Analyzing Control Flow Integrity with LLVM-CFI

Paul Muntean
Technical University of Munich
paul.muntean@sec.in.tum.de

Matthias Neumayer
Technical University of Munich
matthias.neumayer@tum.de

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Gang Tan
Penn State University
gtan@psu.edu

Jens Grossklags
Technical University of Munich
jens.grossklags@in.tum.de

Claudia Eckert
Technical University of Munich
claudia.eckert@sec.in.tum.de

ABSTRACT

Control-flow hijacking attacks are used to perform malicious computations. Current solutions for assessing the attack surface after a *control flow integrity* (CFI) policy was applied can measure only indirect transfer averages in the best case without providing any insights w.r.t. the absolute calltarget reduction per callsite, and gadget availability. Further, tool comparison is underdeveloped or not possible at all. CFI has proven to be one of the most promising protections against control flow hijacking attacks, thus many efforts have been made to improve CFI in various ways. However, there is a lack of systematic assessment of existing CFI protections.

In this paper, we present LLVM-CFI, a static source code analysis framework for analyzing state-of-the-art static CFI protections based on the Clang/LLVM compiler framework. LLVM-CFI works by precisely modeling a CFI policy and then evaluating it within a unified approach. LLVM-CFI helps determine the level of security offered by different CFI protections, after the CFI protections were deployed, thus providing an important step towards exploit creation/prevention and stronger defenses. We have used LLVM-CFI to assess eight state-of-the-art static CFI defenses on real-world programs such as Google Chrome and Apache Httpd. LLVM-CFI provides a precise analysis of the residual attack surfaces, and accordingly ranks CFI policies against each other. LLVM-CFI also successfully paves the way towards construction of COOP-like code reuse attacks and elimination of the remaining attack surface by disclosing protected calltargets under eight restrictive CFI policies.

CCS CONCEPTS

• **Security and privacy → Systems security**; • **Software and application security**;

KEYWORDS

LLVM, control flow integrity, computer systems, defense.

ACM Reference Format:
Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2019. Analyzing Control Flow Integrity with LLVM-CFI. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359789.3359806

ACSAC '19, December 9–13, 2019, San Juan, PR, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA*, https://doi.org/10.1145/3359789.3359806.

1 INTRODUCTION

Ever since the first Return Oriented Programming (ROP) attacks [23, 39, 48], the cat and mouse game between defenders and attackers has initiated a plethora of research. As defenses improved over time, the attacks progressed with them, as pointed out by Carlini *et al.* [8]. While defenders followed several lines of research when building defenses: control flow integrity [5, 9, 13, 18, 20, 31, 32, 34, 35, 37, 49, 52, 57, 58], binary re-randomization [54], information hiding [3], and code pointer integrity [24], the attacks kept up the pace and got more sophisticated [4, 11, 25, 26, 46].

In principle, even with the myriad of currently available CFI defenses, performing exploits is still possible. This holds even in the presence of hypothetically perfect CFI [8]. For this reason, in this work, we aim to answer the question of how secure are programs which are protected by CFI defenses. Even after CFI defenses are in place, attackers could search the program for gadgets that are allowed (for example, pass under the radar due to imprecision) by CFI defenses to conduct *Code Reuse Attacks* (CRAs); see [7, 46]. As such, these attacks become highly program-dependent, and the applied CFI policies make reasoning about security harder. The attacker/analyst is thus confronted with the challenge of searching (manually or automatically) the protected program's binary or source code for gadgets which remain useful after CFI defenses have been deployed. As a result, there is a growing demand for defense-aware attack analysis tools, which assist security analysts when analyzing CFI defenses.

To the best of our knowledge, there are neither tools for statically modeling and comparing static CFI defenses against each other, nor static CRA crafting tools which are aware of a set of applied defenses. Existing tools, including static pattern-based gadget searching tools [12, 55] and dynamic attack construction tools [10, 14, 22, 51, 53], all lack deeper knowledge of the protected program. As such, they can find CRA gadgets, but cannot determine if the gadgets are usable after a defense was deployed.

Consequently, with each applied defense, a more capable assessment tool needs ideally to: (1) model the defense as precisely as possible, (2) use program metadata in order not to solely rely on runtime memory constraints, (3) use precise semantic knowledge of the protected program, and (4) provide absolute analysis numbers w.r.t. the remaining attack surface after a defense was deployed. This allows an analyst to provide precise and reproducible measurements, to decide which CFI defense is better suited for a given situation, and to defend against or craft CRAs by searching available gadgets. Lastly, none of the existing static and dynamic program analysis tools can be used to compare static CFI defenses against each other w.r.t. the offered protection level after deployment.

In this paper, we present LLVM-CFI, which to the best of our knowledge, is the first static Clang/LLVM based compiler framework used for modeling and analyzing static state-of-the-art CFI defenses. LLVM-CFI can determine the security level these protection techniques offer as well as the remaining attack surface after such a defense was deployed. LLVM-CFI automates one step of COOP-like attacks by finding protected targets towards which program control-flow can transfer. As such, LLVM-CFI provides the first step towards searching for COOP-like gadgets, but its main purpose is to evaluate static CFI policies against each other. Further, LLVM-CFI cannot build CRAs automatically but rather assist on how one could go about when constructing CRAs. We envisage LLVM-CFI to be used as a tool to analyze conceptual or deployed CFI defenses by either an analyst—to better compare existing CFI defenses against each other—or by an attacker (*e.g.,* red team attacker)—to help craft attacks which can bypass existing in-place CFI defenses.

LLVM-CFI is a *unified framework* to evaluate different CFI defenses, enabling a head-to-head comparison. To achieve this, we introduce a new CFI defense analysis metric dubbed calltarget reduction (*CTR*), which tells precisely, without averaging the results, how many calltargets are still available after a CFI defense was enforced. LLVM-CFI is capable to analyze CFI policies w.r.t. several metrics and thus compare CFI policies w.r.t. different aspects. *CTR* is one example metric along other three metrics presented in this paper.

Further, we are particularly interested in calltarget reduction analysis as this is the most used metric (see AIR [59], fAIR [49], and AIA [16] — however, these metrics average the results) to compare CFI defenses against each other. At the time of writing this paper, none of the existing CFI metrics can tell how secure a program is after a certain CFI policy was applied; as such, we do not claim that by using our *CTR* metric we can provide more security guarantees than other metrics, but rather *CTR* provides absolute values rather than averaging them. Even though the calltargets could be used or not during an attack, we opt in this work to not further investigate this avenue as the usability of a target depends on the type of CRA performed. Instead, for each protected callsite, we show additional calltarget features (see Section 5.6), such that the analyst could with ease figure out if the targets are usable for a particular attack.

By using different compilers, compiler flag settings or OSs, the results of CFI policy analysis could not be comparable against each other. For this reason, LLVM-CFI relies on the insight that, by precisely modeling a CFI defense into a comprehensive policy, the introduced constraints on callsites and calltargets can be assessed during program compile time, by an unified analysis framework. Further, in order to support this task, LLVM-CFI provides a set of program *expressive primitives*, which help to characterize a wide range of static CFI policies. For example, LLVM-CFI offers static primitives related to variable types, class hierarchies, virtual table layouts and function signatures. These primitives can be used as building blocks to model a wide range of CFI policies. Further, LLVM-CFI provides the legitimate calltarget set for each callsite under different CFI defenses. This set can be evaluated by a lower and upper target bound. The lower bound is represented by the set of all calltargets which are, according to the analyzed policy (*e.g.,* sub-hierarchy policy [5]), legitimate to be called by a protected callsite during benign program execution. Accordingly, the upper bound is represented by the set of all calltargets that can be called (both legitimately or not) by a protected callsite during benign program execution (*e.g.,* all virtual tables policy [58]). Further, LLVM-CFI paves the way towards automated control-flow hijacking attack construction, *e.g.,* the control flow bending attack, see Carlini *et al.* [7], or to refine the analysis of existing attack construction or defense tools.

LLVM-CFI analyzes statically the CFI defenses, as these are more commonly deployed against control-flow hijacking attacks than dynamic defenses. Further, LLVM-CFI focuses on source code (LLVM's IR and Clang metadata is pushed into compiler's LTO phase and analyzed) rather than on binary code, as comparing various static CFI defenses against each other is feasible only in this way. Moreover, the binary CFI policy implementations can be more precisely expressed with source code at hand. Therefore, we opt not to analyze the machine code of the protected programs as source code provides more semantics and precision w.r.t. the constraints imposed by each CFI defense. Thus, LLVM-CFI models static CFI policies for binaries more precisely than the binary instrumentation tools which were used to enforce them as these operated mostly on the binary only. Lastly, LLVM-CFI models source code based CFI policies as precise as the original policies as these were implemented either atop Clang/LLVM or GCC compilers.

We evaluated LLVM-CFI with real-world programs including Google's Chrome, NodeJS, etc., and with eight state-of-the-art static CFI policies which were previously thoroughly discussed and clarified with their original authors w.r.t. how these were modeled within LLVM-CFI. We selected eight representative binary and source code based CFI policies based on the criteria that the policies should be static, state-of-the-art, and available as open source (published). Further, we are aware that there are other CFI policies which cannot currently be modeled with LLVM-CFI. For this reason, in this paper, we do not aim to give a full presentation of which CFI policies can be modeled with LLVM-CFI.

LLVM-CFI can help the assessment of CFI defenses and it can help at finding gadgets, even with highly restrictive CFI defenses deployed. Further, we demonstrate how LLVM-CFI can be utilized to pave the way towards automated CRA construction. We also show how it can be effectively used to empirically measure the real attack surface reduction after a certain static CFI defense policy was used to harden a program's binary.

