LAVA: Large-scale Automated Vulnerability Addition

Brendan Dolan-Gavitt*, Patrick Hulin†, Tim Leek†, Fredrich Ulrich†, Ryan Whelan†

(Authors listed alphabetically)

*NYU
brendandg@nyu.edu
†MIT Lincoln Laboratory
{patrick.hulin,tleek,rwhelan}@ll.mit.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. To begin to address this, we present LAVA, a system for automatically and quickly injecting large numbers of realistic bugs into program source code. LAVA employs a pair of taint-based measures to identify program quantities that both depend upon specific input bytes in a simple way yet do not overly influence control flow. These DUAs (dead-uncomplicated and available data) are employed, via source-to-source transformation, to perturb program quantities at later program points that are likely to cause vulnerabilities. Every LAVA vulnerability is accompanied by a input that triggers it, whereas normal inputs are extremely unlikely to do so. Further, every injected bug is validated, and thus every working bug comes with both a proof-of-concept input and a known manifestation point. 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. In order for an automated tool to discover them, it would have to be able to reason correctly and precisely about all the code executed up to the DUA. Using LAVA, we have injected thousands of bugs into popular programs such as file, readelf, bash, and tshark. We believe LAVA can form the basis of an approach for generating extremely high quality ground truth vulnerability corpora on demand.**

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 [6] 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 [17]. 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. The author's own experience creating such a corpus was that it is a difficult and lengthy process; a corpus of only fourteen very well annotated historic bugs with triggering inputs took about six months to construct. In addition, public corpora have the disadvantage of already being released, and thus rapidly become stale. 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 [13], 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 [7], [1].

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 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 all or a large fraction of inputs it is trivially discoverable by simply running the program.

Distribution A: Public Release. This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

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. We call this implementation LAVA for 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 a 15 seconds. LAVA bugs manifest all along the execution trace, in all parts of the program, shallow and deep, and make use of mostly completely normal data flow. By construction, a LAVA bug comes with an input that triggers it, and no other input can have this effect upon the program.

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. Most automated vulnerability discovery systems work with source code, and we can easily test binary analysis tools by simply compiling the modified source. Injecting bugs into binaries or byte code 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. We produce a proof-of-concept input to trigger any bug we successfully inject, although we do not attempt to produce an actual exploit.

III. LAVA Overview

At a high level, LAVA adds bugs to programs in the following manner.

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.
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

In a little more detail, the first step, in which DUAs are identified, is accomplished as follows.

```
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==0x76697461)*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.

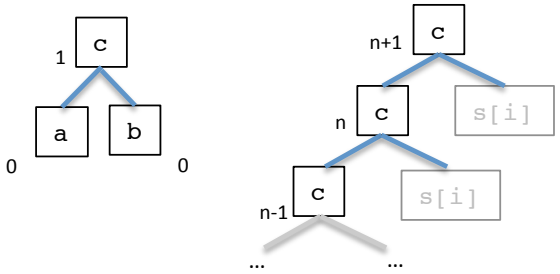

Fig. 2: Taint Compute Number examples from the running example. TCN is simply the depth of the tree of computation that produces the value from tainted inputs. TCN(c) after line 2 is 1, and after line 6 (upon exiting the loop), it is n+1.

The program is executed under a dynamic taint analysis 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 and a direct copy of input bytes, 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 is 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 basis. Whenever the taint system needs to union 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 can have 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, $LIV(8..11) = n$.

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.

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 useable 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 manifest 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.

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 pass), but it is not a reliable source of bugs. There is no reliable 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 [4]: 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.

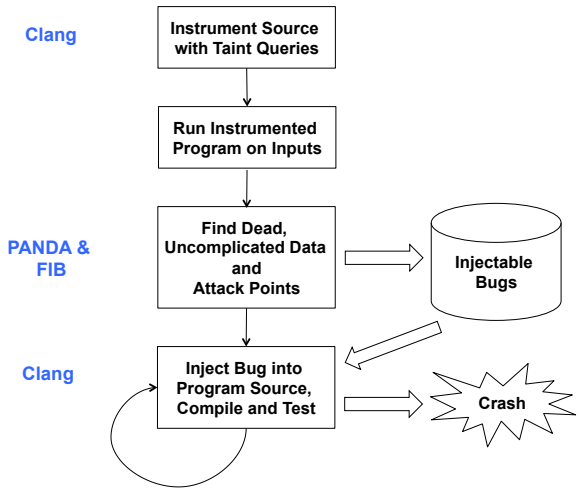

Fig. 3: LAVA Implementation Architecture. PANDA and Clang are used to perform a dynamic taint analysis which identifies potential bug injections as DUA attack point pairs. Each of these is validated with a corresponding source code change performed by Clang as well. Finally, every potentially buggy binary is tested against a targeted input change to determine if a buffer overflow actually results.

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 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.

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 3

A. Taint queries

LAVA's taint queries rely on the PANDA dynamic analysis platform [3], 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 that 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 [2], 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. LAVA employs a simple PANDA plugin `file_taint` that is able to apply taint labels to bytes read from files in Linux. This 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 is how LAVA can 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 Clang plugin to insert taint queries into the source before and after function calls. Each function argument is deconstructed into constituent lvals, and, for each, Clang adds a taint query 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.

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*. 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 4. `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
    &&
    len(tainted_byte.file_offsets) <= max_card
    &&
    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. 4: 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. Tainted branch information in the pandalog updates liveness for all input bytes involved, in the function `update_liveness`. 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 a potentially injectable bugs. In the current implementation of LAVA, an attack point is 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

```
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. 5: PANDA taint analysis and the `FIB` algorithm determines that the first four bytes of `buf` are suitable for use in creating a bug. This is the 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.

```
...
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. 6: 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`.

value by a large amount may trigger a buffer overflow.

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 Clang plugin. 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, an argument to a function call, we insert code that retrieves the DUA value, determines if it matches a magic value, and if so adds it to one of the argument. 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 5, and 6. 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.

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	39.0%	16
readelf	2.25	12	21052	3849	266	276367	1064	53.4 %	354
bash	4.3	143	98871	3832	604	447645	192	9.5%	153
tshark	1.8.2	1272	2186252	9853	1037	1240777	354	15.3%	542

TABLE I: LAVA Injection results for open source programs of various sizes. For each, a single input file was used to perform a taint analysis with PANDA. Various program and dynamic trace statistics are reported as well as DUA, attack point (ATP), and yield (fraction of injected bugs that result in a segmentation violation).

VI. Results

We evaluated LAVA in two ways. First, we injected large numbers of bugs into four open source programs: file, readelf (from binutils), bash, and tshark (command-line verison of 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.

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 each of the target programs, we attempted to validate 2000 potential bugs. Note that larger targets had larger numbers of potential bugs: `tshark` has over a million. Larger targets also take longer to test: almost 10 minutes for `tshark`. 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.

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 caused an overflow.

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 is a two-dimensional histogram with the 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 also, those input bytes have 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.

B. Bug Distribution

It would appear as though LAVA can inject a very large number of bugs. If we extrapolate from yield numbers in Table I, we estimate there would be almost 400,000 real bugs if all were tested.

$$0.390 \times 17518 + 0.095 \times 447645 + 0.534 \times 276367 + 0.153 \times 1240777 \approx 386777$$

But how well distributed is this set of bugs? For programs like

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

TABLE II: Yield as a function of both $mLIV$ and $mTCN$. Yield is highest for DUAs with low values for both of these measures, i.e., that are both a relatively uncomplicated function of input bytes and also that derive from input bytes involved in deciding fewer branches. Cells for which there were no samples are indicated with the contents '–'.

Fig. 7: 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, $I(TOT)$.

`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). For `tshark`, much of the code for which is devoted to parsing esoteric network protocols, coverage is probably the issue since we use only a single input. Similiary, we only use a single hand-written script with `bash`, with little attempt to cover a majority of language features. We are unsure why so few of the source files in `readelf` involve a potential bug.

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. 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 7. 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 8 and 9, 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. The portion of the trace *between* the $I(DUA)$ and $I(ATP)$ is of particular interest since, currently, LAVA 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 10 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.

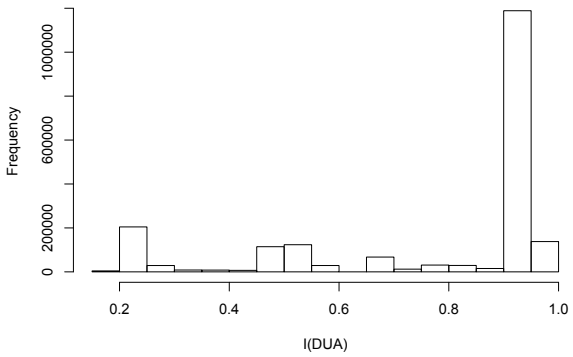

Fig. 8: Normalized DUA trace location

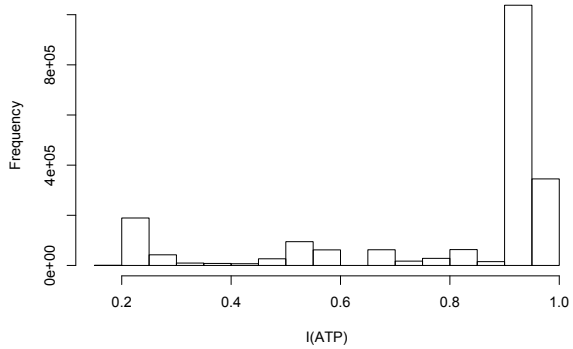

Fig. 9: Normalized DUA trace location

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 [11], drawing from existing bug report databases [8], [9],

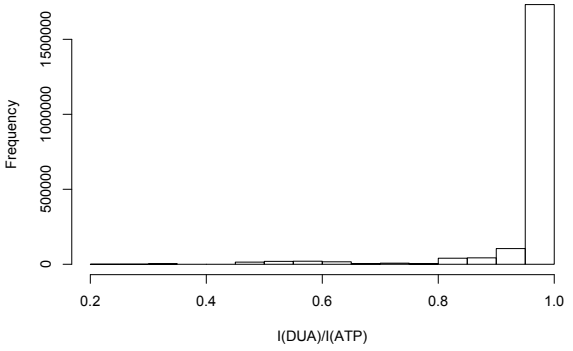

Fig. 10: Fraction of trace with perfectly normal or realistic data flow, $I(DUA)/I(ATP)$

and creating a public bug registry [5]. 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 [14] and 20 different buffer overflow attacks [15] 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. [17] 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 [16]. Although these are real bugs from actual software, the corpus is small both in terms of the number of bugs (14) but also in terms of program size. Even modest sized programs like `sendmail` were too big for some of the static analyzers and so much smaller models capturing the essense 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 [6]. Their evaluation corpus inclues Juliet [1], 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 & data flow. The corpus also includes the IARPA STONESOUP data set [12], 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.

Finally, the general approach of automatic program transformation to introduce errors was also used by Rinard et al. [10]; 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

Future work for LAVA largely involves making the generated corpora look more like the bugs that are found in real programs. First, LAVA currently injects only buffer overflows. 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, the addition of 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 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 dataflow that leads up to the capture of the DUA, which is often nontrivial.

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 CPython plugin 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 evaluatory work. In particular, we want to determine if the bugs that LAVA injects can be found by current vulnerability discovery tools, both open source and commercial. It should be possible to measure both miss and false alarm rates for these tools. Miss rate is straightforward as we can merely compute the fraction of injected, validated bugs that a tool finds. False alarm rate will be trickier to estimate, as it requires determining with certainty if a bug claimed by a tool is real or not. If we do this, the cost for evaluation will be high. Alternately, we may choose to run the tools on the targets and investigate all vulnerability claims. All such claims are very likely false alarms and should persist even when LAVA injects bugs. These can be used to estimate false alarm rates.

IX. Conclusion

In this paper, we have introduced LAVA, a fully automated system that can inject large numbers of realistic bugs into C programs. LAVA has already been used to introduce over 2000 realistic buffer overflows into open-source Linux C programs of between 10,000 and 2 million lines of code. The taint-based measures employed by LAVA to identify attacker-controlled data for use in creating new vulnerabilities should be usable to inject other classes of vulnerabilities than the buffer overflow we demonstrate here, and we will pursue that actively. We believe LAVA will be of immense value as an on-demand source ground truth corpora of very large size. The availability of these corpora should energize research and development into automated vulnerability discovery tools and techniques, as well as the evaluation thereof.

References

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

[2] 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.

[3] Brendan Dolan-Gavitt, Joshua Hodosh, Patrick Hulin, Timothy Leek, and Ryan Whelan. Repeatable reverse engineering for the greater good with panda. CUCS 23, Columbia University, 2014.

[4] Keromytis et al. Tunable cyber defnsive security mechanisms, August 2015.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] Vlad Tsyrklevich. Hacking team: A zero-day market case study. Blog post, July 2015.

[14] 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.

[15] 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.

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

[17] 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.