# Poster: CRAFTED: Code Reuse Analysis for Trusted and Effective Defense

Ethan Johnson
*University of Rochester*
*Email: ethanjohnson@acm.org*

Tianqin Zhao
*University of Rochester*
*Email: tzhao8@ur.rochester.edu*

John Criswell
*University of Rochester*
*Email: criswell@cs.rochester.edu*

***Abstract*—Return-oriented programming (ROP) and similar "code-reuse" attacks have led to intense interest in low-overhead Control-Flow Integrity (CFI) defenses that detect or prevent corruption of control data. However, weaker forms of these defenses have already been broken, and it is unclear whether stronger variants will continue to repel attacks.**

**We propose a program analysis infrastructure for automatically evaluating the efficacy of defenses against code-reuse attacks. Our design uses static analysis to determine whether a particular security policy can prevent an attacker from stitching together the data and control flows necessary to perform a particular malicious computation.**

## 1. Introduction

Return-oriented programming (ROP) [1] and similar code-reuse attacks have led to intense interest in defenses against control-flow hijacking. These defenses are attractive because they can be applied to existing C/C++ code and have tolerable overheads (one implementation of Control-Flow Integrity (CFI) [2] has 16% average overhead). However, the efficacy of such defenses is in doubt: prior work shows that some CFI defenses still permit sufficient control-flow corruption to allow Turing-complete code reuse attacks [3]. Furthermore, even defenses which protect all control data from corruption e.g., Code Pointer Integrity (CPI) [4], do not mitigate non-control data attacks [5] which are not only viable in practice but sufficiently expressive to permit Turing-complete computation [6].

Control-flow integrity research has, like many subjects in the security field, fallen into an "arms race." New defenses are proposed and thought to be sufficient only to be exploited by a more clever attack later. This pattern is difficult to transcend because there is no effective evaluation methodology for these defenses.

We propose an infrastructure, named CRAFTED (Code Reuse Analysis for Trusted and Effective Defense), that will automatically evaluate the efficacy of defenses against code-reuse attacks. Our design uses static analysis to determine whether a particular security policy can prevent an attacker from stitching together the data and control flows necessary to perform a specified malicious computation. Our prototype utilizes the LLVM compiler infrastructure [7].

## 2. Principles of Code Reuse Attacks

Defenses such as CFI [8] and CPI [4] seek to prevent attacks by limiting the order in which instructions within a program can execute. These defenses force control flow to enter and exit blocks of code, rather than arbitrarily jump to any instruction, and to follow a limited set of transitions. These blocks typically correspond to the program's *basic blocks* - its largest units of straight-line (non-branching) code - which are constructed during compilation of the program and linked together by the compiler to form a *control flow graph* (CFG) representing the possible paths that control flow can take. Different defense policies enforce variously strict over-approximations of the CFG, ranging from "coarse-grained" CFI (which enforces only basic restrictions, e.g., a return must transfer to an instruction following a call [9]) to CPI (which completely prevents deviation from normal control flow [4]).

A *code-reuse attack* exploits memory corruption to manipulate the control flow of a program to perform a malicious computation on behalf of an attacker by stringing together *reusable instructions* taken from the original code of a target program; manipulating control flow can be done by corrupting control data (such as return addresses and function pointers) [1], [10] and non-control data [6]. An instruction within a target program is (in general) *reusable* if an attacker can manipulate control flow so that the instruction is executed. An instruction is *reusable for a particular attack* if it is generally reusable and data flows are present in the program so that the result(s) of each step in the attack can reach the input(s) of the next step(s) without being overwritten by intervening instructions that are not of interest to the attacker. Importantly, it is acceptable to execute irrelevant instructions that happen to be present in the original code between reusable instructions of interest so long as they do not interfere with the attacker's computation.

An effective defense must prevent an attacker from executing a desired set of instructions in the desired order or ensure that other instructions within the program overwrite inputs to the attacker's instructions before those instructions can execute. Current methods of evaluating defenses do not ensure that all code reuse opportunities are blocked, leaving the door open for attackers to continue to evade defenses with more sophisticated techniques. In order to truly know that a defense will thwart a particular attack, an evaluation

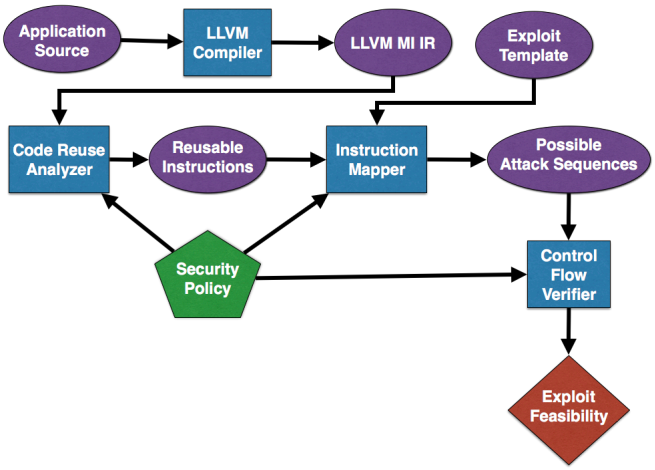


Figure 1. CRAFTED Architecture

must account for all potentially reusable instructions and all possible data-flows between them.

