Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach

Zhiqiang Lin Xiangyu Zhang Dongyan Xu
Department of Computer Sciences and CERIAS
Purdue University
{*zlin, xyzhang, dxu*}*@cs.purdue.edu*

Abstract

Software vulnerabilities are the root cause of a wide range of attacks. Existing vulnerability scanning tools are able to produce a set of suspects. However, they often suffer from a high false positive rate. Convicting a suspect and vindicating false positives are mostly a highly demanding manual process, requiring a certain level of understanding of the software. This limitation significantly thwarts the application of these tools by system administrators or regular users who are concerned about security but lack of understanding of, or even access to, the source code. It is often the case that even developers are reluctant to inspect/fix these numerous suspects unless they are convicted by evidence. In this paper, we propose a lightweight dynamic approach which generates evidence for various security vulnerabilities in software, with the goal of relieving the manual procedure. It is based on data lineage tracing, a technique that associates each execution point precisely with a set of relevant input values. These input values can be mutated by an offline analysis to generate exploits. We overcome the efficiency challenge by using Binary Decision Diagrams (BDD). Our tool successfully generates exploits for all the known vulnerabilities we studied. We also use it to uncover a number of new vulnerabilities, proved by evidence.

1 Introduction

Vulnerabilities in software, especially those that are remote exploitable, are the root cause of wave after wave of security attacks, such as botnet, zero-day worms, non-control data corruptions, and even server-break-ins. Thus, analyzing and exposing software vulnerabilities has become one of the most active research areas today.

In the past, software vulnerability detection/exposing approaches could be divided into two categories: *dynamic* and *static*. Dynamic approaches monitor program execution and detect attempts of attacking a software system. Many promising approaches have been proposed in this category, such as TaintCheck [15], Control Flow Integrity [16], and Data Flow Integrity [17]. However, most of these techniques are often active during program execution, thereby incurring non-trivial runtime overhead. Moreover, they aim to detect attacks, and thus vulnerabilities that are not under attack are invisible.

The second type of approaches are static analysis, and notable examples include BOON [18], Splint [19], and Archer [21]. Static analysis is not bound to execution and thus often capable of identifying potential vulnerabilities in a program, and also it imposes no overhead at runtime. Thus, these techniques are more desirable compared with dynamic approaches if they can live to their promise. Unfortunately, most static techniques suffer from a high false-positive rate and generate a large volume of warnings. For example, the static analysis tool Splint has nearly 50% false positive [20], and tools like Flawfinder [1] and RATS [2] often produce hundreds of warnings, in which only a few of them are the real defects. The procedure of convicting real defects and vindicating false positives remains a highly demanding manual effort, requiring understanding of the source code. With respect to system administrators and regular software users who are concerned about security, the lack of the understanding of (even the access to) source code significantly diminishes their enthusiasm about these techniques. With respect to developers, a long list of suspects with only some being true rapidly wears out their patience. Therefore, it becomes a pressing need to develop new techniques to automatically or semiautomatically generate evidence to convict real vulnerabilities.

Random test generation (e.g., fuzz testing [7, 8]) that randomly mutates benign inputs has been used to construct exploits. However, it is known that random test generation is not effective in many cases, e.g. it might take 2^{32} tries to satisfy a simple predicate as "`P1:if (`x`==c)`" because x is a 32 bit random value. Thus, recently, there has been significant advance in combining static software verification principles with symbolic execution in test generation to identify software errors including vulnerabilities

[9, 11, 10, 12, 13, 14]. These techniques aim to explore all feasible program paths to expose potential defects. Such an ambitious goal with symbolic execution incurs scalability issues. For instance, using symbolic execution an execution taking the true branch of `P1` is modeled by the constraint of `C1`:x==`c`. The technique tries to mutate a benign execution through negating constraints and resolve them by a solver, e.g., solving the negated constraint $\neg$`C1` provides a new input value satisfying x!=`c`, which drives the execution to take the false branch of `P1`. The state of the art [13] is capable of handling hundreds of millions of instructions, which only accounts for a few seconds of execution. Furthermore, it often requires the user to annotate symbolic variables (e.g., EXE [9]), which implies understanding of program semantics.

In this paper, we propose a practical *dynamic* approach that is intended to use in combination with other static tools. We observe that although the suspect pool produced by existing static tools has a high false positive rate, it is nonetheless much smaller than the whole population. Therefore, we use existing static tools as the frontend to generate a set of suspects. Our technique then tries to generate exploits for these suspects. A suspect is convicted only when an exploit can be acquired as the evidence. Such exploits significantly assist regular users and administrators to evaluate the robustness of their software and convince vendors to debug and patch. The key idea is to use data lineage tracing to identify a set of input values relevant to the execution of a vulnerable code location. Exploit-specific mutations are applied to the relevant input values in order to trigger an attack, for example, changing an integer value to MAXUINT to induce an integer overflow. Since these inputs are usually a very small subset of the whole input sequence, mutating the whole input, like in random test generation, is avoided. Our technique does not rely on symbolic execution and constraint solving and thus can easily handle long execution. In case an execution that covers a vulnerable code location cannot be found, our tool also allows user interactions to mutate an input so that the execution driven by the mutated input covers the vulnerable code location.

Our technique addresses a wide range of vulnerabilities including buffer overflow, integer overflow, format string, etc. Our dynamic analysis works at binary level, which greatly facilitates users who do not have the source code access but are concerned about software vulnerabilities. Note that a static analysis used as a frontend may or may not require source code access. Using our system, we are able to reproduce exploits of all the known vulnerabilities we studied. We also successfully identify a set of new vulnerabilities and *prove them by evidence*. They were all promptly confirmed by the developers.

The contributions of our paper are highlighted as follows.

