,                 //file to read
        uinT8 *pixels,          //pixels of image
        inT32 xsize,            //size of image
        inT32 ysize,
```

```
        inT8 bpp,                   //bits per pixel
        inT32                       //bytes per line   ) {
2.  uinT32 bpl;                     //bytes per line
3.  uinT32 wpl;                     //words per line
4.  uinT32 nread;                   //current bits
5.  inT32 index;                    //to cols
6.  unsigned char bug1 = 0;
7.  char * ptr = NULL;
8.  bpl = (xsize * bpp + 7) / 8;      //bytes per line
9.  wpl = (bpl + 3) / 4;
10. wpl *= 4;
11. if( xsize > 100 )
12.     bug1=1;
13. for (index = 0; index < ysize; index++) {
14.   nread = read (fd, pixels + bpl * (ysize - 1 - index), bpl);
15.   if (nread != bpl)
16.     return -1;
17.   if (wpl != bpl)
18.     lseek (fd, wpl - bpl, SEEK_CUR);
19. }
20. if(bug1)
```

21. ***ptr = 1;** //causes access violation

This bug can now be found via fuzzing using CF. After all three stages of EM finish, the colored IDA pro disassembly (graph view) shown in Figure 6 is obtained. Finally, visualization (described below) directs analysts to the bug.

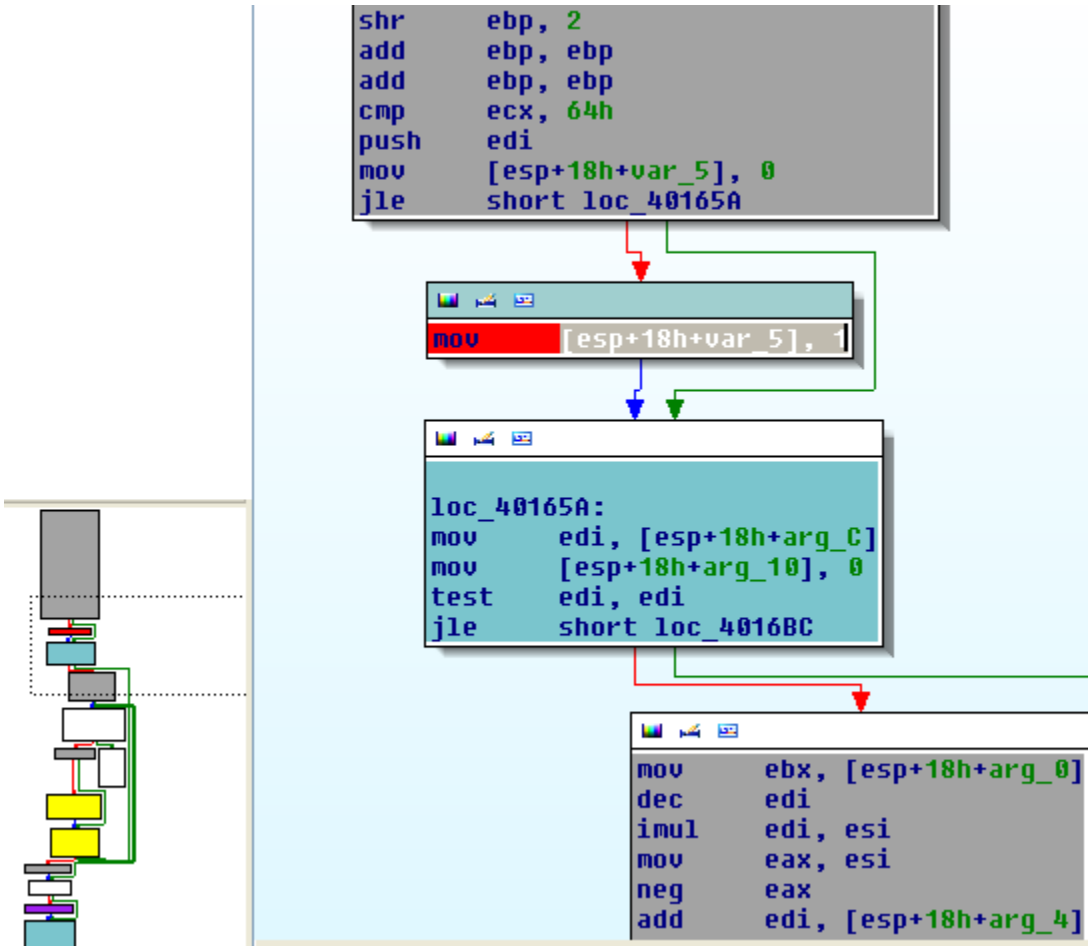

Figure 6: An Example with Covered Code Differences

There are two sections of code that contain the bug: lines 12-13 and 21. EM successfully identified both offending locations as the top two likely suspicious basic blocks. The bug is the small red block in the middle of Figure 6 (the block with one assembly instruction, which is the compiled code for line 13). The following coloring scheme was developed to aid the analyst in bug analysis:

- Grey blocks of code indicate both passing and failing traces executed the block.
- Blue blocks indicate only passing traces executed the code block.
- Brown blocks indicate only failing traces executed the block.
- Red blocks indicate a highly suspicious location (top 2 locations).
- Yellow blocks indicate suspicious locations (next 8 locations).
- A purple block indicates the final location executed (used to identify crash locations).
- An orange block is the most common final block (used to indicate a common crash location).
- All other blocks are left white, the default color in IDA pro.

Table 2: Top 5 Suspicious Code Blocks from Figure 6

Rating	Passing	Failing	Basic Block Virtual Address (Start, Stop)	Color
1.0000	0	155	4016c3, 4016ca	Red
1.0000	0	155	401655, 40165a	Red
0.7806	25	39	4016a5, 4016b8	Yellow
0.7773	76	116	4016a2, 4016a5	Yellow
0.7773	76	116	401694, 4016a5	Yellow

The data shown in Table 2 appears in the IDA pro tool *Output window* (at the bottom of IDA pro by default). That data shows the top 5 suspicious code blocks. The two locations responsible for the bug show up in all 155 failing traces, and never in any of the passing traces. Equation (1) is easily able to operate on the passing and failing information from Table 2 and detect the bug, under the straightforward mathematical conditions of this simple example.

Example 2: A Simple Data-only Bug

Coverage-based analysis shines on Example 1, but suppose the bug had no code coverage implications, meaning different inputs can trigger a bug with no difference in the statements executed. This dissertation refers to this as a *data-only* bug. Algorithms that solely depend on code coverage differences cannot reliably find these bugs because control-flow provides insufficient information—data difference not code coverage needs to be considered. To illustrate, another contrived bug was created in the open-source code from Example 1. Inspect the changes made to a similar region of code, shown below in Code Listing 3.

CODE LISTING 3. Data-only Bug

```
1. inT8 read_bmp_image(        //read header
       int fd,                 //file to read
       uinT8 *pixels,          //pixels of image
       inT32 xsize,            //size of image
       inT32 ysize,
       inT8 bpp,               //bits per pixel
       inT32                   //bytes per line       ) {
```