Comparing binary and software based policy results against each other is out of scope of this paper. Rather, the goal of our tool is to show how the analyzed CFI policies compare against each other and to provide insights on their precision and benefits. Applications of LLVM-CFI go beyond a CFI defense assessment framework, and we envision LLVM-CFI as a tool for defenders and software developers to highlight the residual attack surface of a program. As such, a programmer can evaluate if a bug at a certain program location enables a practical CRA.

In summary, our contributions include the following:

- We implement LLVM-CFI[1], a novel framework for empirically analyzing and comparing CFI defenses against each other.
- We introduce our comprehensive formalization framework for formalizing static state-of-the-art binary and source code based CFI defenses.
- We show evaluation results based on real-world programs by comparing existing static CFI defenses against each other.
- We present an attacker model that is powerful and drastically lowers the bar for performing attacks against state-of-the-art CFI defenses with LLVM-CFI at hand.

[1]The source code is available at https://github.com/TeamVault/LLVM-CFI.git

The remainder of this paper is organized as follows. Section 2 contains the brief overview of the required technical background knowledge. Section 3 describes the design of LLVM-CFI, and Section 4 presents implementation details of LLVM-CFI, while Section 5 contains the evaluation of LLVM-CFI. Section 6 highlights related work and Section 7 offers concluding remarks.

2 BACKGROUND

Control Flow Integrity. Control-Flow Integrity (CFI) [1, 2] is a state-of-the-art technique used successfully along other techniques to protect forward and backward edges against program control-flow hijacking based attacks. CFI is used to mitigate CRAs by, for example, pre-pending an indirect callsite with runtime checks that make sure only legal calltargets are allowed by an as-precisely-as-possible computed control flow graph (CFG).

Protection Schemes. Alias analysis in binary programs is undecidable [40]. For this reason, when protecting CFG forward edges, defenders focus on using other program primitives to enforce a precise CFG during runtime. These primitives are most commonly represented by the program's: class hierarchy [20], virtual table layouts [38], reconstructed-class hierarchies from binary code [37], binary function types [52] (callsite/calltarget parameter count matching), etc. They are used to enforce a CFG which is as close as possible to the original CFG being best described by the program control flow execution. Note that state-of-the-art CFI solutions use either static or dynamic information for determining legal calltargets.

Static Information. CFI defenses which are based on static information allow, for example, callsites to target: (1) function entry points, *e.g.,* [59], map callsite types to target function types by creating a mask which enforces that the number of provided parameters (up to six) has to be higher than the number of consumed parameters, *e.g.,* [52], (2) a rebuilt-class hierarchy (no root node(s) and the edges are not oriented) can be recuperated from the binary and enforced, *e.g.,* [37], (3) all virtual tables that can be recuperated and enforced, *e.g.,* [38], only certain virtual table entries are allowed, *e.g.,* [57] based on a precise function type mapping, and (4) sub-class hierarchies are enforced, *e.g.,* [5, 20, 49]. Thus, in order for such techniques to work, program metadata should be available or it should be possible through program analysis to reconstruct it.

3 DESIGN

3.1 Overview

LLVM-CFI is designed to assist an analyst evaluating the attack surface after different types of static CFI defenses were applied and to pave the way towards automated detection of gadgets. To achieve this, LLVM-CFI applies a static black box strategy in order to statically retrieve the set of attacker-controllable forward control flow graph (CFG) edges. The forward-vulnerable CFG edges are expressed as a callsite with a variable number of possible target functions. Further, these CFG edges can be reused by an attacker to call arbitrary functions via arbitrary read or write primitives. To call such series of arbitrary functions, an attacker can chain a number of edges together by dispatching fake objects contained in a vector. See, for example, the COOP [46] attack which is based on a dispatcher gadget used to call other gadgets through a single loop iteration. The COOP attack uses gadgets which are represented by whole virtual functions.

LLVM-CFI supports a wide range of code reuse defenses based on user-defined policies, which are composed of constraints about the set of possible calltargets allowed by a particular applied CFI defense. The main idea behind LLVM-CFI is to compile the target program with different types of CFI policies and get the allowed target set per callsite for each constraint configuration. Note that we assume the program was compiled with the same compiler as the one on which LLVM-CFI is based. Moreover, LLVM-CFI's policies are reusable and extensible; they model security invariants of important CRA defenses. Essentially, under these constraints, virtual pointers at callsites can be corrupted to call any function in the program. Thus, in this paper, we focus on the possibility to bend [7] a pointer to the callsite's legitimate targets. Further, we assume that large programs contain enough gadgets for successfully performing CRAs. Bending assumes that it is possible for an attacker to reuse protected gadgets during an attack making the applied defense of questionable benefit.

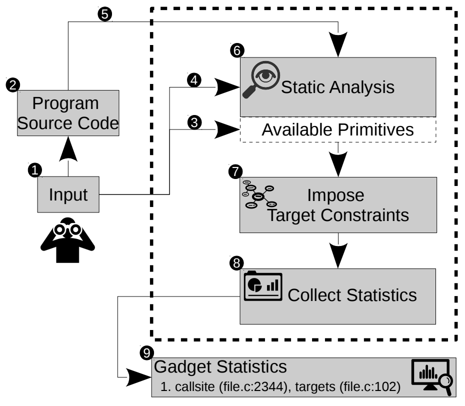

Figure 1: Design of LLVM-CFI.

Figure 1 depicts the main components of LLVM-CFI and the workflow used to analyze the source code of a potentially vulnerable program in order to determine CRA statistics, as follows. To use LLVM-CFI the analyst has to provide as input ❶ to LLVM-CFI a program's source code ❷. In addition, the analyst needs to choose the desired defense policies. A subset of defense policies can be selected by switching on flags inside the LLVM-CFI source code, which can also be implemented as compiler flags, if desired. Fundamentally, the analysis performed by LLVM-CFI is dependent on the implemented defense models and on the available primitives. Therefore, the analyst can choose from the 8 policies currently available in the LLVM-CFI tool. In case a desired defense is not available, the analyst can extend the list of primitives ❸, and model his defense as a policy (set of constraints) in the analysis component of LLVM-CFI ❹. In order to do this, he needs to know about the analysis internals of LLVM-CFI. After selecting/modeling a defense, the analyst forwards the application's source code ❺ to LLVM-CFI which will analyze it using its static analysis component. During static analysis ❻ the previously selected defense will be applied when compiling the program source code. As the analysis is performed, each callsite is constrained with only the legitimate calltargets. Note that both the protected callsites as well as the legitimate calltargets per callsite are dependent on the currently selected defense model. The result of the analysis contains information about the residual target set for each individual callsite after a CFI policy was assessed ❼. This list contains a set of gadgets (callsites + calltargets) that can, given a certain defense model, be used to bend the control flow of the application. These target constraints are collected and clustered in the statistics collection component of LLVM-CFI ❽. Finally, at the end of the gadget collection phase, a list of calltargets containing potential usable gadgets statistics ❾ based on the currently applied defense(s) will be reported.

3.2 Analysis Primitives

LLVM-CFI provides the following program primitives, which are either collected or constructed during program compile time. These primitives are used by LLVM-CFI to implement static CFI policies and to perform calltarget constraint analysis. Briefly, the currently available primitives are as follows:

Virtual table hierarchy (see [20] for a more detailed definition) allows performing virtual table inheritance analysis of only virtual classes as only these have virtual tables. Finally, a class is virtual if it defines or inherits at least one virtual function.

Vtable set is a set of vtables corresponding to a single program class. This set is useful to derive the legitimate set of calltargets for a particular callsite. The set of calltargets is determined by using the class inheritance relations contained inside a program.

Class hierarchy (see Tip *et al.* [50] and Rossie *et al.* [42] for a more formal definition) can be represented as a class hierarchy graph with the goal to model inheritance relations between classes. Note that a real-world program can have multiple class hierarchies (*e.g.,* Chrome, Google's Web browser). Note that the difference between virtual table hierarchy and class hierarchy is that the class hierarchy contains both virtual and non-virtual classes, whereas the virtual table hierarchy can only be used to reason about the inheritance relations between virtual classes.

Virtual table entries allow LLVM-CFI to analyze the number of entries in each virtual table with the possibility to differentiate between virtual function entries, offsets in vtables, and thunks.

Vtable type is determined by the name of the vtable root for a given vtable. A vtable root is the last derived vtable contained in the vtable hierarchy.

Callsites are used by LLVM-CFI to distinguish between direct and indirect (object-based dispatch and function-pointer based indirect transfers) callsites.

Indirect callsites are based on: (1) object dispatches or (2) function pointer based calls. Based on these primitives, LLVM-CFI can establish different types of relations between callsites and calltargets (*i.e.,* virtual functions). At the same time, we note that it is possible to derive backwards relationships from calltargets to legitimate callsites based on this primitive.

Callsite function types allow LLVM-CFI to precisely determine the number and the type of the provided parameter by a callsite. As such, a precise mapping between callsites and calltargets is possible.

Function types allow LLVM-CFI to precisely determine the number of parameters, their primitive types and return type value for a given function. This way, LLVM-CFI can generate a precise mapping between compatible calltargets and callsites.

These primitives can be used as building blocks during the various analyses that LLVM-CFI can perform in order to derive precise measurements and a thorough assessment of a modeled static CFI policy. We note that in order to model other CFI defenses, other (currently not available) simple or aggregated analysis primitives may need to be added inside LLVM-CFI.

