Adding Diversity and Realism to LAVA, a Vulnerability Addition System

by

Rahul Sridhar

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

Master of Engineering in Electrical Egineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by .....
Timothy Leek
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chairman, Department Committee on Graduate Theses

Adding Diversity and Realism to LAVA, a Vulnerability Addition System

by

Rahul Sridhar

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Egineering and Computer Science

Abstract

In this thesis, I designed and implemented several modifications to LAVA, a vulnerability addition system, with the goal of improving realism and diversity of the injected bugs. Specifically, I describe three separate improvements: a method to add fake bugs alongside real ones in order to decrease bug discoverability, two approaches to increase the complexity of the data flow of the inserted bugs, and a standalone program that uses equality saturation to diversify C-source codebases that can be added as a final stage to LAVA. Finally, I present two instances of bug-finding competitions—AutoCTF and Rode0day—that I helped design and run, which leveraged LAVA and the augmentations described in this thesis in order to accomplish their respective goals.

Thesis Supervisor: Timothy Leek
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

I would like to thank Professor Brendan Dolan-Gavitt for reviewing this thesis and providing me with valuable feedback, and Timothy Leek for the mentorship, guidance, and wisdom he's given me over the past two years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Vulnerability discovery, the process of identifying potentially exploitable bugs in software has been and remains an active area of research. However, progress in the field is hampered by a lack of ground truth data with which to evaluate, compare, and improve different bug-finding tools and techniques. While datasets of known vulnerabilities exist, they are time-consuming to compile by hand and often too small to be truly useful for evaluation.

One promising alternative is to use *automated vulnerability injection*, the process of programmatically adding known bugs to a piece of software, in order to create large and high-quality ground truth corpora of known vulnerabilities.

Of course, this approach is most useful if the injected bugs are highly realistic: that is, similar in nature to mistakes a human developer might make when building an application. The best vulnerability insertion approaches would have the ability to quickly inject a large number of realistic bugs into a source program, which could then be used as a benchmark to compare the effectiveness of different bug-finding approaches.

LAVA (Large-scale Automated Vulnerability Addition), created by researchers at NYU, Northeastern University, and MIT Lincoln Laboratory [4], is one such bug insertion tool. LAVA injects vulnerabilities in C source code by identifying points in the source program's trace when user-controlled data is available and artificially changing the value of a pointer involved in a memory read or write if that user-

controlled data is equal to a particular magic value.

The user-controlled data and the memory read/write are identified through the use of a dynamic taint analysis using the PANDA dynamic analysis platform [5], which is itself based on QEMU, a full-system emulator.

This thesis describes an array of modifications and extensions made to LAVA over the course of several years in order to improve the realism and diversity of LAVA's injected bugs.

Chapter 2 discusses prior work in the fields of bug injection and software diversification, and chapter 3 explains the design and implementation of the original LAVA system.

Chapter 4 describes a method to inject "fake bugs" alongside real bugs using LAVA with the purpose of making the real bugs more difficult to locate. Chapter 5 describes several techniques designed to complicate the *data flow* of the injected LAVA bugs, in order to increase their realism and similarity to true vulnerabilities. Chapter 6 describes a technique developed to increase the diversity of LAVA bugs.

Finally, chapter 7 examines two different bug-finding competitions that both leverage LAVA and the extensions described in prior chapters in order to achieve their specific objectives.

1.1 Objectives for injected bugs

At a high level, the primary purpose of LAVA is to improve the state of the art in vulnerability discovery. Given this, we want a tool optimized to identify LAVA bugs to also be successful at discovering vulnerabilities in real software. Specifically, we'd like LAVA bugs to exhibit *realism*, namely, that they are similar in syntax and discoverability to bugs in real programs.

We'd also like to make the bugs we inject be *diverse*. In the context of vulnerabilities, diversity can refer to two distinct concepts: a diversity of bug classes (buffer overflows, integer overflows, logical bugs) or a diversity of manifestations (injecting the same bug in different syntactic ways). Both concepts are valuable to have in a

bug injection system, but this thesis focuses only on the latter.

Finally, we'd also like the bugs LAVA injects to not be trivial to discover. Specifically, we'd like to be able to exert some control over an injected bug's *difficulty*. Note that we don't necessarily want to inject only the most difficult bugs: if the state of the art in vulnerability discovery cannot currently discover any LAVA bugs, the corpora produced by LAVA would not be particularly useful for evaluation and improvement.

1.2 Applications of Vulnerability Addition

The primary application of LAVA is to produce ground truth datasets. These corpora can be used to compare the effectiveness of different bug-finding tools. They can also be used to quickly gauge whether a new feature is an improvement or a regression.

On a more fundamental level, these corpora are vital for the long-term progress of vulnerability discovery as a field of scientific research. Several basic questions in the field such as "what makes a bug 'easy' or 'hard'?" and "which techniques are best suited to finding which bugs?" are impossible to answer without ground truth data for conducting experiments.

Finally, LAVA can be used to run bug-finding competitions. In other fields like computer vision [14], speech recognition [2], and SAT-solving [1], regularly-held competitions are a way to drive innovation and reveal the current state-of-the-art in the field.

In the field of vulnerability discovery, there have already been a handful of such contests—most notably, DARPA's Cyber Grand Challenge held in 2016 [1]. Relatedly, hundreds of *Capture the Flag* competitions (CTFs), which are generally targeted at assessing manual (human-driven) bug-finding and exploitation skills, are held by various organizations every year.

These competitions are time-consuming to create and run: the organizers expend large amounts of effort to create creative new challenges. LAVA enables us to cheaply and more easily run a similar type of competition.

[1] http://archive.darpa.mil/cybergrandchallenge/

Chapter 2

Related Work

2.1 Vulnerability Corpora

There have been several attempts to compile a corpus of software vulnerabilities that can be used for vulnerability-discovery analysis. In 2004, Zitser et al. built a dataset of 14 known buffer overflow vulnerabilities in open source programs and measured detection and false alarm rates across five different static analysis tools [19].

Building on this work, Kratkiewicz et al. [8] created a taxonomy of 22 types of buffer overflow vulnerabilities and used this classification to synthesize 291 small, vulnerable test programs. The authors evaluated the same five static analysis tools on the same five test programs and evaluated their detection rates.

Finally, several large databases of vulnerabilities exist and are actively maintained and grown. Most notably, MITRE's Common Vulnerabilities and Exposures [1] (CVE) system assigns a unique ID and description to publicly known security vulnerabilities.

Each of these efforts fall short in one or more ways that prevent the resulting corpus from being truly useful as ground truth data for progress in the field of automated vulnerability discovery. Manual compilation can produce high-quality results, but is costly to execute and results in small datasets—the authors in [19] estimate that it took them 6 months to build their dataset of 14 bugs). Worse still, the produced dataset can quickly become stale once the state of the art in the field manages to

[1] https://cve.mitre.org/

achieve complete or significant success on the corpus.

Automatically generating a corpus of programs from a classification of vulnerabilities as in [8] can produce large corpora, but the resulting programs are often too small to be representative of the task faced by tools trying to find bugs in large code bases.

In contrast, MITRE's CVE system is both large and utterly realistic, given that it is composed solely of discovered vulnerabilities, but since it doesn't provide a standardized method to isolate the vulnerable source code, it's difficult to transform it into a usable corpus.

2.2 Automated Vulnerability Insertion

Automated vulnerability insertion is a relatively novel concept and as a result there is relatively little literature on similar systems. Of the few systems that do exist, most either take radically different approaches than LAVA or inject bugs in order to solve a different problem.

Evilcoder [12], created by researchers at the Horst-Görtz Institute, is another vulnerability insertion tool. It works by attempting to identify and then disable various security checks such as input sanitization, thereby making the program vulnerable to exploitation via user-controlled data. Evilcoder was evaluated on several open-source projects and successfully identified hundreds of possible bug insertion points.

Unlike LAVA, Evilcoder relies solely on static program analysis: namely, it operates by examining and processing the source code but without ever executing it. In contrast, LAVA is based around a dynamic analysis that is conducted while the source program is run on a particular input. This dynamic approach allows LAVA to reason very precisely about the program's state for the one program execution under analysis, which in turn allows LAVA to provide better assurances that the bugs it inserts will only trigger on a particular user input.