```
2.  uinT32 bpl;                 //bytes per line
3.  uinT32 wpl;                 //words per line
4.  uinT32 nread;               //current bits
5.  inT32 index;                //to cols
6.  unsigned char bug3 = 1;
7.  unsigned int my_size = 0;
8.  bpl = (xsize * bpp + 7) / 8;   //bytes per line
9.  wpl = (bpl + 3) / 4;
10. wpl *= 4;
11. bug3 = (101 - xsize);
12. for (index = 0; index < ysize; index++) {
13.   nread = read (fd, pixels + bpl * (ysize - 1 - index), bpl);
14.   if (nread != bpl)
15.     return -1;
16.   if (wpl != bpl)
17.     lseek (fd, wpl - bpl, SEEK_CUR);
18. }
19. ptr = (char *)malloc(my_size+bug3);
20. *(ptr+xsize)=12;
```

Lines 6-7, add a couple local variables. Line 11 is the bug. Lines 19-20 activate the bug, if the *xsize* from the BMP picture has various unexpected values. The bug results when either no,

or an improper amount of memory is allocated, and the pointer is dereferenced for a region of memory of improper size. The issue illustrated is that there are no differences between passing and failing traces in terms of statements covered. All code paths through this function will execute each statement (e.g. identical code coverage); only different data inputs trigger the bug.

The large grey block shown in Figure 7 (containing "mov al, 65h"), is part of the bug. Neither this block, nor the other portion of the bug, is correctly colored. The failure is because there is no code flow difference in this *data-only* bug. Coverage-based analysis does still find a difference between passing and failing traces, and therefore presents erroneous data as shown in Table 3.

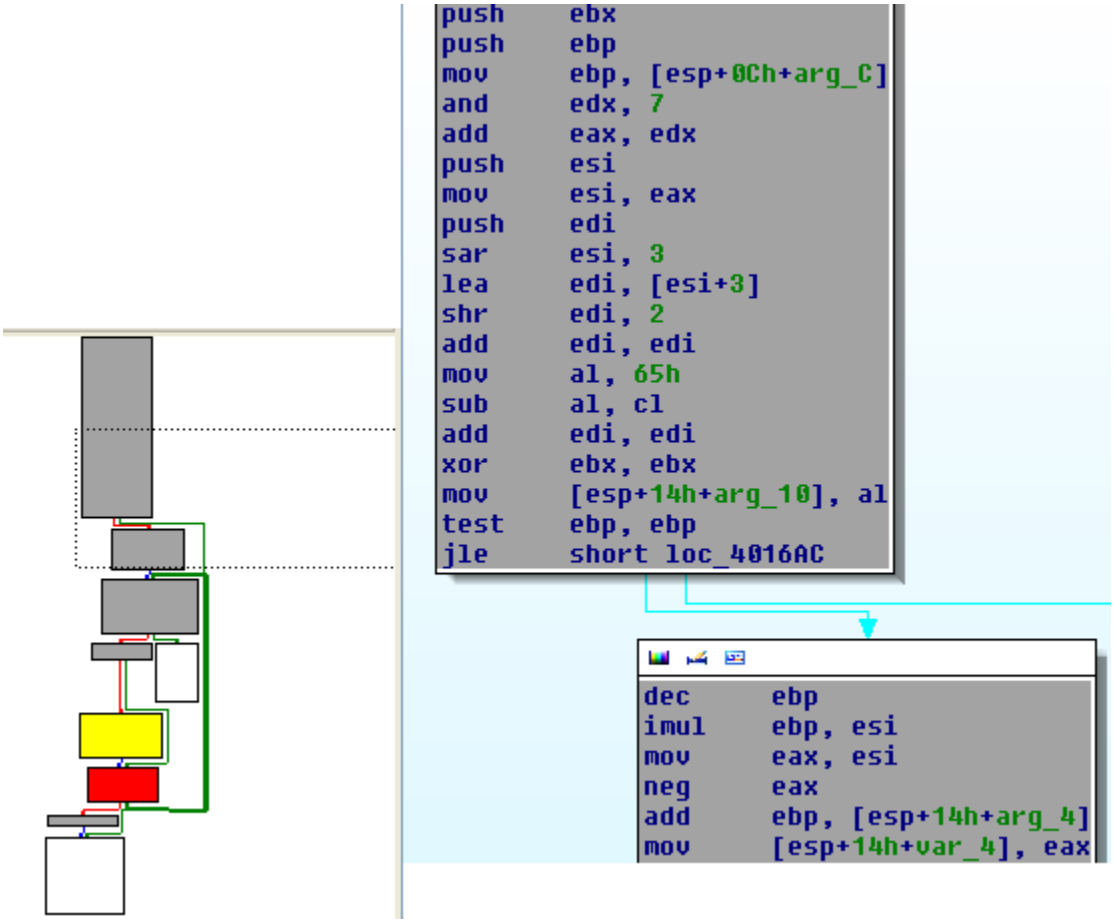

Figure 7: Data-only Example

The data shown in Table 3 indicates line 17 from Code Listing 3 as the most likely root cause. This selection is not the correct choice (line 11 is). In larger programs, happening to get this close could be sufficient, but for this simple example the bug should be directly identified as was done in Example 1.

Table 3: Top 5 Suspicious Code Blocks from Figure 7

Rating	Passing	Failing	Basic Block Virtual Address (Start, Stop)	Color
0.7806	25	39	40169d, 4016a8	Red
0.6654	107	85	40169a, 40169d	Red
0.6654	107	85	401688, 40169d	Yellow
0.6029	132	124	405a94, 405aaa	Yellow
0.6029	132	124	405a8a, 405aaa	Yellow

Other fault localization approaches that only consider code coverage information would also fail to identify the buggy basic block, since there is no distinguishing information present. To locate this type of bug, this research includes a data-flow tracking technique to key in on important code blocks. The ability to localize this difficult data-only bug is later demonstrated.

Motivation for an Enhancement

This research proposes that by addressing the data-only issue the issue of noise can also be addressed. Noise happens when the failing traces contain too many uniquely covered code regions when compared to the passing traces, resulting in many suspicious locations being reported that are not bugs[11]. Noise is the opposite problem from the data-only issue illustrated

---

[11] In general, we believe that binary/assembly based systems are more susceptible to noise than high level language traces would be, but on sufficient scale noise would be present in either scenario.

in the previous section. However, data-only and noise are related: either not enough signal in the data, or too much noise, which hides the signal.

Both issues could be generalized as *focus* issues. The focus of the test cases, and the collected trace data should be as small and relevant as possible. Strategies that (a) include a more complete and focused test suite, and that (b) perform the FL only on the program modules in which the bug occurs, will do better since the captured trace data generated is more likely to be relevant. Those approaches are helpful, and are utilized when possible. For example, if *program.dll* crashes inside of *program.exe*, EM will be performed against just *program.dll*, rather than all of the loaded modules[12]. However, that particular DLL still may be very large, which is why those two techniques alone are insufficient, or not always readily available. The FL improvement in this research seeks enhancements in which:

1. FL uses dynamic information about *tainted* input data.
2. FL can more precisely pinpoint suspicious code blocks, in the presence of noise and/or with data-only bugs, because code regions that operate on tainted input are focused on.

---

[12] Modern operating systems load many dynamic modules and to perform FL program wide would end in poor results because of the diversity in each trace.

CHAPTER 6 ENHANCING FAULT LOCALIZATION WITH SELECT INPUT TAINTING AND TRACKING

The new approach in this research uses a Ochiai coverage-based approach, but applies a score modifier (increased suspiciousness) to each basic block that uses tainted data. In this work, taint is defined as:

- Tagging some or all input data.
- Following marked (tainted) data via a data map, which indicates which memory locations and registers are tainted.
- If a tainted element is used in an operation that modifies an untainted location or register, taint is transferred.

Tainting is important because it allows the new algorithm to tag suspicious inputs and follow them as they are operated on throughout the life of the program. Tainting will be further discussed as each of the following is described:

1. Background information on dynamic information flow tracking
2. The tools upon which we build.
3. The sources of taint.
4. How code blocks are emitted, as they operate on tainted input.
5. How the FL system works (Algorithm 2).

Dynamic Information Flow Tracking

Dynamic information flow is a technique that tracks the flow of information along dynamic dependences starting from the program inputs to the values computed by the program [Gupta et al. 2008]. Applications include software attack detection and prevention, fault localization, and

data validation. Depending on the exact implementation the overhead associated with this technology can be very expensive. In our work we use a dynamic binary instrumentation framework to capture only select information, making our approach scalable to real-world programs.

Dynamic Binary Instrumentation

Dynamic binary instrumentation (DBI) allows developers to modify the behavior of fully compiled programs as they execute. The DBI framework we use to collect code coverage is called Pin [2011]. We also use Pin to apply, track, and selectively report on tainted input data. Pin is a framework that allows testers to instrument binary applications in arbitrary ways by writing a *Pintool*. Similar results could be obtained by using scripts within a debugger, but the performance penalty would be prohibitive when testers desire near real-time analysis.

Data Flow Tracking with Dynamic Binary Instrumentation

The concept of tracking tainted data has been shown useful in security research [Piromsopa 2006]. More recently, researchers at Columbia University [libdft 2011] released a Data Flow Tracking (DFT) toolkit called Libdft that works with Pin. DFT deals with the tagging and tracking of interesting data (normally user input) as they propagate during program execution. DFT may also be referred to as Information Flow Tracking (IFT) or Dynamic Taint Analysis (DTA). DFT has been implemented by a variety of tools for numerous purposes, including protection from buffer overflow and cross-site scripting attacks, analysis of legitimate and malicious software, detection and prevention of information leaks, etc. Libdft is a dynamic DFT framework that unlike previous work, is fast, reusable, and works with commodity software and hardware. Libdft provides an API, which can be used to create and deliver DFT-enabled tools

that can be applied on unmodified binaries running on common operating systems and hardware, thus facilitating research and prototyping.

Our sources of Tainted Data

Libdft was designed to work on network sockets. Since the majority of the experiments in this research operate on files and standard input, the Libdft code base was extended to also work on files and standard input (STDIN). For some programs, if all input data is tagged as tainted, taint propagates throughout much of the code—compounding the noise problem. For file inputs, the taint impact can be decreased, because the input bytes of interest are known: EM localized the offending bytes in the test set creation phase (Figure 5). Thus, just those bytes are marked as tainted. This strategy provides a small amount of targeted data to track, and from the set creation phase this is known to be the critical data. There are situations where marking all data as tainted are required. For example, when a network program is examined later in this dissertation, all input from socket files are tracked. The decision on how to tag is part of the test configuration, which is developed by the user at the start of the test. The internal operation of Libdft is relied upon to properly propagate taint information from memory/registers to other memory locations/registers as they are copied around throughout the life of the program.

How We Emit Tainted Basic Blocks

To minimize the code bocks that get a suspiciousness boost, we choose to record only the address of basic blocks that read from tainted memory locations. Our system works with fully compiled code. Within our Pintool, the following two functions from Pin are used to determine if there is a read for a particular assembly instruction: *INS_IsMemoryRead* and

INS_HasMemoryRead2. As an example, if the following instruction were inside a basic block "*movzx edx, [ebp+tainted_byte]*" its address would be emitted as tainted, so that FL-DFT (the name of our new approach–described in the next section) can later increase the suspiciousness score of that basic block. The Intel assembly move instruction just shown, transfers the data from a tainted pointer to the EDX register.

We also include an option to output basic blocks (BBs), which have both of the following two properties:

1. The BB contains the use of a dangerous APIs, such as *strcpy*, *strncpy*, *memcpy*, etc.
2. The source parameter of the dangerous API includes tainted user input.

The reason we output BBs that use suspicious API functions (with tainted data) is because, in general, our system is interested in finding and analyzing security critical memory corruption bugs. The list of dangerous functions includes APIs likely to play a role in memory corruption.

Fault Localization Algorithm Details

Algorithm 2 details the process from discovered bugs, until the final ordered list of suspicious basic block locations is output. Afterwards, the output file is loaded into IDA Pro for visualization.

Algorithm 2. Coverage-based Fault Localization with Input Data Tainting and Tracking

---

Input: *Results* from Algorithm 1. While the FL algorithm was tested on multiple input types, the end-to-end system only works on files at this time. When a bug is located with a fuzzed input, that input is automatically expanded to create a test set appropriate for the FL system. This new test set is referred to as the *NarrowTestSet.*

Output: Ordered list of suspicious basic blocks addresses and supporting data (*FL-Basic, FL-Enhanced, and FL-DFT*) ready to be loaded in to IDA pro for visualization.

#Execution Mining (Step 1: Automatic Test Set Creation.)

for each input pair *i*, *i´* in *Results* **do**

FileOffset = LocateDifference(*i*, *i´*);

NarrowTestSet = CreateSimilarDataInputs(*i*, *i´, FileOffset*);

for each input *f* in *NarrowTestSet* **do**

RetVal= CreateTraceFile(*f, DFT_Pintool, TraceName, TraceDir, FileOffset, ...*);

if (*RetVal*) == EXCEPTION

then

AppendLabelsFile_Failing(*TraceName*);

else

AppendLabelsFile_Passing(*TraceName*);

end

end

end

#Execution Mining (Step 2: Trace Collection)

passing, failing = dictionary{ *BasicBlockAddress: CountCovered, TaintedData* };

passing_data, failing_data = dictionary{ *InstructionAddress: BasicBlock, set(unique data at InstructionAddress)* };

FL-Basic, FL-Enhanced, FL-DFT = dictionary{ BasicBlockAddress: score};

tainted_data = dictionary{ *BasicBlockAddress: LengthOfSymDiff, InstructionAddress, Data* };

for each trace *t* in *TraceDir* **do**

if (*t*) == EXCEPTION

then

UpdateDictionaries(*failing*, *failing_data t*);

else

UpdateDictionaries(*passing*, *passing_data*, *t*);

end

end

#Execution Mining (Step 3: Fault Localization Algorithms)

FL-Basic = Ochiai(*failing*, *passing*);

SetGlobalModifierValues(*FL-Basic*);

FL-Enhanced = FilterBasicNoise(*FL-Basic*);

tainted_data = SymmetricDifference(*failing_data*, *passing_data*);

FL-DFT = IncreaseSusForTainted(*FL-Enhanced, tainted_data*);

Subroutine SetGlobalModifierValues(*FL-Basic*):

top_10_percent = 0.1 * length(*FL-Basic.keys()*));

top_10_as_int = int(round(top_10_percent));

high_score = *FL-Basic*[FIRST_KEY];

try:

lower_score = *FL-Basic*[*top_10_as_int*];

except:

lower_score = *FL-Basic*[LAST_KEY];

score_range = *high_score* – *lower_score*;

if *score_range* < .0001

then

score_range = .1;

end

big_modifier = *score_range* * 0.85;

medium_modifier = *score_range* * 0.65;

small_modifier = *score_range* * 0.45;

Return

Subroutine FilterBasicNoise (*FL-Basic*):

for each basic block address *k* and score *s* in *FL-Basic* **do**

passing_count, *passing_executions_per_trace* = *passing*[*k*];

failing_count, *failing_executions_per_trace* = *failing*[*k*];

```
    if (passing_count + failing_count) < (.1 * total_blocks)
    then
        s -= medium_modifier;
    end
    if (passing_executions_per_trace != failing_executions_per_trace)
    then
        s += small_modifier;
    end
    FL-Enhanced[k] = s;
  end
Return FL-Enhanced

Subroutine SymmetricDifference (failing_data, passing_data):
  for each <BasicBlock:Instruction> address pair k in common and failing traces do
    try:
        bb_pass, pass_data = passing_data[k];
    except:
        bb_pass = 0;
        pass_data = Set(empty);
    try:
        bb_fail, fail_data = failing_data[k];
    except:
```

```
        bb_fail = 0;
        fail_data = Set(empty);

    diff_data = pass_data.SymmetricDifference(fail_data)

    if bb_pass != bb_fail
    then
        if length(pass_data) < length(fail_data)
        then
            bb_pass = bb_fail;
        end
    end
    InstructionAddress = k.split(":")[1];
    tainted_data[bb_pass] = (length(diff_data), InstructionAddress, diff_data);
  end
Return tainted_data