3.3 Constraints

The basic concept of any CRA is to divert the intended control flow of a program by using arbitrary memory write and read primitives. As such, the result of such a corruption is to bend [7] the control flow, such that it no longer points to the intended (legitimate) calltarget set. This means that the attacker can point to any memory address in the program. While this type of attack is still possible, we want to highlight another type of CRA in which the attacker uses the intended/legitimate per callsite target set. That is, the attacker calls inside this set and performs her malicious behavior by reusing calltargets which are protected, yet usable during an attack. As previously observed by others [51], CRA defenses try to mitigate this by mainly relying on one or two dimensions at a time, as follows:

Write constraints limit the attacker's capabilities to corrupt writable memory. If there is no defense in place, the attacker can essentially corrupt: pointers to data, non-pointer values such as strings, and pointers to code (*i.e.,* function pointers). In this paper, we do not investigate these types of defenses as these were already addressed in detail by Veen *et al.* [51]. Instead, we focus on target constraints as these represent a big class of defenses which in our assessment need a separate and detailed treatment. This obviously does not mean that our analysis results cannot be used in conjunction with dynamic write constraint assessing tools. Rather, our results represent a common ground truth on which runtime assessing tools can build their gadget detection analysis.

Target constraints restrict the legitimate calltarget set for a callsite which can be controlled by an attacker. With no target constraints in place, the target set for each callsite is represented by all functions located in the program and any linked shared library. The key idea is to reduce the wiggle room for the attacker such that he cannot target random callsites. As most of these defenses impose a one-to-*N* mapping, an attacker being aware of said mapping could corrupt the pointer at the callsite to bend [7] the control flow to legitimate targets in an illegitate order to achieve her malicious goals. Thus, all static CFI defenses impose target constraints.

Static Analysis. LLVM-CFI is based on the static analysis of the program which is represented in LLVM's intermediate representation (IR). The analysis is performed during link time optimization (LTO) inside the LLVM [28] compiler framework to detect callsites and legitimate callees under the currently analyzed CFI defense. LLVM-CFI uses the currently available primitives and the implemented defenses to impose target constraints for each callsite individually. Currently, eight defenses are supported, see Section 4, but this list can easily be extended since all defenses are based on similar mechanisms which are assessable during a whole program analysis.

Generic Target Constraints. As mentioned above, LLVM-CFI can be used to impose existing generic calltarget constraints (defenses) based on class hierarchy relations and callsites and calltarget type matching with different levels of precision depending on the currently modeled CFI defense. Further, LLVM-CFI allows extending and combining existing policies or applying them concurrently.

3.4 Describing CFI Defenses

In this section, we present our CFI defense formalization framework and how this was used to formalize eight static CFI defenses inside LLVM-CFI. These defenses are stemming from published research papers and are used to constrain forward edge program control flow transfers to point to only legitimate calltargets. Note that each CFI defense description is an idealized representation and very close to how the original CFI defense policy was implemented in each tool. Further, in case of the binary based CFI defenses, the CFI defense descriptions are more precise than their implementations in the respective tools. Lastly, note that each modeled defense was previously discussed with the original authors and only after the authors agreed with these descriptions we modeled them into LLVM-CFI. Next, we give the formal definitions of each of the CFI defenses as these were modeled inside LLVM-CFI and the descriptions of the performed analyses.

Symbol	Description
P	the analyzed program
Cs	set of all indirect callsites of P
Cs_{virt}	set of P virtual callsites
V	all virtual func. contained in a virtual table hierarchy
V_{sub}	a virtual table sub-hierarchy
v_t	a virtual table
v_e	a virtual table entry (virtual function)
vc_s	a virtual callsite
nv_f	a non-virtual function
v_f	a virtual function (virtual table entry)
C	a class hierarchy contained in P
C_{sub}	a class sub-hierarchy contained in P
c_s	an indirect callsite
nt_{pcs}	callsite's number and type of parameters
nt_{pct}	calltarget's number and type of parameters
F	set of all virtual and non-virtual functions in P
F_{virt}	set of all virtual functions in P
S	set of function signatures
M	calltarget matching set based on the policy rules

Table 1: Symbol descriptions.

Notation. Table 1 depicts the set of symbols used by LLVM-CFI to model CFI defenses. Note that M is determined by applying all rules defined by a CFI defense and represents, at the same time, the matching criteria for each policy. This means that LLVM-CFI increments the count of its analysis by one when such a match is found.

Bin Types. (TypeArmor) [52] We formalize this CFI policy Ψ as the tuple $\langle Cs, F, V, M \rangle$ where the relations hold: (1) $V \subseteq F$, (2) $v_e \in V$, (3) $nv_f \in F$, (4) $c_s \in C$, and (5) $M \subseteq Cs \times V \times F$.

LLVM-CFI's Analysis. For each indirect callsite c_s (1) count the total number of virtual table entries v_e which reside in each virtual table hierarchy V contained in program P, and also, (2) count the number of non-virtual functions nv_f residing in F, which need at most as many function parameters as provided by the callsite and up to six parameters. Further, if F contains multiple distinct virtual table hierarchies (islands) then continue to count them too and take them also into consideration for a particular callsite. An island is a virtual table hierarchy which has no father-child relation to another virtual table hierarchy contained in the program P.

Safe src types. (Safe IFCC) [49] We formalize this CFI policy Ψ as the tuple $\langle Cs, F, F_{virt} S, M \rangle$ where the relations hold: (1) $V \subseteq F$, (2) $v_f \in F_{virt}$, (3) $nv_f \in F$, (4) $nt_{pcs} \in S$, (5) $nt_{pct} \in S$, (6) $f_{rt} \in S$, (7) $c_s \in Cs$, and (8) $M \subseteq Cs \times F \times S$.

LLVM-CFI's Analysis. For each indirect callsite c_s count the number of virtual functions v_f and non-virtual functions nv_f located in the program P for which the number and type of parameters required by the calltarget nt_pct matches the number and type of parameters provided at the callsite nt_pcs. The function return type f_{rt} of the matching function is not taken into consideration. All parameter pointer types are considered interchangeable, *e.g.*, **int*** and **void*** pointers are considered interchangeable.

Src types. (IFCC/MCFI) [34] We formalize this CFI policy Ψ as the tuple $\langle Cs, F, F_{virt} S, M \rangle$ where the relations hold: (1) $V \subseteq F$, (2) $v_f \in F_{virt}$, (3) $nv_f \in F$, (4) $nt_{pcs} \in S$, (5) $nt_{pct} \in S$, (6) $f_{rt} \in S$, (7) $c_s \in Cs$, and (8) $M \subseteq Cs \times F \times S$.

LLVM-CFI's Analysis. For each indirect callsite c_s count the number of virtual functions and non-virtual functions located in the program F for which the number and type of parameters required at the calltarget nt_{pct} matches the number and type of arguments provided by the callsite nt_{pcs}. The return type of the matching function is ignored. Compared to *Safe src types*, this policy distinguishes between different pointer types. This means that these are not interchangeable and that the function signatures are more strict. Neither the return value of the matching function nor the name of the function are taken into consideration.

Strict src types. (vTrust) [57] We formalize this CFI policy Ψ as the tuple $\langle Cs, V, F, F_{virt}, S, M \rangle$ where the relations hold: (1) $V \subseteq F$, (2) $v_f \in F_{virt}$, (3) $ntf_{pcs} \in S$, (4) $f_{rt} \in S$, (5) $c_s \in Cs$, and (6) $M \subseteq Cs \times S \times F_{virt} \times V$.

LLVM-CFI's Analysis. For each indirect callsite c_s compute the function signature of the function called at this particular callsite using the number of parameters, their types, and the name of the function ntf_{pcs} (the literal name used in C/C++ without any class information attached). Match this function type identifier with each virtual function v_f contained in each virtual table hierarchy V of P. The name of the function is taken into consideration when building the hash, but not the function return type f_{rt}, as this can be polymorphic. We have a match when the signature of a function called by a callsite matches the signature of a virtual function v_f.

All vtables. (vTint) [58] We formalize this CFI policy Ψ as the tuple $\langle P, Cs, F_{virt}, V, M \rangle$ where the relations hold: (1) $V \subseteq F$, (2) $v_e \in V$, (3) $v_f \in F_{virt}$, (4) $c_s \in Cs$, and (5) $M \subseteq Cs \times V$.

LLVM-CFI's Analysis. For each indirect callsite cs count each virtual function v_f corresponding to a virtual table entry v_e contained in each virtual table present in the program P.

vTable hierarchy/island. (Marx) [37] We formalize this CFI policy Ψ as the tuple $\langle P, F_{virt}, C, Cs, V, M \rangle$ where the relations hold: (1) $V \subseteq F$, (2) $v_e \in V$, (3) $v_f \in F_{virt}$, (4) $v_t \in V$, (5) $V \in C$, (6) $C \in P$, (7) $c_s \in Cs$, and (8) $M \subseteq Cs \times V \times C$.

LLVM-CFI's Analysis. For each indirect callsite c_s count each virtual function v_f corresponding to each virtual table v_t entry v_e having the same index in the virtual table as the index determined at the callsite c_s by Marx. Perform this matching for each virtual table v_t where the index matches the index determined at the callsite c_s and which is located in the class hierarchy C which contains the class type of the dispatched object. Note that abstract classes are not taken in consideration within this policy, this can be recognized though by virtual tables having pure virtual function entries.

Sub-hierarchy. (VTV) [49] We formalize this CFI policy Ψ as the tuple $\langle P, F_{virt}, C, C_{sub}, V, M\rangle$ where the relations hold: (1) $v_t \in V$, (2) $V \subseteq C$, (3) $C \subseteq P$, (4) $C_{sub} \in C$, (5) $v_f \in F_{virt}$, (6) $vc_s \in P$, and (7) $M \subseteq Cs_{virt} \times C_{sub} \times V \times F_{virt}$.

LLVM-CFI's Analysis. For each virtual callsite vc_s build the class sub-hierarchy C_{sub} having as root node the base class (least derived class that the dispatched object can be of) of the dispatched object. From the classes located in the sub-hierarchy consider, for the currently analyzed callsite, each virtual table v_t. Further, within these virtual tables v_t's consider only the virtual function v_f entries located at the offset used by the virtual object dispatch mechanism. Next, count all virtual functions to which these entries point to.

Strict sub-hierarchy. (ShrinkWrap) [20] We formalize this CFI policy Ψ as the tuple $\langle P, F_{virt}, C, V, V_{sub}, M\rangle$ where the relations hold: (1) $V \subseteq C$, (2) $v_e \in V$, (3) $v_f \in F_{virt}$, (4) $v_t \in V$, (5) $V \subseteq C$, (6) $V_{sub} \subseteq V$, (7) $C \subseteq P$, (8) $c_s \in P$, and (9) $M \subseteq Cs_{virt} V \times V_{sub} \times F_{virt}$.

LLVM-CFI's Analysis. For each virtual callsite vc_s identify the virtual table v_t type used. Take this virtual table v_t from the base class C of the dispatched object and build the virtual table v_t sub-hierarchy V_{sub} having this virtual table v_t as root node. From the virtual tables in this v_t sub-hierarchy find the virtual function v_f entries located at the offset used by the virtual object dispatch mechanism for this particular callsite c_s. Next, count each virtual function v_f, to which these virtual table entries v_e point to. Finally, after LLVM-CFI computes for each callsite the total calltarget set count, as described above for each policy, it sums up all results for each callsite to generate several statistics.

4 IMPLEMENTATION

4.1 Data Collection and Aggregation

Collection. LLVM-CFI collects the virtual tables of a program in the Clang front-end and pushes them through the compilation pipeline in order to make them available during link-time optimization (LTO). For each virtual table, LLVM-CFI collects the number of entries. The virtual tables are analyzed and aggregated to virtual table hierarchies in a later step. Other data such as direct/indirect callsites and function signatures are collected during LTO.

Aggregation. Next, we present the program primitives which are constructed by LLVM-CFI. First, virtual table hierarchies are built based on the previously collected virtual table metadata within the Clang front-end. The virtual table hierarchies are used to derive relationships between the classes inside a program (class hierarchies), determine sub-hierarchy relationships and count, for example, how many virtual table entries (virtual functions) a certain virtual table sub-hierarchy has. Second, virtual table sets are constructed for mapping callsites to legitimate class hierarchy-based virtual calltargets. Third, callsite function types are constructed. These are composed of the number of parameters provided by a callsite, their types, and if the callsite is a void or non-void callsite. Lastly, function types are built. These are composed of the function name, the expected number of parameters, their types and an additional bit used to indicate if the function is a void or non-void function.

4.2 CFI Defense Modeling

LLVM-CFI implements a set of constraints for each modeled CFI-defense, which are defined as analysis conditions that model the behavior of each analyzed CFI-defense. These constraints are particular for each CFI-defense and operate on different primitives. More specifically, different constraints of a CFI-defense are implemented inside LLVM-CFI. The steps for modeling a CFI defense are addressed by answering the five questions listed in the following. (1) Which of LLVM-CFI's primitives are used by the policy? (2) Is there a nesting or subset relation between these primitives? (3) Does the policy rely on hierarchical metadata primitives? (4) What are the callsite/calltarget matching criteria? (5) How to count a callsite/calltarget match? Note that there is no effort needed to port LLVM-CFI from one policy to another as all policies can operate in parallel during compile time. As such, the measurement results obtained for each policy are written in one pass in an external file for later analysis.

Next, we provide a concrete example of how a CFI defense, *i.e.*, TypeArmor's *Bin types* policy [52], was modeled inside LLVM-CFI by following the steps mentioned above. For more details, see Section 3.4 for a description on how this policy works. More specifically, for TypeArmor the following applies. (1) The policy uses the callsite, indirect callsite, callsite function type, and function type primitives provided by LLVM-CFI. (2) From all functions contained in the program, we analyze only the virtual functions which expect up to six parameters to be passed by the callsite. Next, from all callsites, we filter out the ones which are not calling virtual functions and which provide more than six parameters to the calltarget. We check if the callsite is a void or non-void callsite. Further, we check if each analyzed calltarget is a void or non-void target. (3) The policy does not rely on hierarchical metadata. (4) A callsite matches a calltarget if it provides less or the same number of parameters as the calltarget expects. (5) In case the matching criteria holds, we increment the total count by one for each found match.

Lastly, these constraints are implemented as a LLVM compiler module pass performed during LTO. Thus, even with limited knowledge constraints of an CFI policy can be modeled by observing how other existing policies were implemented inside LLVM-CFI.

4.3 CFI Defense Analysis

LLVM-CFI performs for each implemented CFI defense a different analysis. Each defense analysis consists of one or more iterations through the program primitives which are relevant for the CFI defense currently being analyzed. Depending on the particularities of a defense, LLVM-CFI uses different previously collected program primitives. More specifically, class hierarchies, class sub-hierarchies, or function signatures located in the whole program or in certain class sub-hierarchy are individually analyzed. During a CFI-defense analysis, statistics are collected w.r.t. the number of allowed calltargets per callsite taking into account the previously modeled CFI-defense.

As such, for a certain CFI defense (*e.g.*, TypeArmor's CFI policy **Bin types**) it is required to determine a match between the number of provided parameters (up to six parameters) of each indirect callsite and all virtual functions present in the program (object inheritance is not taken into account) which could be the target (may consume up to six parameters) of such a callsite. In order to analyze this CFI defense and collect the statistics, LLVM-CFI visits all indirect callsites it previously detected in the program and all virtual functions located in all previously recuperated class hierarchies. Afterwards, each callsite is matched with potential calltargets (virtual functions). Lastly, after all virtual callsites/functions were visited, the generated information is shown to the analyst.

4.4 Implementation Details

We implemented LLVM-CFI as three link time optimization (LTO) passes and some code inside the Clang compiler to push metadata into the compiler's LTO. LLVM-CFI is built atop the Clang/LLVM (v.3.7.0) compiler [29] framework infrastructure. The implementation of LLVM-CFI is split between the Clang compiler front-end (part of the metadata is collected here), and several link-time passes, totaling 4.2 KLOC. LLVM-CFI supports separate compilation by relying on the LTO mechanism built in LLVM [29]. By using Clang, LLVM-CFI collects front-end virtual tables and makes them available during LTO. Next, virtual table hierarchies are built which are used to model different CFI defenses. Other LLVM-CFI primitives such as function types are constructed during LTO. Finally, each of the analyzed CFI defenses are separately modeled inside LLVM-CFI by using the previously collected primitives and aggregated data to impose the required defense constraints.

5 EVALUATION

In this section, we address the following research questions (RQs).

- **RQ1:** What type of metrics are supported by the LLVM-CFI framework (§5.1)?
- **RQ2:** What is the residual attack surface of NodeJS after applying independently eight CFI defenses (§5.2)? For answering this RQ, we performed a use case analysis focused on NodeJS.
- **RQ3:** What score would each of the analyzed CFI defenses get (§5.3)?
- **RQ4:** How can LLVM-CFI be used to rank CFI policies based on the offered protection level (§5.4)?
- **RQ5:** What is the residual attack surface for several real-world analyzed programs (§5.5)?
- **RQ6:** How can LLVM-CFI pave the way towards attack construction (§5.6)?

Test Programs. In our evaluation, we used the following real-world programs: Nginx [33] (Web server, usable also as: reverse proxy, load balancer, mail proxy and HTTP cache, v.1.13.7, C code), NodeJS [36] (cross-platform JavaScript run-time environment, v.8.9.1, C/C++ code), Lighttpd [27] (Web server optimized for speed-critical environments, v.1.4.48, C code), Httpd [21] (cross-platform Web server, v.2.4.29, C code), Redis [41] (in-memory database with in-memory key-value store, v.4.0.2, C code), Memcached [30] (general-purpose distributed memory caching system, v.1.5.3, C/C++ code), Apache Traffic Server [15] (modular, high-performance reverse proxy and forward proxy server, v.2.4.29, C/C++ code), and Chrome [17] (Google's Web browser, v.33.01750.112, C/C++ code).

Experimental Setup. The experiments were performed on an Intel i5-3470 CPU with 8GB of RAM running on the Linux Mint 18.3 OS. All experiments were performed ten times to provide reliable values. If not otherwise stated, we modeled each of the eight CFI defenses inside LLVM-CFI according to the policy descriptions provided in Section 3.4.

LLVM-CFI's CTR Analysis Capabilities. LLVM-CFI can conduct different types of analysis based on several metrics as such CFI policies can be compared w.r.t. different aspects. In this paper, we decided to focus on the CTR metric as it is one of the simple ones and it is comparable with the AIR, fAIR, AIA and other related metrics. Further, note that CTR is an example metric which does not fit all needs. Other supported metrics are presented in Section 5.1.

5.1 LLVM-CFI Supported Metrics

In this section, we present four novel CFI metrics, which can be used within LLVM-CFI in order to perform different types of CFI policy-related analysis. Note that another set of metrics was introduced in a recent survey by Burow *et al.* [6]. Complementing related work, LLVM-CFI helps to provide precise and reproducible measurement results when performing CFI-related investigations. This section introduces several alternatives to existing CFI metrics.

Symbol	Description
ics	indirect call site (*i.e.,* x86 `call` instruction)
irs	indirect return site (*i.e.,* x86 `ret` instruction)
P	program
VT	virtual table
VTI	virtual table inheritance
CH	class hierarchy
CFG	control flow graph
CG	code reuse gadget
CTR	indirect calltarget reduction
RTR	indirect return target reduction
fCGA	forward-edge based *CG* availability
bCGA	backward return-edge based *CG* availability

Table 2: Symbols and associated descriptions.

Table 2 contains the abbreviations which we used in Figure 2.

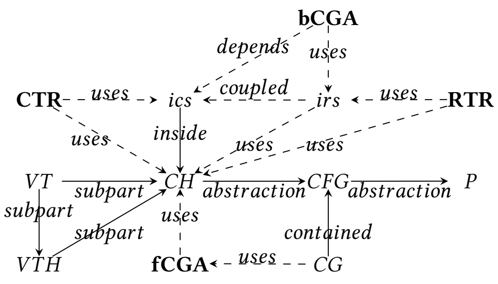

Figure 2: Our four metrics (bold text), & program primitives.

Figure 2 depicts the relationships between our four metrics (bold text) metrics and program metadata primitives. Next, we introduce our metrics that can be used within LLVM-CFI.

*Definition 5.1 (**CTR**).* Let ics_i be a particular indirect callsite in a program P, ctr_i is the total number of legitimate calltargets for an ics_i after hardening a program with a certain CFI policy.

Then, the *iCTR* metric is: $CTR = \sum_{i=1}^{n} ctr_i$. Note that the lower the value of CTR is for a given program, the more precise the CFI policy is. The optimal value of this metric is equal to the total number of callsites present in the hardened program. This means that there is a one-to-one mapping. We can also capture the distribution of the numbers of calltargets using min, max, and standard deviation functions. Minimum: $\min_i \{ctr_i\}$; Maximum: $\max_i \{ctr_i\}$; and Standard Deviation (SD): $CTR_{SD} = \sqrt{\frac{\sum_{i=1}^{n}(ctr_i - \overline{ctr_i})^2}{n}}$.

*Definition 5.2 (**RTR**).* Let irs_i be a particular indirect return site in the program P, then rtr_i is the total number of available return targets for each irs_i after hardening the backward edge of a program with a CFI policy.

Then, the RTR metric is: $RTR = \sum_{i=1}^{n} rtr_i$. Note that the lower the value of RTR is for a given program, the better the CFI policy is. The optimal value of this metric is equal to the total number of indirect return sites present in the hardened program. This means that there is a one-to-one mapping. Other key properties are: Minimum: $RTR_{MIN} = \min_i \{rtr_i\}$; Maximum: $RTR_{MAX} = \max_i \{rtr_i\}$; and Standard Deviation (SD): $RTR_{SD} = \sqrt{\frac{\sum_{i=1}^{n}(rtr_i - \overline{rtr_i})^2}{n}}$.

*Definition 5.3 (**fCGA**).* Let cgf_i be the total number of legitimate calltargets that are allowed and which contain gadgets according to a gadget finding tool. Then, the forward code reuse gadget availability $fCGA$ metric is: $fCGA = \sum_{i=1}^{n} cgf_i$.

Note that the lower the value of $fCGA$ is, the better the policy is. This means that every time a calltarget containing a code reuse gadget is protected by a CFI check, this gadget is not reachable. The reverse is true when the calltarget return contains a gadget and there are indirect control flow transfers which can call this indirect return site unconstrained.

*Definition 5.4 (**bCGA**).* Let cgr_i be the total number of legitimate callee return addresses which contain code gadgets according to a gadget finding tool. Then, the backward code reuse gadget availability $bCGA$ metric is: $bCGA = \sum_{i=1}^{n} cgr_i$.

Note that the lower the value of $bCGA$ is, the better the policy is. This means that every time when a calltarget return address, which contains a code reuse gadget that is protected by a CFI check, then this gadget is not reachable. The reverse is true when the calltarget return contains a gadget and there are indirect control flow transfers which can call this indirect return address in an unconstrained.

Important advantages of these metrics are: (1) provide absolute numbers without averaging the results, (2) can be used to assess backward-edge target set reduction, and (3) can be used to assess the forward-edge and backward-edge control flow transfers w.r.t. gadget availability.

So far, the available CFI metrics have failed to assess how likely an attack still is after a CFI policy was applied. Therefore, we presented examples of metrics that can be incorporated into our framework. The goal of these metrics is to point out that multiple CFI-related measurements are relevant and that these can be performed with LLVM-CFI depending on the type of CFI policy which one wants to analyze. Further, our metrics in contrast to others can be used to analyze the protection of backward edges and the availability of gadgets after a CFI policy was deployed.

One major benefit of our metrics is that LLVM-CFI can use these to assess CFI policies w.r.t. several dimensions (*e.g.*, forward-edge and backward-edge transfers, and gadget availability) and combine this information into meaningful data. Note that currently no available CFI metric can do this. Further, the security implications of these metrics can be used to not only tell how many targets per callsite or return site exist but also to correlate this information to gadgets which may be reachable after such an indirect transfer was executed. This can be achieved by using an additional gadget finding tool which may be used to search for CRA gadgets in order to map these within the analyzed binary. In this way, LLVM-CFI is capable of taking into account past, current and future attacks and assess their likelyhood after deploying a CFI policy.

Further, note that we did not use all four metrics in conjunction with the eight CFI policies assessed in this paper as we wanted to thoroughly focus only on forward-edge transfers in this work. Moreover, we are not aware of any CFI policy assessing tool which can take into account all the dimensions introduced by our four metrics. Lastly, by using these metrics, experiments become more reproducible and results better comparable.

5.2 NodeJS Use Case

In this section, we analyze the residual attack surface after each of the eight CFI policies was applied individually to NodeJS. Note that three out of the eight assessed CFI policies used in the following tables are the same as reported by Veen *et al.* [51] (we share the same names). For the other five CFI policies, we use names which reflect their particularities. We selected NodeJS as it is a very popular real-world application and it contains both `C` and `C++` code. As such, LLVM-CFI can collect results for the `C` and `C++` related CFI polices.

Table 3 depicts the static target constraints for the NodeJS program under different static CFI calltarget constraining policies. Further, Table 3 provides the minimum and maximum values of virtual calltargets which are available for a virtual callsite after one of the eight CFI policies is applied. MKSnapShot contains the Chrome V8 engine and is used as a shared library by NodeJS after compilation. We decided to add MKSnapshot in Table 3 as this component is used considerably by NodeJS and represents a source of potential calltargets. The NodeJS results were obtained after static linking of MKSnaphot. Further, the target median entries in Table 3 (left hand side) indicate the median values obtained for independently applying one of the eight CFI policies to NodeJS. For both NodeJS and MKSnaphot, the best median number of residual targets is obtained using the following policies: (1) *vTable hierarchy*, (2) *sub-hierarchy*, and (3) *strict sub-hierarchy*. The results indicate that these three CFI policies provide the lowest attack surface, while the highest attack surface is obtained for the *bin types* policy, which allows the highest number of virtual and non-virtual targets.

The targets distribution in Table 3 (right hand side) shows the minimum, maximum and 90 percentile results for the same eight policies as before. While the minimum value is 0, the highest values for both NodeJS and MKSnapshot are obtained for the *bin types* policy, while the lowest values are obtained for the following policies: (1) *vTable hierarchy*, (2) *sub-hierarchy*, and (3) *strict sub-hierarchy*. Further, the 90th percentile results show that on the tail end of the distribution, a noticeable difference between the three previously mentioned policies exists. We can observe that for these critical callsites the *strict sub-hierarchy* policy provides the least amount of residual targets and therefore the best protection against CRAs. Meanwhile, the 90th percentile results for the *strict src type* and *vTable hiearchy* policies indicate that the residual attack surface might still be sufficient for the attacker.

5.3 CFI Defenses Scores

Figure 3 depicts the scores obtained by each of the eight policies, which were analyzed for the Chrome Web browser. The scores are depicted in logarithmic scale in order to better compare the values against each other. The optimal score has the value of one depicted on the left hand side Y axis. We opted to depict the values for only

P	Targets Median					Targets Distribution					
						NodeJs			MKSnapshot		
	NodeJS	*MKSnaphot*	*Total*	*Min*	*Max*	*Min*	*90p*	*Max*	*Min*	*90p*	*Max*
(1)	21,950	15,817	15,817	15,817	21,950	12,545	30,179	32,478	8,714	21,785	23,376
	(21,950)	(15,817)	(20,253)	(15,817)	(21,950)	(885)	(30,179)	(32,478)	(244)	(21,785)	(23,376)
(2)	2,885	2,273	2,273	2,273	2,885	0	5,751	5,751	1	4,436	4,436
	(88)	(495)	(139)	(88)	(21,950)	(0)	(5,751)	(5,751)	(0)	(4,436)	(4,436)
(3)	1,511	1,232	1,232	1,232	1,511	0	5,751	5,751	1	4,436	4,436
	(56)	(355)	(139)	(56)	(355)	(0)	(5,751)	(5,751)	(0)	(4,436)	(4,436)
(4)	3	2	3	2	3	0	499	730	0	507	756
(5)	6,128	2,903	6,128	2,903	6,128	6,128	6,128	6,128	2,903	2,903	2,903
(6)	2	1	2	1	2	0	54	243	0	16	108
(7)	2	1	1	1	2	0	7	243	0	11	108
(8)	2	1	1	1	2	0	6	243	0	9	108

Table 3: Legitimate calltargets per callsite for each of the eight CFI policies for NodeJS after each CFI defense was individually applied. The values not contained in round brackets are obtained for only virtual callsites and all targets (*i.e.,* virtual and non-virtual), while the values in round brackets are obtained for all indirect callsites (*i.e.,* virtual and function pointer based calls) and all targets. For the *Bin types*, *Safe src types*, and *Src types* policies depicted above the targets can be virtual or non-virtual, for the remaining policies the targets inherently can only be virt. functions. Targets median: (min. and max.) number of legal function targets per callsite. Target distribution: minimum/90th percentile/maximum number of targets per callsite. This 90th percentile is determined by sorting the values in ascending order, and picking the value at 90%. Thus 90% of the sorted values have a lower or equal value to 90th percentile. P: Policy (Static target constraints), *(1)* Bin types [52], *(2)* Safe src types [49], *(3)* Src types [34], *(4)* Strict src types [57], *(5)* All virtual tables [58], *(6)* virtual Table hierarchy [37], *(7)* Sub-hierarchy [5], and *(8)* Strict sub-hierarchy [20].

P	*Value*	Callsites	Targets Baseline		Virtual Function Targets							
		Write cons.	Base all fun	Base vFunc	*(1)*	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*	*(7)*	*(8)*
	Min				12,545 (1,956)	0 (0)	0 (0)	0 (0)	6,128	0	0	0
	90p				30,179 (4,078)	5,751 (810)	5,751 (810)	499 (10)	6,128	54	7	6
JS	Max	none	32,478	6,300	32,478 (4,455)	5,751 (810)	5,751 (810)	730 (243)	6,128	243	243	243
	Med				21,950 (3,106)	2,885 (426)	1,511 (121)	3 (3)	6,128	2	2	2
	Avg				19,395 (2,793)	2,406 (414)	2,113 (354)	86 (12)	6,128	14	8	8
	Min				2,608 (232)	1 (0)	1 (0)	0 (0)	788	0	0	0
	90p				4,085 (546)	1,315 (97)	1,315 (97)	17 (13)	788	34	7	7
TS	Max	none	6,201	796	6,201 (710)	1,315 (159)	1,315 (159)	18 (16)	788	42	18	18
	Med				2,608 (232)	1,315 (97)	1,315 (97)	17 (13)	788	7	1	1
	Avg				3,122 (321)	928 (76)	923 (74)	11 (9)	788	10	3	3
	Min				97,041 (37,873)	0 (0)	0 (0)	0 (0)	68,560	0	0	0
	90p				201,477 (63,816)	64,315 (24,661)	64,315 (24,661)	48 (30)	68,560	192	25	15
C	Max	none	232,593	78,992	232,593 (71,000)	64,315 (24,661)	64,315 (24,661)	3,029 (509)	68,560	4,486	4,486	4,486
	Med				97,041 (37,873)	8,672 (4,593)	7,633 (4,593)	3 (2)	68,560	6	2	2
	Avg				128,452 (45,731)	29,315 (11,119)	29,127 (11,013)	57 (19)	68,560	78	37	32

Table 4: Legitimate calltargets per callsite for only virtual callsites and for only the C++ programs after each CFI defense was individually applied. *Baseline all func.* represents the total number of functions, while *Baseline virtual func.* represents the number of virtual functions. The first four policies, from left to right in italic font (*Bin types*, *Safe src types*, *Src types*, and *Strict src types*) allow virtual or non-virtual targets, while the remaining four policies inherently allow only virtual targets. This is not a limitation of LLVM-CFI but rather how these were intended, designed and used in the original tools from where these are stemming. The values in round brackets show the theoretical results after adapting the first four policies to only allow virtual targets. Each table entry contains five aggregate values: minimal, 90th percentile: minimum/90th percentile/maximum, maximal, median and average (Avg) number of targets per callsite. P: program, JS: NodeJS, TS: Traffic Server, C: Chrome, *(1)* - *(8)* see Table 3.

the Chrome browser since this represents the largest (approx. 10 million LOC) analyzed program. The numbers on the gray shaded bars represent the 90th percentile values, while the values on the black shaded bars represent the average values for the Chrome Web browser. These values are reported in Table 4 on the last row from top to bottom for the Chrome browser as well. The optimal score is one and means that each callsite is allowed to target a single calltarget. This is the case only during runtime. The lower the bar is or the closer the value is to one, the better the score is.

The best score w.r.t. the 90th percentile and average values is obtained for the *strict sub-hierachy*, which appears to be the best CFI defense from the eight analyzed policies. It is interesting to note that the best function signature based policy *strict src types* has a slightly worse score than the second class based CFI defense (*sub-hierarchy*). Lastly, note that these CFI-based forward-edge

P	*(1)*			*(2)*			*(3)*			*(4)*			*(5)*			*(6)*			*(7)*			*(8)*		
	Avg	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*
JS	59.72	21.0	92.92	7.41	6.32	17.71	6.51	6.44	17.71	0.26	0.54	1.54	97.27	0.0	97.27	0.23	0.63	0.86	0.13	0.46	0.11	0.13	0.46	0.1
TS	50.35	15.79	65.88	14.97	8.89	21.21	14.89	9.01	21.21	0.18	0.12	0.27	98.99	0.0	98.99	1.26	1.27	4.27	0.34	0.51	0.88	0.34	0.51	0.88
C	55.23	19.08	86.62	12.6	12.16	27.65	12.52	12.22	27.65	0.02	0.11	0.02	86.79	0.0	86.79	0.1	0.43	0.24	0.05	0.41	0.03	0.04	0.41	0.02
Avg	55.1	18.62	81.8	11.66	9.12	22.19	11.3	9.22	22.19	0.15	0.25	0.61	94.35	0.0	94.35	0.53	0.77	1.79	0.17	0.46	0.34	0.17	0.46	0.33

Table 5: Normalized results with the baseline (B) using only virtual callsites. Note that virtual callsites can be used for all eight assesses CFI policies as these were designed in the original papers to be used for these types of callsites as well. Baseline: Total number of possible virtual targets. Each entry contains three aggregate values: average-, standard deviation (SD) and 90th percentile number of targets per callsite. The lower the *Average* value is the better the CFI defense is. P: program, JS: NodeJS (Baseline 6.3K), TS: Traffic Server (Baseline 796), C: Chrome (Baseline 78,992).

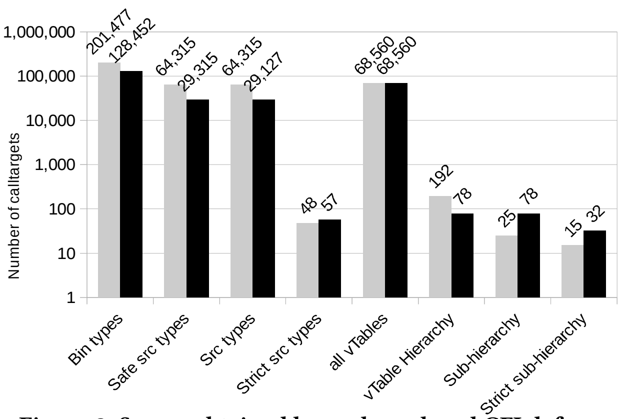

Figure 3: Scores obtained by each analyzed CFI defense.

P	B	Bin types			Safe src types			Src types		
		Avg	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*
a	32,478	64.0	20.43	92.92	3.82	5.83	17.71	3.38	5.64	17.71
b	6,201	54.03	18.76	87.89	13.54	9.27	21.21	13.46	9.36	21.21
c	232,593	56.83	19.84	86.62	11.71	12.11	27.65	11.64	12.16	27.65
d	1,949	52.18	26.5	92.0	2.7	3.01	8.21	2.46	3.01	8.21
e	594	65.25	27.81	97.98	2.94	3.18	7.41	2.93	3.19	7.41
f	225	69.75	7.11	68.89	1.0	0.97	0.89	1.0	0.97	0.89
g	1,270	54.91	24.85	92.28	6.38	4.56	11.73	6.36	4.57	11.73
h	2,880	65.19	16.51	84.62	1.25	2.52	1.88	1.2	2.52	1.88
Avg	34,773	60.3	34.39	87.9	5.4	5.18	12.09	5.3	5.17	12.08

Table 6: Normalized results using all indirect callsites.

policies are not optimal (*i.e.,* they provide values larger than one) and the desired goal is to develop policies, which provide one-to-one mappings similar to shadow stack based techniques.

5.4 Ranking of CFI Policies

In this section, we normalize the results presented in RQ2 using the *baseline* values (*i.e.,* the number of possible target functions), in order to be able to compare the assessed CFI policies against each other w.r.t. calltarget reduction. This allows LLVM-CFI to compare the analyzed CFI defenses on programs with different sizes and complexities which would not be possible otherwise.

Table 5 depicts the average, standard deviation and 90th percentile results obtained after analyzing only virtual callsites. Unless stated otherwise, we use the *CTR* metric. For these callsites, all eight CFI policies can be assessed. Here, we calculated the average over the three C++ programs after normalization. By considering these aggregate average values, the eight policies can be ranked (from best (smallest aggregate average) to worst (highest aggregate average)) as follows: (1) *strict src types* (0.15), (2) *strict sub-hierarchy* (0.17), (3) *sub-hierarchy* (0.17), (4) *vTable hierarchy* (0.53), (5) *src types* (11.3), (6) *safe src types* (11.66), (7) *bin types* (55.1), and (8) *all vTables* (94.35).

From the class hierarchy-based policies *strict sub-hierarchy* performed best in all three aggregate results (Avg, SD and 90th percentile). In comparison, *strict sub-hierarchy* performs better w.r.t. average and standard deviation but worse w.r.t. 90th percentile. The results indicate that these two policies are the most restrictive, but a clear winner in all evaluated criteria cannot be determined.