2.3 Software Diversification

Software diversification is the practice of altering a program at either the binary or source-code level to produce a different, yet functionally-equivalent version of the program that can be used in the original's place. Diversification can be done at the source-code, binary or any intermediate level of the compilation process.

Diversification can be useful for security. Analogous to a biological ecosystem where every individual is susceptible to the same communicable pathogen, in a software environment in which all users run identically-vulnerable versions of a program, an exploit or piece of malware can quickly propagate through a network. If instead, each system is running a unique, but functionally-equivalent version of the program, an attack against one version is unlikely to work against all other versions. This significantly increases the work factor of an attacker, who now must develop a unique exploit for each possible version of the program.

Diversification can also be useful for code obfuscation. Suppose a developer discovers a vulnerability with her software. If she simply fixes the error and releases a patch, malicious actors can analyze the patch to reverse engineer the vulnerability and develop an exploit against all systems still running the vulnerable version of the program. Indeed, Brumley et al. found that it might be possible to *automatically* generate an exploit given the original program and a patched copy [3]. If instead the developer uses diversification to mutate her entire codebase along with the patch, it becomes more difficult for an attacker to identify the particular change that corrected the vulnerability.

Similarly, our primary motivation for exploring diversification is to obfuscate source code changes introduced by the LAVA system. Currently the bugs that LAVA injects are easy to spot visually and as a result are relatively easy to "discover" by hand. If however, we augmented the LAVA-injected bugs with a source-code level diversification we would potentially hide the existence of the bugs and make them more difficult to suss out.

One technique for diversification, known as superdiversification, is inspired by a

corresponding technique in the field of compiler optimization. *Superoptimization* is an optimization technique first introduced by Massalin [10] which involves conducting an exhaustive search over all instruction sequences less than a certain length to find the optimal sequence that has identical functionality as the original sequence of instructions. Massalin introduced this concept as a way to produce shorter instruction sequences, but later researchers have adjusted the cost function to find the fastest, rather than shortest instruction sequence.

A *superdiversifier* builds on this idea, but instead of considering only the shortest or fastest instruction sequence, the diversifier uses the generated sequences to produce alternate, but functionally-equivalent, versions of the original program.

More recently, researchers at Stanford [15] built a superoptimizer called STOKE [2] which uses a randomized, Markov-Chain Monte Carlo sampler to explore the search space of all loop-free sequences of a subset of x86-64 instructions. STOKE is able to optimize for both executable size and performance, and in some cases can outperform a domain-specific compiler at certain tasks.

Jacob et al. [7] were the first to apply the technique of superoptimization to the diversification problem. Their superdiversifier operates on the byte-code level and searches over all instruction sequences less than a certain length to identify substitute sequences that can be used to create unique, but functionally equivalent versions of a given executable.

[2] http://stoke.stanford.edu/

Chapter 3

LAVA

In the context of LAVA, a vulnerability or bug is an unsafe memory read or write that can be reliably triggered by a particular user-provided input to the program, but is not triggered for all or almost all other inputs. To accomplish this, LAVA must be able to reason about how user input affects the execution of the program. This is done using a dynamic taint analysis which tracks every program variable whose value has been derived from user input. We say that a variable is *tainted* by user input if its value was computed using bytes of user input.

As shown in Figure 3-1, LAVA first uses the Clang C Compiler to insert taint-queries in the source code, and then uses PANDA, a dynamic-analysis platform, to run the modified program on a user input and execute the taint-analysis. PANDA's taint analysis is precise: each byte of user input is assigned a different positional label and for a particular byte of a program variable PANDA can determine the exact set of input bytes that it were involved in computing its current value.

Next, LAVA analyzes the results of the taint analysis in order to identify two critical pieces of the program: the DUA and the ATP. The DUA, which stands for Dead, Uncomplicated, and Available data, is a variable that has been tainted by user input but does not significantly affect the program's control flow when its value is changed. We expect that when the input is modified in a precise way, the value of the DUA will also change predictably, so that we can check the value of the DUA against a magic value and trigger the vulnerability if and only if the two are equal.

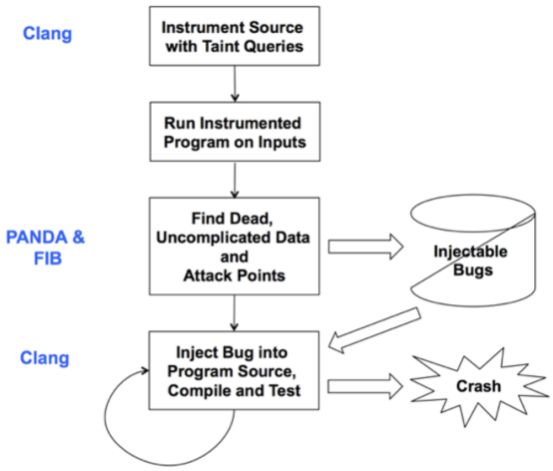

Figure 3-1: First LAVA uses Clang to insert taint queries around each line of the code. Then it uses PANDA to perform the dynamic taint analysis, and identify DUAs and ATPs. Then it uses Clang again to modify the source code to inject the bugs, causing a program crash. Taken from [4].

The second half of the vulnerability involves finding an "attack point" (ATP), which is a program instruction involving a memory read or write whose pointer can be modified. Note that an ATP must occur after a DUA in the program's execution.

LAVA modifies the source code as follows:

1. Insert code after the DUA to copy the tainted bytes to a global variable.

2. Insert code at the ATP , to check if the global variable is equal to some fixed value, such as 0xDEADBEEF.

3. If it is, then modify the pointer at the ATP to be larger than originally intended, causing an overflow when the pointer is read from or written to.

Suppose for the sake of example, that the DUA were simply a copy of the 5th through 8th bytes of user input. Now, LAVA can modify the user input that was used in the dynamic taint-analysis by changing these bytes to 0xDEADBEEF. This ensures that the value of the DUA, and hence the value of the global variable will equal 0xDEADBEEF, which will later trigger the buffer overflow at the ATP and cause a crash.

```
protected int
file_encoding(struct magic_set *ms,
              ..., const char **type) {
...
   else if
     (({int rv =
          looks_extended(buf, nbytes, *ubuf, ulen);
        if (bug) {
          int lava = 0;
          lava |= ((unsigned char*)(buf))[0]<<(0*8);
          lava |= ((unsigned char*)(buf))[1]<<(1*8);
          lava |= ((unsigned char*)(buf))[2]<<(2*8);
          lava |= ((unsigned char*)(buf))[3]<<(3*8);
          lava_set(lava);
        }; rv;})) {
...
```

Figure 3-2: An example of the DUA modifications of a LAVA-injected bug; last five lines injected. LAVA has identified `buf` as a DUA and it copies its first four bytes into lava. The global variable's value is set with the call to `lava_set`. Modified from [4].

Figures 3-2 and 3-3 present an example of the DUA and ATP source code modifications LAVA made for a particular bug inserted in the Linux `file` program.

For a DUA to be useful in a LAVA bug, it must satisfy its namesake 3 properties. First, it must be *dead*, which refers to the property that changing the DUA's value doesn't affect the program's control flow. This is needed so that if we change the DUA's value to 0xDEADBEEF, the program will still execute the ATP which will trigger the crash. LAVA ensures that its DUAs are dead by computing the *liveness* of its constituent bytes, a taint-based metric that keeps track of how many times the byte was used to decide a branch in the control flow. A byte with a liveness of 0 was never considered in any branch up to this point in the code and is therefore *dead*.

