LAVA: Large-scale Automated Vulnerability Addition

Brendan Dolan-Gavitt*, Patrick Hulin†, Engin Kirda‡, Tim Leek†, Andrea Mambretti‡,
Wil Robertson‡, Frederick Ulrich†, Ryan Whelan†

(Authors listed alphabetically)

*New York University
brendandg@nyu.edu
†MIT Lincoln Laboratory
{patrick.hulin, tleek, frederick.ulrich, rwhelan}@ll.mit.edu
‡Northeastern University
{ek, mbr, wkr}@ccs.neu.edu

***Abstract*—Work on automating vulnerability discovery has long been hampered by a shortage of ground-truth corpora with which to evaluate tools and techniques. This lack of ground truth prevents authors and users of tools alike from being able to measure such fundamental quantities as miss and false alarm rates. In this paper, we present LAVA, a novel dynamic taint analysis-based technique for producing ground-truth corpora by quickly and automatically injecting large numbers of realistic bugs into program source code. Every LAVA bug is accompanied by an input that triggers it whereas normal inputs are extremely unlikely to do so. These vulnerabilities are synthetic but, we argue, still realistic, in the sense that they are embedded deep within programs and are triggered by real inputs. Using LAVA, we have injected thousands of bugs into eight real-world programs, including bash, tshark, and the GNU coreutils. In a preliminary evaluation, we found that a prominent fuzzer and a symbolic execution-based bug finder were able to locate some but not all LAVA-injected bugs, and that interesting patterns and pathologies were already apparent in their performance. Our work forms the basis of an approach for generating large ground-truth vulnerability corpora on demand, enabling rigorous tool evaluation and providing a high-quality target for tool developers.**

I. Motivation

Bug-finding tools have been an active area of research for almost as long as computer programs have existed. Techniques such as abstract interpretation, fuzzing, and symbolic execution with constraint solving have been proposed, developed, and applied. But evaluation has been a problem, as ground truth is in extremely short supply. Vulnerability corpora exist [10] but they are of limited utility and quantity. These corpora fall into two categories: historic and synthetic. Corpora built from historic vulnerabilities contain too few examples to be of much use [27]. However, these are closest to what we want to have since the bugs are embedded in real code, use real inputs, and are often well annotated with precise information about where the bug manifests itself.

Creating such a corpus is a difficult and lengthy process; according to the authors of prior work on bug-finding tool evaluation, a corpus of fourteen very well annotated historic bugs with triggering inputs took about six months to construct [26]. In addition, public corpora have the disadvantage of already being released, and thus rapidly become stale; as we can expect tools to have been trained to detect bugs that have been released. Given the commercial price tag of new exploitable bugs, which is widely understood to begin in the mid five figures [20], it is hard to find real bugs for our corpus that have not already been used to train tools. And while synthetic code stocked with bugs auto-generated by scripts can provide large numbers of diagnostic examples, each is only a tiny program and the constructions are often considered unrepresentative of real code [2], [11].

In practice, a vulnerability discovery tool is typically evaluated by running it and seeing what it finds. Thus, one technique is judged superior if it finds more bugs than another. While this state of affairs is perfectly understandable, given the scarcity of ground truth, it is an obstacle to science and progress in vulnerability discovery. There is currently no way to measure fundamental figures of merit such as miss and false alarm rate for a bug finding tool.

We propose the following requirements for bugs in a vulnerability corpus, if it is to be useful for research, development, and evaluation. Bugs must

1) Be cheap and plentiful
2) Span the execution lifetime of a program
3) Be embedded in representative control and data flow
4) Come with an input that serves as an existence proof
5) Manifest for a very small fraction of possible inputs

The first requirement, if we can meet it, is highly desirable since it enables frequent evaluation and hill climbing. Corpora are more valuable if they are essentially disposable. The second and third of these requirements stipulate that bugs must

This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002 and/or #FA8702-15-D-0001. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

be realistic. The fourth means the bug is demonstrable and serious, and is a precondition for determining exploitability. The fifth requirement is crucial. Consider the converse: if a bug manifests for a large fraction of inputs it is trivially discoverable by simply running the program.

The approach we propose is to create a synthetic vulnerability via a few judicious and automated edits to the source code of a real program. We will detail and give results for an implementation of this approach that satisfies all of the above requirements, which we call LAVA (Large-scale Automated Vulnerability Addition). A serious bug such as a buffer overflow can be injected by LAVA into a program like `file`, which is 13K LOC, in about 15 seconds. LAVA bugs manifest all along the execution trace, in all parts of the program, shallow and deep, and make use of normal data flow. By construction, a LAVA bug comes with an input that triggers it, and we can guarantee that no other input can trigger the bug.

II. Scope

We restrict our attention, with LAVA, to the injection of bugs into source code. This makes sense given our interest in using it to assemble large corpora for the purpose of evaluating and developing vulnerability discovery techniques and systems. Automated bug discovery systems can work on source code [1], [8], [9], [24] or on binaries [3], [21]; we can easily test binary analysis tools by simply compiling the modified source. Injecting bugs into binaries or bytecode directly may also be possible using an approach similar to ours, but we do not consider that problem here. We further narrow our focus to Linux open-source software written in C, due to the availability of source code and source rewriting tools. As we detail later, a similar approach will work for other languages.

We want the injected bugs to be serious ones, i.e., potentially exploitable. As a convenient proxy, our current focus is on injecting code that can result in out-of-bounds reads and writes that can be triggered by an attacker-controlled input; in Section VIII we consider extensions to LAVA to support other bug classes. We produce a proof-of-concept input to trigger any bug we successfully inject, although we do not attempt to produce an actual exploit.

For the sake of brevity, in this paper we will use the words *bug* and *vulnerability* interchangeably. In both cases, what we mean is vulnerabilities (in particular, primarily out-of-bounds reads and writes) that cause potentially exploitable crashes.

III. LAVA Overview

At a high level, LAVA adds bugs to programs in the following manner. Given an execution trace of the program on some specific input, we:

1) Identify execution trace locations where input bytes are available that do not determine control flow and have not been modified much. We call these quantities DUAs, for Dead, Uncomplicated and Available data.

```
1 void foo(int a, int b, char *s, char *d, int n) {
      int c = a+b;
3     if (a != 0xdeadbeef)
          return;
5     for (int i=0; i<n; i++)
          c+=s[i];
7     memcpy(d,s,n+c); // Original source
      // BUG: memcpy(d+(b==0x6c617661)*b,s,n+c);
9 }
```

Fig. 1: LAVA running example. Entering the function `foo`, `a` is bytes 0..3 of input, `b` is 4..7, and `n` is 8..11. The pointers `s` and `d`, and the buffers pointed to by them are untainted.

2) Find potential attack points that are temporally after a DUA in the program trace. Attack points are source code locations where a DUA might be used, if only it were available there as well, to make a program vulnerable.
3) Add code to the program to make the DUA value available at the attack point and use it to trigger the vulnerability.

