# Control Flow Integrity: Security Precision and Performance - A Summary

Boakye Dankwa

January 30, 2018

Control-Flow Integrity (CFI) shows promise in the fight against control-flow modification attacks such as remote code injection, Return-Oriented Programing (ROP) and code-reuse. These attacks expoit memory corruption vulnerabilities in C/C++ programs. The technique has been researched and improved upon by researchers for some time now, and solutions have been integrated into products such as LLVM and CFGuard. To evaluate these solutions, researchers rely on Average Indirect-target Reduction (AIR) to measure security precision. Control-Flow Integrity: Security Precision and Performance [1] proposes an alternative qualitative and a quantitative security metrics for evaluating CFI mechanisms. They presented an evaluation of existing CFI mechanisms against the novel matrics and provided detailed performance study of various CFI implementations using the SPEC CPU2006 benchmark.

The authors described CFI mechanisms as consisting of an analysis and runtime components. 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 explained that static analysis nature of CFG generally makes it subject to over-approximation of the true control-flow transfers in a given program. This makes the precision of CFG analysis a determining factor in the security assurance of a given CFI mechanism. To aid in the precision analysis of CFGs, they provided a classification of control-flow transfers in a program into nine categories namely:

- CF.1 (backward control-flow )
- CF.2 (forward control-flow using direct jumps )
- CF.3 (forward control-flow using direct calls )
- CF.4 (forward control-flow using indirect jumps)
- CF.5 (forward control-flow using indirect calls supporting function pointers )
- CF.6 (forward control-flow using indirect calls supporting vtables )
- CF.7 (forward control-flow using indirect calls supporting Smalltalk-style method dispatch )
- CF.8 (complex control-flow to support exception handling )

– CF.9 (control-flow supporting language features such as dynamic linking, separate compiling, etc)

Additionally, the proposed a taxonomy for the static analysis precision of CFGs as shown below in increasing order of precision for forward and backward control-flow transfers:

– SAP.F.0 (No forward branch validation )
– SAP.F.1a (ad-hoc algorithms and heuristics )
– SAP.F.1b (context- and flow-insensitive analysis )
– SAP.F.1c (labeling equivalent classes )
– SAP.F.2 (class-hierarchy analysis)
– SAP.F.3 (rapid-type analysis )
– SAP.F.4a (flow sensitive analysis )
– SAP.F.4b (context-sensitive analysis )
– SAP.F.5 (context- and flow-sensitive analysis )
– SAP.F.6 (dynamic analysis (optimistic)

– SAP.B.0 (No backward branch validation)
– SAP.B.1 (labeling equivalent classes)
– SAP.B.2 (Shadow stack)

They hope the proposed classifications enable researchers evaluate CFI mechanisms more precisely and consistently compared to the overused and imprecise fine-gained/coarse-grained CFI security classification used within the CFI research community.

Using the above metrics, 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 based on a novel quantitative measure shown in Equation 1. The CFI mechanisms were evaluated qualitatively on four dimensions namely: supported control-flow transfers they support (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 which are grouped into equivalent classes. They explained that this leads to loss of precision in the static analysis since, for instance, all callers of a function must carry the same label. A more precise approach would be each

caller of a given function carrying a unique label. They pointed out that the only exception to this loss of precision is $\pi$CFI which uses dynamic information to activate CFG edges, scoring SAP.F.6. $\pi$CFI received the highest precision. According to them, 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. The authors proposed a novel quantitative measure for CFI security. The metric 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 because smaller classes provide less attack surface. 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 evaluated four different open-source CFI mechanisms. These implementations were instrumented to capture the number and sizes of sets. The authors described some known vulnerabilities in CFI mechanisms. These vulnerabilities exist in both fine-grained and coarsed-grained CFI mechanisms, some of which are leveraged by attackers to make system calls or implement Turing complete computations using gadgets.

The authors performed extensive performance study of various publicly available CFI implementations. They replicated the performance methodology described in the various publications but used a single hardware platform, operating system and the SPEC CUP2006 benchmark for the study where possible 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 also reported that the results of their performance study suggest that both precision and performance of CFI mechanisms have improved over the years. For example, recent compiler based CFI mechanisms have overhead within the 5% threshold for widespread adoption. They showed that LLCM-CFI 3.7 and VTI had the least measured performance overhead, followed by CFGuard, $\pi$CFI 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 comprehensive binary-level mechanisms such as a shadow stack, increases performance overhead. Lockdown was given as an example of such mechanisms. At the end, they suggested researchers utilize all the benchmarks in SPEC CPU2006 when conducting CFI performance analysis to get a more complete picture. They also recommended more frequently targeted applications be made part of the SPEC CPU2006 benchmark.

Finally, the authors 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 enforcement. They described successful hardware-based CFI mechanisms in the research and production environment including techniques which uses LBR inspection, HMAC authentication of pointers and those that utilize special Intel processor instructions in their implementations. 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, they identified protecting operating system kernels, Just-in-Time (JIT) compiled code and method dispatch in object-oriented languages as challenging areas for CFI research.

# 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.