## 3. CRAFTED Design

To evaluate whether a defense policy will effectively stop a given attack, CRAFTED must determine all possible ways to construct a malicious computation using the reusable instructions within a target program under the execution ordering and data-flow constraints enforced by the defense. CRAFTED accepts three inputs: 1) a specification of the restrictions on reusable code imposed by a defense policy; 2) a template for a malicious computation; and 3) a program to be attacked. A malicious computation must satisfy constraints on data and control flow to perform its desired task. Static analysis can determine whether it is possible for any sequence of reusable code to satisfy these constraints. If it is not, then the defense is effective in preventing this attack; if it is, CRAFTED can demonstrate this by identifying the specific sequences of reusable instructions that could potentially permit the malicious computation.

CRAFTED consists of three main components. The **Code Reuse Analyzer** performs static analysis on the machine code of a program to identify the set of all potentially reusable instructions permitted by a defense policy. The **Instruction Mapper** identifies sequences of reusable instructions which provide the necessary data-flows that connect together the inputs and outputs of instructions needed by the attack. The **Control Flow Verifier** checks whether identified instruction sequences will be permitted to execute under the control flow restrictions of a defense policy. Figure 1 depicts the general structure of CRAFTED and its dependencies on LLVM components.

### 3.1. Code Reuse Analyzer

To identify reusable instructions, we start with the local (intra-procedural) *control flow graph* (CFG) of the target program, which is explicit in LLVM's in-memory representation of its machine code [11]. Local control flow is static (hard-coded by the compiler) and not manipulable by an attacker. We then extend this into an *augmented global CFG* which reflects all possible control-flow transfers between functions that are permitted by the defense policy. In the most basic case - a defense policy which permits no control flow corruption whatsoever (such as CPI [4]) - the augmented CFG will be the same as the normal global CFG that would be computed by the compiler, reflecting all paths that can be taken under benign execution (or some reasonable over-approximation thereof, depending on the precision of available static analyses). Under policies that allow some (usually limited) control flow corruption (such as CFI [8]), the augmented CFG will have additional edges representing the full flexibility permitted the attacker. The augmented CFG may have multiple entry points, reflecting the attacker's options at the point of attack. The set of reusable instructions available to the attacker consists of all instructions which are reachable from any entry point in this augmented CFG.

### 3.2. Instruction Mapper

Using the augmented CFG constructed by the Code Reuse Analyzer, CRAFTED performs an iterative "may-reach" data flow analysis [12] to compute the set of all definitions that may reach each input of each reusable instruction. This allows CRAFTED to determine whether reusable instructions can be "strung together" to perform sequential steps of a larger computation. The malicious computation template provided as input to CRAFTED specifies the data-flow constraints required to perform the attacker's computation. Using the reaching-definitions analysis results, CRAFTED builds all possible chains of reusable instructions which satisfy the attack's data-flow constraints. From these, it generates all possible execution sequences of reusable instructions that respect the attack's data-flow.

### 3.3. Control Flow Verifier

To determine if it is possible to execute the instruction sequences identified by the Instruction Mapper without violating the control-flow restrictions of the defense policy, we construct a context-free grammar[1] based on the augmented CFG and use a parser generator to create a verifier that checks whether a given sequence of instructions respects its constraints. CRAFTED uses the verifier to check each instruction sequence to determine which (if any) of the sequences will be permitted to execute under the policy. Any sequences that survive this check represent a potential code-reuse attack that will succeed despite the enforcement of this defense policy. Conversely, if no instruction sequence can be found that satisfies all the constraints of the attack template and defense policy, CRAFTED concludes that this policy effectively stops the attack.

1. Both "control-flow graph" and "context-free grammar" are commonly abbreviated "CFG." We use "CFG" to abbreviate "control-flow graph."

## References


[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information Systems Security*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[2] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2005, pp. 340–353.

[3] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security*, vol. 14, 2015, pp. 28–38.

[4] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 147–163. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685061

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *14th USENIX Security Symposium*, August 2004, pp. 177–192.

[6] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 969–986.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the Conference on Code Generation and Optimization*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information Systems Security*, vol. 13, pp. 4:1–4:40, November 2009.

[9] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proceedings of the Thirty-Fifth IEEE Symposium on Security and Privacy*, May 2014.

[10] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 121–141. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23644-0_7

[11] "The LLVM Target-Independent Code Generator," http://llvm.org/docs/CodeGenerator.html.

[12] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.