These three steps will be discussed in the following three sections, which refer to the running example in Figure 1.

A. The DUA

Because they ensure that attacker-controlled data is available to influence program behavior, DUAs form the raw material from which we construct bugs. We identify DUAs in a program by running that program under a dynamic taint analysis [14] for a specific input. That taint analysis has a few important features:

- Each byte in the input is given its own label. Thus, if an internal program quantity is tainted, then we can map that quantity back to a specific part of the input.
- The taint analysis is as complete and correct as possible. All program code including library and kernel is subject to taint analysis. Multiple threads and processes are also handled correctly, so that taint flows are not lost.
- The taint analysis keeps track of a *set* of labels per byte of program data, meaning that it can represent computation that mixes input bytes.

Every tainted program variable is some function of the input bytes. We estimate how complicated this function is via a new measure, the Taint Compute Number (TCN). TCN simply tracks the depth of the tree of computation required to obtain a quantity from input bytes. The smaller TCN is for a program quantity, the closer it is, computationally, to the input. If TCN is 0, the quantity is a direct copy of input bytes. The intuition behind this measure is that we need DUAs that are computationally close to the input in order to be able to use them with predictable results.

Note that TCN is not an ideal measure. There are obviously situations in which the tree of computation is deep but the resulting value is both completely predictable and has as much entropy as the original value. However, TCN has the advantage that it is easy to compute on an instruction-by-instruction

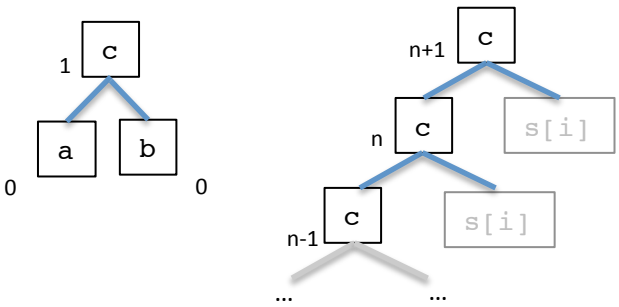

Fig. 2: Taint Compute Number examples from the running example. TCN is simply the depth of the tree of computation that produces the value from tainted inputs. TCN(c) after line 2 is 1, and after line 6 (upon exiting the loop), it is n+1.

basis. Whenever the taint system needs to compute the union of sets of taint labels to represent computation, the TCN associated with the resulting set is one more than the max of those of the input sets. In the running example, and illustrated in Figure 2, $TCN(c) = 1$ after line 1, since it is computed from quantities `a` and `b` which are directly derived from input. Later, just before line 7 and after the loop, $TCN(c) = n + 1$ because each iteration of the loop increases the depth of the tree of computation by one.

The other taint-based measure LAVA introduces is *liveness*, which is associated with taint labels, i.e., the input bytes themselves. This is a straightforward accounting of how many branches a byte in the input has been used to decide. Thus, if a particular input byte label was never found in a taint label set associated with any byte used to decide a branch, it will have liveness of 0. A DUA entirely consisting of bytes with 0 or very low liveness can be considered *dead* in the sense that it has little influence upon control flow for this program trace. If one were to fuzz dead input bytes, the program should be indifferent and execute the same trace. In the running example, $LIV(0..3) = 1$ after line 3, since `a` is a direct copy of input bytes 0..3. After each iteration of the loop, the liveness of bytes 8..11, the loop bound, increase by one, and so after the loop $LIV(8..11) = n$.

Figures 3 and 4 are plots of liveness and taint compute number for the program `file` processing the input `/bin/ls`. In both plots, the horizontal axis is the number of replay instructions processed, and so corresponds, roughly, to time. The vertical axis is file position, so at bottom is the first byte in `/bin/ls` and at top is the 70th byte. Obviously, `/bin/ls` is bigger than 70 bytes. However, as expected, only the 64-byte ELF header has any interesting liveness or taint compute number values and so we restrict our attention, in these plots, to that section. Higher values for liveness and taint compute number are represented on both plots by darker patches. Thus, portions that are very light for large horizontal stretches starting at the left on both plots are DUAs. For instance, bytes 28-32, which point to the start of the program header table, are uninvolved in branches or computation. Presumably

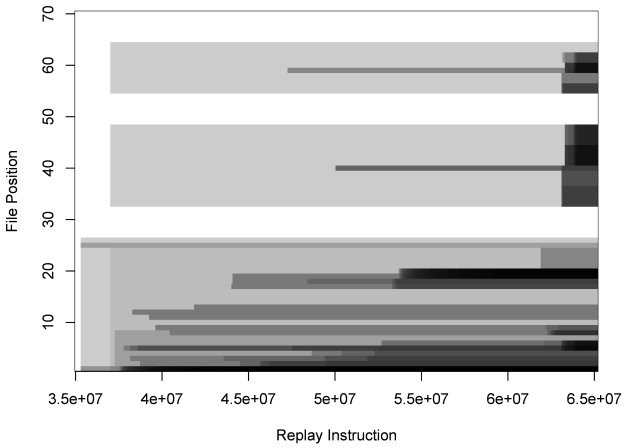

Fig. 3: Liveness plotted, over time, for the input bytes of '/bin/ls' being processed by the program 'file'.

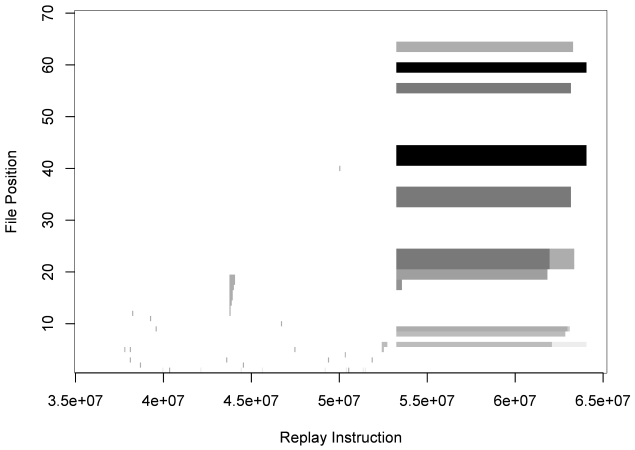

Fig. 4: Taint compute number, over time, for the input bytes of '/bin/ls' being processed by the program 'file'.

these would make a fine DUA, and it seems reasonable that `file` simply doesn't have much use for this information. Whereas bytes 19 and 20, which indicate the instruction set, are very live after about 55M instructions. This is reasonable since `file` needs to report this information in human-readable form, such as `x86_64`. However, consider bytes 10-16, which are apparently being used in a number of branches. This is odd considering they are marked as unused in the ELF header spec. These kinds of disparities make a good argument for using taint-based measures to inject bugs rather than trusting published specs.