Subroutine IncreaseSusForTainted (FL-Enhanced, tainted_data):
  FL-DFT = FL-Enhanced;
  for each basic block address k and count c in tainted_data do
    if c == 0
    then
```

FL-DFT[*k*] += *small_modifier*;

else

FL-DFT[*k*] += *big_modifier*;

end

end

Return *FL-DFT*

To test our FL methodology three approaches were created, each providing a list of ordered suspicious location: basic, enhanced, enhanced with tainting. Each list of ordered suspicious locations is referred to as: FL-Basic, FL-Enhanced, and FL-DFT. Each result set is output as a file, which can be loaded into IDA Pro to pinpoint and visualize the fault locations. Each approach is described in the following sections.

FL-Basic

FL-Basic follows Equation 1, an Ochiai coverage-based algorithm, which is implemented in EM. All of the FL techniques in this dissertation operate on basic blocks (BBs), since access to source code (and subsequent line number information) is not assumed.

FL-Enhanced

FL-Enhanced is the first improvement researched to increase the accuracy of the FL-Basic algorithm. As is later shown in the results, it improves performance over the FL-Basic approach, but does not yet address the data-only problem.

FL-Enhanced adds enhancements to the basic coverage-based algorithm to filter out noise associated with large amounts of basic block tracing. This filtering is achieved by adding score modifiers. The most effective modifier in the *FilterBasicNoise* subroutine of Algorithm 2 adds a penalty for basic blocks that do not appear in a significant number of runs (*s* -= *medium_modifier*). The intuition is simple: if a particular code block is recorded only a couple times out of 100 total traces, it is assumed the block is unimportant (e.g. noise). The other modifier in that subroutine increases a basic block's suspiciousness score by a smaller amount if the number of times that block was executed per trace differs between passing and failing traces, indicating something was different about the block, probably the data it operated on. These techniques may seem simple, but prove to be effective when combined.

FL-DFT

FL-DFT is the novel fault localization technique in this research, which includes the tainted input tracking enhancements:

- The *SymmetricDifference* function gathers together the locations (basic block and instruction address pairs) that read from tainted locations, or that were emitted because of tainted API parameters.
- The tainted data is collected as a symmetric difference (SD) set between passing and failing traces. That is the set of elements which are in either of the sets and *not* in their intersection. The intuition is that, the unique tainted data, in particular unique tainted data that only shows up in failing traces is the most important. Specifically, from *IncreaseSusForTainted* of Algorithm 2:

1. The length of the SD set is maintained along with the tainted data at that basic block location.
2. If the SD set is of length (count variable *c*) zero a small modifier is added to the suspiciousness of that location, else a larger modifier is added. The intuition: if different values are being used (thus the non-zero SD set), the chance that this BB is related to important (changing) data is even higher.

For all these enhancements, the key is to allow the FL algorithm to ignore the large amount of irrelevant statements, and focus on those that matter.

CHAPTER 7 TEST RESULTS

The effectiveness of the proposed approaches is demonstrated over standard FL techniques on three different kinds of problems: contrived problems for clarity of presentation, on the popular Siemens test set, and finally on large real-world problems.

Contrived

To illustrate how FL-DFT works, a contrived example is presented, by again performing FL against Code Listing 3 which showed a data-only bug. The fault can now correctly be found because:

- The *xsize* variable is at the offset position in the file input, which is related to the bug. That makes *xsize* suspicious data.
- Therefore, *xsize* gets tainted and tracked.
- Subsequently, *bug3* would also be tainted because it uses *xsize* in its initialization.
- This increases the suspiciousness of the basic block that includes line 11.
- The scores of lines 19 and 20 increase as well since they use *xsize* or its descendants.

A standard coverage-based approach could not differentiate among the blocks in Code Listing 3 since the code statements were executed an equal number of times in both passing and failing traces. However, taint tracking differentiates the suspiciousness of the blocks because of the flow of the data through the blocks. As a result, lines 11, 19, and 20 are the top suspicious code blocks, demonstrating the validity of this approach.

GZIP

Knowing that the Software-artifact Infrastructure Repository [SIR 2011] includes test programs used by fault localization researchers, those programs were considered to validate the FL approach in this dissertation. Grep, Flex, Space, GZIP, and the Siemens test suite were among the programs examined. However, since fault localization techniques often ignore, or cannot handle tests that cause a crash, very few of the programs include crashing tests. A crash was found in one of the versions included with GZIP (also in 8 Siemens versions discussed later).

The GZIP compression program (based on version 1.0.7) has a vulnerability based on a stack buffer overflow. The vulnerable version (gzip/ versions.alt/ versions.seeded/ v1) has a function called *get_suffix,* which when the FAULTY_F_KL_2 bug is enabled and compiled as part of the program, exhibits memory corruption because of a misused *strcpy* function. This is shown on lines 13-15 and 18 of Code Listing 4.

CODE LISTING 4. *Get_suffix* function

```
1.  local char *get_suffix(char *name) {
2.    int nlen, slen;
3.    char suffix[MAX_SUFFIX+3]; /* last chars of name, forced to lower case */
4.    static char *known_suffixes[] =
5.      {z_suffix, ".gz", ".z", ".taz", ".tgz", "-gz", "-z", "_z",
6.  #ifdef MAX_EXT_CHARS
7.        "z",
8.  #endif
```

```
9.          NULL};
10.    char **suf = known_suffixes;


11.    if (strequ(z_suffix, "z")) suf++; /* check long suffixes first */
12.    nlen = strlen(name);
13. #ifdef FAULTY_F_KL_2
14.    if (nlen > MAX_SUFFIX+2)
15. #else
16.    if (nlen <= MAX_SUFFIX+2)
17. #endif
    {
18.       strcpy(suffix, name);
19.    } else {
20.       strcpy(suffix, name+nlen-MAX_SUFFIX-2);
21.    }
22. #ifdef SUFFIX_SEP
23.    /* strip a version number from the file name */
       {
24.      char *v = strrchr(suffix, SUFFIX_SEP);
25.      if (v != NULL) *v = '\0', nlen = v - name;
       }
```

```
26. #endif
27.     strlwr(suffix);
28.     slen = strlen(suffix);
29.     do {
30.       int s = strlen(*suf);
31.       if (slen > s && suffix[slen-s-1] != PATH_SEP
32.           && strequ(suffix + slen - s, *suf)) {
33.           return name+nlen-s; }
34.     } while (*++suf != NULL);
```

The block with the vulnerable *strcpy* is *gzip+38f9* which is colored yellow in Figure 8.

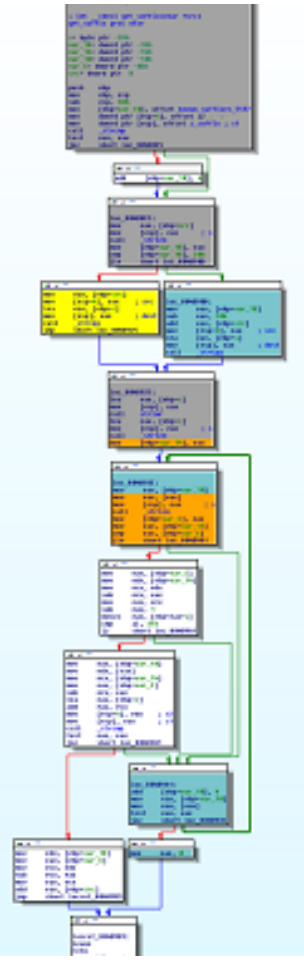

Figure 8: *get_suffix* Function from GZIP

Figure text not meant to be readable, for color and block reference only.

Table 4 shows the top 10 suspicious code blocks from the three EM FL algorithms[13].

---

[13] Note in early portions of this dissertation the actual virtual address was used, where here an offset relative to the executable module is shown. Either a relative address within a program as shown here or the absolute virtual address is handled properly within EM.

Table 4: Top 10 Suspicious Code Blocks from FL-Basic, FL-Enhanced, and FL-DFT

FL-Basic				FL-Enhnd				FL-DFT			
Basic Block	Rating	Pass	Fail								
gzip+d5e	1	0	4	gzip+d5e	1	0	4	gzip+8de0	1.20	0	4
gzip+d58	1	0	4	gzip+d58	1	0	4	gzip+8db6	1.20	0	4
gzip+8de0	1	0	4	gzip+8de0	1	0	4	**gzip+38f9**	1.20	0	4
gzip+8db6	1	0	4	gzip+8db6	1	0	4	gzip+d5e	1	0	4
gzip+8db1	1	0	4	gzip+8db1	1	0	4	gzip+d58	1	0	4
gzip+390b	1	0	4	gzip+390b	1	0	4	gzip+8db1	1	0	4
gzip+38f9	1	0	4	**gzip+38f9**	1	0	4	gzip+390b	1	0	4
gzip+f81	0.755	3	4	gzip+d98	0.755	3	4	gzip+dbe	0.963	3	4
gzip+f60	0.755	3	4	gzip+bc8	0.755	3	4	gzip+8def	0.963	3	4
gzip+ed0	0.755	3	4	gzip+98da	0.755	3	4	gzip+3b0d	0.963	3	4

The interesting property of these results is that while all of the three algorithms technically found the correct basic block as the first answer, there were many other blocks with the same score. According to the Ochiai algorithm, ties must be manually resolved. Note that FL-DFT helps in this case by creating a small tie of only three locations, as opposed to the seven locations indicated by FL-Basic and FL-Enhances. This improved score results from the fact that *name* variable is tainted when used in the *strcpy*. Since, *strcpy* is one of the APIs our Pintool can recognize and record, that block receives a boost in its score according to the FL-DFT algorithm.

Pure-FTPd Network Server

To further test our new FL algorithms, a bug was added to an open source FTP server called Pure-FTPd, version 1.0.21. This example allows us to highlight the following three points:

1. Though the focus of this fuzzing and analysis work has been on desktop applications, it was also important to show that our tools could operate on networked code.
2. To illustrate the benefits of visualization during manual bug analysis after FL is complete.
3. To illustrate the potentially subjective nature of analyzing fault localization techniques.

A stack buffer overflow was added in the TYPE command of the FTP server. Since EM currently works for file-based inputs, a passing and failing test suite was also created. Writing a tool to automatically create similar tests would be possible for network inputs as well, but was not addressed for this research.

Tainting network input is similar to working with file and standard input data on UNIX. The primary change is that network socket file descriptors are added to the tracked list. Reads (and other related system calls such as the *recv* function) on tracked file descriptors sets taint on those memory locations. By describing this bug in detail, the information display is shown to make fault localization more useful than a simple top X suspicious locations that would result otherwise.

Code Listing 5 shows code for the *dotype* function. The bug occurs on lines 18-19, which was inserted for this research. If the input to the TYPE command is greater than DEFAULT_MAX_TYPE (400 bytes), then up to DEFAULT_MAX_TYPE_LEN (1024 bytes)

will be copied into a static local buffer, creating a stack buffer overflow. The input argument length is checked, and if it is above 400 bytes, it is copied into the local buffer (an easy coverage-based condition to identify). The fix would be to remove lines 18-19. However, when operating a coverage-based fault localization program, is the bug line 18 or 19? Furthermore, since compiled code is operated on, there could be multiple basic blocks associated with a line. When presenting the results of the FL algorithm, which basic block(s) should the algorithm present? In this case there is one block that contains the *strncpy*, which we call the buggy basic block.

CODE LISTING 5. Relevant code from *dotype* function

```
1.  void dotype(const char *arg) {
2.      replycode = 200;
3.      char TYPE[DEFAULT_MAX_TYPE];
4.      if (!arg || !*arg) {
5.          addreply(501, MSG_MISSING_ARG "\n" "A(scii) I(mage) L(ocal)");
6.      } else if (tolower((unsigned char) *arg) == 'a')
7.          type = 1;
8.      else if (tolower((unsigned char) *arg) == 'i')
9.          type = 2;
10.     else if (tolower((unsigned char) *arg) == 'l') {
11.         if (arg[1] == '8') {
12.             type = 2;
```

