A Backsolving Technique Using Dynamic Taint Analysis

Clark Wood

Florida State University

clark.w.wood@gmail.com

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

1. Abstract

2. Introduction

Finding exploitable bugs in binaries is a difficult but valuable endeavour. Fuzzing, or the feeding of controlled input into processes to try and detect vulnerable conditions, is one technique used by researchers and analysts. Applications tested for vulnerabilities are often black boxes because source code is unavailable. This piles difficulties onto an already complex problem, of which much of current research attempts to solve or reduce. Among these issues are i) efficient and thorough dynamic taint analysis, ii) the path explosion problem [U], and iii) the code coverage problem. Dynamic taint analysis is also difficult because taint can propagate directly via assignment, or indirectly via affecting control flows which affect assignment [D], which make proper implementation tricky. The path explosion problem arises from the nature of assembly languages. Jump instructions, which may make up a decent percentage of machine instructions in programs, each multiply the total number of existing paths in a binary by two. Considering the size of binaries, this quickly leads to an un-tenably large set of possible paths to explore via conventional techniques like symbolic execution. And even if it is possible to explore all paths in a binary, analysts still need sample input which can trigger the vulnerable condition, perhaps proving that the binary is exploitable along the way. Such automated exploit generation is the subject of much research [H, I].

This work aims to reduce the severity of the latter two problems while being wary of the first during implementation. Instead of trying to attain more reasonable performance by using symbolic execution less or more wisely [J], I focus instead on approximating symbolic execution by using dynamic taint analysis to drive mutation. This is possible by tracking taint at the instruction level, then backsolving from the jump to the source of taint to determine smarter ways to mutate user input and create new test cases. For instance, in the figure 1 example:

```
figure 1:
call read
mov eax, [ebp - val]
inc eax
mul eax, 5
cmp eax, 42
jnz 0x8BADMEAT
```

If we already had a test case where the value read onto the stack, after the transformations, did not equal 42, then the cmp instruction would not set the zero flag, and we would jump to the memory location 0xBADMEAT. With dynamic taint analysis in place, we see taint introduced with the read syscall, then moved into the eax register (which is subsequently tainted). After some number of transformations, perhaps zero, to the value in the eax register, we see a compare instruction, which checks eax's value against an immediate value. Then we can keep track of specific taint sources, and can record any movements or transformations, we have all the necessary information to "solve" the branch. Once we reach the tainted branch, we know that user input can affect whether we jump or not, and consequently we can backsolve to find out how to affect the branch instruction. In particular, we know the conditional jump instruction jnz depends on the Z flag in the EFLAGS register. We record whatever instruction last modified each flag in EFLAGS, so we handle the compare instruction next. Because we have kept track of taint propagation, we know we can affect the value of eax, but not 42, since it is an immediate value, so we know we need to make the value in eax equal 42. We follow the instructions that have modified the value of eax and use any appropriate operands involved with the inverse operation of the one originally performed on the value to reverse the transformations to eax's value. Finally, we notice eax originally received taint when a value was moved from memory, which was read into memory using the read syscall.

This works similarly to a constraint solver, but constraint solving appears to be a traditional bottleneck, requiring generating symbolic input, usually in a different language like LLVM intermediate representation [V] and passing execution to the constraint solver.

I propose to avoid using a constraint solver entirely, and instead backsolve completely in memory. This is easier to implement than many of the clever optimizations used by S2E and could result in speedup over symbolic execution.

Because there are a finite number of conditional jumps, all of which depend on different flags or combinations of flags in the EFLAGS register, and a finite number of instructions that modify the EFLAGS register in x86, I can create cases for all possible ways jumps are decided. I only want to prove this is possible however, so I will only implement a subset and instead comment on how to implement the other comparisons.