Next, the DUA must be an *uncomplicated* function of the tainted input bytes, otherwise it is difficult to predict how to set the input bytes in order to ensure the DUA ends up with the magic value. LAVA ensures this using a different taint-based metric: the *taint compute number* (TCN). The TCN measures the depth of the

```
...
protected int
file_trycdf(struct magic_set *ms,
            ..., size_t nbytes) {
  ...
  if (cdf_read_header
      (( (&info)) + (lava_get())
       * (0x6c617661 == (lava_get())
          || 0x6176616c == (lava_get())), &h) == -1)
    return 0;
```

Figure 3-3: An example of a LAVA-injected bug. LAVA has identified a read of the pointer `&info`. The global variable's value is accessed with `lava_get`. If its value equals the magic value 0x6c617661, the pointer is modified which will cause a crash. Modified from [4].

computation tree for a particular byte. For example, if a byte was a direct copy of the input, its TCN would be 0. If a byte was computed by adding 12 to an input byte, its TCN would be 1. Good candidates for DUAs should consist of bytes with low liveness and a small TCN.

Finally, the DUA must be made *available* at the attack point. It's unlikely that the DUA will happen to be in scope at the attack point, so LAVA accomplishes by copying its value to a global variable and then checking that global variable's value at the attack point.

3.1 The LAVA-M Dataset

The authors of the original LAVA paper used the system to inject hundreds of bugs into four `coreutils` programs (`base64`, `md5sum`, `uniq`, and `who`), and then released the modified source code as the *LAVA-M* corpus. In the years since the corpus was released, several new tools have been created and evaluated using the dataset and have progressively found a greater proportion of the injected bugs [9, 13, 11].

Notably, three of these tools (Steelix [9], VUzzer [13], and Angora [11]) have also found one or more 0-day vulnerabilities in real, unmodified programs. This

suggests that success at discovering LAVA bugs does translate into success at finding vulnerabilities in real programs, a promising result for LAVA.

Chapter 4

Fake Bugs (Improving "Difficulty")

4.1 The "diff" Problem

One major criticism of the original LAVA system is that an attacker who knows the source code of an unmodified program can compare this code with the LAVA-modified code and simply infer the locations of all the bugs based on the source code modifications. We refer to this difficulty as the "diff" problem.

In order to reduce the effectiveness of this technique, we developed a modification to LAVA that allows it to insert fake bugs alongside real ones. These fake bugs are at first glance indistinguishable from real bugs, but these source code modifications are highly unlikely to trigger a segmentation fault.

Now, if a theoretical adversary is given a LAVA-modified program with say, 50 fake bugs and 50 real bugs, diffing the modified program with the original source code would reveal all the modifications, but would not help in distinguishing real bugs from fake bugs. We argue that if an adversary can differentiate real modifications from fake ones with a success rate greater than random chance, then the adversary should be equally successful in isolating bugs in real programs.

4.2 Fake Bugs from Fake Duas

Our approach for inserting fake bugs is simple: fake LAVA bugs are simply regular LAVA bugs where the DUA consists of *untainted* data. Recall that in standard LAVA bugs, the DUA consists of tainted data that is often a copy of user input; this is so that modifying user input appropriately can modify the DUA to equal a magic value and trigger a crash. However, if the DUA consists of untainted data, no amount of user input modification can alter the value of the DUA, so it can never equal the magic value, and the program shouldn't crash.

4.3 Fake Bugs for security

While it may seem that the idea of introducing fake bugs is simply a way to mitigate flaws in LAVA's original bug insertion process, we believe that the concept is more broadly useful. If fake bugs are truly able to trick a human or automated vulnerability finder into wasting time attempting to compose a trigger for them, it can make it harder for an attacker to discover potential real bugs in a program.

For example, in the 2016 DARPA Cyber Grand Challenge, one competing cyber reasoning system deliberately introduced an obvious buffer overflow vulnerability in its patched programs [18]. This vulnerability, while easy to discover, was not actually exploitable because the bug could not be reached specifically on the custom CGC infrastructure.

Chapter 5

Data Flow (Improving "Realism")

In the version of LAVA presented in [4], the DUA and ATP were connected via the global variable `lava`. This presents a couple of drawbacks. Firstly, bug-finding software that is familiar with LAVA-style vulnerabilities, could simply check for the use of the `lava_set` function call in order to locate where the bug was inserted. This can be mitigated in a number of ways (for example, by inserting a number of fake bugs along with real bugs, see chapter 4), but a larger issue remains: the use of a global variable is rather unrealistic when compared to real programs.

5.1 Data Flow

The term *data flow*, in the context of program analysis, refers to the propagation of data from variable to variable over the execution trace of a program.

When we deem the original LAVA bugs as unrealistic, we're specifically referring to their data flow: they involve exactly one copy between the DUA and the ATP. Real programs—and hence real bugs—tend to exhibit far more complex data flow with more copies and modifications.

Aside from being simplistic, LAVA's original data flow is also somewhat artificial. Since the DUA and ATP are independently chosen out of a set of candidate DUAs and candidate ATPs, the two points in the program are not likely to have any semantic relationship with each other. The `lava_get` and `lava_set` methods will then

introduce an arbitrary connection.

5.2 Function Argument Addition

Our first approach involves modifying function signatures to include an additional argument that can carry the tainted data. Consider the call graph shown in figure 5-1, and suppose we inserted an additional argument `int* lava` to each of the shaded functions. Now, we can insert code at each of the parents of the DUA to propagate the value of the DUA to the function's caller and insert code at each of the parents of the ATP to propagate the value to the function's callee. Then, by the time we get to the function involving the ATP, the added function argument should contain a copy of the DUA. This approach brings with it a number of limitations: since we cannot modify library functions or their signatures, if control flow ever passes to a library function, which then calls a user function (say, when starting a thread), all subsequent functions under that node in the call tree are inaccessible for data flow and bug insertion. We could potentially overload each called library function with a stub that propagates the data flow and then calls the real function, but this is a sufficiently rare event that we did not attempt this.

Another limitation involves function pointers: currently we modify function pointer signatures and types to include the additional argument needed for dataflow. However, if a program ever uses a function pointer that points to a library function, our approach will fail. Once again, we find that this event occurs sufficiently rarely that we are comfortable with the tradeoff.

We implemented this approach by using Clang to identify all non-library function signatures and modify them to include our additional argument. We then added code before and after every non-library function call that first checks if the current function's argument is equal to the magic value, and if so, propagates this value to the next function call.

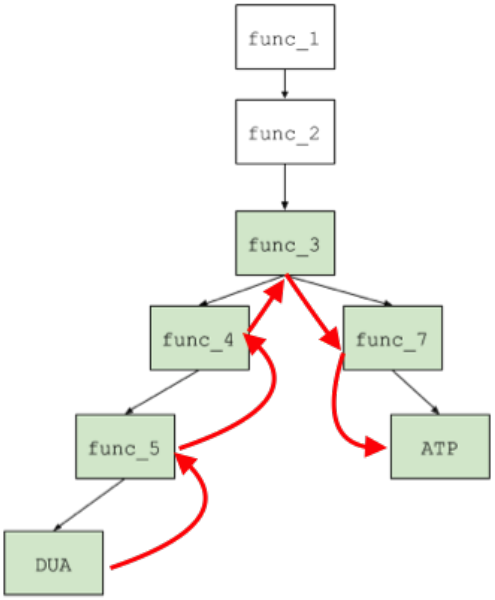

Figure 5-1: Each of the boxes represents a function in the program, and a thin arrow from A to B indicates that function A calls function B. By modifying the function signatures of the shaded functions, we can introduce data flow between the DUA and ATP, shown here as thick arrows that propagate back up the call graph and back down to the ATP.

5.3 Unused Data Hopping

The second approach we implemented involved leveraging unused variables already present in the program that we can use to store and propagate our own data. At a particular point in a program's execution, a variable is considered *unused* if its value is never read before the next time it is written. Crucially, this means that we can safely overwrite this variable with our own data without affecting the program's execution because no subsequent portion of the program relies on or is affected by the variable's current value.

Hypothetically, imagine that there existed a particular variable `var1` which was in scope at both the DUA and ATP, and which was unused for the entire program execution between the two points in time. Since `var1` is never read, we can safely copy the DUA's value into it without affecting the program's execution. Then, at the ATP, we can copy the value back from `var1` and check it against the magic value to decide if we should trigger the overflow. Now suppose that `var1` is not in scope at the ATP, so its value cannot be accessed at that point in the program, but a different unused variable, `var2`, is. If `var1` and `var2` are ever in the same scope between the DUA and ATP, we can first copy the DUA to `var1`, then copy `var1` to `var2`, and finally copy `var2` into the check for the trigger at the ATP. The bottom panel of figure 5-2 shows what this process would look like with three intermediate unused variables. The horizontal axis represents program execution time and the shaded rectangles represent the scopes of the unused variables. The red lines show how the value of the DUA can be copied from variable to variable to get to the ATP.

