Grammar Inference via Dynamic Taint Tracing

Pavel Zakopaylo, ANU
Supervisors: Steve Blackburn, Tony Hosking, Michael Norrish, Shane Magrath

Australian National University

Security analysis is most effective when the structure of the input is known.
By dynamically tracking data flow around a program's memory space, we can more effectively infer the grammar that it accepts.

Motivation

There exist a number of approaches (e.g. fuzzing) that allow for automated detection of security-critical bugs in an application. By crafting inputs based on an understanding of its input, we can enable better analysis by traversing more of the code's branches. However, this is difficult to do when no source code or debugging information is provided apart from the target binary.

Taint Analysis

Microsoft Research's TUPNI paper outlines a grammar inference algorithm that build on top of a Data Flow Graph, which extends on the concept of taint tracing. Rather than tracking if a memory location or register is tainted, analysis of data flow also keep track of which bytes of the input affect each tainted location.
In effect this is a map between memory locations and sets of byte-offsets in the input file. This data structure is updated after every instruction and can be queried at any point during program execution.

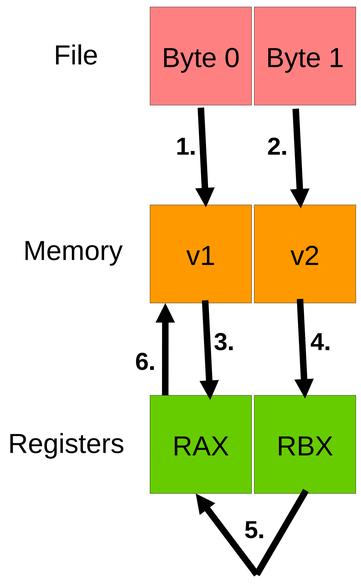
Contribution

TUPNI provides a reasonable level of detail in regards to the inference algorithm, but leaves the taint analysis mostly unspecified. Hence, my work focusses on replicating the taint tracer TUPNI used.
Whereas TUPNI utilized Valgrind for dynamic instrumentation, my tool is built on Intel PIN. This theoretically has a performance benefit, since we are not adding performance overhead by IR-translation or synthetic execution.

Below: Fragment of assembly and corresponding data flow graph.

```
01: read(byte 0 -> v1) // syscall
02: read(byte 1 -> v2)

03: mov rax, v1
04: mov rbx, v2
05: add rax, rbx
06: mov v1, rax
```



Implementation

The taint tracer works by maintaining a tree structure of which memory locations are tainted. Every time an instruction is executed, a note is made of all the memory locations and registers that are read. These are checked against the known taints, and the combined set of taints for that instruction are hence determined. Then, the locations that are written to are gathered, and the total set of taints recorded earlier is either appended to or overwrites the sets of taints of the written locations, as appropriate.
PIN's API provides methods for obtaining the list of memory locations and registers that are read or written. However, blindly following these results produces incorrect output. For example, suppose we push a tainted value onto the stack. PIN will tell us that it reads the tainted value, that it writes to a point in the stack, and that is writes to the stack pointer. Following the algorithm blindly, this would taint the stack pointer. However, we know that the stack pointer is decremented by a fixed amount, and hence its value does not depend on the original tainted value. Such cases (e.g. CALL, RET, XADD etc.) must be handled individually.
In order to test that the taint tracer handles cases correctly, I write toy parsers in assembly language; these are very simple and hence the memory model that they utilize is precisely known. Hence, they can be used as "unit tests", in that the output from the taint tracer can then be marked as correct or incorrect.

Performance

The taint tracers that TUPNI cites do not actually provide the same functionality as TUPNI uses – they usually only store taint information as boolean tainted / not tainted, instead of the data flow graph discussed. This is most likely why they seem to have "negligible" performance overhead. For example, for this type of taint tracking, you only need to store one bit per byte of memory, which limits overhead to at most 12.5%. The same reasoning does not apply to my tool. Indeed, as a result of this and other factors (e.g. after each instruction we execute an O(log N) search of the taint structure), we end up with orders of magnitude more memory and CPU usage. This is, at present, not usable with most large-scale programs.

Future Work

To make this useful, the performance overhead needs to be reduced to the point where something meaningful can be extracted from it. Also, it needs to interface to either some grammar inference system. Ideally, we would be able to replicate the results of the TUPNI grammar learner.

FOR FURTHER INFORMATION
u6055204@anu.edu.au

REFERENCES
Weidong Cui, Marcus Peinado, Karl Chen, Helen Wang, Luis Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS).