Efficient inter-module CFG construction using code pointer restrictions

Nirupama Talele, Frank Capobianco, Xinyang Ge,
Gang Tan, and Trent Jaeger

Attackers can enable a legitimate program to execute an arbitrary and malicious piece of code by hijacking the control flow of the programs. **Control Flow Integrity** (CFI) is the defense mechanism used to prevent these kind of exploits that compromise the program by enforcing the restriction on the targets of control transfer in a program.

An accurate **Control Flow Graph** (CFG) should be computed to ensure the efficiency of the CFI enforcement.

- ✧In general the problem of valid target computation is an un-decidable problem, various approximate approaches to produce coarse grained and fine grained results have been used to enforce CFI.
- ✧It is found that most of the coarse grained approach are still susceptible to code reuse attacks like ROP, while the fine grained approach tend to be performance intensive.

We propose an approach to produce fine grained forward edge control flow policy by enforcing some restrictions on the code pointers in the programs, this enables us to compute a precise policy without performance intensive processing. We observe that these restrictions are obeyed in most cases and propose a remedy to handle the programs which violate the restrictions.

Approach

We compute the legal targets for an indirect invocation by performing static taint propagation without the extensive points to analysis.

We apply the following restrictions on the usage of function pointers in the code:

- ✧**[A1]:** The only allowed operation on function pointer are assignment and dereferencing.
- ✧**[A2]:** There exists no data pointer to a function pointer.

The above restrictions have been found to hold largely for kernel software like FreeBSD and MINIX microkernel [1].

We test whether this approach can be applied effectively for user-space programs, and found that a majority of the programs comply with the restrictions.

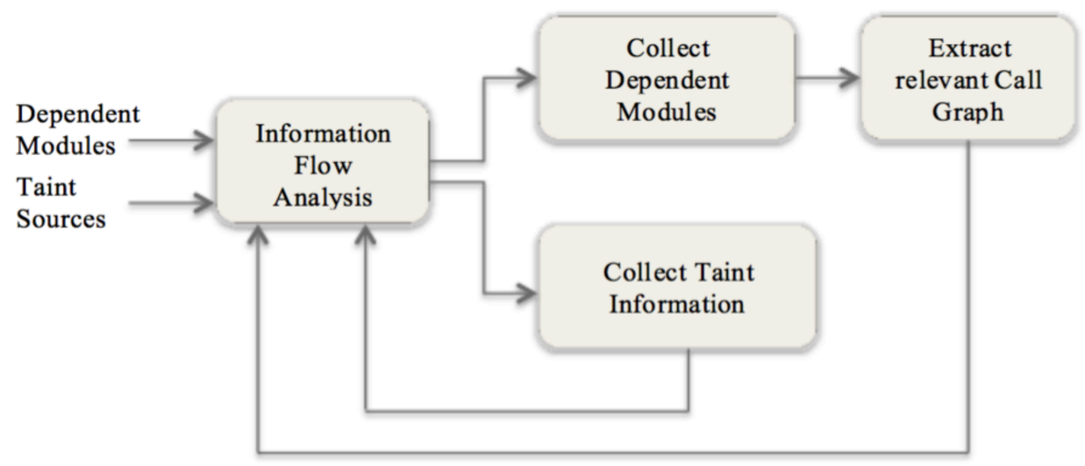
We observed that to ensure soundness in user-space programs we need to add another restriction for isolating data and code pointers.

- ✧**[A3]**: Function pointers may not be cast to or cast from a data pointer.

Inter-module Analysis

Another challenge for user space programs is the policy computation across dependent and dynamically loaded modules.

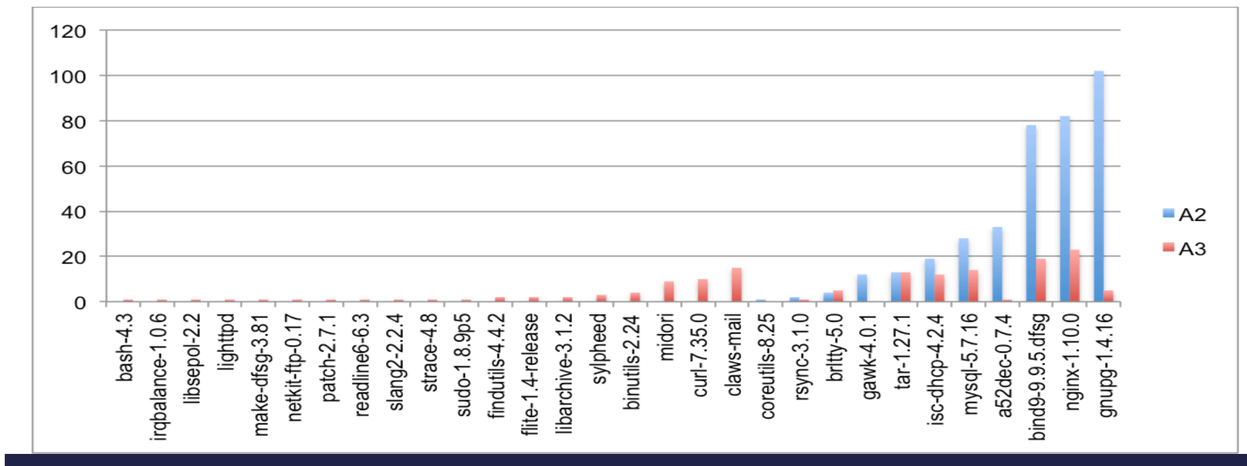
We iteratively process the dependent modules for code pointer information exchange and compute the complete policy for the application.



Violation Analysis

We perform our test on base packages that come with Ubuntu 16.04 distribution, a total of **950** packages with **2,349** binaries.

Out of those, **871** binaries have either indirect calls or pass function pointers to libraries out of which we found violations in **147** binaries, i.e. about **16.9%**.



Results

The following table shows the comparison between targets computed using signature matching approach and our taint based approach.

Method	Binary Files Analyzed	Resolved Indirect Calls	Total Targets	Average Targets per call
Taint Based	455	19,532	81,352	4.16
Signature Based	455	19,532	388,226	19.87

We also evaluate the gain in precision due to inter-module analysis with and without statically linking.

Binaries	Indirect Calls	Inter-mod Targets	Merged Targets	Percentage Increase
170	11,766	42,577	43,882	3.1

Related Work

[1] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, 2016.

Supported by National Science Foundation