```
13.       } else if (isdigit((unsigned char) arg[1])) {
14.           addreply_noformat(504, MSG_TYPE_8BIT_FAILURE);
15.       } else {
16.           addreply_noformat(0, MSG_MISSING_ARG);
17.           type = 2;
          }
18.   } else if (strlen(arg) > DEFAULT_MAX_TYPE) {
19.       strncpy(TYPE, arg, DEFAULT_MAX_TYPE_LEN);
      }
20.   else {
21.       addreply(504, MSG_TYPE_UNKNOWN ": %s", arg);
      }
22.   addreply(0, MSG_TYPE_SUCCESS " %s", (type > 1) ? "8-bit binary" : "ASCII");
```

Figure 9 shows the IDA disassembly of the buggy procedure, the *dotype* function.

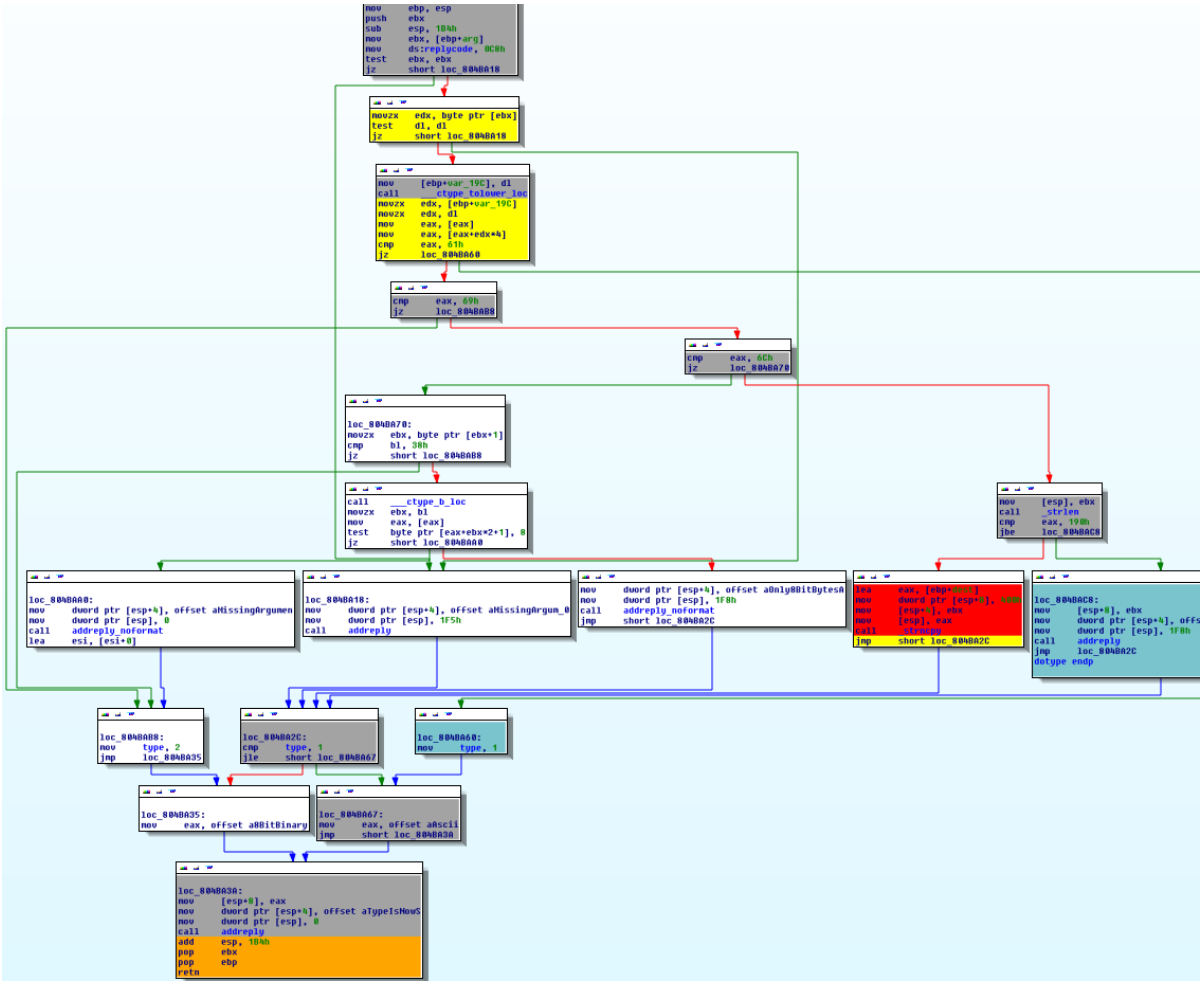

Figure 9: *dotype* Function from Pure-FTPd

Figure text not meant to be readable, for color and block reference only.

The coloring is from FL-DFT. The *strncpy* function (line 19) is the left side of the branch trio all the way to the right in the picture, colored red by FL-DFT. A branch trio is the visualization of a condition check and both of the possible branch blocks. Table 5 shows FL-DFT did better than the other two approaches.

Table 5: Results for Pure-FTPd

	FL-Basic	FL-Enhanced	FL-DFT
Rank in suspicious file	2	2	1

In this case, the *src* pointer of the *strncpy(dst, src, n)* API call points to tainted information. Thus, the *strncpy* block gets a tainted boost to its suspiciousness score (as do other unrelated blocks). That is why FL-DFT scored the buggy block higher in the list of suspicious code blocks.

Finally, the merits of visualization shown in Figure 9 are discussed. Note that the second half of the final basic block is orange. As stated earlier, orange indicates that this code region is the common final ending[14]. Note that the key assembly directive in this block is a return instruction (RET). The reason for the crash is that the stored return pointer is corrupted by a stack buffer overflow. Anyone with experience with memory corruption bugs will quickly check to see if there are any local buffers, and if so, note where they are used. The local buffer gets

[14] Since this dissertation is concerned with crashing and non-crashes test cases, orange areas indicate the most common location of the crash. Note that this location is generally not the bug, and may or may not be near the actual bug.

used by the *strncpy*[15] function. The color helps because the analyst can quickly visualize important forensic details about the program execution, such as:

- Blocks executed
- Suspicious blocks
- Crash location
- Flow through branches
- Rough top-to-bottom execution sequence

For example, the block all the way to the right in Figure 9 is blue, meaning that the block was executed only in passing traces. Conversely, note that the other branch target of that same branch trio is red, which indicates two things: (1) the red block was executed at least once by a failing trace, and (2) that block is considered suspicious. Debugging intuition added to the provided coloring, points to this branch and the *strncpy* block as part of the coding mistake.

Thus, the top X suspicious locations are only a portion of the helpful information that can be provided. Even if the FL algorithm had failed to find the correct basic block (by identifying it in the top 5 choices), an analyst familiar with stack bugs could look at the coloring in this function and surmise the real code error. This feature is a useful side benefit.

---

[15] The *strncpy* function is a safer version of the traditionally misused *strcpy* function, because of the addition of a maximum size parameter. As seen here, it is still possible to misuse most functions.

Siemens Benchmark

The new FL approaches were tested using the Siemens benchmarks [SIR 2011], which are commonly used to demonstrate the effectiveness of fault localization techniques. The Siemens test set originates from work done at Siemens Corporate Research on data-flow and control-flow test adequacy [Hutchins et al. 1994].

This research targets the subset of bugs known as vulnerabilities, especially memory corruption bugs. If these vulnerabilities are first found in the field (as opposed to the code developers), those systems can be exploited. Because the focus here is on vulnerabilities, only the program versions in the Siemens test set which include a memory corruption bug were used.

Table 6: Comparing FL Types (1=Success; 0=Failure)

	FL-Basic	FL-Enhanced	FL-DFT	Top-K LEAP
print_tokens2: v10	1	1	1	1
replace: v23	1	1	1	1
schedule: v1	0	1	1	1
schedule: v5	0	1	1	1
schedule: v6	0	1	1	1
schedule2: v8	0	0	1	0
tot_info: v18	1	1	1	1
tot_info: v20	0	1	1	1
Average	37.50%	87.50%	100%	87.50%

Table 6 compares our three test variations against one of the best fault localization techniques, Top-K LEAP [Cheng et al. 2009]. A “1” indicates that the known bug was

successfully found; "0" indicates the FL approach failed to find the bug. Observe that FL-Enhanced is able to match the effectiveness of Top-K LEAP, and that FL-DFT outperforms Top-K LEAP.

Of particular interest are the results for the schedule2, version 8 bug. That bug is not properly localized by 3 of the approaches. To be properly localized, the code block must be in the top 5 suspicious locations. The rest of this section describes why FL-DFT is able to successfully localize the bug in schdeule2:v8.

The particular bug in question results from missing code. The routine within the schedule2 program with the bug is a function called *put_end*. The code is shown in Code Listing 6.

CODE LISTING 6. Relevant code from *put_end* function

```
1. int put_end(int prio, struct process * process) /* Put process at end of queue */ {
2.     struct process **next;
       /*if(prio > MAXPRIO || prio < 0) return(BADPRIO); */
       /* find end of queue */
3.     for(next = &prio_queue[prio].head; *next; next = &(*next)->next);
4.     *next = process;
5.     prio_queue[prio].length++;
6.     return(OK);
```

The bug occurs between lines 2 and 3; a commented input validation check that needs to be uncommented. FL-Basic found the erroneous area as the 19th choice, FL-Enhanced as the 12th

pick, and FL-DFT found the bug as the first choice. FL-DFT performs better because it has access to relevant information (usage of tainted data), that other FL algorithms do not. In particular, the information in *prio* and *prio_queue* are tainted, both of which increase the suspiciousness scores of the relevant basic blocks in the *put_end* function. Table 7 summarizes the results.

Table 7: Results for Schdule2, Version 8

	FL-Basic	FL-Enhanced	FL-DFT
Rank in suspicious file	19	12	1

VLC Media Player

The VLC Media Player [2011] is a free and open source cross-platform multimedia player and framework that plays most multimedia files as well as DVD, Audio CD, VCD, and various streaming protocols. VLC had a bug, which was documented by Klein [2011]. The version 0.9.4 had a stack buffer overflow when a particular size field in the .ty+ file format (TiVo's digital video) was larger than a local 32 byte buffer. Code Listing 7 portrays a section of this large code base.

CODE LISTING 7. Relevant code from *parse_master* function

```
1. static void parse_master(demux_t *p_demux) {
2.   demux_sys_t *p_sys = p_demux->p_sys;
3.   uint8_t mst_buf[32];
4. int i, i_map_size;
5. int64_t i_save_pos = stream_Tell(p_demux->s);
```

```
6.  int64_t i_pts_secs;
7.  free(p_sys->seq_table);
    /* parse header info */
8.  stream_Read(p_demux->s, mst_buf, 32);
9.  i_map_size = U32_AT(&mst_buf[20]); /* size of bitmask, in bytes */
10. p_sys->i_bits_per_seq_entry = i_map_size * 8;
11. i = U32_AT(&mst_buf[28]); /* size of SEQ table, in bytes */
12. p_sys->i_seq_table_size = i / (8 + i_map_size);
    /* parse all the entries */
13. p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
14. for (i=0; i<p_sys->i_seq_table_size; i++) {
15.     stream_Read(p_demux->s, mst_buf, 8 + i_map_size); …
```

In the *parse_master* function, the first *stream_Read* function (line 8) inputs 32 bytes of the relevant header. Parsed out of that information is the *i_map_size* variable, which is later misused (line 15) on the next *stream_Read*, leading to a local stack buffer corruption if the size is great than 32.

Table 8: Results for VLC

	FL-Basic	FL-Enhanced	FL-DFT
Rank in suspicious file	456	288	3

Table 8 shows the results of our three approaches. The actual buggy block (containing the second *stream_Read*) was correctly identified by only FL-DFT. Recall, the buggy basic block is required to be in the top 5 to be considered successfully identified. The top yellow block in Figure 10 is the buggy block. All three got close to the actual basic block, by reporting a nearby basic block (the one in red). However, by closely inspecting each of the top 10 blocks reported by the FL-DFT approach, one would note that all 10 are better, in the sense that all of the basic blocks are in functions that operate on actual user data, rather than basic blocks that just happen to show up more in failed runs. In other words, apart from the red block, FL-Basic and FL-Enhanced more or less report random (noisy) blocks that just happened to show up more in failing traces, but were not relevant to the bug. On the other hand, nearly all of the top ten blocks reported by FL-DFT are meaningful code used to parse input data.

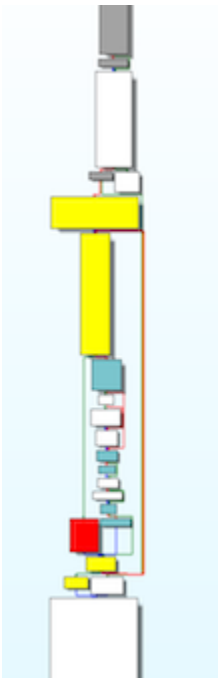

Figure 10: Visualized *parse_master* Function in IDA Pro via FL-DFT

The CF/EM system works particularly well with this example. The bug is easily found via fuzzing. A sample input can be found at: http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty+. One mutation fuzzing heuristic, which moves a DWORD (0xffffffff is common) through the input file, will find this bug when the pertinent size field gets changed. EM will then isolate the changed byte at offset 0x0030017 in the file. The original byte was the size 0x02, and when changed to 0xff, will crash VLC (trigger the bug). The beauty of our taint tracking is that it does not over taint. That is, if all user input (the whole file) is tainted as it is read in, many basic blocks will be tagged, and thus the suspiciousness of many basic blocks will increase. Tainting everything may be better than no tainting, but it is not as helpful as only tagging the blocks that deal with the byte (0x02 → 0xff) at offset 0x00300017. By tagging just that one byte, only the highly relevant blocks are tagged as tainted, and pushed to the top of the suspiciousness list. EM does this automatically, since the byte offset position is known (from the test set creation phase of EM). Note also that that automatic test set creation works as intended. As new files are created, each the same, except that this particular byte varies in value (0x00 – 0xff), some will trigger the bug (with value > 32) and others will not (with value < 32).

Note that this bug also shows off a data-only situation. The only way this BB can be distinguished is by considering data, since both passing and failing traces flow through this block.

Firefox

To test our FL techniques against a large, real-world program the open source Mozilla Firefox browser [Firefox 2011] was chosen. Version 3.5.8 was used because there was a public PNG

(graphic format) bug, which had an in-depth description on Bugzilla detailing the manual process developers followed to determine the root of the crash [Bugzilla-Mozilla 2011]. The cause of the bug comes from code in the file *nsPNGDecoder.cpp* in the location *../ modules/ libpr0n/ decoders/ png /*. The *row_callback* function (line 684, revision 1.9.1) omits an important check to see if the *row_num* is in bounds. Code Listing 8 shows the small portion of relevant code from the repaired revision 1.9.2, with the added code on lines 6-7.

CODE LISTING 8. Relevant code from *row_callback* function