3. Related Work

The major impetus for this work is outlined in "A Taint Based Approach for Smart Fuzzing". This paper proposes a fuzzing tool architecture, involving vulnerability detection, followed by taint analysis based upon found vulnerabilities, which drives the generation of intelligent tests. As tests are run they are checked to ensure they provide adequate code coverage, and results are carefully monitored for interesting situations like crashes [A]. The paper also serves as an excellent explanation of the current state and future of fuzzing. In particular, they suggest combining taint analysis with backward slicing. The technique discussed in this paper boils down to an attempt to automatically solve slices like those introduced in [AA], which have been derived from taint analysis as much as possible. This data is put to use by solving for the input state necessary to explore new blocks in the binary by taking the opposite path for a particular branch in the previous run.

Often, research leveraging taint analysis requires access to source code or even programming optimization. However, minimizing programmer workload is more in spirit with fuzzing. Fuzzers should be fire and forget, allowing expensive programmers to work on something else while testing runs in the background. Work such as [E] has kept the spirit of this, although they focus at a different granularity of taint collection and propagation. The authors argue for following taint at the function level instead of at the instruction level, primarily to improve performance. The paper proposes several performance-saving heuristics, such as statically constructing a finite state machine, which is executed with information gleaned during dynamic analysis, to help follow taint propagation within loops. However, perhaps because of the reliance on QEMU, or the initial translation of code to an intermediate representation without side-effects, significant overhead is introduced.

4. Assumptions and Scoping

I assume ASLR is disabled for the binary to be fuzzed. This allows the program to easily track and compare execution paths. I plan to implement the technique for 32-bit x86 assembly, because x86 is widespread, and vulnerabilities resulting from unsafe C code should persist in binaries, regardless of both the size and flavor of architecture (32 vs. 64-bit, ARM vs. x86 vs. PowerPC etc.), since compilers try to adhere to the C standard and vulnerabilities like buffer overflows exist at a C source code level.

I also assume the presence of an algorithm which will detect an exploitable condition, provided it is given the proper binary and input to the binary. The program will then attempt to concretely execute each path within the binary by backsolving, hopefully at a greater speed than symbolic execution.

Because current selective symbolic execution engines still appear to introduce significant overhead (Between 6 and 78X overhead more than QEMU for S2E. Just QEMU is between 4 and 10 times slower on some benchmarks, closer to 15 on others) [J, K, L]. In contrast, basic block counting using Intel Pin, the framework I am leveraging, introduced between 2 and 4X overhead [M]. However, the pintool needed for my proposed technique will introduce greater levels of overhead, since I will be working at the instruction level instead of the basic block level, requiring more instrumentations for a given binary.

I am leveraging Intel PIN to perform dynamic binary instrumentation. I am first testing dummy programs with many paths that are easy to backsolve, similar to the example above. A C program which uses sequential switch statements to check user input against a secret, when compiled with tcc, which performs very little optimization, is such an example binary. I then plan to test against S2E and Peach Fuzzer [Z].

In addition, there are allowable situations in programs which are outside the scope of this program. For instance, programs can loop forever [X]. These can be dealt with using reasonable heuristics and could be the subject of future research.

5. Techniques

Several valuable techniques to improve fuzzers' code coverage have evolved over the years. Two, in particular, inspire this work directly: Dynamic Taint Analysis and Concolic Execution. Dynamic Taint Analysis, or DTA, provides valuable information for guided fuzzing. Concolic execution, which has spawned out of the desire to use symbolic execution more wisely due to performance constraints, aims to ameliorate the path explosion problem while maintaining good code coverage. These each come with their own drawbacks, a primary one of which is the lack of automated instrumentation. Historically, symbolic execution required a programmer to directly instrument source code, but much research is now focused on automating as much as possible.

5.1 Dynamic Taint Analysis

DTA is the process of locating sources of taint and following their propagation through a binary to data sinks. Since, for exploitation, bugs must be triggered by user input, DTA usually identifies all user input (files, command line arguments, UI interactions, ...) as tainted, and marks it accordingly.

The granularity of taint marking is variable. Assigning everything from a generic binary value of tainted/not tainted to bit-level marking of which part of user input affected which variable is conceivable, although finer-grained taint analysis is more computationally expensive and difficult to determine.