The combination of uncomplicated (low TCN) and dead (low liveness) program data is a powerful one for vulnerability injection. The DUAs it identifies are internal program quantities that are often a direct copy of input bytes, and can be set to any chosen value without sending the program along a different path. These make very good triggers for vulnerabilities. In the running example, bytes 0..3 and 8..11 are all somewhat live, because they have been seen to be used to

decide branches. Arguments `a` and `n` are therefore too live to be useful in injecting a vulnerability. Argument `b`, on the other hand, has a TCN of 0 and the bytes from which it derives, 4..7 are completely dead, making it an ideal trigger to control a vulnerability.

Precisely which DUAs should be included, based on their liveness and TCN, is a configurable threshold of LAVA. We explore the impact (in terms of whether the bug can be successfully injected and verified) of various thresholds in Section VI-A.

B. The attack point

Attack point selection is a function of the type of vulnerability to be injected. All that is required is that it must be possible to inject a bug at the attack point by making use of dead data. This data can be made available later in the trace via new dataflow. Obviously, this means that the attack point must be *temporally after* an appearance of a DUA in the trace. If the goal is to inject a read overflow, then reads via pointer dereference, array index, and bulk memory copy, e.g., are reasonable attack points. If the goal is to inject divide-by-zero, then arithmetic operations involving division will be attacked. Alternately, the goal might be to control one or more arguments to a library function. For instance, in the running example, on line 7, the call to `memcpy` can be attacked since it is observed in the trace after a usable DUA, the argument `b`, and any of its arguments can be controlled by adding `b`, thus potentially triggering a buffer overflow.

C. Data-flow bug injection

The third and final step to LAVA bug injection is introducing a dataflow relationship between DUA and attack point. If the DUA is in scope at the attack point then it can simply be used at the attack point to cause the vulnerability. If it is not in scope, new code is added to siphon the DUA off into a safe place (perhaps in a static or global data structure), and later retrieve and make use of it at the attack point. However, in order to ensure that the bug only manifests itself very occasionally (one of our requirements from Section I), we add a guard requiring that the DUA match a specific value if it is to be used to manifest the vulnerability. In the running example, the DUA `b` is still in scope at the `memcpy` attack point and the only source code modification necessary is to make use of it to introduce the vulnerability if it matches a particular value. If we replace the first argument to the call to `memcpy`, `d`, with `d+(b==0x6c617661)*b` then there will be an out of bounds write only when bytes 4..7 of the input exactly match 0x6c617661.

Although this mechanism for ensuring that each bug is only triggered for one specific input has worked well for us so far, there are other ways we could accomplish the same task. For example, we could instead guard the call to the buggy code with an `if` statement, or perform a comparison with the input bytes that make up the DUA one by one. Although these are functionally equivalent, the exact mechanism used may make it easier for certain tools to find the bug. Comparing the input bytes one by one, for example, would allow coverage-maximizing fuzzers to incrementally discover the bug by guessing one byte at a time, rather than having to guess the entire 32-bit trigger at once. For a full-scale tool evaluation, it would be best to inject bugs with a variety of different trigger mechanisms; however, for our current prototype we use only the one described in this section.

IV. Roads Not Taken

Given the goal of adding bugs to real-world programs in an automated way, there are a large number of system designs and approaches. In order to clarify our design for LAVA, in this section we will briefly examine alternatives.

First, one might consider compiling a list of straightforward, local program transformations that reduce the security of the program. For example, we could take all instances of the `strlcpy` and `strncpy` functions and replace them with the less secure `strcpy`, or look for calls to `malloc` and reduce the number of bytes allocated. This approach is appealing because it is very simple to implement (for example, as an LLVM transformation pass), but it is not a reliable source of bugs. There is no easy way to tell what input (if any) causes the newly buggy code to be reached; and on the other hand, many such transformations will harm the correctness of the program so substantially that it crashes on *every* input. In our initial testing, transforming instances of `strncpy` with `strcpy` in `bash` just caused it to crash immediately. The classes of bugs generated by this approach are also fundamentally limited and not representative of bugs in modern programs.

A more sophisticated approach is suggested by Keromytis [6]: targeted symbolic execution could be used to find program paths that are potentially dangerous but currently safe; the symbolic path constraints could then be analyzed and used to remove whatever input checks currently prevent a bug. This approach is intuitively promising: it involves minimal changes to a program, and the bugs created would be realistic in the sense that one could imagine them resulting from a programmer forgetting to correctly guard some code. However, each bug created this way would come at a high computational cost (for symbolic execution and constraint solving), and would therefore be limited in how deep into the program it could reach. This would limit the number of bugs that could be added to a program.

By contrast, the approach taken by LAVA is computationally cheap—its most expensive step is a dynamic taint analysis, which only needs to be done once per input file. Each validated bug is guaranteed to come with a triggering input. In our experiments, we demonstrate that even a single input file can yield thousands of bugs spread throughout a complex program such as `tshark`.

V. Implementation

The LAVA implementation operates in four stages to inject and validate buffer overflow vulnerabilities in Linux C source code.

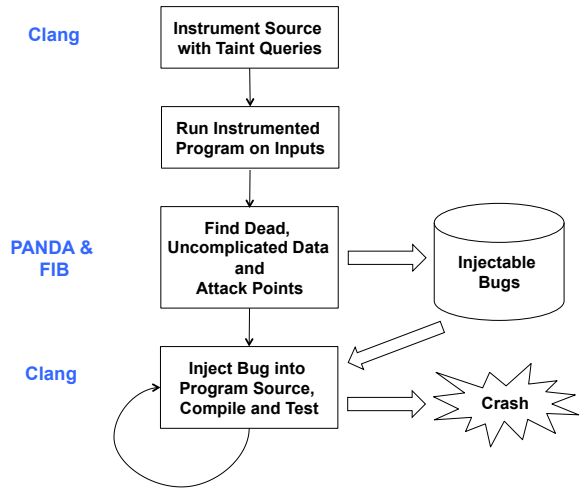

Fig. 5: LAVA Implementation Architecture. PANDA and Clang are used to perform a dynamic taint analysis which identifies potential bug injections as DUA attack point pairs. Each of these is validated with a corresponding source code change performed by Clang as well. Finally, every potentially buggy binary is tested against a targeted input change to determine if a buffer overflow actually results.

1) Compile a version of the target program which has been instrumented with taint queries.
2) Run the instrumented version against various inputs, tracking taint, and collecting taint query results and attack point information.
3) Mine the taint results for DUAs and attack points, and collect a list of potential injectable bugs.
4) Recompile the target with the relevant source code modifications for a bug, and test to see if it was successfully injected.

These stages are also depicted in Figure 5.

A. Taint queries

LAVA's taint queries rely on the PANDA dynamic analysis platform [5], which is based on the QEMU whole-system emulator. PANDA augments QEMU in three important ways. First, it introduces deterministic record and replay, which can be used for iterated and expensive analyses (e.g. taint) that often cannot be performed online. Second, it has a simple but powerful plugin architecture that allows for powerful analyses to be built and even built upon one another. Third, it integrates, from S2E [4], the ability to lift QEMU's intermediate language to LLVM for analysis.