```
1. void row_callback(png_structp png_ptr, png_bytep new_row, png_uint_32 row_num,
   int pass) {
2. nsPNGDecoder *decoder =
   static_cast<nsPNGDecoder*>(png_get_progressive_ptr(png_ptr));
3.  // skip this frame
4. if (decoder->mFrameIsHidden)
5.    return;
6. if (row_num >= decoder->mFrameRect.height)
7.    return;
8. …
```

The FL-DFT technique was able to correctly localize the bug, while the FL-Basic and FL-Enhanced techniques were not. The primary reason is that in large programs there is often too much noise and not enough signal. Coverage-based fault localization presumes there is a

reasonable amount of signal to noise. Conversely, if code is widely variable between traces, even with similar inputs, the Ochiai algorithm may key in on unrelated, but unique code from failing traces as the source of the bug.

To avoid such problems it is necessary to focus on the portion of large code bases that are *interesting* in order to reduce the variability in analysis. One technique used is to note which module (DLL in Windows) the exception occurs in, and only perform the FL within that module. Isolation of the problem solves many variability issues. However, in this case Firefox always uses a very large module called *libxul*, which is a combination of many other libraries designed to decrease the start time when Firefox is launched, making the code to analyze still very large. (When compiled, *libxul.so* is 482MB which is a very large compiled object.) Thus, only analyzing one module is not a satisfying solution in this case. This problem was overcome by honing in on important code, using tainted input tracking to key in on the code operating on relevant data.

FL-Basic technique found the correct code as the 100th choice in the ordered list of suspicious locations. FL-Enhanced found it as the 34th choice. FL-DFT identified the correct function with the 1st choice. That information is summarized in Table 9.

Table 9: Results for Firefox PNG

	FL-Basic	FL-Enhanced	FL-DFT
Rank in suspicious file	100	34	1

CHAPTER 8 SIGNIFICANCE OF OUR PRIMARY FAULT LOCALIZATION ENHANCEMENT

In this chapter, we want to establish that our primary fault localization contribution is significant. We will show that the data-flow faults that we can find occur with sufficient frequency that it is a problem worth solving. We begin by first showing a need to incorporate data information as part of a code coverage-based FL algorithm—a need that has been observed by others. We finish by illustrating the effectiveness of our FL improvements shown in Chapter 7.

We will also estimate the frequency of the data-flow bugs that would not be well localized without our improvement. We combine our earlier results from the Siemen's test set with a count from a large software project. Because the best, coverage-based FL studies showed precisions as high as 75%, we expect that the remaining bugs to be less than 25%, and we expect our data-flow bugs to make up a significant portion of that remaining 25%. If we can find data-flow bugs to make up greater than 5% of total bugs, we will conclude that they are significant.

Lack of Focus on Data

The seminal Siemens paper [Hutchins et al. 1994] is the original source of the benchmark programs used in Chapter 7. In that paper, Hutchins et al. sought to determine if data-flow and control-flow testing strategies would aid traditional specification-based approaches. Not surprisingly, they found that both techniques aided in the discovery of bugs when supplementing the popular informal methods of the day. Given their observations it is interesting that so many fault localization techniques have focused primarily on control-flow, and have left data information by the wayside.

Consider a quote from the Hutchins paper: "*Finally, we saw an approximately equal split between faults that were detected at higher ratios by Edge coverage and by DU [data definition-usage pair] coverage, leading us to conclude that the two methods, according to the way they are measured by Tactic, are frequently complementary in their effectiveness*." To be clear, our primary improvement (FL-DFT) is an overall fault localization improvement. Because it is paired with a coverage-based technique it works well for code and data bugs. The analysis in this chapter is not to indicate that our new technique works only against data-flow and noisy bugs, but to understand how significant this new improvement is.

The researchers of Top-K LEAP noted in their conclusion, "*In this work, we only consider the control-flow information when building graphs and extracting discriminative signatures. We try to make the signature rich enough by considering various edge types so that it has enough expressiveness to discriminate the buggy cases. We find that at times this is not sufficient as the data-flow information matters. The same control-flow could be executed by both buggy and normal behaviors, which might differ only on the data that flows in along this control-flow*" [Chang et al. 2009]. In our initial investigations, we also noticed bugs that were not well localized by existing techniques. That is why we sought to improve existing FL techniques by incorporating data information.

Improvement on Data-only and Noise Problems

In Chapter 7, we compared our FL techniques (FL-Basic, FL-Enhanced, and FL-DFT) to Top-K LEAP. Top-K LEAP properly localized 7 out of 8 bugs in the Siemens benchmark set. Our FL-DFT, which includes data information, solved 8 out of 8. The difficult bug required data-flow

information to solve well. That experiment provides some insight into the prevalence of data-flow bugs.

When investigating our FL techniques on other problems our enhancements again showed improvement. The following is a summary, based on the sections in Chapter 7:

1. FL-DFT did the best on GZIP, primarily because FL-DFT was able to better score the important code block because that block used tainted input. The GZIP code base was smaller than VLC and Firefox, so noise was not as big a problem, but as shown, data-flow was.
2. FL-DFT did the best on the VLC Media Player code. Noise and data-flow were both problematic. FL-Enhanced helped with the noise (37% improvement), but only FL-DFT was able to correctly locate the bug (99% improvement). FL-DFT did this by keying in on tainted data used by the API function at fault.
3. FL-DFT did the best on Firefox. Noise and data-flow were both problematic. Again, FL-Enhanced significantly outperformed FL-Basic (by 66% in this case). But only FL-DFT was able to cut through the many potential buggy code blocks, and identify the real fault by using data-flow information (99% improvement).

Thus, we surmise that the data-flow issue and the noise introduced by large or variable code sets, are important considerations that have previously been left unsolved. Our FL-Basic builds on a prior FL algorithm (Ochiai) by incorporating a bug hunting front end. FL-Enhanced improves FL-Basic by filtering out unimportant code blocks. FL-DFT performs the best because it builds on prior techniques, and also includes the ability to highlight the most important code blocks by using input data tracking.

Size Estimate

From the Siemen benchmark tests, 1 out of 8 (12.5%) of the faults examined were not properly localized without data information. The Siemen's benchmark set came from a commercial code base so it is a well-accepted test set. Recall that there were more than eight tests, but there were only eight that led to memory corruption vulnerabilities—the bugs we are most interested in. Therefore, the Siemens set indicates 12% as our initial estimate on the size of the set of bugs not well localized without our enhancement.

Second Estimate

To get another data point we used the Red Hat Linux Bugzilla repository [Bugzilla-Redhat 2012]. We used this data source because it was one of the few repositories that included (1) a variety of real-world bugs, and (2) sufficient information on the flaw, the original source code, and the required fix to the source code, such that we could get a reasonably accurate fix on characteristics of each bug.

Out of thousands of bugs, 51 of the most relevant bugs were extracted from the Red Hat project (shown in Appendix: Significance Data). These 51 were found by searching the repository for memory corruption bugs—bugs that are most useful to an attacker. The 51 bugs are from different components and different versions of Linux, so they provide breadth in our estimates. The bugs occurred from 2007 to 2012.

We examined each bug manually to determine if data-flow information would likely be required to properly localize the bug. We choose a manual approach for a number of reasons. One is that manual examination allows us to better reason about the circumstances of the bug—had we simply run the examples through our code we would have only found what our code

found and that might be different. There was also a practical reason: setting up and running 51 samples through our test framework would have taken months without yielding better results.

We will explain our methodology using two of the Linux bugs. The first example shows code that contains no control-flow logic near the bug. That is, control-flow alone cannot isolate the bug. Therefore, we classify this bug as a data-flow bug because it would require data information to localize the buggy line of code. In the second example, we show the opposite situation: the block of code that contains the bug has an if-statement (relevant control-flow), and would likely be correctly identified by code-coverage FL techniques. We showed similar examples earlier in Chapter 5 via Example 1 and Example 2.

Methodology Examples

As an example of our methodology for determining if a bug is a control or data-flow candidate, consider the Enterprise Linux 5 kvm bug number 72938 (Appendix: Significance Data). If we examine the flaw, and the fix, we see no control-flow that would aid an algorithm in localizing to this location. The flaw is shown in the top of Figure 11, and fix is shown in the bottom of Figure 11. Since all paths must flow through line 801, we mark this bug as best being localized with data information. In other words, regardless of the flows through this function the *depth* variable may be in error, until the fix is applied.

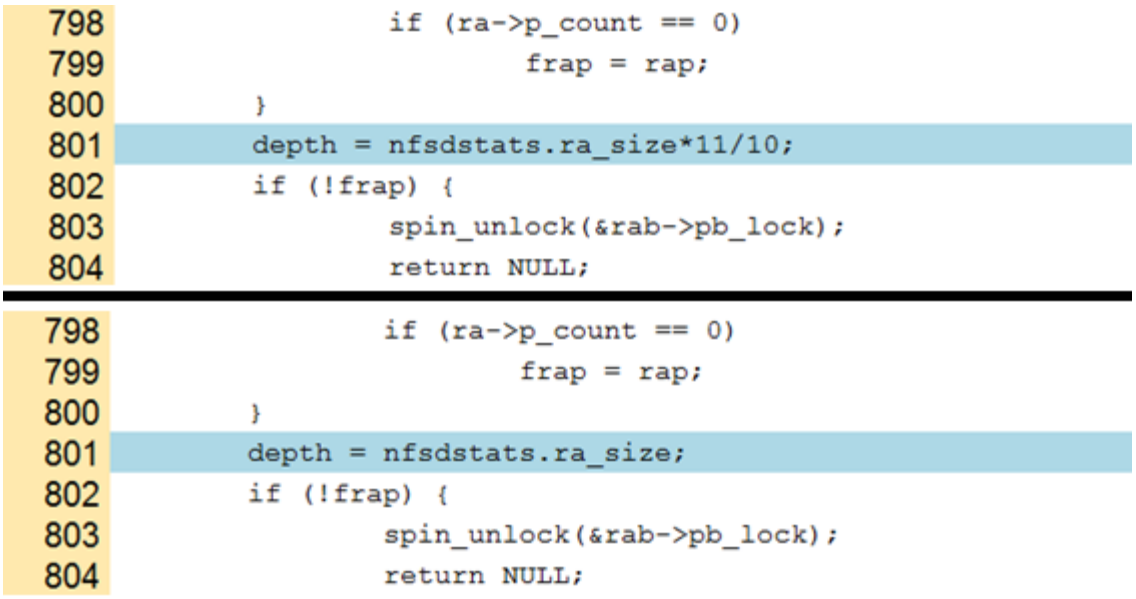

Figure 11: File ./fs/nfsd/vfs.c

For an example of a control-flow bug, consider the Linux Xen bug number 222467 (Appendix: Significance Data). The flaw is the top of Figure 12, and the fix is the bottom of Figure 12. The bug happens when code within the branch statement is executed (lines 445-450). Ochiai-based algorithms are proficient at localizing code errors when there is an obvious control-flow difference. The fix moved the erroneous code to a different location in the code base.

```
441         goto error_out;
442     }
443
444     /* HVM domains must be put into shadow mode at the start of
day */
445     if ( xc_shadow_control(xc_handle, domid,
XEN_DOMCTL_SHADOW_OP_ENABLE,
446                            NULL, 0, NULL,
447                            XEN_DOMCTL_SHADOW_ENABLE_REFCOUNT  |
448                            XEN_DOMCTL_SHADOW_ENABLE_TRANSLATE |
449                            XEN_DOMCTL_SHADOW_ENABLE_EXTERNAL,
450                            NULL) )
451     {
452         PERROR("Could not enable shadow paging for domain.\n");
453         goto error_out;
454     }
455
456     memset(ctxt, 0, sizeof(*ctxt));
457
458     ctxt->flags = VGCF_HVM_GUEST;
```

