

A Diff-based Data Tainting Method

Xin Li
xinli@ece.rochester.edu

1 Overview

Dynamic data tainting a means of locating accesses to certain data located in computer main storage or register space, which entails obtaining key characteristics of access patterns. Some typical uses of memory tracking can be seen in [3]. The most notable application of such taintings is preventing insecure data from being executed.

However, the conventional tainting methods are inherently limited. In this proposal we will take a look at a diff-based tainting method primarily in the context of memory error propagation. We will then discuss the possibility of applying the idea in the generic tainting scenarios.

2 The limitation of current taint policy

In taint analysis works, sensitive data are tainted to track their flow. The taint tracking sounds like an easy idea. Initially some locations of system memory or registers are marked as tainted, and when an instruction processes a tainted location, the output of the instruction would be marked as tainted too, such as this example, y becomes tainted if x is tainted:

```
int x, y;
...
y = x + 5;
```

The intuition behind is the causality of operations. Those tainted variable that determines the value of another variable results in taint propagation to it. The bulk of the research works (Qin-Micro06 [4], Ho-Eurosys06 [2] and Mysore-Asplos08 [3]) hand-waves about the model they use with examples just like what we have showed. However, in reality, tracking the dependencies of value change can be challenging if not daunting. Several of these problems are touched upon in [1]. That paper also provides partial solutions for some of these problems. Here we will discuss these in a more thorough fashion.

2.1 Pointer De-reference

The first point not covered by the bulk of papers is, if a pointer is tainted, what happens if that pointer is used to access a memory location? If the tracking stops at that point and the retrieved value is not marked tainted, then you might incur quite a few false negatives in table look-up like operations. Consider the following example.

```
int i;
int x, y;
int array[10];

for (i=0; i<10; i++)
    array[i] = i;
...
/* x was tainted in the ellipsis*/
```

```
y = array[x];
```

In this case, let's assume the variable x comes from an external source which will be marked tainted. Essentially the program looks up a table and determines the value of y . Here the program happens to be equivalent to a single $y = x$ statement, but the taint tag for y is dropped, which is counter-intuitive.

[1] gives the solution for this by tainting the value of the output whenever a tainted value is involved in the address computation of a source memory operand. However, it doesn't address the issue presented below. Again, consider an example.

```
uid_t buffer[N], uid;
int i, j;

...
/* put uid in a buffer place */
buffer[i] = uid;
...
/* j is supposed to be equal to i*/
targetuid = buffer[j];

setuid(targetuid);
```

Let's suppose *buffer* is a temporary place to hold user IDs. initially the array *buffer* is filled with 0's. Further suppose i comes from a tainted source, and it is altered from the original value. The wrong *buffer* element receives the update *uid*. The wrong place may be tagged, but the original is not. When later the value at the original place is read out, it contains the value 0 and could later be used to assigns a process the uid of the root user's. And since the value is not tainted, the system is not warned against this behavior. The tainting is inherently limited in this type of cases, because the taint doesn't track the values. What it knows is just a tag of the data saying if it's tainted or not. This limitation may be acceptable in the context of that paper, which deals with tracking how sensitive info such as credit card number. In the context of memory errors, however, this scenario can be common.

2.2 Control flow

So far we have seen data flow about tainted value only. But what if a tainted data is used in a condition that dictates control flow? The following example is adapted from [1].

```
int x, y;
...
if (x == 1)
    y = 1;
else if (x == 2)
    y = 2;
...
else if (x == 255)
    y = 255;
```

Like what is pointed out in that paper, this is essentially equivalent to a table look up. [2] clearly states that taint is

not tracked across comparisons and arithmetic operations and therefore on a x86 architecture, it is impossible for them to track the flow in this case. [1] does nothing about this case. In fact, they admit that attacks that launder data like this do exist, and they don't want to protect the system from crafted malicious code of this sort. In this case the semantics is simple and clear. It is only straightforward to seek help from the compiler to identify the fork of executions and taint the data touched in the different branches. However, it would be much more confusing when the control flow takes complicated jumps. For example, what if a function that never returns is called? Or worse even, what if `setjmp` and `longjmp` are called?

3 Exploiting the nature of error tracking

3.1 Error propagation nature

In the context of memory error studies, the tainting differs on one extremely important aspect from the generic scenarios. When the taint is tracked for security reasons on suspected data from the Internet, the values read from the network are suspected to cause abnormal behavior of the system, but there is no "normal" behavior to compare against for the "abnormal". Therefore only very obvious attacks such as jumping to tainted code can be detected. In our scenario, when an error is injected into the memory, we have the knowledge what was the correct value. Therefore whether an error causes a new generation of error(s) largely depends on whether after some operations involving the erroneous value, the new value(s) produced would be DIFFERENT from those that would have been produced if the correct input value had been used. In other words, if we take an original clean image of a system, and inject some errors in some places, after some operations are executed, the new errors generated would be the diff across the two images at the end of the execution.

Knowing the difference between what is right and what is wrong is a powerful weapon. We will have better solutions to the problems presented in the previous section. For example, the constant function problem is no longer a problem. Simply comparing the outputs of a series of instructions for right and wrong inputs would suffice to see if those outputs should be tagged as "erroneous". In the following example,

```
int x, y;
...
/* x is tainted */
y = x & 0xff;
```

in general, taint analysis must mark `y` as tainted too. In our case, if the most significant bit of `x` is wrong, the value of `y` would stay the same, which would not mark `y` as "erroneous".

Following this line of thinking, the pointer problem completely disappears as well. We won't go over the details for the sake of space limit.

The control flow problem is somewhat troublesome. In simple cases as the one stated in the last section, a diff can be made before and after the branches on the system images, which is an effective way to convert control flow deviation to data deviation. The difference (the value of the variable `y` in our case) would be noted and marked. For the difficult cases like `setjmp/longjmp` pairs, our method can introduce a time-out mechanism could be used when the two execution paths are going along each other for too long, which is impossible for the static compiler methods. However, in general control flow is still a nasty beast which is hard to tame. Again, for the sake of space, we will not discuss this in details, and hopefully we will devote a few minutes of our demo to this issue.

3.2 Applying the method to generic tainting

The major obstacle that stands in our way to generalize the idea of diff-based tainting is that in these cases normally there is no notion of right and wrong, and thus there is no reference executions to compare against. However, we can perform value sampling on the tainted locations and see if different values can cause different execution results. In this case we will only cover the propagations probabilistically. However, our diff-based approach steps into the realm like pointer-based accesses where the conventional are completely inept.

References

- [1] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation (awarded best paper!). In *USENIX Security Symposium*, pages 321–336, 2004.
- [2] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. of the First European Systems Conference*, pages 29–41, Leuven, Belgium, Apr. 2006.
- [3] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. pages 211–221, Seattle, WA, Mar. 2008.
- [4] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of the 39th Int'l Symp. on Microarchitecture*, pages 135–148, 2006.