Table 6 depicts (similar to Table 5) normalized results with the difference that all indirect callsites (both virtual and pointer based) are analyzed. Shortnames: B: baseline, a) NodeJS, b) Apache Traffic Server, c) Google's Chrome, d) Httpd, e) LightHttpd, f) Memcached, g) Nginx, h) Redis. Thus, the *baseline* values used for normalization include virtual and non-virtual targets. By taking into account the aggregate averages and the standard deviation of the three policies in Table 6, we can rank the policies as follows (from best to worse): (1) *src types* (Avg 5.3 and SD 5.17), (2) *safe src types* (Avg 5.4 and SD 5.18), and (3) *bin types* (Avg 60.3 and SD 34.39). In contrast, by considering the 90th percentile values, we conclude that for the most vulnerable 10% of callsites, *bin types* only restricts the target set to 87.9% of the unprotected target set. As such, these callsites essentially remain unprotected. Meanwhile, the *safe src type* and *src type* policies restrict to only around 12% of the unprotected target set.

5.5 General Results

Table 7 depicts the results for the three policies which can provide protection for both C and C++ programs. Abbreviations: P: program, (1) bin types, (2) safe src types, (3) src types; a) NodeJS, b) Apache Traffic Server, c) Google's Chrome, d) Httpd, e) LightHttpd, f) Memcached, g) Nginx, and h) Redis. In contrast to Table 4, all indirect calls are taken into account (including virtual calls). Therefore, the targets can be virtual or non-virtual. Intuitively, the residual attack surface grows with the size of the program. This can be observed by comparing the results for large (*e.g.,* Chrome) with smaller (*e.g.,* Memcached) programs.

In contrast, Table 4 depicts the overall results obtained after applying the eight assessed CFI policies to virtual callsites only. The first four policies (italic font) cannot differentiate between virtual and non-virtual calltargets. Therefore, for these policies the baseline of possible calltargets includes all functions (both virtual and non-virtual). This is denoted with *baseline all func.* Since the remaining four policies can only be applied to virtual callsites, they restrict

P	*Value*	*Callsite write cons.*	*Baseline all func.*	**Targets (Non-) & virt. func.** *(1)*	*(2)*	*(3)*
a	Min	none	32,478	885	0	0
	90p			30,179	5,751	5,751
	Max			32,478	5,751	5,751
	Med			21,950	88	56
	Avg			20,787	1,242	1,099
b	Min	none	6,201	357	0	0
	90p			5,450	1,315	1,315
	Max			6,201	1,315	1,315
	Med			2,608	1,315	1,315
	Avg			3,350	840	835
c	Min	none	232,593	3,612	0	0
	90p			201,477	64,315	64,315
	Max			232,593	64,315	64,315
	Med			97,041	8,672	7,394
	Avg			132,182	27,238	27,074
d	Min	none	1,949	99	0	0
	90p			1,793	160	160
	Max			1,915	160	160
	Med			1,070	18	16
	Avg			1,017	53	48
e	Min	none	594	37	0	0
	90p			582	44	44
	Max			582	44	44
	Med			395	6	6
	Avg			388	17	17
f	Min	none	225	92	0	0
	90p			155	2	2
	Max			221	17	17
	Med			155	2	2
	Avg			157	2	2
g	Min	none	1,270	422	1	1
	90p			1,172	149	149
	Max			1,259	149	149
	Med			719	75	75
	Avg			697	81	81
h	Min	none	2,880	1,266	1	1
	90p			2,437	54	54
	Max			2,635	391	391
	Med			1,994	16	14
	Avg			1,877	36	35

Table 7: Results for virtual and pointer based callsites.

the possible calltargets to only virtual functions. Thus, the baseline for these policies includes only virtual functions (*baseline virtual function*). For a better comparison between the first and second categories of policies, we also calculated the target set when restricting the first four policies to only allow virtual callsites. For *bin types*, *safe src types*, *src types*, and *all vTables* the results indicate that there is *no* protection offered. The three-class hierarchy-based policies perform best when considering the median and average results. In addition, the *strict src type* policy performs surprisingly well, especially after restricting the target set to only virtual functions.

5.6 Towards Automated CRA Construction

In this section, we show how LLVM-CFI is used to automate one step of a COOP-like attack, namely finding protected targets which can be legitimately called. This attack bypasses a state-of-the-art CFI policy-based defense, namely VTV's *sub-hierarchy policy*. This case study is architecture independent, since LLVM-CFI's analysis is performed at the IR level during LTO time in LLVM. Note that LLVM IR code represents a higher level representation of machine code (metadata), thus our results can be applied to other architectures (*e.g.,* ARM) as well. Our case study assumes an ideal implementation of VTV/IFCC. Breaking the ideal instrumentation shows that the defense can be bypassed in any implementation. More specifically, we present the required components for a COOP attack by studying the original COOP attack [46] against the Firefox Web browser and demonstrate that such an attack is easier to perform when using LLVM-CFI.

For example, the original COOP attack presented by Schuster *et al.* [46] consists of the following four steps: (1) a buffer overflow filled with six fake counterfeit objects by the attacker, (2) precise knowledge of the Firefox `libxul.so` shared library layout, (3) knowledge about a COOP dispatcher and other gadgets (`ML-G`) resides in `libxul.so`, and (4) how to pass information from one gadget to the other in order to open a Unix shell. As demonstrated by COOP, the attacker first needs to find an exploitable memory corruption (*e.g.,* buffer overflow, etc.) and fill it with fake objects. Next, the attacker calls different gadgets (virtual C++ functions) located in `libxul.so`. Note that these functions would be in the benign execution not callable as these reside in distinct class hierarchies. Further, with fine-grained CFI defenses in place these calltargets would be protected during an attack. LLVM-CFI helps with identifying the protected targets (see step (3) above) and if desired the attacker can use other targets depending on his goals and the type of deployed CFI defense.

As such, we assume that NodeJS contains an exploitable memory vulnerability (*i.e.,* buffer overflow), and that the attacker is aware of the layout of the program binary. Next, we assume that the attacker wants to bend the control flow to only per callsite legitimate calltargets since in this way he can bypass the in-place CFI policy. Next, the attacker wants to avoid calling targets located in other program class hierarchies or protected targets. Therefore, he needs to know which calltargets are legitimate for each callsite located in the main NodeJS binary and which targets are protected.

CS	**#**	**Base only vFunc**	**Base all func**	**Eight Target Policies** *(1)*	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*	*(7)*	*(8)*
a	5	6,300	32,478	31,305	4	4	1	6,128	1	1	1
b	2	6,300	32,478	21,950	719	719	49	6,128	57	53	49
c	3	6,300	32,478	27,823	136	136	1	6,128	1	1	1
d	1	6,300	32,478	12,545	810	810	1	6,128	72	12	12
e	1	6,300	32,478	1,956	810	810	1	6,128	72	13	13
f	1	6,300	32,478	1,956	810	810	6	6,128	20	19	19
g	3	6,300	32,478	1,956	810	810	6	6,128	20	19	19
h	2	6,300	32,478	3,106	35	35	8	6,128	48	13	5
i	2	6,300	32,478	3,106	2,984	2,984	49	6,128	53	53	49
j	2	6,300	32,478	3,106	719	719	49	6,128	53	53	19

Table 8: Ten controllable callsites & their legitimate targets under the *Sub-hierarchy* CFI defense. #: passed parameters. CS: Ten controllable callsites, for *(1)-(8)*, see Table 3 caption.

Table 8 depicts ten controllable callsites (in total LLVM-CFI found thousands of controllable callsites) for which the legitimate target set, depending on the used CFI policy (1-8), ranges from one to 31,305 calltargets: a) debugger.cpp:1329:33, b) protocol.cpp:839:60, c) schema.cpp:133:33, d) handle_wrap.cc:127:3, e) cares_wrap.cc:642:5, f) node_platform.cc:25:5, g) node_http2_core.h:417:5, h) tls_wrap

.cc:771:10, i) protocol.cpp:839:60, and j) protocol.cpp:836:60. For each calltarget, LLVM-CFI provides: file name, function name, start address and source code line number such that it can be easily traced back in the source code file. The calltargets (right hand side in Table 8 in italic font) represent available calltargets for each of the eight assessed policies. Further, the information shown in Table 8 demonstrates the usefulness of LLVM-CFI when used by an analyst. By using LLVM-CFI it can drastically reduce the time needed to search for COOP-like protected and unprotected gadgets after a certain CFI policy was deployed. Lastly, this helps to better tailor attacks w.r.t. deployed CFI-based defenses.

6 RELATED WORK

6.1 Defense Assessment Metrics

AIR [59], fAIR [49], and AIA [16] metrics have limitations (see Carlini *et al.* [7]) and are currently the available CFI defense assessment metrics which can be used to compare the protection level offered by state-of-the-art CFI defenses w.r.t. only forward-edge transfers. These metrics provide average values which shed limited insight into the real offered protection level and thus cannot be reliably used to compare CFI-based defenses. Most recently, ConFIRM [56] also attempted to evaluate CFI, especially the compatibility, applicability, and relevance of CFI protections with a set of microbenchmarking suites. In contrast, LLVM-CFI is not a benchmark suite but rather a framework for modeling CFI defenses and comparing them against each other w.r.t. protection level these offer. Burow *et al.* [6] propose two metrics: (1) a qualitative metric based on the underlying analysis provided by each of the assessed techniques, and (2) a quantitative metric that is the product of the number of equivalence classes (EC) and the inverse of the size of the largest class (LC). In contrast, we propose LLVM-CFI, a CFI defense assessment framework and *CTR*, a new CFI defense assessment metric based on absolute forward-edge reduction set analysis, without averaging the results. *CTR* provides precise measurements and facilitates comprehensive CFI defense comparison.

6.2 Static Gadget Discovery

Wollgast *et al.* [55] present a static multi-architecture gadget detection tool based on the analysis of the intermediate language (IL) of VEX, which is part of the Valgrind [45] programming debugging framework. The tool can find a series of CFI-resistant gadgets. Compared to LLVM-CFI, both tools leave the gadget chain building as a manual effort. In contrast, when using LLVM-CFI, it is possible to define a specific CFI-defense policy and search for available gadgets, while the tool of Wollgast et al. specifies CFI-resistant gadgets by defining their boundaries (start and end instructions). These have to conform to some constraints and respect the normal program control flow of the program in order to be considered CFI resistant. These types of gadgets are more thoroughly described by Goktas *et al.* [19], Schuster *et al.* [47].

RopDefender [12], ROPgadget [43], and Ropper [44] are non-academic gadget detection tools based on binary program analysis. These tools are used to search inside program binaries with the goal to find consecutive machine code instructions, which are similar to a previously specified set of rules that define a valid gadget. While allowing a fast search, these tools cannot detect defense-aware gadgets, since these tools do not model the defense applied to the program binary. As such, these tools cannot determine which gadgets are usable after a certain defense was applied.

6.3 Dynamic Attack Construction

Newton [51], is a runtime binary analysis tool which relies on taint analysis to help significantly simplify the detection of code reuse gadgets defined as callsite and legal calltarget pairs. Newton can model part of the byte memory dependencies in a given program. Newton is further able to model a series of code reuse defenses by not focusing on a specific attack at a time. Newton is able to craft attacks in the face of several arbitrary memory write constraints. A substantial difference compared to Newton is that LLVM-CFI uses program source code, which captures more precise information about the caller-callee pair than binary analysis based approaches.

StackDefiler [10] presents a set of stack corruption attacks that leverage runtime object allocation information in order to bypass fine-grained CFI defenses. Based on the fact that Indirect Function-Call Checks (IFCC) [49] (also valid for VTV) spill critical pointers onto the stack, the authors show how CRAs can be built even in presence of a fine-grained CFI defense. Compared to LLVM-CFI, which is based on control flow bending to legitimate targets, StackDefiler shows an alternative approach for crafting CRAs. More specifically, the authors show that information disclosure poses a severe threat and that shadow stacks which are not protected through memory isolation are an easy target for a skilled attacker.

ACICS [14] gadgets are detected during runtime by the ADT tool, in a similar way as Newton detects gadgets. Note that the ACICS gadgets are more constrained then those of Newton. For example, only attacks where the function pointer and arguments are corruptible on the heap or in global memory are taken into consideration. Similar to LLVM-CFI, the ADT tool is able to craft an attack in the face of IFCC's CFI defense policy by finding pairs of indirect callsites that match to certain functions which can be corrupted during runtime. In contrast, LLVM-CFI, is not program input dependent as it is not a runtime tool. Therefore, it can find all corruptible indirect callsite and function pairs under a certain modeled CFI policy.

Revery [53] crafts attacks by analyzing a vulnerable program and by collecting runtime information on the crashing path as for example taint attributes of variables. Revery fails in some cases to generate an attack due to complicated defense mechanisms of which the tool is not aware. Lastly, in some cases, Revery cannot generate exploits due to dynamic decisions which have to be made during exploitation.

7 CONCLUSION

We have presented LLVM-CFI, a control-flow integrity (CFI) defense analysis framework that allows an analyst to thoroughly compare conceptual and deployed CFI defenses against each other. LLVM-CFI paves the way towards automated control-flow hijacking attack construction. We implemented LLVM-CFI, atop of the Clang/LLVM compiler framework which offers the possibility to precisely analyze real-world programs during compile time. We have released the source code of LLVM-CFI. By using LLVM-CFI, an analyst can drastically cut down the time needed to search for gadgets which are compatible with state-of-the-art CFI defenses contained in many real-world programs. Our experimental results indicate that most of the CFI defenses are too permissive. Further, if an attacker does not only rely on the program binary when searching for gadgets and has a tool such as LLVM-CFI at hand to analyze the vulnerable application, then many CFI defenses can easily be bypassed.

ACKNOWLEDGMENTS

We are grateful to Elias Athanasopoulos, our shepherd, who provided highly valuable comments that significantly improved our paper. Further, we also would like to thank the anonymous reviewers for their constructive feedback. Zhiqiang Lin is partially supported by US NSF grant CNS-1834215 and ONR award N00014-17-1-2995. Gang Tan is partially supported by US NSF grant CNS-1801534 and ONR award N00014-17-1-2539.

REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005. Control Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2009. Control Flow Integrity Principles, Implementations, and Applications. In *Transactions on Information and System Security (TISSEC).*

[3] M. Backes and S. Nuerenberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the USENIX Security Symposium (USENIX Security).*

[4] BlueLotus. 2015. BlueLotus Team, bctf challenge: Bypass vtable read-only checks. https://github.com/ctfs/write-ups-2015/tree/master/bctf-2015/exploit/zhongguancun.

[5] D. Bounov, R. G. Kici, and S. Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS).*

[6] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. In *CSUR.*

[7] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T.-R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the USENIX Security Symposium (USENIX Security).*

[8] N. Carlini and D. Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the USENIX Security Symposium (USENIX Security).*

[9] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. 2014. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS).*

[10] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[11] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[12] L. Davi, A.-R. Sadeghi, and M. Winandy. 2011. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS).*

[13] M. Elsabagh, D. Fleck, and A. Stavrou. 2017. Strict Virtual Call Integrity Checking for C ++ Binaries. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS).*

[14] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskosr. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[15] Apache Foundation. 2019. Apache Traffic Server. http://trafficserver.apache.org/.

[16] X. Ge, N. Talele, M. Payer, and T. Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the European Symposium on Security and Privacy (Euro S&P).*

[17] Google. 2019. Google Chromium. https://www.chromium.org/.

[18] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy.* ACM, Scottsdale, Arizona, USA.

[19] E. Göktas, E. Athanasopoulos, and H. Bos. 2014. Out Of Control: Overcoming Control-Flow Integrity. In *Proceedings of the Symposium on Security and Privacy (S&P).*

[20] I. Haller, E. Göktas, E. Athanasopoulos, G. Portokalidis, and H Bos. 2015. ShrinkWrap: VTable Protection without Loose Ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC).*

[21] Httpd. 2019. Httpd. https://httpd.apache.org/docs/2.4/programs/httpd.html.

[22] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[23] S. Krahmer. 2005. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. https://users.suse.com/~krahmer/no-nx.pdf.

[24] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014. Code-Pointer Integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[25] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng. 2015. Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. In *Proceedings of IEEE Trustcom/BigDataSE/ISPA.*

[26] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. 2016. Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC).*

[27] Lighthttpd. 2019. Lighthttpd. https://www.lighttpd.net/.

[28] LLVM. 2017. The LLVM Compiler Infrastructure. https://llvm.org/.

[29] LLVM. 2018. Clang/LLVM compiler framework. https://clang.llvm.org/.

[30] Memcached. 2019. Memcached. https://memcached.org/.

[31] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. 2018. CFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID).*

[32] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert. 2018. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[33] Nginx. 2019. Nginx. https://nginx.org/en/.

[34] B. Niu and G. Tan. 2014. Modular Control-Flow Integrity. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).*

[35] B. Niu and G. Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[36] NodeJS. 2019. NodeJS. https://nodejs.org/en/.

[37] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS).*

[38] A. Prakash, X. Hu, and H. Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS).*

[39] A. Pslyak. 1997. Return-into-libc overflow exploit. https://seclists.org/bugtraq/1997/Aug/63.

[40] G. Ramalingam. 1994. The Undecidability of Aliasing. In *Transactions on Programming Languages and Systems (TOPLAS).*

[41] Redis. 2019. Redis. https://redis.io/.

[42] J. Rossie Jr. and D. Friedman. 1995. An Algebraic Semantics of Subobjects. In *Proceedings of the Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[43] J. Salwan. 2011. ROPgadget - Gadgets Finder and Auto-roper. http://shell-storm.org/project/ROPgadget/.

[44] S. Schirra. 2017. Ropper. https://github.com/sashs/Ropper.

[45] S. Schirra. 2018. Valgrind Home. http://valgrind.org/.

[46] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the Symposium on Security and Privacy (S&P).*

[47] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. 2014. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proceedings of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID).*

[48] H. Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (On the x86). In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[49] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *Proceedings of the USENIX Security Symposium (USENIX Security).*

[50] F. Tip, J.-D. Choi, J. Field, and G. Ramalingam. 1996. Slicing Class Hierarchies in C++. In *Proceedings of the Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[51] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[52] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the Symposium on Security and Privacy (S&P).*

[53] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In *Proceedings of the Conference on Computer and Communications Security (CCS).*

[54] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. 2016. Shuffler: Fast and Deployable Continous Code Re-Randomization. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[55] P. Wollgast, R. Gawlik, B. Garmany, B. Kollenda, and T. Holz. 2016. Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[56] X. Xu, M. Ghaffarinia, W. Wang, K. Hamlen, and Z. Lin. 2019. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, 1805–1821.

[57] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. 2016. vTrust: Regaining Trust on Virtual Calls. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[58] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. 2015. vTint: Protecting Virtual Function TablesÍntegrity. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[59] M. Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.