```
441         goto error_out;
442     }
443
444     memset(ctxt, 0, sizeof(*ctxt));
445
446     ctxt->flags = VGCF_HVM_GUEST;
```

Figure 12: File ./tools/libxc/xc_hvm_build.c

We choose the above approach based on our prior observations of when code coverage-based techniques fail. In Chapter 5 we previously showed two contrived examples of when code-coverage works, and when it fails. The underlying difference is that distinguishable control logic should reside near the bug, for code coverage-based techniques to operate well.

Second Estimate Results

Of the Red Hat bugs, 6 out of 51 (11.76%) could be best localized with the presence of data information. For a combined estimate, we total the number of control-flow and data-flow bugs with the Siemens estimate. This information is summarized in Table 10. Since the three percentages are similar, we conclude that 12% is a reasonable estimate.

Table 10: Data-flow Estimate

	Control-flow	Data-flow	Percent
Red Hat	45	6	11.76%
Siemens	7	1	12.50%
Total	52	7	11.86%

Reasonableness

We note that the precision of two prior fault localization studies was 72.5% (for RAPID) and 74.3% (for Top-K Leap) [Chang et al. 2009]. These two techniques are among the best available coverage-based fault localization implementations. The first thing to note about each precision is that there is room for improvement. That fact is expected, since they do not claim to be perfect approaches. Improvement might come if a class of bugs being missed is found. FL-DFT did this

when it found 8 of 8 in our Siemens study. Second, consider these two statistics together: the set size estimate from Table 10 and the best precision (Top-K). By adding the two numbers (74.3%+11.86% = 86.16%) we note that there is still room for improvement, if 100% of faults are to be localized. The fact that a chance for further improvement still exists is expected. If we had argued that the size estimate was a number that, when added to the best precision, summed to a number greater than 100%, our proposed set size estimate would have seemed unrealistically high. It is reasonable to assume that even if Top-K included the ability to identify data-only bugs, there would still be imperfection in the approach, though it would likely perform better. This information is summarized in Table 11.

Table 11: Reasoning about the Accuracy of the Data-flow Estimate

	FL Precision	
RAPID	72.5%	
Top-K	74.3%	
	Best FL Precision	74.3%
	+ Data-flow Estimate	11.86%
	Reasonably, sum < 100%	86.16%

Summary

In this chapter we showed that our primary FL enhancement was needed. We also showed that it is significant both in terms of its ability to better localize a subset of bugs, and that the size of that subset is significant. We support the need of our new algorithm, based on prior researcher

observations. We show the ability of our approach to better localize based on our automated, empirical results. We estimated the size of the set of bugs not well localized without our enhancement in the following manner:

1. We obtained the set size based on our empirical Siemens study, and on a manual study of Red Hat Linux bugs. Both show the size to be approximately 12%.
2. We presume this number to be reasonable, because it is within the thresholds we originally described: it is less than 25% and more than 5%.

Therefore, we say that our estimate is both reasonable and significant. We desired this knowledge as further credence to our FL contribution.

CHAPTER 9 CONCLUSIONS

This dissertation presents enhancements to existing coverage-based fault localization (CBFL) techniques. The best approach utilizes dynamically collected taint information on important data inputs. Specifically, tainted blocks receive an addition to their base suspiciousness score. This technique puts emphasis on code regions processing untrusted input, with the basic premise that those blocks are more likely to be at fault, and more likely to contain bugs which could be translated into an important subclass of bugs known as vulnerabilities or software exploits.

Our new CBFL is also combined with a distributed or parallel fuzzing framework. Finding bugs in parallel scales. Our framework also sorts and prioritizes discovered bugs. Finally, a user-friendly graphical interface allows testers or security researchers to enter notes about the particular bug so that team testing is encouraged.

Merit

This dissertation research provides a number of merits for the field of software engineering and software security research. A distributed bug hunting and analysis system was created and enhanced and has the following key contributions:

1. Fault localization is improved by incorporating input data tainting, married with a traditional coverage-based technique. The presented empirical studies show that the approach is important, and obtains superior results to prior techniques, by testing against:
    a. Contrived problems
    b. The Siemens test set
    c. Large and real-world programs.

2. New vulnerabilities were discovered in Java, showing the effectiveness of the distributed testing platform.
3. The closed source (assembly) visualization was provided to aid the analyst in understanding correctly and even incorrectly localized bugs.

Threats to Validity

Fault categorization has been noted to be difficult. Consider the following quote: "*Different authors propose different fault categories, and relatively few arguments exist that would clearly support their particular choices … Furthermore, we conclude that the distribution of software faults depends heavily on project-specific factors, such as the maturity of the software, operating environment or programming language*" [Ploski et al. 2007]. Therefore, the estimate on the set of data-only bugs would likely vary if the language and environment changes significantly.

Santelices et al. [2009] note that different types of bugs might be best localized by different FL techniques. However, since the bug type is generally not known ahead of time, multiple coverage types performed better in their research. Clearly the "correct" approach to all of fault localization has yet to be discovered and, just as clearly, more research needs to be done. Moreover, developers need to integrate Fault Localization more closely with the development or exploitation phases to become more commonly useful. This research takes a step forward with the CF/EM system.

Bug Hunting Limitations

Our fuzzer is only capable of finding bugs typically found with dynamic testing: denials-of-Service, crashes, use-after-free, race-conditions, and all manner of memory corruption (off-by-

one, buffer overflows, print format errors, etc.). Design flaws and other architectural mistakes are unlikely to be uncovered by fuzzing. Additionally, dynamic testing rarely covers all code, but focuses on the interfaces.

Fault Localization Limitations

Our FL scheme is focused on the typical memory corruption bugs discovered by a fuzzer. Therefore, our pass/fail oracle is no-crash/crash. Additionally, our system is tuned for compiled code on the x86 architecture. Porting our techniques to other oracles and architectures should be a straight-forward endeavor.

Future Work

This dissertation leads to many possible future research projects. The following are the most likely:

1. Future work can focus on continued data augmentations. Coverage-based approaches are popular, effective, and quick, but can fail because of unfocused analysis. This research suggests that continued work that supplements coverage-based work via data techniques hold promise.
2. Future work can focus on new or enhanced techniques to locate bugs. Work also done at Michigan State University by the author, relates to the use of genetic algorithms to enhance fuzzing programs [Takanen et al. 2008].
3. Future work that focuses on further automation of the bug hunting and analysis pipeline is related and of high interest, particularly to the security community. Symbolic execution [Avgerinos et al. 2011], automated exploit development [Heelan 2009], and

automated crash analysis [Miller 2010] are related to the pipeline approach taken by our distributed bug hunting framework. Incorporating those notions into our framework would provide value in the fields of software testing and security research.

APPENDICES

APPENDICES

ClusterFuzz GUI Walkthrough

The following is a brief walkthrough of an early version of the ClusterFuzz (CF) graphical user interface (GUI). The GUI is accessed via a web browser. The GUI is capable of controlling all of the CF system. The discussion and figures add more detail to the prior CF description. Only major features of the tool are discussed. Harris has since renamed the tool to *CyberFuzz* and continues work on it. Further detail or inquiries can be made by contacting Gary Kay (gkay@harris.com).

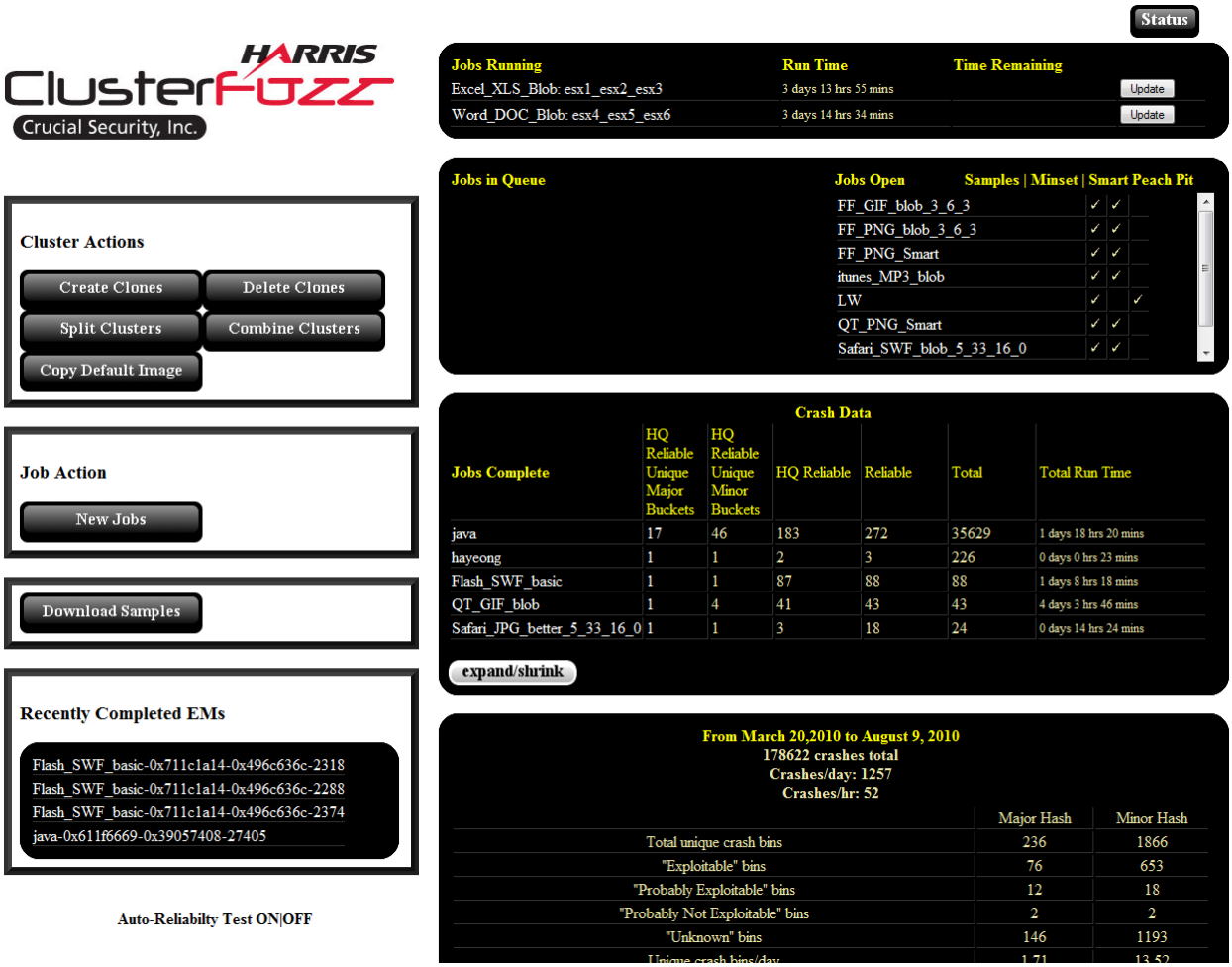

Figure 13: ClusterFuzz Primary View

Figure text not meant to be readable, for visual reference only.

In Figure 13, the *Cluster Actions* (upper left) controls the segmenting or cohesion of available ESX servers. New jobs are created with the *Job Action*. The top right block shows the *Jobs Running*. The *Jobs in Queue* shows the current jobs waiting to run. The *Crash Data* block shows a summary of the "high quality" results. Runs that did not discover at least *probably exploitable* results or better are not shown. The final window on the bottom right, shows an overview of all the results collected.

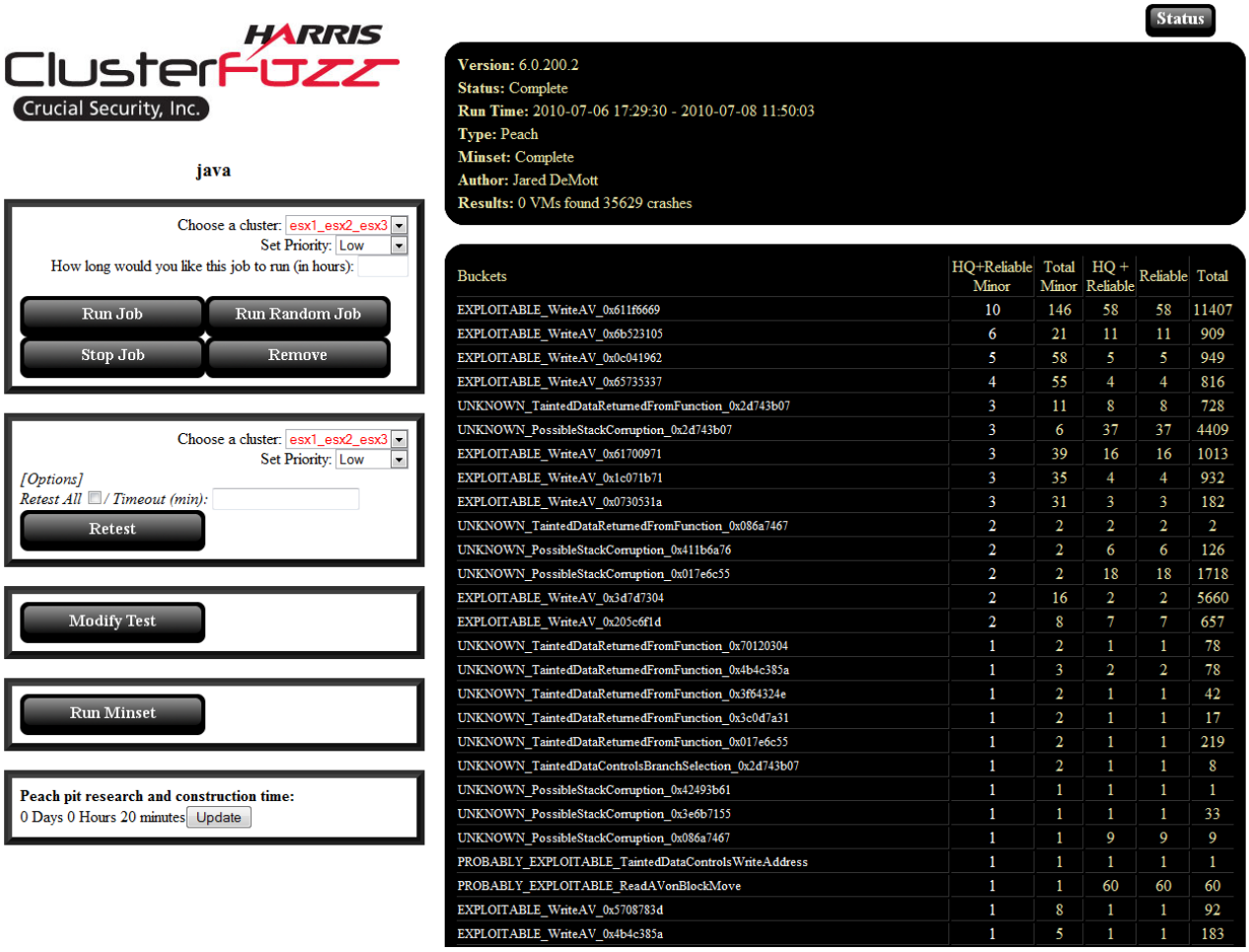

Figure 14: Clicked on Job Results for Java

Figure text not meant to be readable, for visual reference only.

In Figure 14, the upper left contains actions for the created job, such as *Run Job* or *Stop Job*. Once a job is complete, the pane below that can be manually used to retest results for reliability. By default, this happens automatically, called Auto-Retest. The MinSet feature can be controlled below that. To the right, note the results from the job ordered by reliability, number in each

bucket, and severity. The *HQ* stands for "high quality" (an !exploitable rating of probably exploitable or better), and the *Reliable* means the bug is repeatable.

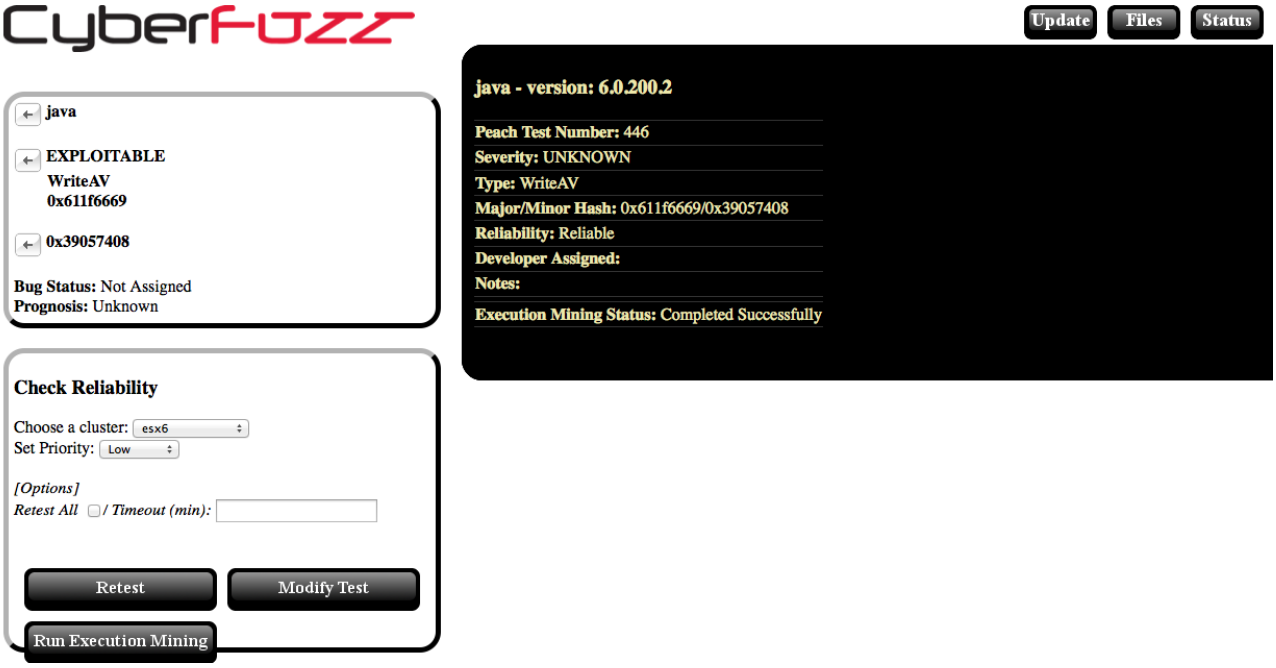

Figure 15: Clicked on Specific Java Results

Figure text not meant to be readable, for visual reference only.

In Figure 15, the left window shows the job name, the type and major bucket ID, and the minor hash bucket ID. Execution Mining can be manually controlled from the window below, but runs automatically for file inputs. On the right, notes about the result can be entered, or the relevant files can be downloaded for further inspection (click on *Files* in upper right). Files include a crash trace, the original input, the fuzzed input, and a python script to manually exercise the bug.

Significance Data

Table 12 provides the details for the 51 bugs used in Chapter 8.

Table 12: Bugzilla-Redhat Details

Red Hat	Bug ID/Link	Component	Data-only	Date fixed	Date reported	Days to repair	Issue
Enterprise Linux 5	220592	Kernel-xen	0	11/7/07	12/22/06	320	kernel boot panic with >=64GB memory
All Linux	222467	xen	0	2/14/07	1/12/07	33	[XEN-VT]'Cannot allocate memory' error
Enterprise Linux 5	236121	ipsec-tools	0	6/27/07	4/11/07	77	LSPP: racoon has a buffer overflow when receiving large security context from kernel
Enterprise Linux 5	316371	Kernel-xen	0	5/21/08	11/30/07	173	32-bit PAE HV hardware limitation > 4GB memory
Enterprise Linux 5	423521	kernel	0	1/20/09	12/13/07	404	memory leak on size-8192 buckets with NFSV4
Enterprise Linux 5	432867	kernel	0	1/20/09	2/14/08	341	memory corruption due to portmap call succeeding after parent rpc_clnt has been freed
Enterprise Linux 5	435271	anaconda	0	6/11/08	2/28/08	104	RuntimeError: Error running lvm: Cannot allocate memory
Enterprise Linux 5	435291	kernel	0	1/20/09	2/28/08	327	LTC41974-Pages of a memory mapped NFS file get corrupted.
Enterprise Linux 5	445348	kernel	0	5/29/08	5/6/08	23	Please port the mtrr fixes from 2.4.25 so that some hardware can boot with >4GiB memory
Enterprise Linux 5	446188	kernel	0	1/20/09	5/13/08	252	BUG: Don't reserve crashkernel memory > 4 GB on ia64

Table 12 (cont'd)							
Enterprise Linux 4	448046	kernel	0	5/18/09	5/23/08	360	memory corruption due to portmap call succeeding after parent rpc_clnt has been freed
Enterprise Linux 5	452756	kernel	0	10/7/09	6/24/08	470	panic at intel_i915_configure() when amount of shared video memory set to 1M
Enterprise Linux 5	453392	virt-manager	0	1/16/09	6/30/08	200	virt-manager.py is taking all the memory!
Enterprise Linux 5	470267	kernel	0	1/20/09	11/6/08	75	cifs: data corruption due to interleaved partial writes timing out
All Linux	470769	hfs filesytem	1	12/23/10	11/10/08	773	CVE-2008-5025 kernel: hfs: fix namelength memory corruption
Enterprise Linux 5	471613	kernel	0	1/6/10	11/14/08	418	Corruption on ext3/xfs with O_DIRECT and unaligned user buffers
Enterprise Linux 5	478643	kernel	0	9/2/09	1/2/09	243	multipath test causes memory leak and eventual system deadlock
Enterprise Linux 5	485173	kernel	0	2/20/12	2/11/09	1104	kernel/module-verify-sig.c with memory uncleaned bug
Enterprise Linux 5	487672	kernel	0	9/2/09	2/27/09	187	slab corruption with dlm and clvmd on ppc64
All Linux	491988	gfs2-utils	0	3/27/09	3/24/09	3	GFS2 File System Corruption in RHEL5.3
Enterprise Linux 5	499553	kernel-xen	0	1/13/11	5/7/09	616	Cannot generate proper stacktrace on xen-ia64
Enterprise Linux 5	503062	ksh	0	4/8/10	5/28/09	315	ksh trashes its internal memory, can dump core
Enterprise Linux 5	503905	kernel	1	9/2/09	6/3/09	91	kernel: TPM: get_event_name stack corruption [rhel-5.4]
Enterprise Linux 5	507846	kernel-xen	0	1/13/11	6/24/09	568	Balloon driver gives up too easily when ballooning up under memory pressure
Enterprise Linux 5	514412	autofs	0	3/30/10	7/28/09	245	RHEL 5.5 RFE - autofs add configuration option to increase max open files and max stack size
Enterprise	521865	kernel	0	3/30/10	9/8/09	203	Xen fails to boot on ia64 with > 128GB memory

Table 12 (cont'd)							
Fedora	532302	pam	1	11/11/09	11/1/09	10	pam_console memory corruption
Enterprise Linux 5	546133	autofs	0	7/21/11	12/10/09	588	autofs - memory leak on reload
All Linux	548249	perl	0	2/21/12	12/16/09	797	Perl threads leak memory when detached
Enterprise Linux 5	552211	dhcp	0	1/14/10	1/4/10	10	dhcpd memory leak when failover configured
Enterprise Linux 5	558809	kernel	0	3/30/10	1/26/10	63	e1000 & e1000e: Memory corruption/paging error when tx hang occurs
Enterprise Linux 5	565973	kernel	0	1/13/11	2/16/10	331	[EMC 5.6 bug] security and PSF update patch for EMC CKD ioctl
Enterprise Linux 5	568912	All Linux	0	3/9/10	2/26/10	11	Occasional data corruption using openssl.i686 AES ciphers on HP dc5850 systems
Enterprise Linux 5	580509	All Linux	0	1/13/11	4/8/10	280	Give correct memory amount back to dom0 when ballooning fails
Enterprise Linux 5	581933	kernel	0	1/13/11	4/13/10	275	pci_mmcfg_init() making some of main memory uncacheable
Enterprise Linux 5	585979	kexec-tools	0	1/13/11	4/26/10	262	kdump fails to save vmcore on machine with 1TB memory
Enterprise Linux 5	590073	libvirt	0	1/13/11	5/7/10	251	Memory leak in libvirtd
Enterprise Linux 5	601800	kernel	0	1/13/11	6/8/10	219	NFS-over-GFS out-of-order GETATTR Reply causes corruption
Enterprise Linux 6	606335	corosync	1	11/10/10	6/21/10	142	malloc(): memory corruption
Enterprise Linux 5	606919	xen	0	1/13/11	6/22/10	205	Memory leak in xenstore
Enterprise Linux 5	623219	gdb	0	1/13/11	8/11/10	155	gdb aborts with a 'double free or corruption' when calling inferior functions with print or call command
Enterprise Linux 5	625828	kexec-tools	0	1/13/11	8/20/10	146	kdump will not start on ppc64 server with 256G of memory enabled

Table 12 (cont'd)							
Enterprise Linux 5	660292	m2crypto	0	1/13/11	12/6/10	38	memory leak in the attached script
Enterprise Linux 5	663563	kernel	0	7/21/11	12/16/10	217	[ext4/xfstests] 011 caused filesystem corruption after running many times in a loop
Enterprise Linux 5	665011	xen	0	7/21/11	12/22/10	211	backport various fixes for memory leaks
All Linux	672812	xorg-x11-drv-mga	0	12/6/11	1/26/11	314	xorg-x11-drv-mga
Enterprise Linux 5	674314	kernel	1	6/21/11	2/1/11	140	NFSD: memory corruption due to writing beyond the array
Enterprise Linux 5	678308	kexec-tools	0	2/20/12	2/17/11	368	kexec kernel crashes due to use of reserved memory range
Enterprise Linux 5	681440	xenpv-win	0	6/8/11	3/2/11	98	xenpv-win drivers leak memory on (misaligned I/O && backend is busy)
Enterprise Linux 5	690555	kernel	0	7/21/11	3/24/11	119	GFS2: resource group bitmap corruption resulting in panics and withdraws
Enterprise Linux 5	729381	kvm	1	4/8/12	8/9/11	243	memory corruption handling the acceptable command line option, leads to segfault
		Percent FL-DFT:	11.76%		Avg Fix Days:	263	

BIBLIOGRAPHY

BIBLIOGRAPHY

ABREU R., ZOETEWEIJ P., and VAN GEMUND, A. J. C. 2007. On the Accuracy of Spectrum-Based Fault Localization. *In Proc. TAIC PART*.

AGRAWAL, H. and HORGAN, J. 1990. Dynamic Program Slicing. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* pages 246-256.

AGRAWAL, H., DEMILLO, R., and SPAFFORD, E. 1993. Debugging with dynamic slicing and backtracking. *Software Practice and Experience* (SP&E), Vol. 23, No. 6, pages 589-616.

AGRAWAL, H., HORGAN, J.R., LONDON, S., WONG, W.E. 1995. Fault Localization using Execution Slices and Dataflow Tests. *Software Reliability Engineering. Proceedings., Sixth International Symposium on*, vol., no., pp.143-151.

AMINI, P. and PORTNOY, A. 2010. Sulley Fuzzing Framework. http://www.fuzzing.org/wp-content/SulleyManual.pdf

ARTZI, S., DOLBY, J., TIP, F., and PISTOIA, M. 2010. Directed Test Generation for Effective Fault Localization. In *Proceedings of the 19th international symposium on Software testing and analysis* (ISSTA '10). ACM, New York, NY, USA, 49-60.

AVGERINOS, T., CHA, S., K., HAO, B., L., T., and BRUMLEY, D. 2011. AEG: Automatic Exploit Generation. *18th Annual Network and Distributed System Security Symposium*. San Diego, CA.

BUGZILLA. 2011. https://bugzilla.mozilla.org/show_bug.cgi?id=570451

BUGZILLA. 2012. https://bugzilla.redhat.com/

CHENG, H., LO, D., ZHOU, Y., WANG, X., and YAN, X. 2009. Identifying Bug Signatures using Discriminative Graph Mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (ISSTA '09). ACM, New York, NY, USA, pages 141-152.

CLARKE, R.A. AND KNAKE, R.K. 2010. Cyber War: The Next Threat to National Security and What to Do. Ecco, New York.