The main feature of PANDA used by LAVA is a fast and robust dynamic taint analysis plugin that works upon the LLVM version of each basic block of emulated code. This LLVM version includes emulated versions of every x86 instruction that QEMU supports. QEMU often implements tricky processor instructions (e.g. MMX and XMM on x86) in C code. These are compiled to LLVM bitcode using Clang, and, thereby made available for taint analysis by PANDA as well. This process ensures that PANDA's taint analysis is *complete* in the sense that it can track dataflow through all instructions.

LAVA employs a simple PANDA plugin named `file_taint` that is able to apply taint labels to bytes read from files in Linux. The plugin, in turn, leverages operating system introspection and system call plugins in PANDA to determine the start file offset of the read as well as the number of bytes actually read. This allows LAVA to make use of taint information that maps internal program quantities back to file offsets.

Before running a target program under PANDA, LAVA first invokes a custom Clang tool to insert taint queries into the source before and after function calls. Each function argument is deconstructed into its constituent lvalues, and Clang adds a taint query for each as a *hypervisor call* which notifies PANDA to query the taint system about a specific source-level variable. The function return value also gets a taint query hypercall. LAVA also uses Clang to insert source hypervisor calls at potential attack points. It should be noted that the query points employed by LAVA are by no means exhaustive. There is every reason to expect that querying at pointer dereferences, e.g., might yield a number of additional DUAs.

B. Running the program

Once the target has been instrumented with taint queries, we run it against a variety of inputs. Since our approach to gathering data about the program is fundamentally dynamic, we must take care to choose inputs to maximize code coverage. To run the program, we load it as a virtual CD into a PANDA virtual machine and send commands to QEMU over a virtual serial port to execute the program against the input.

As the hypervisor calls in the program execute, PANDA logs results from taint queries and attack point encounters to a binary log file, the *pandalog*. Information about control flow transfers that depend on tainted data is also recorded in the pandalog so that it can be used to compute the liveness of each input byte. Note that because the pandalog is generated by hypercalls inserted into program source code, it can connect source-level information like variable names and source file locations to the taint queries and attack points. This allows bug injection, later, to make use of source-level information.

C. Mining the Pandalog

We then analyze the pandalog in temporal order, matching up DUAs with attack points to find potentially injectable bugs. The program that does this is called `FIB` for "find injectable bugs", and is detailed in Figure 6. `FIB` considers the pandalog entries in temporal order.

Taint query entries are handled by the function `collect_duas` which maintains a set of currently viable DUAs. Viable DUAs must have enough tainted bytes, and those bytes must be below some threshold for taint set cardinality and TCN. Additionally, the liveness associated with all the input bytes which taint the DUA must be below a threshold. Note that a DUA is associated with a

```
def check_liveness(file_bytes):
  for file_byte in file_bytes:
    if (liveness[file_byte]
        > max_liveness):
      return False
  return True

def collect_duas(taint_query):
  retained_bytes = []
  for tainted_byte in taint_query:
    if tainted_byte.tcn <= max_tcn
    and
    len(tainted_byte.file_offsets) <= max_card
    and
    check_liveness(tainted_byte.file_offsets)):
      retained_bytes += tainted_byte.file_offsets
  duakey = (taint_query.source_loc,
    taint_query.ast_name)
  duas[duakey] = retained_bytes

def update_liveness(tainted_branch):
  for tainted_file_offset in tainted_branch:
    liveness[tainted_file_offset]++

def collect_bugs(attack_point):
  for dua in duas:
    viable_count = 0
    for file_offset in dua:
      if (check_liveness(file_offset)):
        viable_count ++
    if (viable_count >= bytes_needed):
      bugs.add((dua, attack_point))

for event in Pandalog:
  if event.typ is taint_query:
    collect_duas(event);
  if event.typ is tainted_branch:
    update_liveness(event);
  if event.typ is attack_point:
    collect_bugs(event);
```

Fig. 6: Python-style pseudocode for FIB. Panda log is processed in temporal order and the results of taint queries on values and branches are used to update the current set of DUAs and input byte liveness. When an attack point is encountered, all currently viable DUAs are considered as potential data sources to inject a bug.

specific program point and variable name, and only the last encountered DUA is retained in the viable set. This means that if a DUA is a variable in a loop or in a function that is called many times, the set will only have one entry (the last) for that variable and source location, thus ensuring that value is up to date and potentially usable at an attack point.

Information about the liveness of file input bytes is updated whenever a log entry describing a tainted branch instruction is encountered. Tainted branch information in the pandalog updates liveness for all input bytes involved, in the function `update_liveness`.

Finally, when `FIB` encounters an attack point in the pandalog, the function `collect_bugs` considers each DUA in the set, and those that are still viable with respect to liveness are paired with the attack point as potentially injectable bugs. In the current implementation of LAVA, an attack point is

```
protected int
file_encoding(struct magic_set *ms,
              ..., const char **type) {
...
    else if
      (({int rv =
           looks_extended(buf, nbytes, *ubuf, ulen);
       if (buf) {
         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;})) {
...
```

Fig. 7: Code injected by Clang into file's `src/encodings.c` to copy DUA value off for later use. The function `lava_set` saves the DUA value in a static variable. PANDA taint analysis and the `FIB` algorithm determines that the first four bytes of `buf` are suitable for use in creating the bug.

```
...
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;
```

Fig. 8: Code injected into file's `src/readcdf.c` to use DUA value to create a vulnerability. The function `lava_get` retrieves the value last stored by a call to `lava_set`.

an argument to a function call that can be *made vulnerable by adding a DUA to it*. This means the argument can be a pointer or some kind of integer type; the hope is that changing this value by a large amount may trigger a buffer overflow. Note that as with taint queries, LAVA attack point selection is clearly far from complete. We might imagine attacking pointer reads and writes, their uses in conditionals, etc.

D. Inject and test bugs

For each DUA/attack point pair, we generate the C code which uses the DUA to trigger the bug using another custom Clang tool. At the source line and for the variable in the DUA, we inject code to copy its value into a static variable held by a helper function. At the attack point, we insert code that retrieves the DUA value, determines if it matches a magic value, and if so adds it to one of the function arguments.

The final step in LAVA is simply compiling and testing the modified program on a proof-of-concept input file, in which the input file bytes indicated as tainting the DUA have been set to the correct value. An example of the pair of source code insertions plus the file modifiction in order to inject a

bug into the program `file` can be seen in Figures 7, and 8. The original input to `file` was the binary `/bin/ls`, and the required modification to that file is to simply set its first four bytes to the string 'lava' to trigger the bug. Note that the taint analysis and FIB identifies a DUA in one compilation unit and an attack point in another compilation unit.

VI. Results

We evaluated LAVA in three ways. First, we injected large numbers of bugs into four open source programs: file, readelf (from binutils), bash, and tshark (the command-line version of the packet capture and analysis tool Wireshark). For each of these, we report various statistics with respect to both the target program and also LAVA's success at injecting bugs. Second, we evaluated the distribution and realism of LAVA's bugs by proposing and computing various measures. Finally, we performed a preliminary investigation to see how effective existing bug-finding tools are at finding LAVA's bugs, by measuring the detection rates of an open-source fuzzer and a symbolic execution-based bug finder.

Counting Bugs

Before we delve into the results, we must specify what it is we mean by an injected bug, and what makes two injected bugs distinct. Although there are many possible ways to define a bug, we choose a definition that best fits our target use case: two bugs should be considered different if an automated tool would have to reason about them differently. For our purposes, we define a bug as a unique pair $(DUA, attackpoint)$. Expanding this out, that means that the source file, line number, and variable name of the DUA, and the source file and line number of the attack point must be unique.

Some might object that this artificially inflates the count of bugs injected into the program, for example because it would consider two bugs distinct if they differ in where the file input becomes available to the program, even though the same file input bytes are used in both cases. But in fact these should be counted as different bugs: the data and control flow leading up to the point where the DUA occurs will be very different, and vulnerability discovery tools will have to reason differently about the two cases.

A. Injection Experiments

The results of injecting bugs into open source programs are summarized in Table I. In this table, programs are ordered by size, in lines of C code, as measured by David Wheeler's `sloccount`. A single input was used with each program to measure taint and find injectable bugs. The input to `file` and `readelf` was the program `ls`. The input to `tshark` was a 16K packet capture file from a site hosting a number of such examples. The input to `bash` was a 124-line shell script written by the authors. $N(DUA)$ and $N(ATP)$ are the number of DUAs and attack points collected by the `FIB` analysis. Note that, in order for a DUA or attack point to be counted, it must have been deemed viable for some bug, as described in Section V-C. The columns *Potential Bugs* and *Validated Bugs* in Table I give the numbers of both potential bugs found by `FIB`, but also those verified to actually return exitcodes indicating a buffer overflow (-11 for segfault or -6 for heap corruption) when run against the modified input. The penultimate column in the table is *Yield*, which is the fraction of potential bugs what were tested and determined to be actual buffer overflows. The last column gives the time required to test a single potential bug injection for the target.

Exhaustive testing was not possible for a number of reasons. Larger targets had larger numbers of potential bugs and take longer to test; for example, `tshark` has over a million potential bugs and each takes almost 10 minutes to test. This is because testing requires not only injecting a small amount of code to add the bug, but also recompiling and running the resulting program. For many targets, we found the build to be subtly broken so that a `make clean` was necessary to pick up the bug injection reliably, which further increased testing time. Instead, we attempted to validate 2000 potential bugs chosen uniformly at random for each target. Thus, when we report in Table I that for `tshark` the yield is 17.7%, this is because 306 out of 2000 bugs were found to be valid.

As the injected bug is designed to be triggered only if a particular set of four bytes in the input is set to a magic value, we tested with both the original input and with the modified one that contained the trigger. We did not encounter any situation in which the original input triggered a crash.

Yield varies considerably from less than 10% to over 50%. To understand this better, we investigated the relationship between our two taint-based measures and yield. For each DUA used to inject a bug, we determined $mTCN$, the maximum TCN for any of its bytes and $mLIV$, the maximum liveness for any label in any taint label set associated with one of its bytes. More informally, $mTCN$ represents how complicated a function of the input bytes a DUA is, and $mLIV$ is a measure of how much the control flow of a program is influenced by the input bytes that determine a DUA.

Table II shows a two-dimensional histogram with bins for $mTCN$ intervals along the vertical axis and bins for $mLIV$ along the horizontal axis. The top-left cell of this table represents all bug injections for which $mTCN < 10$ and $mLIV < 10$, and the bottom-right cell is all those for which $mTCN >= 1000$ and $mLIV >= 1000$. Recall that when $mTCN = mLIV = 0$, the DUA is not only a direct copy of input bytes, but those input bytes have also not been observed to be used in deciding any program branches. As either $mTCN$ or $mLIV$ increase, yield deteriorates. However, we were surprised to observe that $mLIV$ values of over 1000 still gave yield in the 10% range.

TABLE II: Yield as a function of both $mLIV$ and $mTCN$

	$mLIV$			
$mTCN$	$[0, 10)$	$[10, 100)$	$[100, 1000)$	$[1000, +\inf]$
$[0, 10)$	51.9%	22.9%	17.4%	11.9%
$[10, 100)$	–	0	0	0
$[100, +\inf]$	–	–	–	0

TABLE I: LAVA Injection results for open source programs of various sizes

Name	Version	Num Src Files	Lines C code	N(DUA)	N(ATP)	Potential Bugs	Validated Bugs	Yield	Inj Time (sec)
file	5.22	19	10809	631	114	17518	774	38.7%	16
readelf	2.25	12	21052	3849	266	276367	1064	53.2 %	354
bash	4.3	143	98871	3832	604	447645	192	9.6%	153
tshark	1.8.2	1272	2186252	9853	1037	1240777	354	17.7%	542

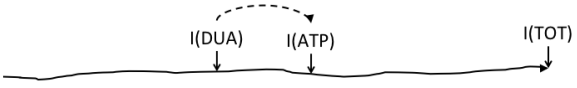

Fig. 9: A cartoon representing an entire program trace, annotated with instruction count at which DUA is siphoned off to be used, $I(DUA)$, attack point where it is used, $I(ATP)$, and total number of instructions in trace, $I(TOT)$.

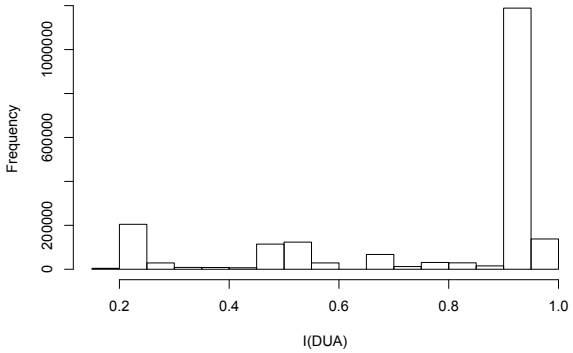

Fig. 10: Normalized DUA trace location

B. Bug Distribution

It would appear that LAVA can inject a very large number of bugs into a program. If we extrapolate from yield numbers in Table I, we estimate there would be almost 400,000 real bugs if all were tested. But how well distributed is this set of bugs?

For programs like `file` and `bash`, between 11 and 44 source files are involved in a potential bug. In this case, the bugs appear to be fairly well distributed, as those numbers represent 58% and 31% of the total for each, respectively. On the other hand, `readelf` and `tshark` fare worse, with only 2 and 122 source files found to involve a potential bug for each (16.7% and 9.6% of source files).