- We propose a novel dynamic technique which generates evidence to convict a wide range of real vulnerabilities. Compared with the state of the art of test generation techniques [9, 10, 11], it is less expensive. The output of our tool is a runnable program input to the whole software system instead of a module, and such an input can be easily turned into an exploit.
- The technique is built upon a dynamic program analysis called data lineage tracing. It traces the set of input that is relevant to a particular execution point. The lineage information is used to guide our evidence generation procedure. The challenge of efficiency is overcome by using reduced ordered Binary Decision Diagrams (roBDDs).
- Data lineage on its own is not sufficient in producing evidence. We design a search algorithm that makes use of lineage information and looks for a mutation of a benign program input that triggers a suspicious vulnerability.
- We apply our technique on a set of real software applications and our results show that we are able to reproduce all the known vulnerabilities that we collected for our experiment. Our case study also presents the effectiveness of our tool by convicting suspects which have not been brought to "justice" before.
- Our performance evaluation indicates that our technique has reasonable overhead.

2 Overview

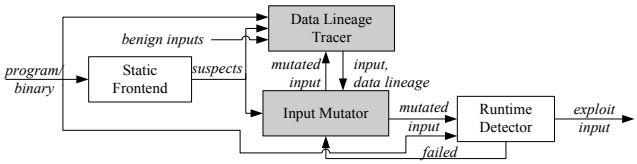

Figure 1. System Overview.

The overview of our system is presented in Figure 1. It consists of four components with the shaded ones being our contributions. The system relies on static analysis to produce a set of suspects, which are potential vulnerable code, represented in the forms of instruction addresses at binary level or source code locations. Benign program inputs are needed to begin with, which may come from a random test case generator or from the test suite shipped with the software. Provided with a program input and a suspect, the data lineage tracing component computes lineage for the execution. This information is consumed by the input mutation component that searches for a way to mutate the previous program input such that the vulnerability is manifested

through a crash. We call such a mutated input an *exploit* in the paper. Note that how to add payload to the crash-inducing input to gain control of the host program is beyond the scope of this paper. The runtime detector is to check if the vulnerability is triggered. If so, the suspect is convicted. Otherwise, the suspect is considered innocent. Since the runtime detector is not our focus, we simply use a segmentation fault detector. More advanced detectors such as TaintCheck [15] can be adopted for higher accuracy.

Our technique does not rely on a specific static analysis tool, which provides flexibility to the system. More specifically, it can be easily shaped into a system handling buffer overflow, format string, integer overflow, or other attacks, depending on the frontend analysis. Although the static and runtime detectors may need source code, our lineage tracing and input mutation components only require binary. The precision of the static analysis is not a major concern as well. For example, the user may choose to subject all buffer accesses to the conviction procedure.

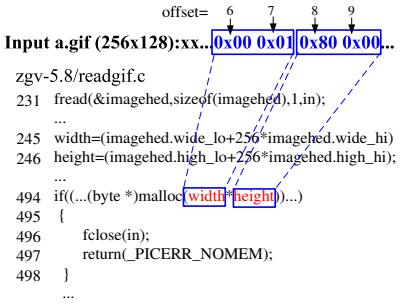

Figure 2. An Illustrative Example.

Next we use a real example to demonstrate the working of our system. Figure 2 shows one of the integer overflow vulnerabilities in CVE-2004-0994. By providing a malicious `gif` file header, remote attackers can exploit the integer overflow at line 494 and eventually launch a heap overflow attack.

In our system, suppose static analysis tools are able to point out that there is an overflow suspect at line 494. Note that many static tools can generate such warnings. Now given a benign test input (in this case, any `gif` file input touches line 494), a normal `gif` image with the size of 256×128, our system traces the lineage of the execution of line 494 and identifies that the value of `width` comes from "`0x00 0x01`", and the value of `height` comes from "`0x80 0x00`", as shown in the figure. Our mutation algorithm eventually finds that replacing these input values with large numbers triggers this integer overflow vulnerability. Such a mutated input is provided as the evidence for the conviction.

In order to realize the idea, we have to overcome several technical challenges such as scalable lineage tracing, input mutation, and test generation to cover suspects. In the next two sections, we will present our solutions to these issues.

3 Data Lineage Tracing

The first problem that confronts us is to identify the set of input values that are relevant to a particular execution point. Although one can say that all the inputs are related to each point of execution in general, we observe that given a particular execution point, part of the inputs are much more *closely related* than others. A more formal definition of "closely related" will be given in later discussion, but intuitive examples can be found in Figure 2. As we can see from this figure, the binary strings in the rectangles have one-one mappings to the values at line 494. The code excerpt clearly explains how the input values are propagated to line 494[1].

Next, let us formalize our definition of data lineage. The definition is based on the concept of data dependence in the field of program slicing [23]. Given a program execution E, s_i denotes the ith execution instance of a statement s. Note that a statement can be executed multiple times in one execution.

Definition 1 *A statement execution instance s_i* **data depends** *on another statement execution instance t_j if and only if a variable is defined at t_j and then used at s_i.*

For example in Figure 2, assuming all the statements execute only once, 494_1, the first instance of statement 494, data depends on 245_1 and 246_1.

Definition 2 *The* **data lineage** *of a variable v at an execution point of s_i, denoted as $DL(v@s_i)$, is the set of input bytes that are directly or indirectly involved in computation of the value of v at s_i through data dependence.*

In some places of this paper, we also use $DL(s_i)$ to denote the data lineage of the statement instance s_i. For example, $DL(\texttt{width}@494_1) = \{6, 7\}$, with the numbers denoting the values' indices in the input sequence[2]. $DL(494_1) = \{6, 7, 8, 9\}$. One may raise the question whether control dependence [22] needs to be considered. Our experience shows that simply including control dependence in lineage computation often leads to undesirably oversized lineage sets. Therefore we consider control dependence in the search procedure for mutation (Section 4) instead of in lineage computation. In practice, our strategy is sufficient for the cases we studied.

Given the definition, we develop a run-time algorithm to compute data lineage. The basic rule is that *the set of input elements relevant to a statement instance s_i is the union of the relevant input sets of all the statement instances which*

[1] Note that although we present the example in its source code form for readability, our analysis works directly on binary.

[2] We store the input sequence into a global buffer so that input values can be indexed and accessed.

s_i *data depends on.* In other words, all the input values that are relevant to some operand of s_i are considered as relevant to s_i as well.

For the simplicity of explanation, let

$$s_i : \texttt{def} = f(\texttt{use}_0, \texttt{use}_1, ..., \texttt{use}_n)$$

be an executed statement instance, in which s_i defines variable `def` by using the variables of $\texttt{use}_0$,$\texttt{use}_1$, ..., and $\texttt{use}_n$.

For example, the statement instance 245_1 can be represented as

$$245_1 : \texttt{width} = f(\texttt{imagehed.wide\_lo}, \texttt{imagehed.wide\_hi})$$

Let $DEF(x)$ be the latest statement instance that defines variable x. The computation of data lineage can be represented by the following equations:

$$DL(s_i) = DL(\texttt{def}@s_i)$$

$$DL(\texttt{def}@s_i) = \begin{cases} get\_new\_id() \\ \qquad \texttt{if } def \texttt{ is an input value;} \\ (\bigcup_{\forall x} DL(\texttt{use}_x@s_i) \\ = (\bigcup_{\forall x. DEF(\texttt{use}_x) \neq \phi} DL(\texttt{use}_x@DEF(\texttt{use}_x)) \\ \qquad \texttt{otherwise.} \end{cases} \quad (1)$$

As shown by the equations, the lineage of s_i is equivalent to the lineage of the variable `def` defined at s_i. If `def` is considered as an input, function $get\_new\_id()$ is called to assign a unique id for the input instance. If def does not represent an input value, its lineage is computed as the union of the lineage sets of $\texttt{use}_x$s. If a variable $\texttt{use}_x$ was previously defined,
$DL(\texttt{use}_x@s_i) = DL(\texttt{use}_x@DEF(\texttt{use}_x))$.
Otherwise, it is treated as having an empty lineage set, corresponding to statically initialized variables.

Identifying Input Values. It is non-trivial to label input values. In EXE [9], users are required to annotate input variables. We had considered such a strategy. However, since we are working at binary level and we handle whole system inputs such as those read from files or network packets, we found that it was hard to adopt. We took a different path by intercepting input relevant system calls such as system reads and assign unique ids to each input value. More precisely, after each system read, we scan the input buffer, and assign unique ids to each byte in the input buffer. Such an id serves as the lineage for that input byte. The `fseek`-like operations for local file read were challenges for us because a single byte may be read multiple times due to this kind of operations and we have to avoid generating multiple ids for the same byte. Our solution is to intercept other system calls besides reads such as `lseek` to synchronize the state of cursors between input files and our id labeling. For network packet, it does not have such issues since every single byte are sequentially received/processed upon entering the system. We should note there exist some special cases of user-input which cannot be caught via system calls, e.g., the command line option inputs (i.e., `argv`), for which we label all data from the bottom of stack to the frame just before `main` with ids upon entering function `main`.

An Example Of Lineage Tracing. We use the example in Figure 2 to illustrate lineage computation. The procedure is presented in Table 1. The first column presents the control flow trace. To disclose the complete computation, we extend the excerpt in Figure 2 to include some code in library, labeled with `pc1` and `pc2`. Inside the function call `fread`, system call `READ` is first issued to load in the gif file to `buf` with the input length of `size`. The values in `buf` is then copied to the structure `imagehed`.

In Table 1, the column labeled `def` indicates the variables that are defined at the statement instance. Columns $\texttt{use}_x$ and $\texttt{DEF}\ (\texttt{use}_x)$ represent the variables used and the previous statement instances that define these variables, respectively. The last column shows the data lineage. According to Equation 1, after the system call `READ`, each byte is assigned a unique id at $pc1_1$. Then, at $pc2_{7,8,9,10}$, the lineages of corresponding bytes are propagated to variables `wide_hi, wide_lo, high_hi, high_lo`. Note that `*p` points to these variables at the various instances of $pc2$. At 245_1, `wide_hi` and `wide_lo` are used to define `width`, according to the equation, the lineage of `width` at 245_1 is the union of the lineages of `wide_hi` and `wide_lo`. Eventually, at 494_1, we acquire the exact lineage as demonstrated earlier in Figure 2.

Efficient Lineage Representation. Compared with existing techniques with similar functions such as TaintCheck [15], in which one bit is required for one byte, we are facing a much harder space problem because we are computing a set for each byte, which potentially has the same cardinality of the entire input set. Moreover, set operations are performed at each step of execution. Therefore, an efficient set representation is critical to the system performance. A naive link-list based implementation may be devastating. For example, sets with thousands of elements may have to be traversed for the execution of a single instruction. Fortunately, recent research on dynamic slicing [24] reveals that *reduced ordered Binary Decision Diagram* (roBDD) [4] can be used to achieve both space and time efficiency in representing sets, especially when these sets have the characteristics of *overlapping, clustering,* and *reappearing*. Data lineage possesses exactly these characteristics. For example, the execution of statement "`y=x+1`" gives rise to reappearing lineages because both `x` and `y` have the same lineage. A statement like "`z=y+x`" introduces significantly amount of overlap between the lineages of `x, y` and `z`, due to the union operation. The detailed study of these properties is not the focus of this paper.

Table 1. Computation of Data Lineage

s_i		def	use_0	$DEF(use_0)$	use_1	$DEF(use_1)$	$DL(def@s_i)/DL(s_i)$
231_1	`fread(...)`						
$pc1_1^*$	READ (buf,size,...)	$\forall 0 \leq i < size$ buf[i]					$\forall 0 \leq i < size\ DL(buf[i]@pc1_1) = get\_new\_id()^{***}$
$pc2_7^{**}$	`*p = buf[i]`	wide_lo	buf[6]	$pc1_1$			$DL(*p@pc2_7) = DL(buf[6]@pc1_1) = \{6\}$
$pc2_8$	`*p = buf[i]`	wide_hi	buf[7]	$pc1_1$			$DL(*p@pc2_8) = DL(buf[7]@pc1_1) = \{7\}$
$pc2_9$	`*p = buf[i]`	high_lo	buf[8]	$pc1_1$			$DL(*p@pc2_9) = DL(buf[8]@pc1_1) = \{8\}$
$pc2_{10}$	`*p = buf[i]`	high_hi	buf[9]	$pc1_1$			$DL(*p@pc2_{10}) = DL(buf[9]@pc1_1) = \{9\}$
245_1	`width=...`	width	wide_hi	$pc2_8$	wide_lo	$pc2_7$	$DL(width@245_1) = DL(wide\_hi@pc2_8) \cup DL(wide\_lo@pc2_7) = \{6,7\}$
246_1	`height=...`	height	high_hi	$pc2_{10}$	high_lo	$pc2_9$	$DL(height@246_1) = DL(high\_hi@pc2_{10}) \cup DL(high\_lo@pc2_9) = \{8,9\}$
494_1	`...width*height...`		width	245_1	height	246_1	$DL(494_1) = DL(width@245_1) \cup DL(height@246_1) = \{6,7,8,9\}$

* pc1, pc2 are statements in *libc* functions.
** the input byte with offset 7, with the value "0x00", is loaded to buf[6] by the 6th instance of pc2
*** since the input sequence starts with buf[0] and the id assignment starts at 0, $DL(buf[i]@pc1_1) \equiv i$.

As roBDD is capable of efficiently representing the power set domain of a universal set (here the universal set is the set of input values), it benefits us in the following respects. First, each unique lineage set is indexed by a unique integer in roBDD. In other words, two sets are represented by the same integer number if and only if they are identical. This is critical to our system, because instead of storing a set for each byte in memory to represent its lineage, we only need to store an integer. Furthermore, performing the equivalence test on two sets can be achieved in $O(1)$ time by comparing the corresponding integers. Second, roBDD also promises time efficiency because set operations can be translated into roBDD operations. For instance, binary operations (e.g., union) of two sets whose roBDD representations contain n and m roBDD nodes can be performed in time $O(n \times m)$ [5]. Note that the number of roBDD nodes is often much smaller than the number of elements in the represented set.

Binary Instrumentation. In order to trace lineage, we have to instrument the binary of the program such that lineage information is updated during program execution. According to Equation 1, we need to update the DL set of the left hand side variable at every step of the execution and store it somewhere. In our system, we use *shadow space* to store lineage sets. Specifically, if the variable is stored at a specific stack/heap location, a corresponding *shadow memory (SM)* is allocated and used to store the set associated with the variable. Similarly, we use the *shadow register file (SRF)* to store the sets for variables in registers. Both shadow memory and shadow registers are implemented by software.

4 Input Mutation

The lineage tracing component collects runtime information about the random generated input (benign input). This information is used to direct the other key component of our system, the input mutator, to generate an exploit. In this paper, an exploit refers to an input that leads to unsafe

Algorithm 1 *Input Mutation*

```
1: Driver (s, T, SCD) /* s:suspect, T:benign input, SCD: static control dependence*/
2: {
3:     T_x = T;
4:     if (s is not executed with input T)
5:         T_x = DirectedTGen (s,T, SCD);
6:     if (T_x) {
7:         s_i = the last execution instance of s;
8:         return Search(s,s_i,T,SCD);
9:     }
10:    return NULL;
11: }
12: DirectedTGen (s, T,SCD)
13: {
14:    DL = TraceDL(T); /* TraceDL () is the lineage tracing procedure*/
15:    if (s is not executed with input T) {
16:        wl = {s}; /*wl is a worklist*/
17:        while (t = wl.removeNext()) {
18:            for each p ∈ SCD(t), p is executed with T) {
19:                T' = MMutate(p_1, t, T, DL);
20:                if (T' and T_x=DirectedTGen(s,T',SCD))
21:                    return T_x;
22:            }
23:            wl.add(SCD(t));
24:        }
25:    } else return T;
26:    return NULL;
27: }
28: Search (s, t_i, T, SCD) /*s:suspect, t_i:the execution point to start search*/
29: {
30:    DL = TraceDL(T);
31:    if (DL(t_i) ≠ φ)
32:        if (T'=Mutate (s, DL(t_i), T))
33:            return T';
34:    /* Facilitated by SCD*/
35:    p_j = a predicate instance that controls the definition of t_i;
36:    if (T'=Search (s, p_j, T,SCD)) return T';
37:    return NULL;
38: }
39: Mutate (s, DL, T)/* s:suspect, DL:the lineage relevant to the suspect */
40: {
41:    /*Heuristic One: change input values*/
42:    for v in {MAXINT, '%n', 0, ...} {
43:        T'= replace DL part in T with v;
44:        if (AttackDetected (s, T')) return T';
45:    }
46:    /*Heuristic Two: change input lengths*/
47:    X = DL; Threshold = 0;
48:    while ( Threshold++ < 16) {
49:        X = X · X;
50:        T'= replace the DL part in T with X;
51:        if (AttackDetected(s, T')) return T';
52:    }
53:    /*More heuristics*/
54: }
```

memory writes; gaining control of the host program through these unsafe writes is beyond the scope of this paper.

The overall procedure is illustrated by the algorithms presented in Algorithm 1. Method `Driver` (line 1-11) serves as the driver. It checks if the program execution with the benign input T covers the suspect s. SCD contains the static control dependence information, which is precomputed from the binary. Readers who are interested in computing static control dependence are referred to [22]. The implementation of our SCD component is discussed in Section 5. If s is not covered by the benign execution, the driver calls the method `DirectedTGen` (line 2-27), which is a directed input generation procedure that produces a T_x to cover s. More details about the `DirectedTGen` procedure will be disclosed at the end of this section.

Now, let us focus on the `Search` (line 28-38) method and the `Mutate` (line 39-54) method. These two methods aim to mutate the benign input that covers the suspect s to generate an exploit. If they fail to produce one, our system considers s innocent.

Given the suspect s and its last execution instance s_i in the benign execution with input T, the `Search` method is called in the `Driver` function to look for the lineage that is relevant to s_i and then automatically mutate it to generate an exploit. The `Search` method first checks if the current search point t_i has a non-empty lineage. If so, it calls `Mutate` to change the $DL(t_i)$ part of input T. If not, or the mutation is not successful, meaning the suspicious vulnerability is not triggered, the search procedure looks for the predicate instance p_j that controls the definition of t_i's value. Line 35 makes use of SCD information and looks for the control dependence of the definition of t_i. Essentially, the search algorithm takes into account the effect of control dependence without incurring oversized lineage sets.

The `Mutate` method takes the benign test input T and the lineage DL that is found by the `Search` method, and tries to mutate T by replacing the DL with something else. The mutation method applies several heuristics in changing T, as shown in line 41-53, the first heuristic is to change the integer values in DL to the maximum unsigned integer (MAXUINT, i.e., `0xffffffff`), several other boundary values such as $0, -10, -100, -1000, 1, 10, 1000$, and '%n', etc. Changing to a boundary integer is to exploit integer overflow or integer value related vulnerabilities; changing to '%n' is to trigger format-string vulnerability. The second heuristic is to change the size of input, with the goal of triggering buffer overflows. This is done by duplicating DL in T after each iteration till a threshold is hit (we set the threshold as 16 since we seldom find the buffer length is greater than 2^{16}). In other words, this heuristic replaces DL in T with $DL \cdot DL$, $DL \cdot DL \cdot DL \cdot DL$, and so on. Users also have the freedom to insert their own heuristics. We observe that simple heuristics turn out to be highly effective in producing exploits.

Directed Input Generation. If the benign input T does not cover the suspect s, the `Driver` function calls `DirectedTGen` to generate a new program input to cover s. Starting from s, the input generation procedure transitively searches along s's static control dependencies until it sees a direct/indirect control dependence p that has been executed with input T, then it tries to mutate the lineage of the first execution instance p_1 such that program execution with the new input takes the edge $p \rightarrow t$, which leads to s. In other words, the new execution is one step closer to s. The procedure repeats until s gets covered. Compared with the state of the art of test generation techniques, our input generation is more directed, meaning that we only try to cover a specific program point instead of all feasible program paths. Another difference is that our technique is facilitated by data lineage.

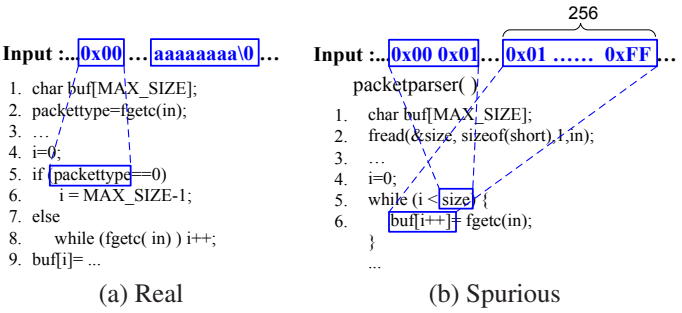

Figure 3. Correlations

Currently, this procedure involves user interactions, namely, the function `MMutate` requires the user to inspect predicate conditions to construct a replacement of the relevant lineage. Compared with the automated `Mutate`, the possible mutations in `MMutate` (line 19) are not bounded by the types of considered vunerabilities. While source code access would be beneficial to `MMutate`, our experience shows that the desired mutation can also be inferred from the binary predicate instruction and its lineage, as demonstrated by the evaluation case in Section 5.1.3.

Correlated Inputs. So far, we have presented our technique on trying to mutate a test case by replacing one lineage. However, if the inputs have correlations, multiple lineages may have to be mutated in order to successfully trigger a vulnerability. Consider the example in Figure 3(a). The suspect is at statement 9. Since variable `packettype` is set to 0 according to the lineage, variable `i` is set to `MAX_SIZE-1`. If our system only tries to mutate DL(`packettype`) to generate the exploit, it would fail because the lineage of `fgetc(in)`, as shown inside the second rectangle, needs to change simultaneously in order to trigger the attack. This is due to input correlation. To handle such correlated input mutations, a potential solution is to mutate the input in multiple rounds, namely, recursively

search for mutations of mutated inputs.

Also sometimes, suspicious input correlations turned out to be spurious. Consider the example in Figure 3(b), which presents a `packetparser` function. Since the size of the packet body, delimited by the second rectangle, is decided by a value in the packet header, delimited by the first rectangle. Although our system fails to generate an exploit by mutating the lineage of `buf[i++]`, it successfully generates an exploit by solely changing the lineage of `size` because it results in some bytes that were tailing the `buf` are now treated as part of the `buf`. We call this type of correlation *spurious correlation*.

Discussion About Completeness. Our technique is not complete, meaning we have false negatives. The reason is multi-faceted. For instance, our directed input generation may fail to generate an input to cover a suspect. There may exist real input correlations which fail our test mutation procedure. Our mutation procedure `Mutate` is heuristic-based rather than exhaustive. As a result, we cannot conclude a suspect for which our system cannot generate an exploit to be surely innocent. However, we argue that the benefit of convicting some real vulnerabilities with evidence pays off the loss of completeness.

5 Implementation and Evaluation

We have implemented the whole system in Linux. The data lineage tracer module is built on top of Valgrind-2.2 [6] with roBDD [4] support. We instrument data movement (e.g., `LOAD`, `STORE`, `MOV`), arithmetic operation, and logic operation instructions (e.g., `ADD`, `SUB`, `AND`) to keep track of data dependence and generate lineage information. We implement the SCD computation module on top of Diablo-0.3 [3], a retargetable link-time binary rewriting framework which has the capability of constructing the control flow graph of an x86 binary. To be more precise, we implement an post-dominance analysis which facilitates computation of static control dependence for a given function.

To verify the effectiveness and efficiency of our system, we have conducted a number of experiments. The types of vulnerability we studied include a wide range of possible exploitable ones, i.e., stack overflow, heap overflow, format string, and integer overflow. The benchmark programs and their related vulnerabilities are described in Table 2. All the experiments were performed on a machine with two 2.13Ghz Pentium processors and 2G RAM running the Linux kernel 2.6.15, and the vulnerable programs were compiled with `gcc` 3.2.2 (because of some compatibility issues when compiling some old programs), and linked with `glibc` 2.2.

5.1 Effectiveness

Since our major contribution is on the dynamic analysis, the static frontend is not the focus of the evaluation. Thus, in this subsection, we take on a perfect frontend by using the real vulnerabilities reported by CVE and generate exploits for them. Our experience on generating evidence for new (never-reported) vulnerabilities are reported in a later subsection.

Table 2. Description of the Benchmarks

CVE#	Program	#LOC	Vul. Description	Convicted?
CVE-2001-1413	ncompress-4.2.4	1.9K	Stack overflow	√
CVE-2001-1228	gzip-1.2.4	8.2K	Stack overflow	√
CVE-2002-1549	bftpd-1.0.11	1.1K	Stack overflow	√
CVE-2002-1496	nullhttpd-0.5.0	2.5K	Heap overflow	√
CVE-2000-0573	wu-ftpd-2.6.0	23.7K	Format string	√
CVE-2001-0609	cfingerd-1.4.3	5.1K	Format string	√
CVE-2005-0226	ngircd-0.8.2	16.4K	Format string	√
CVE-2004-0904	zgv-5.8	25.4K	Integer overflow	√
CVE-2006-3082	gnuPG-1.4.3	79.3K	Integer overflow	√

In our experimentation, we have successfully generated exploits to trigger all the vulnerabilities shown in Table 2. Indeed we have tried a number of other reported vulnerabilities as well and the result was consistent. Due to the space limit, we are not going to present all the results we have.

5.1.1 Buffer Overflow

`Ncompress` is a utility handling compression and decompression of Lempel-Ziv archives. The code in function `comprexx` of `ncompress-4.2.4` does not properly check bounds on user-supplied input, and thus contains a stack buffer overflow vulnerability. For this benchmark, we started with a benign input (a command line option) with a filename of 4 bytes; our mutator found the lineage is not empty at the buffer access in function `strcpy` which is at line 893 in file `compress42.c`; then it doubled the inputs in every re-execution, and after repeating this process for 10 times, the program was successfully crashed because the buffer size of 1024 was exceeded. The vulnerabilities of `gzip` and `bftpd` are very similar to `ncompress`, and it took our mutator 10 and 4 re-executions, respectively, to generate the exploit. The vulnerability of `nullhttpd-0.5.0`, a multi-threaded web server, is a heap overflow. Three re-executions were required.

Here, we did not encounter any path coverage issue for these 4 buffer overflow tests since all the vulnerable statements are on some common program path. Our experience with other buffer overflow vulnerabilities also certify this observation. This could be due to the nature of buffer overflow vulnerabilities. Another explanation is that attackers/testers rely on existing test cases to study vulnerabilities, just like us, so that only those covered by the provided test cases are reported.

5.1.2 Format String

The root cause of format-string vulnerability lies in the format string, which is an argument to a function in the

printf family, (partly) comes from user input. The mutation heuristics are to change values in the lineage set of the format string to %n (%s also works), which typically leads to a segmentation fault as %n/%s entails a memory write to a random memory location specified by the corresponding parameter (a random value here); if a crash does not occur, we double the length of the %n input subsequence, eventually resulting in an observable segmentation fault if the format string vulnerability is real.

We have applied the above format string evidence generation scheme to three real world applications, i.e., cfingerd-1.4.3, wu-ftpd-2.6.0, and ngircd-0.8.2. Due to the limited space, we only describe how we caught the format string vulnerability in cfingerd-1.4.3. The vulnerable code is at line 245 of file main.c, where syslog directly uses syslog_str as the format string argument, and part of the argument is user-supplied input (e.g., username). In our experiment, we started with a benign username (6 bytes long), and our mutator found syslog called by main contains a non-empty lineage. Then it directly changed all these 6 characters to %n, and consequently, a segmentation fault occurred in syslog. For this benchmark, we only re-executed the program once and successfully generated the exploit. For wu-ftpd and ngircd, our mutator also only used one re-execution based on the benign input.

5.1.3 Integer Bugs

There are four types of integer bugs: overflow, underflow, signedness error, and truncation. Here we focus on integer overflow, meaning the result of an integer expression exceeds the maximum value regarding its type. The other 3 types are similar. The benchmark we used is gnupg-1.0.5, which contains an integer overflow and eventually will cause a heap out-of-bound access. The vulnerable code is shown in Figure 4.

```
397     switch( pkttype ) {
        ...
422     case PKT_USER_ID:          /* PKT_USER_ID = 13 */
423        rc = parse_user_id(inp, pkttype, pktlen, pkt );
        ...
1580  static int
1581  parse_user_id( IOBUF inp, int pkttype,
        unsigned long pktlen, PACKET *packet )
1582  {
1583     byte *p;
1584
1585     packet->pkt.user_id = m_alloc
           (sizeof *packet->pkt.user_id  + pktlen);
        ...
1595     p = packet->pkt.user_id->name;
1596     for( ; pktlen; pktlen--, p++ )
1597         *p = iobuf_get_noeof(inp);
1598     *p = 0;
1599     ...
```

Figure 4. Parse-packet.c of Gnupg-1.0.5

The integer overflow suspect is the expression of m_alloc. We started with a benign input. Unfortunately, the benign input did not lead to execution of parse_user_id, we have to first mutate the benign input such that the desired path is covered. In this case, our diablo based SCD module is called, and it disassembles the binary code of gnupg, and generates an interprocedural static control flow graphThe total time of such a procedure is 238 seconds in our experiment. We display part of the graph in Figure 5. A box node represents a basic block and the instructions of this basic block are displayed inside the box. For *better illustration*, we annotate the graph with the corresponding source code. As we can see, the initial input drives program execution along the path from 0x805d1de to 0x805d1e5 while the desired program point is at line 1597. According to our SCD computation, line 1597 is transitively statically control dependent on the call site to function parse_user_id (0x805d386), which in turn is statically control dependent on the switch statement at line 397 (0x805d1de). This point was executed by our initial input. The lineage of pkttype(%ecx) at this point contains the input value of 6. We manually inspected the path condition and concluded that changing the lineage value to 13 leads to the suspect.

With the mutated benign input that drives the execution to 1597, the mutator found the lineage at 1597 is not empty due to the data dependences between 1597 and 1585, whose lineage contains an input value at pktlen. The value was changed to MAXUINT by our mutator. It caused m_alloc to allocate a buffer with 35 bytes (sizeof *packet->pkt.user_id =36). The number of assignments at 1597, decided by pktlen, exceeded the buffer and resulted in a crash.

Our another integer overflow case study was on zgv-5.8. This case has been explained earlier in Section 2 and will not be repeated here.

5.2 Experience With New Vulnerabilities

So far we have assumed a perfect frontend that only points us to suspects that are guilty. Next we present our experience of connecting our system to a real static vulnerability detection tool called RATS [2], which can detect buffer overflow and even integer overflow with user extensions.

We applied our system to a few most-recent software versions. The first one we tried was ipgrab-0.9.9. RATS reported 106 buffer overflow suspects. We tried to convict these suspects one by one using our system. We found that the 48th suspect is a real one. The vulnerability, which is presented in Figure 6, lies at line 357 in file.c. It is a buffer overflow caused by an integer overflow. To begin with, we used a random generated benign input. The input was not hard to acquire because any input packet will touch the suspect. The mutator altered the lineage of header.inc_len to MAXUINT, and it caused a segmentation fault at line 357 because the parameter to malloc is 0 while fread tries to read MAXUINT bytes. Thus, through one round of mutation, we proved the existence of this vulnerability. Using the same methodology, we have found and proved another two new integer overflows caus-

Table 3. Performance and Space Overhead of Lineage Tracing

Program	Metrics	Performance (seconds)					Space (bytes)		
		Normal	W/O Log	Ratio	With Log	Ratio	Link-list	Bdd	Ratio
ncompress-4.2.4	Time to compress a 4.3k bytes file	0.001	1.740	1740	1.960	1960	4296576	460700	9.33
gzip-1.2.4	Time to compress a 15.7k bytes file	0.004	2.700	675	9.645	2411	3163228	1086220	2.91
bftpd-1.0.11	Response time of an automated user authentication	0.014	0.302	21	0.318	23	6808	4160	1.63
nullhttpd-0.5.0	Response time of processing a 512 bytes post packet	0.007	0.452	65	0.463	66	120736	21980	5.49
wu-ftpd-2.6.0	Response time of an automated user authentication	0.018	0.486	27	0.526	29	11496	6340	1.81
cfingerd-1.4.3	Response time of a normal lookup request	0.015	0.517	34	0.543	36	39508	10820	3.65
ngircd-0.8.2	Response time of an automated 5 sequence `irc` commands	0.021	0.342	16	0.378	18	33032	16060	2.07
zgv-5.8	Time to display a 1.4k bytes malicious gif file	0.011	20.909	1901	21.965	1997	971328	14000	69.38
gnuPG-1.4.3	Time to verify a 1.2k bytes signature file	0.012	29.298	2441	86.926	7243	2898128	279160	10.38

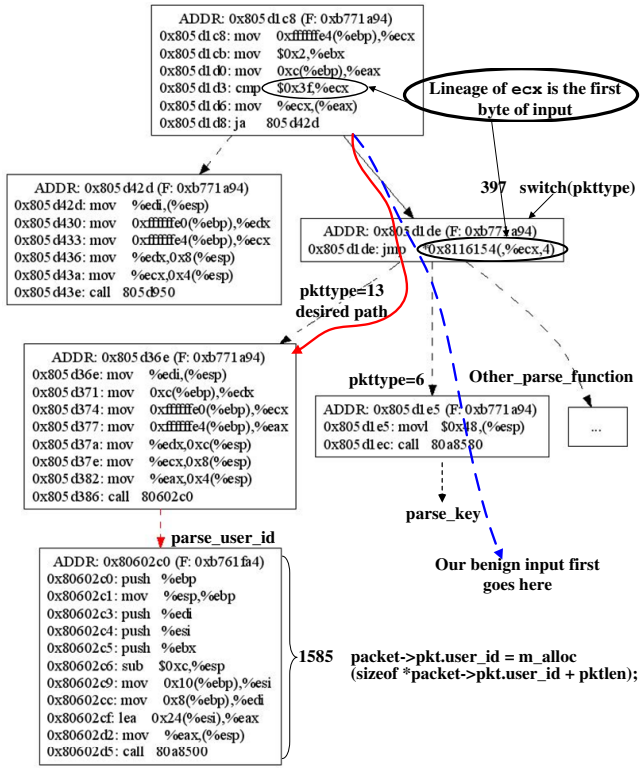

Figure 5. Part of the CFG of gnupg.

ing buffer overflow vulnerabilities in `dcraw-7.94`, and `epstool-3.3`. We have reported these vulnerabilities with evidence to their developers. They replied promptly, admitting the existence of these defects.

```
334 while(1)
335     {
336         /* Read the header */
337         ret = fread((void *) &header,
338                     sizeof(snoop_packet_header_t),1,fp);
            ...
355         /* Get the actual packet */
356         packet = my_malloc(header.inc_len+1);
357         ret = fread((void *)packet,header.inc_len,1,fp);
358         ...
```

Figure 6. File.c of lpgrab-0.9.9

5.3 Performance and Space Overhead

We also used the above 9 benchmark programs to measure performance and space overhead of the lineage tracing module, which is the performance dominator in our system. With respect to performance, we measure three scenarios, without lineage tracing, with lineage tracing but without logging, and with both lineage tracing and logging. For the daemon programs, we indirectly measure their performance by measuring their response time, and for the utility applications, we directly measure the running times. The setup and the result are presented in Table 3.

Without logging, the performance slow down factor varies from 16 to 2441. If logging is enabled, its performance overhead varies from 23 to 7243 times. The large overhead factors for utility programs are mainly due to the fact that the total running times of these programs include the starting and ending times of the `Valgrind` engine, which is significant compared with the real execution time. The numbers for daemon programs, ranging from 16 to 66, are closer to the real slowdown since we excluded the time spent on `Valgrind` by inserting performance monitors to the programs. Note that network latency is not an issue here because we were using the local network interface. We believe the performance has greatly benefited from using roBDD. One can easily imagine the overhead of performing set operations on up to a few thousands elements during each step of execution. Another observation is that if the application is data-intensive (e.g., `gnuPG`), the log file is very large (nearly 10M in this case), causing a lot of runtime overhead. Due to some historical reason, we used an old version of `Valgrind`, which incurs ten times slowdown even without any instrumentation. Furthermore, we have not strived to optimize the system because performance is not yet a critical factor for us.

For the space overhead, as illustrated in Table 3, we can see that a link-list based approach will cost much more space than our roBDD based approach, especially for data-intensive applications.

6 Related Work

In recent years, there have been significant advance in automated code based test generation [9, 10, 11, 13]. Theoretically, these techniques can be applied to our problem of automated evidence generation. However in practice, they have inherent limitations that constrain their application. First, most these techniques are tuned to unit testing

due to the scalability issue. Second, these techniques work by combining concrete execution with constraint solving to explore all potential program paths. Third, solving symbolic constraints requires the user to specify symbolic variables in the source code, demanding not only the access to the source code but also a certain level of understanding.

In contrast, our technique is a light-weight whole program technique that explore a subset of program paths. It does not require source code access and it does not require understanding the program in most cases.

TaintCheck [15] represents another type of dynamic techniques that are relevant. Our technique can be considered as a generalization of TaintCheck. More specifically, TaintCheck uses one bit to color program execution as input-relevant or input-irrelevant. By contrast, we "taint" each program execution point with a set of relevant input values. Our scenario is more challenging because sets may have various numbers of elements, with the upper bound of the universal set of inputs. Furthermore, TaintCheck is proposed as an online technique to detect attacks on the fly. Reducing runtime overhead is its major concern. This is also true for other dynamic techniques such as control flow integrity checking [16] and data flow integrity checking [17]. Our technique aims to generate evidence off-line.

7 Conclusions

In this paper, we propose a data lineage tracing based dynamic approach to generate evidence for remote exploitable vulnerabilities in software. More specifically, it associates an execution point suspect with a set of input bytes, whose values are then mutated offline to generate an exploit (evidence). Our approach delivers both efficiency and effectiveness. Using our system, we are able to reproduce exploits for all the known vulnerabilities we studied. We also successfully identified and convicted a number of new vulnerabilities, which were all promptly confirmed by the developers. Our evaluation also shows that the system has reasonable overhead for the scenario of offline diagnosis.

Acknowledgments
We thank the anonymous reviewers for their detailed, helpful comments. This work is supported in part by NSF grants CNS-0720516, CNS-0708464 and CNS-0716444.

References

[1] http://www.dwheeler.com/flawfinder/

[2] http://www.fortifysoftware.com/security-resources/rats.jsp

[3] http://diablo.elis.ugent.be

[4] Buddy, a binary decision diagram package. Department of Information Technology, Technical Univ. of Denmark.

[5] C. Meinel and T. Theobald. Algorithms and data structures in vlsi design, 1998. Springer.

[6] Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. N. Nethercote and J. Seward. In *Proc. of ACM PLDI*, June 2007.

[7] B.P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM* 33, 12, Dec. 1990.

[8] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proc. of the 4th USENIX Windows System Symposium*, 2000.

[9] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. Exe: automatically generating inputs of death. In *Proc. of ACM CCS*, Nov. 2006.

[10] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proc. of ACM ESEC/FSE-13*, 2005.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. of ACM PLDI*, 2005.

[12] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. of 2007 IEEE Symposium on Security and Privacy*, May 2007.

[13] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proc. of NDSS*, San Deigo, CA, Feb. 2008.

[14] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proc. of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[15] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.. In *Proc. of NDSS*, Feb. 2005.

[16] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proc of ACM CCS*, Nov. 2005.

[17] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proc. of OSDI*, Nov. 2006.

[18] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of NDSS*, Feb. 2000.

[19] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proc. of USENIX Security*, 2001.

[20] M. Zitser, D. Shaw, T. Leek and R. Lippman. Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code. In *Proc. of ACM ESEC/FSE-11*, 2004.

[21] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proc. of ACM ESEC/FSE-10*, 2003.

[22] J. Ferrante, K. Ottenstein, J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3),1987.

[23] M. Weiser. Program slicing. In *Proc. of ICSE*, 1981.

[24] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proc. of ICSE*, 2004.