In general, if we can find enough such variables and determine their scoping, we can create a chain of unused variables connecting the DUA to the ATP in which we copy the value of the DUA from variable to variable as the scope changes.

To accomplish this, we first wrote a custom PANDA plugin that inserts queries to identify all unused data in the program. For each memory address, we tracked the pattern of reads and writes and identified those durations during which the data was unused.

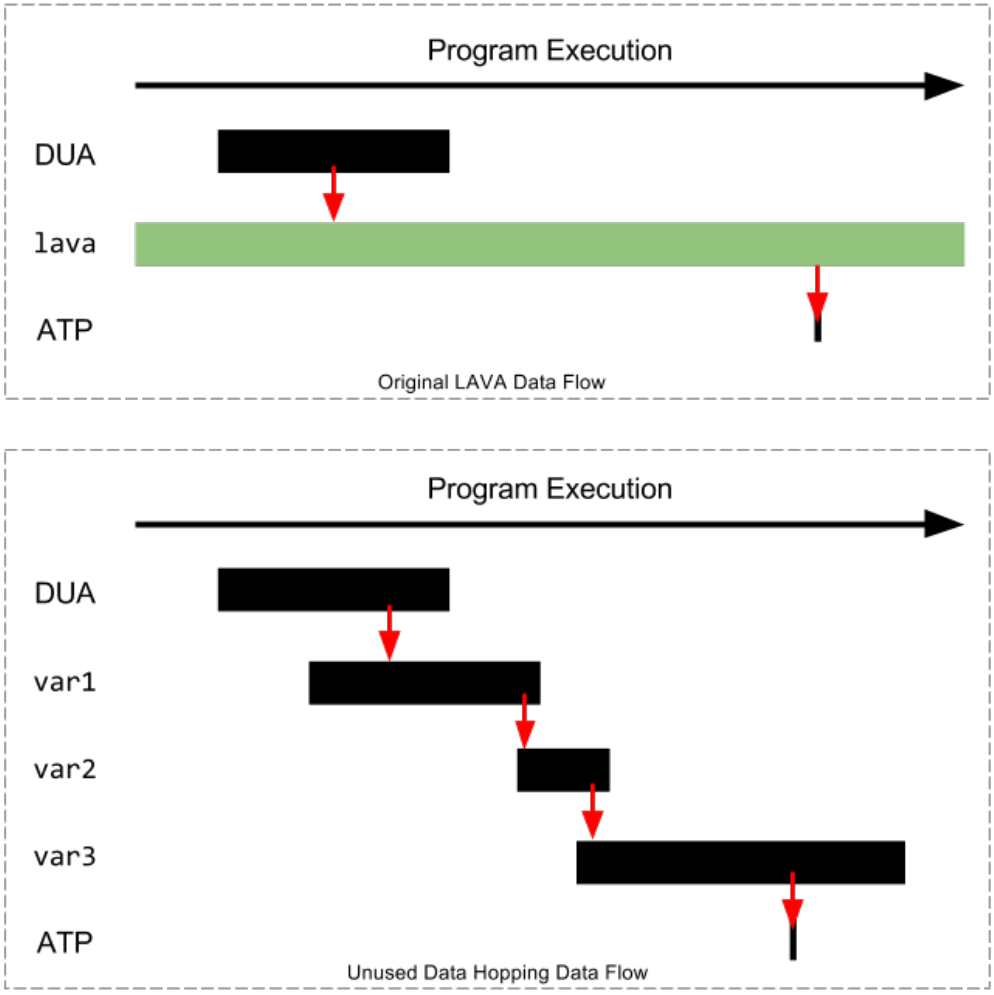

Figure 5-2: The horizontal axis is program execution time and each shaded rectangle represents the scope of the associated variable. The top graph depicts data flow in the first version of LAVA: a global variable is introduced and connects the DUA and ATP. The second chart depicts data flow using three unused variables `var1`, `var2`, and `var3`.

Next, we needed a reliable way to determine the scope of a variable in order to determine at what points in the program trace it was accessible. Instead of conducting a complicated analysis of scoping information, we opted to restrict our attention to function argument variables which have an easily determined scope: the duration of the function. Using PANDA, we inserted queries after each function call to determine the addresses of its arguments. If the arguments were pointers, we also tracked the range of addresses that they pointed to. Then, we cross-referenced these addresses against the unused data identified using the custom plugin to determine unused function arguments.

Finally, we developed an algorithm that would identify a series of these arguments that spanned the program from DUA to ATP, and then inserted code to transfer data from argument to argument to produce the desired data flow. Figures 5-3 and 5-4 show the same LAVA bug presented in figures 3-2 and 3-3 with our modified data flow. In this case, one unused data slot, found in the function argument `ms`, is in scope at both the DUA and ATP. Note that the new injected bug requires no calls to `lava_set` or `lava_get`.

It is theoretically possible that there is no such sequence of data slots available that connect the DUA and ATP, however in practice unused data is so prevalent that we have never encountered such a situation.

5.4 Results

We evaluated both our approaches on a number of metrics to test their effectiveness. First, we considered how our approaches to data flow affected LAVA's *yield*. LAVA is not able to validate 100% of the bugs that it inserts; currently bugs injected by LAVA have somewhere between a 30%-50% yield: that is, there is between a 30 and 50% chance that they will actually cause a crash when given the triggering input. This yield rate varies substantially depending on the nature of the original source code.

When modifying LAVA, we want to ensure that we do not substantially alter its bug yield to maintain its effectiveness. We evaluated both our approaches on two

```
protected int
file_encoding(struct magic_set *ms,
              ..., const char **type) {
  int* dataflow = (int *)ms;
...
   else if
     (({int rv =
          looks_extended(buf, nbytes, *ubuf, ulen);
        if (bug) {
          int lava = 0;
          lava |= ((unsigned char*)(buf))[0]<<(0*8);
          lava |= ((unsigned char*)(buf))[1]<<(1*8);
          lava |= ((unsigned char*)(buf))[2]<<(2*8);
          lava |= ((unsigned char*)(buf))[3]<<(3*8);
          dataflow[6] = lava;
        }; rv;})) {
...
```

Figure 5-3: An example of the DUA modifications of a LAVA-injected bug using our modified data flow. The `ms` function argument has been found to point to some unused data; it is casted to the int array `dataflow` and the DUA's value is copied into its sixth slot.

```
...
protected int
file_trycdf(struct magic_set *ms,
            ..., size_t nbytes) {
  int* dataflow = (int *)ms;
  ...
  if (cdf_read_header
      (( (&info)) + (dataflow[6])
       * (0x6c617661 == (dataflow[6])
          || 0x6176616c == (dataflow[6]), &h) == -1)
    return 0;
```

Figure 5-4: An example of the ATP modifications of a LAVA-injected bug. The DUA's value is retrieved from the `ms` function argument.

	unmodified	func-arg	unused data
`toy.c`	71%	64%	76%
`file`	53%	48%	42%

Table 5.1: Yield rates for unmodified LAVA, function-argument addition, and unused data hopping respectively calculated for two programs: `toy.c` and `file`. 50 bugs were inserted into `toy.c` and 1000 bugs were inserted into `file` for each approach.

test programs: `toy.c`[1], a 72-line C program, and the Unix program `file`, which is around 13,000 lines of code. The results are shown in in table 5.1.

We found that both of our approaches maintain approximately the original yield when run on the same input source programs: that is, a generic LAVA bug has an equal probability of triggering a crash as a LAVA bug with our modified data flow. This is expected as none of the additional code needed for the data flow should reduce the success of a LAVA bug.

Next, since the goal of this project was to introduce more sophisticated data flow, we evaluated the additional complexity introduced by our approaches by counting the number of times the DUA was copied in between it and the ATP trigger. Note that in the original version of LAVA this copy count is always exactly 1 since it requires one copy to siphon the data from the DUA to the global variable.

We injected a total of 500 modified LAVA bugs in `file` for each of the two approaches and counted the average number of copies required. The first approach (function argument addition) required an average of 3.9 copies and the second (unused data hopping) required an average of 4.7 copies. Note that for both approaches this metric is essentially counting the number of function calls in between the DUA and ATP, so it makes sense that the two numbers are similar.

We also considered the data flow diversity that each approach presented: namely, for a given DUA / ATP pair, how many possible different data flow paths does each approach allow. Two data flow paths are considered different if they involve a different sequence of copies from DUA to ATP. The first approach results in no diversity: since data flow is always done through the augmented first function argument, there is only

[1]https://gist.github.com/moyix/93cd687fde9fb965cfb7d508118d27c1

one possible sequence of copies for each DUA / ATP pair.

In contrast, the second approach is much more flexible: at each stage in the chain we counted how many unused variables were available among the arguments of the next function call. Then we multiplied these counts together to produce an estimate of the total possible number of chains. We found that, on average, a DUA / ATP pair could result in an average of 22.3 different data flow paths.

Finally, we performed a subjective analysis based on a hypothetical analysis with complete knowledge of how LAVA and the data flow approaches work to determine how much more effective our approach is.

A human analyst who is faced with a program that has been injected with 1 generic LAVA bug can simply examine references to the `lava_set` calls in order to identify both the DUA and ATP. Then, given both these points the analyst could craft a triggering input. An analyst faced with a LAVA bug with data flow from our first approach has a slightly more difficult task. Now the analyst must examine the uses of the first argument of every function and identify the location where the DUA is copied into it. This is more difficult, but not impossible since the only references to the artificial first argument are from LAVA's own injected code.

However, an analyst faced with a LAVA bug with data flow from our second approach has a much more difficult task: now the analyst must consider the uses of every function argument and identify both the DUA and ATP among these uses. We judge that this approach does substantially increase the work required by the analyst.

Chapter 6

Equality Saturation (Improving "Diversity")

6.1 Diversity vs. Obfuscation

While diversification and obfuscation are related concepts, they are not interchangeable terms. In the context of program analysis, *obfuscation* generally refers to the practice of modifying a program so that its intended function and purpose are harder to divine. In contrast, *diversification* is concerned with simply creating a new, semantically-equivalent version of the original program, without the explicit goal of making the program more difficult to understand.

Indeed, we'd like our diversified program to be statistically "similar" to the original program in some sense. For example, while a possibly compelling method to obfuscate a program is to replace all assembly instructions with `movs` [1], this would not be a particularly successful attempt at diversification, because the new program looks statistically very different from the original.

[1] https://github.com/xoreaxeaxeax/movfuscator

6.2 Source code modifications vs. Binary modifications

Since LAVA operates on C source code programs, we're primarily interested in building a diversifier for C programs. Prior work in this area has largely focused on diversification at the LLVM or assembly layers, but since LAVA's bugs are inserted at the source code layer, we'd like our diversifier to operate on source code as well.

There are a number of challenges present in applying diversification to source code. One is determining a robust technique for traversing the search space of all possible valid code. Another challenge is ensuring that the source code mutations made by the diversifier are still present in the compiled code and not removed by the compiler. Finally, although the performance of the mutated code is not a primary goal of a diversifier, it is important to ensure that diversification does not introduce a significant performance penalty, which would deter widespread adoption.

6.3 Program Snippet Equivalencies

The first phase of our diversifier identifies appropriate snippets of source code to work on. These snippets must be free of loops, other function calls, and certain types of array indexing and pointer arithmetic so that we can feasibly reason about their equivalence with a mutated form of the code.

We can then treat the snippet as essentially a function f mapping a set of input variables $x_1, x_2, \cdots x_m$ to a (possibly overlapping) set of output variables $y_1, y_2, \cdots y_n$ via a use-definition analysis. Variables that are ever *used* before being defined are our x_i's and variables that are *defined* (assigned to) over the course of the snippet are our y_i's. Our task is then to find an equivalent function g such that

$$f(x_1, x_2, \cdots x_m) = g(x_1, x_2, \cdots x_m)$$

for all possible values of the x_i's.

6.4 Equality Saturation

Our approach to diversification is inspired by *equality saturation*, a technique from the field of compiler optimization [17]. Instead of performing a series of optimizations sequentially, equality saturation uses a sophisticated intermediate representation (IR) to keep track of various *equivalencies* between program fragments. Once the IR is saturated with these equivalencies, a profitability heuristic is used to select one option from each equivalency to produce the final program.

In [17], the authors manage to express several sophisticated optimizations in terms of equivalencies in their IR such as loop unrolling, strength reduction, constant propagation, and tail recursion elimination.

6.4.1 Program Expression Graphs

Once we identify a program snippet to diversify, we convert it into our intermediate representation: a *program expression graph* (PEG). In our PEGs, each node represents the result of an intermediate or final expression in the snippet. The *leaves* of the graph represent the variables in scope at the start of the snippet, and the *roots* represent the live variables at the end of the snippet.

Consider the program snippet shown in figure 6-1 which swaps the variables `x` and `y` using a well-known trick. The PEG for this snippet is shown in 6-3(a). The $\oplus$ nodes represent the XOR operation, which are represented as carets in the program snippet. The shaded nodes represent variables which are live at the end of the snippet.

Figure 6-2 shows a program snippet that has the same effect as the snippet in figure 6-1 by introducing two temporary variables, `z` and `w`. Since the two snippets are equivalent, there is likely some series of transformations that can take us from the PEG of one snippet to the other.

Consider 6-3(a). It is well-known that $(a \oplus b) \oplus b = a$. Let's call this the XOR axiom. This axiom allows us to transform the PEG shown in 6-3(a) to the one in 6-3(b). Note that the $:=$ symbol represents assignment / equality.

Now, it is also true that $b = a; c = b$ is equivalent to $b = a; c = a;$. Let's call

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
use(x);
use(y);
```

Figure 6-1: An example of the "XOR Swapping Trick." At the end of this snippet the variables `x` and `y` have swapped value. This snippet is equivalent to the one shown in figure 6-2.

```
w = x;
z = y;
x = z;
y = w;
use(x);
use(y);
```

Figure 6-2: A standard way to swap the values in `x` and `y` using introduced temporary variables. This snippet is equivalent to the one shown in figure 6-1.

this the TRANSITIVITY axiom. Applying this to 6-3(b) allows us to transform it into the PEG shown in 6-3(c). Finally, applying the XOR axiom to 6-3(c) takes us to 6-3(d), which is the PEG for the "swapping" program snippet shown in 6-2.

6.4.2 Axiom Generation

In order for equality saturation to effectively diversify a candidate snippet, it must have a large set of axioms from which to draw from to recognize potential transformations to a PEG. 43 such axioms were manually created, taken largely from the standard rules of arithmetic and logic. Table 6.1 shows a selection of these axioms. The full list is shown in appendix A.

Care must be taken to ensure that the axioms hold for all possible inputs. For example, while associativity always holds over the addition of real numbers, it does not hold for floating point numbers because of their limited precision. Even for integer types, we must carefully consider edge cases such as overflows and underflows and signed vs unsigned types.

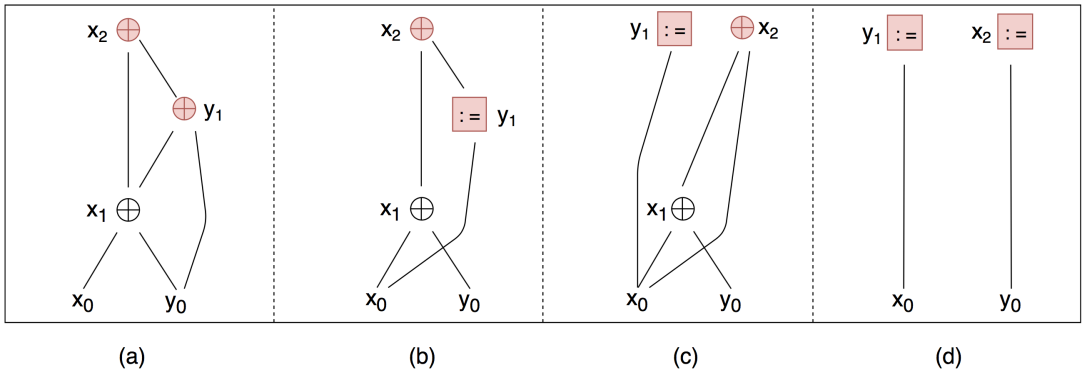

Figure 6-3: (a) The program expression graph for the snippet in figure 6-1. (b) The PEG after applying the XOR axiom. (c) The PEG after applying the TRANSITIVITY axiom. (d) The PEG after applying the XOR axiom. This is now the PEG for the snippet in figure 6-2.

REFLEXIVITY	$a \rightarrow a = a$
SUBSTITUTION	$a = b, a = c \rightarrow b = c$
TRANSITIVITY	$a = b, b = c \rightarrow a = c$
ADD-COMMUTATIVITY	$a, b \rightarrow a + b = b + a$
ADD-ASSOCIATIVITY	$a, b, c \rightarrow a + (b + c) = (a + b) + c$
XOR	$a, b \rightarrow (a \oplus b) \oplus a = b$

Table 6.1: Selected manually created axioms.

We also generated around 20 (and counting!) axioms automatically using a technique similar to superdiversification. Specifically, we randomly generate a small C function f over three or fewer variables, and then randomly mutate it to search for other functions g which have the same effect as f.

We create a C program which includes the source code definitions of our candidate functions g and our original function f. In the main body of the program we wrote a simple tester that evaluates all the g's and f on random inputs and eliminates those functions g which produce incorrect values. The program outputs the set of g's that collided with f for every input tested.

Note that it is not sufficient to perform only execution testing, since many possible pairs of functions exist which collide on all but a fraction of possible inputs. For example, the expression `!(x ^ 1234567890)` is equivalent to 0 for every input except `x = 1234567890`, where it is equal to 1. Candidate axioms were subsequently verified manually to ensure that they were always true.

6.4.3 Implementation

The diversifier was implemented in about 1100 lines of python. We used Eli Bendersky's pycparser [2] project to parse the C source code and create an abstract syntax tree (AST) and control flow graph (CFG). From the AST and CFG we build a program expression graph for each basic block of the code.

For each basic block, we repeatedly run through our list of axioms to try to match them with the PEG. If we find a match, we then probabilistically choose to apply the corresponding transformation to the PEG. We stop after a fixed number of iterations; this is a tunable parameter—a higher number of iterations leads to further diversity at a cost of additional time.

[2] https://github.com/eliben/pycparser

	src	-O1	-O3
`toy.c`	92%	60%	13%
`file`	88%	71%	19%

Table 6.2: Percent of modified basic blocks by the diversifier in the C soruce code, and under compilation with optimization levels -O1 and -O3. Rates were computed for two programs: `toy.c` and `file` with an iteration count of 500.

6.4.4 Results

Table 6.2 shows the percentage of modified basic blocks for two programs: `toy.c` and `file`. Recall that the former is small (less than 100 lines), while the latter is substantially longer (around 13,000 lines). With an iteration count of 500, the diversifier is able to modify around 90% of all basic blocks in both programs.

We also measured how these rates change after compilation. Optimizing compilers are designed to produce the fastest sequence of assembly instructions for any given input, so we would expect compilation to undo at least a fraction of our modifications. Under gcc's -O1 level of of optimization, around 60-70% of compiled basic blocks still differ from the corresponding block of the original program. However, under -O3 (the most aggressive level of optimization), this drops to less than 20% in both cases.

Since the diversifier operates on basic blocks, we can easily calculate what proportion were altered in the C source code. We used BinDiff [3], a binary diffing tool, to calculate the same statistics for the programs post-compilation. Since we never alter the control-flow graph, BinDiff is able to match up the original and diversified binaries and indicate which basic blocks exhibit changes.

[3] https://www.zynamics.com/software.html

Chapter 7

Bug-finding Competitions

7.1 AutoCTF

7.1.1 Capture the Flag Competitions

Each year, various companies, organizations and communities organize more than a hundred *Capture the Flag* (CTF) competitions—security contests featuring multiple challenges in categories including cryptography, web security, binary exploitation, and reverse engineering. The goal of each of these challenges is to obtain a *flag*, a specific string of text that proves success in some sense specific to the challenge. In binary exploitation challenges, often referred to as *pwnables* and the focus of AutoCTF, the flag is often a string located in a file on the server hosting the challenge. The challenge itself is some binary service that competitors can interact with over the network. Acquiring the flag serves as proof that a competitor has gained remote code execution on the server.

The purpose of these contests is primarily education and entertainment; CTF challenges are meant to give competitors the chance to hone their skills on legally "safe" targets.

7.1.2 Objectives

CTFs are expensive to build and run. In [6] we estimate that costs can range as high as $15,000 for the creation of just one challenge.

In principle, LAVA gives us the ability to drastically reduce the cost of challenge creation as it can quickly and easily instrument almost any C program with bugs.

To test this use case, in 2017 we created and ran a a capture the flag contest using LAVA called AutoCTF. We were interested in testing whether such an event would even be feasible, and if it could be pedagogically useful and fun.

7.1.3 Exploitable Bugs

In order to gain remote code execution through binary exploitation, it is not enough to make the program crash. Modern binary exploitation usually requires the combination of a memory leak to bypass address space layout randomization (ASLR), and either a write-what-where bug, in which an attacker can write any chosen value to any chosen memory address, or control of the instruction pointer.

LAVA bugs, as presented in the original paper, verifiably cause a memory corruption but are unlikely to be exploitable, as the attacker has no control over the size of the pointer offset (recall that this offset is identical to the triggering DUA value).

In order to inject exploitable bugs, we modified the attack point code to either give competitors control of the stack pointer, give competitors a write-what-where vulnerability, or to cause a memory leak to bypass ASLR.

While traditional LAVA bugs joined each ATP with one DUA, each of these new ATP types leverage multiple DUAs: one DUA which acts as a trigger or guard, and one or more which give the attacker control over which memory addresses to read or write. For example, the write-what-where ATP type uses three DUAs: one for the trigger, one to control where to write, and one to control what data to write.

7.1.4 Natural Data Flow

A hypothetical competitor faced with a binary injected with traditional LAVA bugs could focus his or her attention on the calls to `lava_get` and `lava_set` and bypass analysis of much of the rest of the binary.

In order to prevent this, we used the "function argument addition" technique described in section 5.2. The resulting binary contains no injected global data and no additional injected functions.

7.1.5 "Chaff" Bugs

To further hide LAVA's modifications, and to surmount the "diff" problem described in chapter 4, we also injected several *chaff* bugs in our programs. Although we could have used the technique described in chapter 4 to create these bugs, we did not need to: because the threshold for success in AutoCTF was remote code execution rather than memory corruption we simply used traditional LAVA bugs as our chaff because of their high probability of non-exploitability.

Specifically, traditional LAVA bugs (as shown in Figure 3-3) involve offsetting a pointer read or write by a large value and give a hypothetical attacker essentially no control other than whether the offset occurs or not. If the offset does occur, the program will almost certainly crash immediately, halting execution and giving the attacker no foothold for exploitation.

7.1.6 Results

In order to measure the difference in difficulty between LAVA-built challenges and traditional CTF pwnables, we ran the contest with eight different binaries based on two original C-source code programs. Four of the binaries were built with our modified LAVA system and the other four were modified by hand.

The CTF teams of four universities participated in the contest. Although competitors were able to exploit both LAVA-built binaries and human generated binaries, the competition was probably too difficult for the skill level of the players.

Interviews were conducted with five players to gather feedback on the event. Players had mixed feelings about the LAVA-binaries. In particular, they felt that the DUA triggers "stuck out" and were unrealistic, and some felt that reverse engineering essentially the same binary multiple times was uninteresting.

Crafting the LAVA-built binaries did take significantly less time than constructing a brand new CTF pwnable, but verifying its exploitability still took up a significant amount of time (on the order of hours).

7.2 Rode0day

7.2.1 Automated Vulnerability Discovery

While CTFs have existed for at least a decade, in recent years there has been a surge of interest in holding similar contests in which the competitors are all fully automated. DARPA's Cyber Grand Challenge (CGC), held in 2016, is far and away the most prominent such competition to be held.

In it, seven cyber reasoning systems (CRSs) competed to find, exploit, and patch vulnerabilities in 250 different binary services written for a modified version of Linux with a limited number of system calls [16].

By and large, the competition was successful: the competing CRSs were able to exploit and patch a significant number of the challenges. The winning CRS, Mayhem, was invited to compete in the DEF CON Finals CTF to play alongside human hacking teams, and while it did come in last place, it was able to capture multiple flags [16].

7.2.2 Objectives

While the CGC drew a large amount of interest to the field, it was designed to be held only once, which limited its ability to serve as an analogue to the similar competitions held in computer vision and other fields. In order to demonstrate steady progress (or potentially the lack thereof) in the state of the art in the field of vulnerability discovery, we plan to design and run a new contest, called Rode0day, on a monthly

basis.

Furthermore, although demonstrating exploitability certainly has its uses, in Rode0day we chose to lower the threshold of success to simply demonstrating memory corruption in the form of crash (specifically a program exit code greater than 128).

While the CGC was run on a simplified Linux system, Rode0day challenges are all standard 32-bit Linux ELF files. As this is a primary target of modern fuzzing efforts, we hope that success on Rode0day challenges will more closely translate to success at finding vulnerabilities in real programs. Indeed this is the over-arching goal for the entire endeavor: we hope that the optimal strategy for winning a Rode0day will be to build a system that is also very successful at discovering bugs in real programs.

7.2.3 LAVA Modifications

As a consequence, it is important that we make it difficult for a hypothetical competitor to "cheat" and discover triggering inputs to inserted LAVA bugs by bypassing critical or challenging steps that they would encounter in attempting to do the same for a real program.

Once again, we use the data flow modifications to add complexity and remove the uses of artificial helper functions. We also rely on the approach discussed in chapter 6 to diversify the released binaries and modify the trigger comparison.

Finally, we also added a compile-time flag to the modified C-source files that would allow us to produce versions of the binaries that logged to standard output whenever a DUA guard was successfully triggered. This allows us to easily determine which bug a competitor triggered when an input causes a crash.

7.2.4 Scoring

We wanted to reward competitors for finding bugs faster than other players, but we also wanted to keep the scoring system simple to make it easy to reason about. In our contest, each found bug is worth 10 points, with one bonus point if you are the first player to trigger that bug.

Note that it is theoretically possible to trigger multiple DUA triggers with one unique input. In this case, we chose to award credit to all the bugs that were triggered in the process.

A different input that triggers an already-discovered bug will not be awarded any additional points.

7.2.5 Website

We built a website [1] which lists the rules for the competition, allows competitors to register an account, and calculates and displays the current score. Once registered, competitors are provided with an API token which allows them to access the contest binaries and submit triggering inputs. Finally, full results from the prior months contests will be released and permanently archived on the website so that the public can perform their own analysis of the data.

7.2.6 Beta testing

Using our modified LAVA system we inserted 21 and 31 bugs into two small programs named `buffalo` and `bill` respectively. We invited a number of academic researchers and other interested parties to participate in a closed beta to evaluate the web interface and our created binaries.

Figure 7-1 shows the current scoreboard for the beta. Nine teams have found at least one bug, and two (itszn and Inventive Mayfly) have found all 52 inserted bugs.

We are currently soliciting feedback and working on preparing two new binaries for a launch scheduled to start in early June.

[1] https://rode0day.mit.edu/

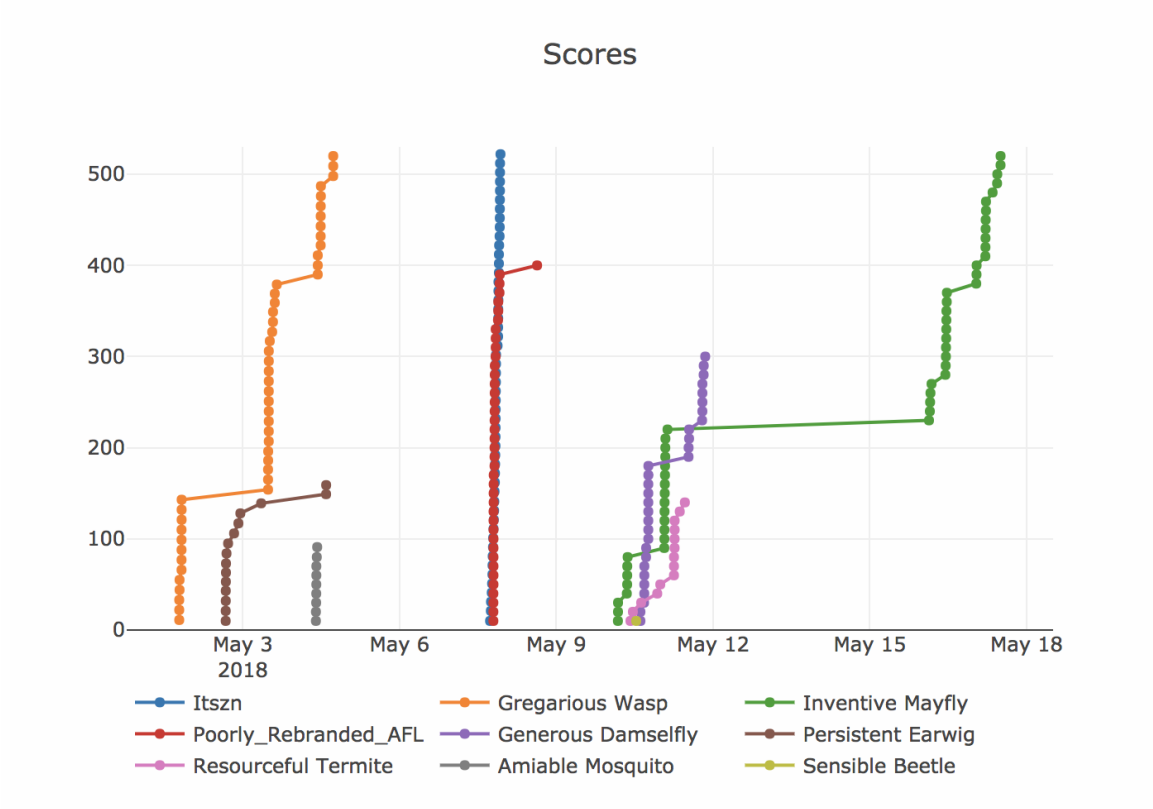

Figure 7-1: Current scoreboard for the beta Rode0day. Nine teams have scored over the course of several weeks and two have found all 52 bugs. Taken from `rode0day.mit.edu`.

Chapter 8

Conclusions

In this thesis, we presented several significant modifications to LAVA in order to improve the diversity and realism of the injected bugs. Specifically, we discussed a technique for injecting fake bugs in order to prevent the ability to find LAVA bugs by simply examining the inserted code. We also discussed two different techniques intended to complicate the data flow of the injected bugs.

In chapter 6 we presented a method to diversify C-source code programs using a variant of equality saturation. We used this to diversify source code after LAVA has injected vulnerabilities as a way to both obscure the bug-related modifications and "normalize" the program.