CLEVE H. and ZELLER, A. May, 2005. Locating Causes of Program Failures, *In proceedings of the International Conference on Software Engineering*, pages 342-351, St. Louis, Missouri.

DEMOTT, J., ENBODY, R., and PUNCH, W. July 2007. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. *BlackHat USA.*

EDDINGTON, M., CACCHETTI, A., KAMINSKY, D. 2011. Showing How Security Has (And Hasn't) Improved, After Ten Years Of Trying. *CanSecWest Security Conference*. http://www.dejavusecurity.com/files/DMK_AC_DD_Cansec_Fuzzmarking.pptx

EXPLOITABLE. 2011. http://msecdbg.codeplex.com/

FIREFOX. 2011. http://www.mozilla.org/projects/firefox/

GALLAGHER, T., Office Security Engineering. 2009. *BlueHat*. http://technet.microsoft.com/en-us/security/ee460903.aspx#gallagher

GASSER, M. May, 1988. Building a secure computer system. *Van Nostrand Reinhold.* ISBN 0-442-23022-2.

GODEFROID, P., LEVIN, M., MOLNAR, D. 2008. Automated Whitebox Fuzz Testing. *NDSS*

GOEL, S. August 2011. Cyberwarfare: connecting the dots in cyber intelligence. Commun. ACM 54, 8, pgs. 132-140.

GREENGARD, S. December 2010. The new face of war. Commun. ACM 53, 12, 20-22.

GUPTA, N., HE, H., ZHANG, X., and R. GUPTA. November, 2005. Locating Faulty Code Using Failure-Inducing Chops. *IEEE/ACM International Conference on Automated Software Engineering*, pages 263-272, Long Beach, California.

GUPTA, R., GUPTA, N., XIANGYU ZHANG, JEFFREY, D., NAGARAJAN, V., TALLAM, S., CHEN TIAN. April 2008 Scalable dynamic information flow tracking and its applications. *Parallel and Distributed Processing*. IEEE International Symposium on , vol., no., pp.1-5, 14-18.

HEELAN, S. 2009. Automatic Generation of Controlflow Hijacking Exploits for Software Vulnerabilities. *MSc Computer Science Dissertation, Oxford University*.

HEX-RAYS. 2011. http://www.hex-rays.com/idapro/

HSU, H., JONES, J. A., and ORSO, A. 2008. Rapid: Identifying Bug Signatures to Support Debugging Activities. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (ASE '08). IEEE Computer Society, Washington, DC, USA, pages 439-442.

HUTCHINS, M., FOSTER, H., GORADIA, T., and OSTRAND, T. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *Proceedings of the 16th international conference on Software engineering* (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, USA, pages 191-200.

JEFFERY, T., GUPTA, N., and GUPTA, R. 2008. Fault localization using Value Replacement. *In Proceedings of the international symposium on Software testing and analysis* (ISSTA '08). ACM, New York, NY, USA, pages 167-178.

JONES, J., A., and HARROLD, M., J. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. *In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (ASE '05). ACM, New York, NY, USA, pages 273-282.

JONES, J. A. April, 2008. Semi-Automatic Fault Localization. *Dissertation*, Georgia Institute of Technology.

KLEIN, T. 2011. A Bug Hunter's Diary: A Guided Tour through the Wilds of Software Security. *San Francisco: No Starch Press.*

KOREL, B., and LASKI, J. 1988. Dynamic program slicing. *Information Processing Letters* (IPL), Vol. 29, No. 3, pages 155-163.

RANDOLPH, K. December, 2009. Fuzzing Reader - Lessons Learned. http://blogs.adobe.com/asset/2009/12/fuzzing_reader_-_lessons_learned.html

LEWIS, W. April, 2000. Software Testing and Continuous Quality and Improvement, 2nd Edition, pp. 29-39. ISBN 0-849-39833-9.

LIBDFT. 2011. http://www.cs.columbia.edu/~vpk/research/libdft/

LIBLIT, B., AIKEN, A., ZHENG, A. X., and JORDAN, M. I. June, 2003. Bug Isolation via Remote Program Sampling, *In Proceedings of the Conference on Programming Language Design and Implementation*, San Diego, California.

LIBLIT, B., NAIK, M., ZHENG, A., AIKEN, A., and JORDAN, M. 2005. Scalable Statistical Bug Isolation, *ACM*.

LIU, C., YAN, X., FEI, L., HAN, J. and MDKIFF, S. P. September 2005. Sober: Statistical Model-Based Bug Localization*, In ESEC/FSE.*

LUCAS, J. 1961. Minds, Machines, and Gödel. Philosophy XXXVI pages 112-127.

MCMINN, P. AND HOLCOMBE, M. March 2006. Evolutionary Testing Using an Extended Chaining Approach. *ACM Evolutionary Computation*, Pgs 41-64, Volume 14, Issue 1

MERCEDES, K. and WINOGRAD, T. October, 2008. Enhancing The Development Life Cycle To Produce Secure Software. *Data & Analysis Center for Software*.

MICROSOFT. 2011. http://www.microsoft.com/security/sdl/

MILLER, B.P., FREDRIKSEN, L., and SO, B. December, 1990. An Empirical Study of the Reliability of UNIX Utilities, *Communications of the ACM*. http://pages.cs.wisc.edu/~bart/fuzz/fuzz.html

MILLER, B.P., KOSKI, D., LEE, C.P., MAGANTY, V., MURTHY, R., NATARAJAN, A., and STEIDL, J. April 1995. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. *Computer Sciences Technical Report #1268*, University of Wisconsin-Madison.

MILLER, B.P., COOKSEY, G., and MOORE, F. July 2006. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. *First International Workshop on Random Testing*. Portland, Maine.

MILLER, C. 2010. Babysitting an Army of Monkeys: An analysis of fuzzing 4 products with 5 lines of Python. *CanSecWest*.

MILLER, C., CABALLERO, J., JOHNSON, N., M., KANG, M., G., MCCAMANT, S., POOSANKAM, P., and SONG, D. 2010. Crash Analysis with BitBlaze. *Black Hat USA*.

MOLNAR, D. and OPSTAD, L. 2010. Effective Fuzzing Strategies. *CERT Vulnerability Discovery Workshop*. http://www.cert.org/vuls/discovery/downloads/CERT-presentation-dmolnar-larsop.pdf

MULLINER, C. and MILLER, C. 2009. Fuzzing the phone in your phone. *Black Hat USA*. http://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf

MYERS, G. J. 1979. The Art of Software Testing. *John Wiley and Sons*. ISBN 0-471-04328-1.

NAGY, B. July, 2009. Finding Microsoft Vulnerabilities by Fuzzing Binary Files with Ruby – A New Fuzzing Framework, *SyScan*. http://www.syscan.org/Sg/program.html

OEHLERT, P. March/April 2005. Violating Assumptions with Fuzzing. *IEEE Security & Privacy*, Pages 58-62.

PAN, H. and SPAFFORD, E. 1992. Heuristics for Automatic Localization of Software Faults. *Technical Report SERC-TR-166-P*, Purdue University.

PEACH. 2011. http://peachfuzzer.com/

PIN. 2011. http://www.pintool.org/

PIROMSOPA, K. and ENBODY, R. October-December, 2006. Secure Bit: Transparent, Hardware Buffer-Overflow Protection. *IEEE Transactions on Dependable and Secure Computing, Vol. 3, No. 4*.

PLOSKI, J., ROHR, M., SCHWENKENBERG, P., AND HASSELBRING, W. 2007. Research Issues in Software Fault Categorization. *SIGSOFT Software Eng. Notes*, vol. 32, no. 6, p. 6.

RENIERIS, M. and REISS, S. 2003. Fault Localization with Nearest Neighbor Queries. *In ASE*.

SANTELICES, R., JONES, J., A., YU, Y., and HARROLD, M., J. 2009. Lightweight Fault-Localization using Multiple Coverage Types. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 56-66.

SAHA, D., NANDA, M. G., DHOOLIA, P., NANDIVADA, V. K., SINHA, V., and CHANDRA, S. 2011. Fault localization for data-centric programs. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 157-167.

SECURE DEVELOPMENT LIFECYCLE TEAM. April 2007. Lessons learned from the Animated Cursor Security Bug. *Microsoft Blog Post*. http://blogs.msdn.com/b/sdl/archive/2007/04/26/ lessons-learned-from-the-animated-cursor-security-bug.aspx

SOFTWARE-ARTIFACT INFRASTRUCTURE and REPOSITORY (SIR). 2011. http://sir.unl.edu/portal/index.html

SUTTON, M. and GREENE, A. 2005. The Art of File Format Fuzzing. *BlackHat USA*.

SUTTON, M. 2006. Fuzzing: Brute Force Vulnerability Discovery. ReCON. www.recon.cx/en/f/msutton-fuzzing.ppt

SUTTON, M., GREENE, A., and AMINI, P. 2007. Fuzzing: Brute Force Vulnerability Discovery. Addison Wesley. ISBN: 0-321-44611-9.

TAKANEN, A., DEMOTT, J., and MILLER, C. 2008. Fuzzing for Software Security Testing and Quality Assurance. *Artech House, Inc. Norwood, MA*. ISBN: 1-596-93214-7.

TASSEY, G. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*. Planning Report 02-3.

TESSERACT OCR. 2011. *http://code.google.com/p/tesseract-ocr/*

VESSEY, I. 1985. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A process analysis*, Vol. 23, No. 5, pages 459-494.

VLC MEDIA PLAYER. 2011. http://www.videolan.org/vlc/

VOAS, J.M. and MILLER, K.W. 1992. The Revealing Power of a Test Case. *Software Testing, Verification and Reliability*, 2(1): 25–42.

WONG, W. and DEBROY, V. 2009. Software Fault Localization. *IEEE Reliability Society 2009 Annual Technology Report*.

YAN, X., CHENG, H., HAN, J., and YU, P., S. 2008. Mining Significant Graph Patterns by Leap Search. In *Proceedings of the ACM SIGMOD international conference on Management of data* (SIGMOD '08). ACM, New York, NY, USA, pages 433-444.

ZELLER A. September, 1999. Yesterday, my Program Worked. Today, it does not. Why? *Seventh European Software Engineering Conference/ Seventh ACM SIGSOFT Symposium on Foundations of Software Engineering* (ESEC/FSE), pages 253-267.

ZHANG, X., HE, H., GUPTA, N., and GUPTA, R. 2005. Experimental evaluation of using dynamic slices for fault location. *In AADEBUG'05: Proceedings of the International Symposium on Automated Analysis-driven Debugging*, pages 33–42, Monterey, California, USA.

ZHANG, X., TALLAM, S., GUPTA, N., and GUPTA, R. June, 2007. Towards Locating Execution Omission Errors, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego.