Challenges faced when Forcing Malware Execution down Hidden Paths

James Gatt, Mark Vella and Mark Micallef

University of Malta

1 Background

Dynamic Malware Analysis involves the observation of a malware sample at runtime, usually inside a sandbox, whereby probes are used to detect different actions performed by the malware in order to categorize its behaviour.

However, Dynamic Analysis is limited in that it can only observe a single run of the malware at a time, and there is no way of telling whether the run demonstrated the complete set of behaviours contained in the malware. Exploitation of this drawback is on the increase by malware authors as the presence of hidden and trigger-based behaviours has become more widespread[1].

2 Unlocking Hidden Behaviour in Malware

Existing work on unlocking hidden behaviour takes an execution path exploration approach with the aim of maximizing precision by excluding infeasible paths and executing paths under correct runtime values, while also keeping performance at scalable levels. Symbolic execution in the form of concolic testing is one of the more popular approaches, as it generates inputs to explore as many feasible paths in the program's execution tree as possible[1]. However, it is difficult to scale to real-world malware binaries, as its use of SAT/SMT[2] solvers for constraint solving is computationally expensive, which downgrades performance[1]. Moreover, symbolic execution suffers from a number of known shortcomings that malware authors can take advantage of.

An alternative is the use of forced sampled execution techniques such as flood emulation, which explores a program's execution tree in a depth-first fashion, enforcing execution iteration limits on blocks of code, and forcing execution down paths that might not normally be taken at runtime[2]. While sacrificing some precision, as analysis could end up exploring infeasible paths[1][2], we have chosen this approach for our work as it significantly improves the performance, and thus the scalability of analysis[1][2].

3 Challenges

While flood emulation is an effective analysis technique, in practice there are situations where forcing code execution blindly down paths can lead to problems. Here we dis-

[1] http://www.fireeye.com/resources/pdfs/fireeye-hot-knives-through-butter.pdf

[2] Satisfiability/Satisfiability Modulo Theories

cuss a few such situations, while demonstrating how they can also be problematic for concolic execution.

3.1 Jump Tables

Jump tables in a binary usually result from the use of switch statements in the source code. A switch statement allows for the divergence of control flow into many possible paths, based on the value of some variable. At a lower level, this results in the creation of a jump table having an address entry for each possible path. Deducing the number of possible destination addresses in a jump table is a problem in itself. Moreover, jump tables generally use the switch variable as an index in order to calculate the address of the path to be taken on the fly, with the final control flow transfer usually taking the form of an unconditional jump to the destination address. This effectively flattens the control flow of the program and both flood emulation and concolic execution will only end up unlocking a single path, the one taken during a particular execution.

3.2 Blocking Behaviour

Any action that puts a program into a suspended state can be described as blocking behaviour. An example is blocking calls, which include any call to a function that halts program execution until the function returns, for example a network socket waiting for data to return from a remote server. Other cases include malware that waits for a certain number of user mouse clicks before continuing execution, or temporary suspension of a thread using sleep functionality. In all the above cases, flood emulation and concolic execution are stopped in their tracks until the malware resumes execution.

3.3 Program/System-wide State Corruption

Any non-adherence to the assumed state of a system or program at any point during execution infers state corruption. This might affect the program-wide state, for example trying to bind a network address to a socket that has not been created, or the system-wide state, for example trying to access a file that does not exist. Such sections of code exhibit path sequence sensitivity, and the order in which functions are executed is essential. Thus, forcing code execution down such paths even if one of the steps fails, as done during flood emulation, leads to executing infeasible paths, and does not reveal anything as the entire code section will not work. In the case of concolic execution, which strictly adheres to feasible paths, if any step fails, it will stop analysis from continuing further down that particular path.

References

1. F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Ling, and Z. Su. X-force: force-executing binary programs for security applications. In *SEC'14 Proceedings of the 23rd USENIX conference on Security Symposium*, pages 829–844, 2014.
2. J. Wilhelm and T. cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *RAID'07 Proceedings of the 10th international conference on Recent advances in intrusion detection*, pages 219–235, 2007.