Control Flow Integrity: Security Precision and Performance - A Summary

Boakye Dankwa

January 28, 2018

1 Summary

Control-Flow Integrity (CFI) shows promise to defeat control flow modification attacks such as remote code injection, Return-Oriented Programing (ROP) and code-reuse, that expoit memory corruption vulnerabilities in C/C++ programs. The technique has been researched and improved upon by researchers and has been integrated into products such as LLVM and some Microsoft products. Current CFI evaluations usually use the SPEC2006 benchmark and Average Indirect-target Reduction (AIR) to measure performance and security precision respectively. Control-Flow Integrity: Security Precision and Performance [1] systematize various CFI implementations and their trade-offs, and propose a novel way of evaluating these implementations against security precision and performance.

The authors first described CFI mechanisms as consisting of an analysis and a runtime component. The analysis component constructs a Control-Flow Graph(CFG), which approximates the set of control-flow transfers in the program, from the source code or binary. The CFG is then used by the runtime to restrict program execution flow by validating indirect control-flow transfer against the CFG edges. They classified control-flow transfer into nine categories from CF.1 (backward control flow such as return instruction) to CF.9 (control flow that supports features such as dynamic linking) based on the kind of control flow (*direct, indirect, jump, return* or *call*) and supported language constructs (*such as function pointers and vtables*). They further explained that CFG is subject to over-approximation, making the precision of the CFG analysis crucial in determining the security precision of a given CFI mechanism. They classified prior work on precision of the static analysis into increasing order of static analysis precision (SAP), from SAP.F.0 (No forward branch validation) to SAP.F.6(supports dynamic analysis) for forward control-flow analysis precision. For backward control-flow precision, the classification are SAP.B.0 (No backward branch validation), SAP.B.1(Labeling equivalent classes) and SAP.B.2 (Shadow stack). They hope to achieve a more precise and consistent taxonomy this way as opposed to the fine-gained/coarse-grained classification used in prior work within the CFI research community.

Using the proposed classifications on control-flow transfers and static analysis precision, they qualitatively analyzed the theoretical security guarantees of different existing CFI mechanisms in publications. They also gave quantitative evaluation of the security precision of several CFI implementations. The CFI mechanisms were evaluated qualitatively on four dimensions namely: supported control-flow transfers (CF), reported implementation overhead (RP), static analysis precision on forward control-flows (SAP.F) and static analysis precision of backward control-flows (SAP.B). Each of the mechanisms were evaluated on four axes on a spider plot, each axis corresponding to one of the dimensions. They stated that the area under the spider plot gives a rough estimate of the precision of a given mechanism. They noted that both forward and backward control-flow transfers must be considered when analyzing the precision of a given implementation. Therefore implementations such as SafeDispatch, T-VIP, VTV, IFCC, vfGuard and VTint have a precision of SAP.B.0 in the SAP.B dimension since they did not consider backward control-flow transfers. They also noted that most implementations use labels to distinguish equivalent classes. This leads to loss of precision in the static analysis since.....with the exception being πCFI which uses dynamic information to activate CFG edges, scoring SAP.F.6. πCFI received the highest precision. Implementations that received high precision are those that use context-sensitivity processing of CFG in the forward control-flow transfer and also has a high backward control flow precision such as shadow stack. Such implementations include RochJIT, MCFI, PathArmor and Lockdown. Their contribution to quantitative measure of CFI security was a metric which is directly proportional to the number of equivalent classes (LC) and inversely proportional to the size of the largest class (LC) as shown in Equation 1.

$$QuantitativeSecurity \;=\; EC * \frac{1}{LC} \tag{1}$$

They argued that an implementation with a large number of small equivalent classes is more secure than an implementation with a small number of large equivalent classes. Hence, Equation 1 provides a more meaningful measure than Average Indirect Target Reduction (AIR) which gives the average reduction of allowed targets. Against this novel quantitative matric, they eval TODO - Quantitative analysis.

The authors studied the performance of various publicly available CFI implementations. They replicated the performance methodology described in the various publications but used the same hardware platform, operating system and the SPEC CUP2006 benchmark for an objective comparison. They observed that benchmarks that make large number of indirect calls or generates large numbers of equivalent classes incur large performance overhead. They explained this as the security precision/performance trade-off that CFI designers must make. They reported that the results of their performance study suggests that both precision and performance of CFI mechanisms have improved over the years. For example, compiler based CFI mechanisms have overhead within the 5% threshold for widespread adoption. Their results showed that LLCM-CFI 3.7 and VTI had the least measured performance

overhead, followed by CFGuard, πCFI and VTV. They also reviewed reported performances of a number of compiler based and binary-level CFI mechanisms and observed that both methods incur similar performance overhead. However, CFI strategies that employ a more complete binary-level mechanisms such as using a shadow stack, increases performance overhead. Lockdown was given as an example of such mechanisms. They suggested researchers utilize all the benchmarks in SPEC CPU2006 when conducting CFI performance analysis to get a more complete picture. They also suggested more frequently targeted applications be made part of the SPEC CPU2006 benchmark.

Finally, they discussed trends in CFI enforcement mechanisms, open challenges facing the CFI research community and the direction of future research. According to paper, CFI enforcement mechanisms have evolved from Program Shepharding type enforcement where mechanisms only check integrity of a subset of indirect branches, through label-based schemes to hardware-supported enforcements. They described successful hardware-based CFI mechanisms in the research and production environment. Some of the examples include LBR inspection, HMAC authentication of pointers and special Intel processor instructions. These hardware solutions usually protect backward control-flow transfers with fully isolated shadow stacks which shows improved performance overhead compared to software based shadow stack. The authors listed imprecision in the construction of CFGs as a major weakness in CFI mechanisms. They recommended the use of flow-sensitive and context-sensitive analysis for forward edges, and shadow stacks for backwards edges. They also recommended coding practices that would result in large number of equivalent classes as improvements that could be adopted by the CFI community. For the future CFI work, the authors identified protecting operating system kernels, JIT compiled code and method dispatch in object-oriented languages as challenging areas for CFI research.

2 Application to Safety Critical Software

References

[1] Nathan Burow, Scott A. Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, abs/1602.04056, 2016.