Taint can propagate via explicit or implicit flows, sometimes also called data and control dependencies respectively [G]. Explicit flows involve a tainted variable, x, which is used in an assignment expression to compute a new variable, y. In this situation, x taints y, and if y is involved in any further assignment to a variable z, then x taints z by transitivity. In contrast, implicit data flows involve a tainted variable used to affect control flow within a program which subsequently sets the value of another variable, for instance at a branch [D].

Handling taint correctly is of great important because of the existence of subtle implicit flow cases referenced by Clause, J. et al [D]. Implicit data flows are not always considered by dynamic tainting techniques [E], but for my project, implicit data flows are very important, so I need to implement DTA which catches both implicit and explicit flows.

DTA has been used to implement smart mutation fuzzers [A, T, MORE], but I think my work is unique from and complementary to Bekrar's work. In [A], DTA is proposed as a way to intelligently decide which parts of user input should be mutated. In [T], dynamic taint analysis identified hot bytes, which were then modified randomly or with boundary values.

I plan to add on to these ideas by allowing smarter mutations to be selected by backsolving to decide which values are most likely to result in new paths being explored. This has been done with symbolic execution before, but traditional symbolic execution is much more heavy-weight, both in time required to setup the environment and in overhead introduced to run the program, than my proposed approach.

5.2 Symbolic Execution

Symbolic execution supplies symbolic instead of concrete values for input [O]. This technique has been used to effectively detect bugs in software [Q], although historically it has required access to source code, which makes traditional symbolic execution infeasible in many vulnerability research situations where source code is unavailable. In addition, symbolic execution often uses instruction translators like QEMU [K] and satisfiability modulo theorem solvers like Z3 [P], which introduce significant overhead. This makes it difficult for symbolic execution to scale past the order of tens of thousands of lines of code.

Without utilizing symbolic execution, however, it is difficult to automatically guarantee that all possible paths in a binary have been explored. Where most current research focuses on improving speed by using symbolic execution less, I propose to achieve similar code coverage to symbolic execution by implementing something akin to symbolic execution, but without using the SMT solvers and instruction emulation which tend to make symbolic execution slow.

Call-chain-backward symbolic execution has been proposed by [R], although this technique achieved a backward symbolic execution by iteratively applying a forward execution from successively farther away points in the program and the reducing the set of possible symbolic inputs. In contrast, I propose to avoid symbolic execution, forward or backward, by backsolving concretely and then applying heuristics to generate concrete input which is highly likely to result in exploring a new path. Symbolic execution never occurs, although I believe in many situations an equal degree of precision in determining new paths can be attained.

5.3 Concolic Execution

Current research focuses on carefully deciding when and how to use symbolic execution [J, T, W]. So-called concolic execution mixes concrete and symbolic input, and various heuristics to determine when to resort to symbolic execution are in research.

Concolic testing has been shown to improve runtime while still allowing both wide and deep inspection of a program's execution tree, although it often still requires source code to be instrumented [X, Y]. What I propose is similar to concolic testing, in that I am augmenting concrete execution with dynamic taint analysis to solve for mutations which will exercise new paths in the code. However, the lack of a heavyweight constraint solver and the ability to work on binaries without instrumenting or using source code in any way have, to the best of my knowledge, not yet been proposed.

6. Implementation

I implement dynamic taint analysis for files as input sources. I am basing my DTA heavily on Jonathan Salwanś work [N]. Currently I instrument the read syscall, watching for all reads executed, then tag user input at the byte level and follow it as it moves throughout the binary. I instrument binaries at the assembly instruction level using Intel Pin. Each instruction is handled differently based upon the following criteria:

- Is the instruction a conditional jump?
- Is the first operand a memory location or a register?
- Is the first operand read or written?
- Is the first (second if applicable) operand tainted?

For example, move operations involving an untainted first operand and a tainted second operand generally spread taint, while those with a tainted first operand and an untainted second operand remove taint. Tainted branches trigger the backsolving technique. Various precautions, such as removing taint for an instruction like "xor eax, eax", must be taken. Lossy operations, like shifts, must be handled carefully to be backsolve-able.

6.1 Testing Harness

Fuzzers require intuitive but powerful test harnesses, so that users can easily control how a binary is fuzzed and understand the results of fuzz testing. Currently, I record input, output, and taint for each test executed, using an auto-incrementing id and separating results at the directory level. I also keep track of the mutations which have already been explored. Users can grep for interesting conditions like segfaults or unexpected output, then cross reference the test number with the observed taint flow and the particular mutation tested.

6.2 Methodology

I am currently instrumenting dummy programs which use a subset of all possible comparisons and arithmetic operations on tainted variables. If results appear promising I can try to make the fuzzer as complete as possible and begin running it on real-world applications. Achieving a noticeable speedup over modern symbolic execution would be the goal.

As of this writing I am moving towards fuzzing Linux utilities. Results for toy binaries are promising, with "crackme" type programs which ask for a secret being solved in approximately 2*n runs, where n is the length of the secret. There is, however, still some overtainting which must be accounted for.

6.3 Parallelization

I have implemented a parallel version of the pintool, mp.c, to fork and exec a user-specified number of processes. Because a tainted branch may only be reachable based upon previous branches taken, and also because of complex conditions involving loops, the problem is not embarassingly parallel. For this project I have tested my program on a personally created binary, simpleCrackme. This binary takes a file containing a password in argv1. If the password matches its secret, then it prints "You win", otherwise, it prints "You lose".

After running my program, one can grep for the win string in the output/ directory. This is similar to concept to running a fuzzer, and grepping around the output recorded for more interesting conditions, like segfaults. Results show a noticeable speedup.

Sequential: After several runs, I've seen performance ranging from 33 seconds to 19 seconds. 26.12user 7.15system 0:33.42elapsed 99%CPU (0avgtext+0avgdata 12964maxresident)k 0inputs+1496outputs (0major+1168365minor)pagefaults 0swaps

14.66user 4.67system 0:19.44elapsed 99%CPU (0avgtext+0avgdata 12964maxresident)k 0inputs+1496outputs (0major+1168305minor)pagefaults 0swaps

Parallel: The parallel implementation seems to decrease time by somewhere around half, which makes sense considering the problem is not embarassingly parellel, and Intel Pin may try and use multiple cores during startup. 30.64user 9.71system 0:10.83elapsed 372%CPU (0avgtext+0avgdata 12952maxresident)k 0inputs+816outputs (0major+1368622minor)pagefau 0swaps

However, I have seen runs as high as 13 seconds elapsed. Since my testing environment has 4GB ram and 4 virtual cores, 4 processes seems to give me the best speedup. I also tested with 2, 8, and 16 processes but didn't see a comparable speedup.

7. Future Work

Dynamic taint analysis, symbolic execution and fuzzing are hot topics of research, and so the area moves quickly. I have been trying to come up with ideas which are both novel and implementable within my graduation timeframe, but this has proved difficult.

I have made progress using the Pin instrumentation framework, and feel comfortable developing code using the API. I have also read quite a few papers which have progressed my knowledge of the area, so in the worst case I have still learned a lot about a subject that interests me and written some code. In addition, using the Pin API to instrument binaries, and debugging my pintools using tools like strace, has taught me more about assembly language, which is valuable for reverse engineering, a branch of information security I want to pursue a career in.

I think, at the least, this can be a project option for my graduation, but I still believe a thesis is possible at this point.

I need to:

- Read more papers to make sure what I am proposing is novel and will not step on any toes.
- Begin writing a prototype and prove that a subset of I am proposing is feasible.
- Benchmark my prototype against modern applications like S2E and Peach Fuzzer to decide whether or not it makes sense to continue.

Possible complications are:

- Difficulty achieving completeness. Since there are many different ways that user input can be modified and compared, there will be many cases to account for. I think I can at least implement a subset of them in order to demonstrate speedup under a specific set of circumstances
- This work is not novel. Until very recently I was looking at DTA more than symbolic execution, so it is hard to say whether my proposal is actually new. I'm doing my best to read all the relevant papers, and so far it appears that the use of a separate constraint solver or other satisfiability prover is a prerequisite of symbolic execution.