Finally, in chapter 7 we discussed two different competitions, AutoCTF and Rode0day, which used LAVA and relied on the modifications discussed in prior chapters in order to accomplish their objectives.

8.1 Future Work

8.1.1 Bug Type Diversity

The work in this thesis has focused almost entirely on changing the expression of injected LAVA bugs by making additional modifications to the program's source code. However, rather than simply altering the way any given LAVA bug manifests itself, we

could also broaden the diversify of *types* of bugs that LAVA can inject. For example, in chapter 7 we discussed modifying LAVA in order to explicitly inject exploitable bugs.

In principle, the LAVA system is capable of injecting a wide variety of bug types including integer overflows, use-after-frees, and race conditions. We managed to create the exploitable bugs for AutoCTF by composing multiple DUAs with the ATP in a deliberate way, but injecting these other bug types would require using PANDA's taint analysis to identify different primitives. For example, injecting an integer overflow would require finding an ATP which involved integer arithmetic instead of a memory read or write.

Attaining this diversity of different types of injected bugs would bring the ground truth data LAVA is able to produce closer in line with the diversity of real vulnerability types.

8.1.2 Control-Flow Diversification

Currently, the work on source code-level diversification described in chapter 6 is largely confined to modifications within a basic block. In other words, there is currently no diversification of the underlying *control flow* of the program. This is not a fundamental limitation of the method—the original equality saturation technique described in [17] was able to make sophisticated changes to control flow such as function inlining or tail call elimination.

This would require a number of changes to the diversifier. First, we would need to modify the PEGs described in 6.4.1 to include control-flow information. Then, we'd need to generate axioms that operate on changes beyond a basic block. Finally, we'd need to convert our new PEGs back into C-source code.

Such changes would in theory complicate methods to match a diversified code base with its original source code that relied on analyzing the control flow graph such as BinDiff.

8.1.3 Measuring Bug Properties

One of the hopes we have of the LAVA system is that the bug corpora it produces will allow us to answer basic scientific questions about the current state of vulnerability discovery and software security. We hope to be able to learn what properties make a given bug difficult to find and what types of bugs are currently being missed by existing bug-finding tools. This in turn can inform us on how to improve the state of the art in the field of vulnerability discovery.

Rode0day, the LAVA-powered bug finding competition discussed in section 7.2, offers the clearest path forward to begin this process. The competition has already drawn the interest of several researchers and tool designers in the relevant fields, and we anticipate that the results of the upcoming monthly contests will allow us make some preliminary investigations into the question of bug difficulty by comparing the discovery-rates between the various bugs LAVA inserted.

Appendix A

Axioms

ADD-COMMUTATIVITY	$a, b \rightarrow a + b = b + a$
MUL-COMMUTATIVITY	$a, b \rightarrow a \cdot b = b \cdot a$
XOR-COMMUTATIVITY	$a, b \rightarrow a \oplus b = b \oplus a$
AND-COMMUTATIVITY	$a, b \rightarrow a \wedge b = b \wedge a$
OR-COMMUTATIVITY	$a, b \rightarrow a \vee b = b \vee a$
EQ-COMMUTATIVITY	$a, b \rightarrow (a == b) = (b == a)$
NEQ-COMMUTATIVITY	$a, b \rightarrow (a \neq b) = (b \neq a)$
NEG-COMMUTATIVITY	$a, b \rightarrow a - b = -(b - a)$
R-ADD-ASSOCIATIVITY	$a, b, c \rightarrow a + (b + c) = (a + b) + c$
L-ADD-ASSOCIATIVITY	$a, b, c \rightarrow (a + b) + c = a + (b + c)$
R-MUL-ASSOCIATIVITY	$a, b, c \rightarrow a \cdot (b \cdot c) = (a \cdot b) \cdot c$
L-MUL-ASSOCIATIVITY	$a, b, c \rightarrow (a \cdot b) \cdot c = a \cdot (b \cdot c)$
R-XOR-ASSOCIATIVITY	$a, b, c \rightarrow a \oplus (b \oplus c) = (a \oplus b) \oplus c$
L-XOR-ASSOCIATIVITY	$a, b, c \rightarrow (a \oplus b) \oplus c = a \oplus (b \oplus c)$
R-AND-ASSOCIATIVITY	$a, b, c \rightarrow a \wedge (b \wedge c) = (a \wedge b) \wedge c$
L-AND-ASSOCIATIVITY	$a, b, c \rightarrow (a \wedge b) \wedge c = a \wedge (b \wedge c)$
R-OR-ASSOCIATIVITY	$a, b, c \rightarrow a \vee (b \vee c) = (a \vee b) \vee c$
L-OR-ASSOCIATIVITY	$a, b, c \rightarrow (a \vee b) \vee c = a \vee (b \vee c)$
REFLEXIVITY	$a \rightarrow a = a$
SUBSTITUTION	$a, b, c \rightarrow a = b, a = c \rightarrow b = c$
TRANSITIVITY	$a, b, c \rightarrow a = b, b = c \rightarrow a = c$
DOUBLE-NEG	$a \rightarrow \sim (\sim a) = a$
DOUBLE-SUB	$a \rightarrow -(-a) = a$
DOUBLE-NOT	$a \rightarrow !(!a) = a$
XOR	$a \rightarrow a \oplus a = 0$
ZERO-ADD	$a \rightarrow a + 0 = a$
ZERO-XOR	$a \rightarrow a \oplus 0 = a$
ZERO-MUL	$a \rightarrow a \cdot 0 = 0$
ZERO-SUB	$a \rightarrow a - 0 = a$
ONE-MUL	$a \rightarrow a \cdot 1 = a$
LEFT-SHIFT	$a \rightarrow a << C = a \cdot 2^C$
RIGHT-SHIFT	$a \rightarrow a >> C = a/2^C$
TO-LEFT-SHIFT	$a \rightarrow a \cdot 2^C = a << C$
TO-RIGHT-SHIFT	$a \rightarrow a/2^C = a >> C$
LESS-THAN-SWAP	$a, b \rightarrow (a < b) = (b > a)$
GREATER-THAN-SWAP	$a, b \rightarrow (a > b) = (b < a)$
LESS-THAN-EQ-SWAP	$a, b \rightarrow (a \leq b) = (b \geq a)$
GREATER-THAN-EQ-SWAP	$a, b \rightarrow (a \geq b) = (b \leq a)$
LESS-THAN-SWAP	$a, b \rightarrow (a < b) = !(a \geq b)$
GREATER-THAN-SWAP	$a, b \rightarrow (a > b) = !(a \leq b)$
LESS-THAN-EQ-SWAP	$a, b \rightarrow (a \leq b) = !(a > b)$
GREATER-THAN-EQ-SWAP	$a, b \rightarrow (a \geq b) = !(a < b)$

Table A.1: Full list of manually created axioms.

Bibliography

[1] Tomas Balyo, Marijn Heule, and Matti Jarvisalo. Sat competition 2016: Recent developments, 2017.

[2] J. Barker, R. Marxer, E. Vincent, and S. Watanabe. The third CHiME speech separation and recognition challenge: Dataset, task and baselines. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 504–511, Dec 2015.

[3] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 143–157, Washington, DC, USA, 2008. IEEE Computer Society.

[4] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, May 2016.

[5] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[6] Patrick Hulin, Andy Davis, Rahul Sridhar, Andrew Fasano, Cody Gallagher, Aaron Sedlacek, Tim Leek, and Brendan Dolan-Gavitt. Autoctf: Creating diverse pwnables via automated bug injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[7] Matthias Jacob, Mariusz H. Jakubowski, Prasad Naldurg, Chit Wei Saw, and Ramarathnam Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Proceedings of the 3rd International Workshop on Security: Advances in Information and Computer Security*, IWSEC '08, pages 100–120, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Kendra Kratkiewicz and Richard Lippmann. Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. 10 2009.

[9] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 627–637, New York, NY, USA, 2017. ACM.

[10] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[11] Hao Chen Peng Chen. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, San Francisco, CA, 5 2018.

[12] Jannik Pewny and Thorsten Holz. Evilcoder: Automated bug insertion. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 214–225, New York, NY, USA, 2016. ACM.

[13] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, February 2017.

[14] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[15] Eric Schkufza, Rahul Sharma, and Aiken Alex. Stochastic superoptimization. In *ASPLOS '13*. ACM, 2013.

[16] J. Song and J. Alves-Foss. The DARPA Cyber Grand Challenge: A Competitor's Perspective. *IEEE Security Privacy*, 13(6):72–76, Nov 2015.

[17] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

[18] M. F. Thompson. Effects of a honeypot on the cyber grand challenge final event. *IEEE Security Privacy*, 16(2):37–41, March 2018.

[19] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, October 2004.