The underlying cause for the low numbers of files in which bugs appear seems to be poor dynamic coverage. For `tshark`, much of the code is devoted to parsing esoteric network protocols, and we used only a single input file. Similarly, we only used a single hand-written script with `bash`, and made little attempt to cover a majority of language features. Finally, we ran `readelf` with a single command line flag (`-a`); this means that functionality such as DWARF symbol parsing was not exercised.

C. Bug Realism

The intended use of the bugs created by this system is as ground truth for development and evaluation of vulnerability discovery tools and techniques. Thus, it is crucial that they be realistic in some sense. Realism is, however, difficult to assess.

Because this work is, to our knowledge, the first to consider the problem of fully automated bug injection, we are not able to make use of any standard measures for bug realism. Instead, we devised our own measures, focusing on features such as how well distributed the malformed data input and trigger points were in the program's execution, as well as how much of the original behavior of the program was preserved.

We examined three aspects of our injected bugs as measures of realism. The first two are DUA and attack point position within the program trace, which are depicted in Figure 9. That is, we determined the fraction of trace instructions executed at the point the DUA is siphoned off and at the point it is used to attack the program by corrupting an internal program value.

Histograms for these two quantities, $I(DUA)$ and $I(ATP)$, are provided in Figures 10 and 11, where counts are for all potential bugs in the LAVA database for all five open source programs. DUAs and attack points are clearly available at all points during the trace, although there appear to be more at the beginning and end. This is important, since bugs created using these DUAs have entirely realistic control and data-flow all the way up to $I(DUA)$. Therefore, vulnerability discovery tools will have to reason correctly about all of the program up to $I(DUA)$ in order to correctly diagnose the bug.

Our third metric concerns the portion of the trace *between* the $I(DUA)$ and $I(ATP)$. This segment is of particular interest since LAVA currently makes data flow between DUA and attack point via a pair of function calls. Thus, it might be argued that this is an unrealistic portion of the trace in terms of data flow. The quantity $I(DUA)/I(ATP)$ will be close to 1 for injected bugs that minimize this source of unrealism. This would correspond to the worked example in Figure 1; the DUA is still in scope when, a few lines later in the same function, it can be used to corrupt a pointer. No abnormal data flow is required. The histogram in Figure 12 quantifies this effect for all potential LAVA bugs, and it is clear that a large fraction have $I(DUA)/I(ATP) \approx 1$, and are therefore highly realistic by this metric.

D. Vulnerability Discovery Tool Evaluation

We ran two vulnerability discovery tools on LAVA-injected bugs to investigate their use in evaluation.

1) Coverage guided fuzzer (referred to as FUZZER)
2) Symbolic execution + SAT solving (referred to as SES)

These two, specifically, were chosen because fuzzing and symbolic execution are extremely popular techniques for find-

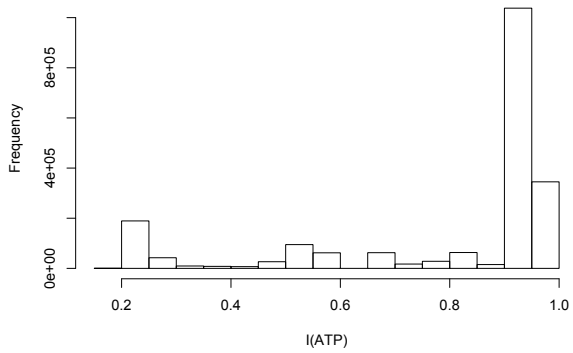

Fig. 11: Normalized ATP trace location

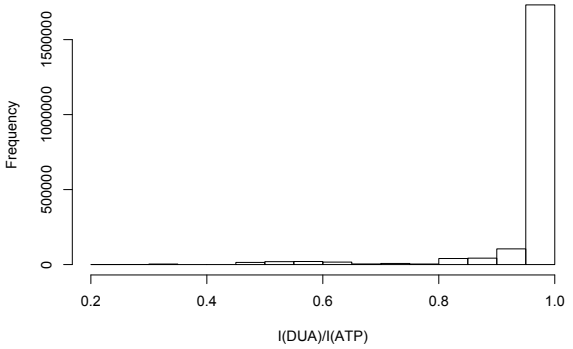

Fig. 12: Fraction of trace with perfectly normal or realistic data flow, $I(DUA)/I(ATP)$

ing real-world bugs. FUZZER and SES are both state-of-the-art, high-profile tools. For each tool, we expended significant effort to ensure that we were using them correctly. This means carefully reading all documentation, blog posts, and email lists. Additionally, we constructed tiny example buggy programs and used them to verify that we were able to use each tool at least to find known easy bugs.

Note that the names of tools under evaluation are being withheld in reporting results. Careful evaluation is a large and important job, and we would not want to give it short shrift, either in terms of careful setup and use of tools, or in presenting and discussing results. Our intent, here, is to determine if LAVA bugs *can be used* to evaluate bug finding systems. It is our expectation that in future work either by ourselves or others, full and careful evaluation of real, named tools will be performed using LAVA. While that work is outside the scope of this paper, we hope to indicate that it should be both possible and valuable. Additionally, it is our plan and hope that LAVA bugs will be made available in quantity and at regular refresh intervals for self-evaluation and hill climbing.

The first corpus we created, *LAVA-1*, used the `file` target, the smallest of those programs into which we have injected bugs. This corpus consists of sixty-nine buffer overflow bugs injected into the source with LAVA, each on a different branch in a `git` repository with a fuzzed version of the input verified to trigger a crash checked in along with the code. Two types of buffer overflows were injected, each of which makes use of a single 4-byte DUA to trigger and control the overflow.

1) **Knob-and-trigger**. In this type of bug, two bytes of the DUA (the *trigger*) are used to test against a magic value to determine if the overflow will happen. The other two bytes of the DUA (the *knob*) determine how much to overflow. Thus, these bugs manifest if a 2-byte unsigned integer in the input is a particular value but only if another 2-bytes in the input are big enough to cause trouble.
2) **Range**. These bugs trigger if the magic value is simply in some range, but also use the magic value to determine how much to overflow. The magic value is a 4-byte unsigned integer and the range varies.

These bug types were designed to mirror real bug patterns. In knob-and-trigger bugs, two different parts of the input are used in different ways to determine the manifestation of the bug. In range bugs, rather than triggering on a single value out of 2^{32}, the size of the haystack varies. Note that a range of 2^0 is equivalent to the bug presented in Figure 8.

TABLE III: Percentage of bugs found in *LAVA-1* corpus

Tool	Bug Type					
	Range					
	2^0	2^7	2^{14}	2^{21}	2^{28}	KT
FUZZER	0	0	9%	79%	75%	20%
SES	8%	0	9%	21%	0	10%

The results of this evaluation are summarized in Table III. Ranges of five different sizes were employed: 2^0 (12 bugs), 2^7 (10 bugs), 2^{14} (11 bugs), 2^{21} (14 bugs), and 2^{28} (12 bugs); we used 10 knob-and-trigger bugs. We examined all output from both tools. FUZZER ran for five hours on each bug and found bugs in the larger ranges (2^{14}, 2^{21}, and 2^{28}). It was also able to uncover 20% of the knob-and-trigger bugs, perhaps because the knob and trigger could be fuzzed independently. SES ran for five hours on each bug, and found several bugs in all categories except the 2^7 and 2^{28} ranges.

The results for the *LAVA-1* corpus seem to accord well with how these tools work. FUZZER uses the program largely as a black box, randomizing individual bytes, and guiding exploration with coverage measurements. Bugs that trigger if and only if a four-byte extent in the input is set to a magic value are unlikely to be discovered in this way. Given time, FUZZER finds bugs that trigger for large byte ranges. Note that for many of these LAVA bugs, when the range is so large, discovery is possible by simply fuzzing every byte in the input a few times. These bugs may, in fact, be trivially discoverable with a regression suite for a program like `file` that accepts arbitrary file input.[1] By contrast, SES is able to find both knob-and-trigger bugs and different ranges, and the size of the range does not affect the number of bugs found. This is because it is no more difficult for a SAT solver to find a satisfying input for

[1]In principle, anyway. In practice `file`'s test suite consists of just 3 tests, none of which trigger our injected bugs.

a large range than a small range; rather, the number of bugs found is limited by how deep into the program the symbolic execution reaches.

Note that having each bug in a separate copy of the program means that for each run of a bug finding tool, only one bug is available for discovery at a time. This is one kind of evaluation, but it seems to disadvantage tools like FUZZER and SES, which appear to be designed to work for a long time on a single program that may contain multiple bugs.

Thus, we created a second corpus, *LAVA-M*, in which we injected more than one bug at a time into the source code. We chose four programs from the `coreutils` suite that took file input: `base64`, `md5sum`, `uniq`, and `who`. Into each, we injected as many verified bugs as possible. Because the `coreutils` programs are quite small, and because we only used a single input file for each to perform the taint analysis, the total number of bugs injected into each program was generally quite small. The one exception to this pattern was the `who` program, which parses a binary file with many dead or even unused fields, and therefore had many DUAs available for bug injection.

We were not able to inject multiple bugs of the two types described above (knob-and-trigger and range) as interactions between bugs became a problem, and so all bugs were of the type in Figure 8, which trigger for only a single setting of four input bytes. The *LAVA-M* corpus, therefore, is four copies of the source code for `coreutils` version 8.24. One copy has 44 bugs injected into `base64`, and comes with 44 inputs known to trigger those bugs individually. Another copy has 57 bugs in `md5sum`, and a third has 28 bugs in `uniq`. Finally, there is a copy with 2136 bugs existing all at once and individually expressible in `who`.

TABLE IV: Bugs found in *LAVA-M* corpus

Program	Total Bugs	Unique Bugs Found		
		FUZZER	SES	Combined
`uniq`	28	7	0	7
`base64`	44	7	9	14
`md5sum`	57	2	0	2
`who`	2136	0	18	18
Total	2265	16	27	41

We ran FUZZER and SES against each program in *LAVA-M*, with 5 hours of runtime for each program. `md5sum` ran with the `-c` argument, to check digests in a file. `base64` ran with the `-d` argument, to decode base 64.

SES found no bugs in `uniq` or `md5sum`. In `uniq`, we believe this is because the control flow is too unconstrained. In `md5sum`, SES failed to execute any code past the first instance of the hash function. `base64` and `who` both turn out more successful for SES. The tool finds 9 bugs in `base64` out of 44 inserted; these include both deep and shallow bugs, as `base64` is such a simple program to analyze.

SES's results are a little more complicated for `who`. All of the bugs it finds for `who` use one of two DUAs, and all of them occur very early in the trace. One artifact of our method for injecting multiple bugs simultaneously is that multiple bugs share the same attack point. It is debatable how well this represents real bugs. In practice, it means that SES can only find one bug per attack point, as finding an additional bug at the same attack point does not necessarily require covering new code. LAVA could certainly be changed to have each bug involve new code coverage. SES could also be improved to find all the bugs at each attack point, which means generating multiple satisfying inputs for the same set of conditions.

FUZZER found bugs in all utilities except `who`.[2] Unlike SES, the bugs were fairly uniformly distributed throughout the program, as they depend only on guessing the correct 4-byte trigger at the right position in the input file.

FUZZER's failure to find bugs in `who` is surprising. We speculate that the size of the seed file (the first 768 bytes of a `utmp` file) used for the fuzzer may have been too large to effectively explore through random mutation, but more investigation is necessary to pin down the true cause. Indeed, tool anomalies of this sort are exactly the sort of thing one would hope to find with LAVA, as they represent areas where tools might make easy gains.

We note that the bugs found by FUZZER and SES have very little overlap (only 2 bugs were found by both tools). This is a very promising result for LAVA, as it indicates that the kinds of bugs created by LAVA are not tailored to a particular bug finding strategy.

VII. Related Work

The design of LAVA is driven by the need for bug corpora that are a) dynamic (can produce new bugs on demand), b) realistic (the bugs occur in real programs and are triggered by the program's normal input), and c) large (consist of hundreds of thousands of bugs). In this section we survey existing bug corpora and compare them to the bugs produced by LAVA.

The need for realistic corpora is well-recognized. Researchers have proposed creating bug corpora from student code [18], drawing from existing bug report databases [12], [13], and creating a public bug registry [7]. Despite these proposals, public bug corpora have remained static and relatively small.

The earliest work on tool evaluation via bug corpora appears to be by Wilander and Kamkar, who created a synthetic testbed of 44 C function calls [22] and 20 different buffer overflow attacks [23] to test the efficacy of static and dynamic bug detection tools, respectively. These are synthetic test cases, however, and may not reflect real-world bugs. In 2004, Zitser et al. [27] evaluated static buffer overflow detectors; their ground truth corpus was painstakingly assembled by hand over the course of six months and consisted of 14 annotated buffer overflows with triggering and non-triggering inputs as well as buggy and patched versions of programs; these same 14 overflows were later used to evaluate dynamic overflow detectors [25]. Although these are real bugs from actual software, the corpus is small both in terms of the number of

[2] In fact, we allowed FUZZER to continue running after 5 hours had passed; it managed to find a bug in `who` in the sixth hour.

bugs (14) but also in terms of program size. Even modestly-sized programs like `sendmail` were too big for some of the static analyzers and so much smaller models capturing the essence of each bug were constructed in a few hundred lines of excerpted code.

The most extensive effort to assemble a public bug corpus comes from the NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project [10]. Their evaluation corpus inclues Juliet [2], a collection of 86,864 synthetic C and Java programs that exhibit 118 different CWEs; each program, however, is relatively short and has uncomplicated control and data flow. The corpus also includes the IARPA STONESOUP data set [19], which was developed in support of the STONESOUP vulnerability mitigation project. The test cases in this corpus consist of 164 small snippets of C and Java code, which are then spliced into program to inject a bug. The bugs injected in this way, however, do not use the original input to the program (they come instead from extra files and environment variables added to the program), and the data flow between the input and the bug is quite short.

Most recently, Shiraishi et al. [17] conducted a quantitative analysis of commercial static analysis tools by constructing 400 pairs of C functions, where each pair consisted of two versions of the same function, one with a bug and one without. The bugs covered a range of different error types, including static/dynamic buffer overruns, integer errors, and concurrency bugs. They then ranked the tools according to effectiveness and, by incorporating price data for commercial tools, produced a measure of efficiency for each tool. As with previous synthetic corpora, however, the functions themselves are relatively short, and may be easier to detect than bugs embedded deep in large code bases.

Finally, the general approach of automatic program transformation to introduce errors was also used by Rinard et al. [15]; the authors systematically modified the termination conditions of loops to introduce off-by-one errors in the Pine email client to test whether software is still usable in the presence of errors once sanity checks and assertions are removed.

VIII. Limitations and Future Work

A significant chunk of future work for LAVA involves making the generated corpora look more like the bugs that are found in real programs. LAVA currently injects only buffer overflows into programs. But our taint-based analysis overcomes the crucial first hurdle to injecting any kind of bug: making sure that attacker-controlled data can be used in the bug's potential exploitation. As a result, other classes of bugs, such as temporal safety bugs (use-after-free) and meta-character bugs (e.g. format string) should also be injectable using our approach. There also remains work to be done in making LAVA's bug-triggering data flow more realistic, although even in its current state, the vast majority of the execution of the modified program is realistic. This execution includes the data flow that leads up to the capture of the DUA, which is often nontrivial.

However rosy the future seems for LAVA, it is likely that certain classes of bugs are simply not injectable via taint-based measures. Logic errors, crypto flaws, and side-channel vulnerabilities, for instance, all seem to operate at a rather different level than the kinds of data-flow triggered flaws LAVA is well positioned to generate. We are not hopeful that these types of vulnerabilities will soon be injectable with LAVA.

We discovered, in the course of our use of LAVA bugs to evaluate vulnerability discovery tools, a number of situations in which LAVA introduces unintended bugs, such as use-after free and dereference of an uninitialized pointer in the code that siphons off a DUA value for later use triggering a bug. In some cases, the tool under evaluation even found these real bugs that were due to LAVA artifacts and we had to remove them and re-run in order to ensure that the evaluation was not compromised. These artifacts are a result of LAVA performing no real static analysis to determine if it is even vaguely safe to dereference a pointer in order to introduce the data flow it requires to inject a bug. It should be possible to remedy this situation dramatically in many cases but a complete solution would likely require intractable whole-program static analysis.

LAVA is limited to only work on C source code, but there is no fundamental reason for this. In principle, our approach would work for any source language with a usable source-to-source rewriting framework. In Python, for example, one could easily implement our taint queries in a modified CPython interpreter that executed the hypervisor call against the address of a variable in memory. Since our approach records the correspondence between source lines and program basic block execution, it would be just as easy to figure out where to edit the Python code as it is in C. We have no immediate plans to extend LAVA in these directions.

We are planning some additional evaluation work. In particular, an extensive evaluation of real, named tools should be undertaken. The results will shed light on the strengths and weaknesses of classes of techniques, as well as particular implementations. It should also be noted that in our preliminary evaluation of vulnerability discovery tools we measured only the *miss rate*; no attempt was made to gauge the *false alarm rate*. For tools that generate a triggering input, as do both SES and FUZZER, measuring false alarm rate should be trivial. Every input can be tested against the program after it has been instrumented to be able to detect the vulnerability. In the case of buffer overflows in C, this could mean compiling in fine-grained bounds checking [16]. However, many bug finding tools, especially static analyzers and abstract interpretation ones, do not generate bug-triggering inputs. Instead, they merely gesture at a line in the program and make a claim about possible bugs at that point. In this situation, we can think of no way to assess false alarm rate without extensive manual effort.

IX. Conclusion

In this paper, we have introduced LAVA, a fully automated system that can rapidly inject large numbers of realistic bugs

into C programs. LAVA has already been used to introduce over 4000 realistic buffer overflows into open-source Linux C programs of up to 2 million lines of code. We have used LAVA corpora to evaluate the detection powers of state-of-the-art bug finding tools. The taint-based measures employed by LAVA to identify attacker-controlled data for use in creating new vulnerabilities are powerful and should be usable to inject many and diverse vulnerabilities, but there are likely fundamental limits; LAVA will not be injecting logic errors into programs anytime soon. Nevertheless, LAVA is ready for immediate use as an on-demand source of realistic ground truth vulnerabilities for classes of serious vulnerabilities that are still abundant in mission-critical code. It is our hope that LAVA can drive both the development and evaluation of advanced tools and techniques for vulnerability discovery.

X. Acknowledgements

Many thanks to Graham Baker, Chris Connelly, Jannick Pewny, and Stelios Sidiroglou-Douskos for valuable early discussions and suggestions. In addition, we thank Amy Jiang for some critical initial Clang development and debugging.

References

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08. USENIX Association, 2008.

[2] Center for Assured Software. Juliet test suite v1.2 user guide. Technical report, National Security Agency, 2012.

[3] Sang Kil Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, 2012.

[4] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[5] Brendan Dolan-Gavitt, Joshua Hodosh, Patrick Hulin, Timothy Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Workshop on Program Protection and Reverse Engineering (PPREW)*, 2015.

[6] Keromytis et al. Tunable cyber defensive security mechanisms. https://www.sbir.gov/sbirsearch/detail/825791, August 2015.

[7] Jeffrey Foster. A call for a public bug and tool registry. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[8] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009.

[9] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*. USENIX, 2013.

[10] Michael Kass. NIST software assurance metrics and tool evaluation (SAMATE) project. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[11] Kendra Kratkiewicz and Richard Lippmann. Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[12] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: A benchmark for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[13] Barmak Meftah. Benchmarking bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[14] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Symposium (NDSS)*, 2005.

[15] Martin Rinard, Cristian Cadar, and Huu Hai Nguyen. Exploring the acceptability envelope. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 21–30, New York, NY, USA, 2005. ACM.

[16] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[17] Shin'ichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. In *Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering*, ISSRE '15, 2015.

[18] Jaime Spacco, David Hovemeyer, and William Pugh. Bug specimens are important. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[19] TASC, Inc., Ponte Technologies LLC, and i_SW LLC. STONESOUP phase 3 test generation report. Technical report, SAMATE, 2014.

[20] Vlad Tsyrklevich. Hacking team: A zero-day market case study. https://tsyrklevich.net/2015/07/22/hacking-team-0day-market/, July 2015.

[21] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[22] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, 2002.

[23] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, 2003.

[24] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy*, 2014.

[25] Michael Zhivich, Tim Leek, and Richard Lippmann. Dynamic buffer overflow detection. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[26] Misha Zitser, Richard Lippmann, and Tim Leek. Personal communication.

[27] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 97–106, New York, NY, USA, 2004. ACM.