- No noticeable speedup. Pin introduces noticeable overhead, and my ideas may simply not be faster than current approaches, even when test cases are reasonably tailored to work in my favor. In this case I simply go the project route unless something else strikes me. Although research with negative results is theoretically publishable, I don't see very much of it, so I assume it's a faux pas in computer science.

8. References

A Bekrar, Sofia, et al. ”A taint based approach for smart fuzzing.” Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, 2012.

B DeMott, J., Enbody, R., and Punch, W. ”Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing”, BlackHat and Defcon 2007.

C Schwartz, E.J., Avgerinos, T., Brumley, D. ”All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask).” 2010 IEEE Symposium on Security and Privacy.

D Clause, J. Li, W., Orso, A. ”Dytan: a generic dynamic taint analysis framework”. 2007 Int'l symposium on Software testing and analysis. ACM, 2007.

E ... ”Beyond Instruction Level Taint Propagation”.

F Yamaguchi, Fabian, Felix Lindner, and Konrad Rieck. ”Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning.” Proceedings of the 5th USENIX conference on Offensive technologies. USENIX Association, 2011.

G Bao, Tao, et al. ”Strict control dependence and its effect on dynamic information flow analyses.” Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010.

H Cha, Sang Kil, et al. ”Unleashing mayhem on binary code.” Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.

I Avgerinos, Thanassis, et al. ”AEG: Automatic Exploit Generation.” NDSS. Vol. 11. 2011.

J Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea. ”S2E: A platform for in-vivo multi-path analysis of software systems.” ACM SIGARCH Computer Architecture News 39.1 (2011): 265-278.

K Bellard, Fabrice. ”QEMU, a Fast and Portable Dynamic Translator.” USENIX Annual Technical Conference, FREENIX Track. 2005.

L Guillon, Christophe. ”Program Instrumentation with QEMU.” 1st International QEMU Users Forum. 2011.

M Luk, Chi-Keung, et al. ”Pin: building customized program analysis tools with dynamic instrumentation.” Acm Sigplan Notices. Vol. 40. No. 6. ACM, 2005.

N Salwan, Jonathan. Shell-storm.org

O King, James C. ”Symbolic execution and program testing.” Communications of the ACM 19.7 (1976): 385-394.

P De Moura, Leonardo, and Nikolaj Bjrner. ”Z3: An efficient SMT solver.” Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008. 337-340.

Q Godefroid, Patrice, Michael Y. Levin, and David Molnar. ”Sage: Whitebox fuzzing for security testing.” Queue 10.1 (2012): 20.

R Ma, Kin-Keung, et al. ”Directed symbolic execution.” Static Analysis. Springer Berlin Heidelberg, 2011. 95-111.

S Boyer, Robert S., Bernard Elspas, and Karl N. Levitt. ”SELECTa formal system for testing and debugging programs by symbolic execution.” ACM SigPlan Notices. Vol. 10. No. 6. ACM, 1975.

T Wang, Tielei, et al. ”Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution.” ACM Transactions on Information and System Security (TISSEC) 14.2 (2011): 15.

U Cadar, Cristian, and Koushik Sen. ”Symbolic execution for software testing: three decades later.” Communications of the ACM 56.2 (2013): 82-90.

V Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. ”KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” OSDI. Vol. 8. 2008.

W McCamant, Stephen, et al. Transformation-aware symbolic execution for system test generation. Tech. Rep. UCB/EECS-2013-125, University of California, Berkeley (Jun 2013), 2013.

X Majumdar, Rupak, and Koushik Sen. ”Hybrid concolic testing.” Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, 2007.

Y Sen, Koushik, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. Vol. 30. No. 5. ACM, 2005.

Z Peach Fuzzer. urlhttp://peachfuzzer.com/

AA Zhang, Xiangyu, Rajiv Gupta, and Youtao Zhang. ”Precise dynamic slicing algorithms.” Software Engineering, 2003. Proceedings. 25th International Conference on. IEEE, 2003.