τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries

Paul Muntean[1[0000−0002−2462−7612]], Matthias Fischer[1],
Gang Tan[3[0000−0001−6109−6091]], Zhiqiang Lin[2[0000−0003−1320−1015]],
Jens Grossklags[1[0000−0003−1093−1282]], and Claudia Eckert[1]

[1] Technical University of Munich, Germany
[2] The Ohio State University, USA
[3] Penn State University, USA

Abstract. Programs aiming for low runtime overhead and high availability draw on several object-oriented features available in the C/C++ programming language, such as dynamic object dispatch. However, there is an alarmingly high number of object dispatch (*i.e.,* forward-edge) corruption vulnerabilities, which undercut security in significant ways and are in need of a thorough solution. In this paper, we propose τCFI, an extended control flow integrity (CFI) model that uses both the types and numbers of function parameters to enforce forward- and backward-edge control flow transfers. At a high level, it improves the precision of existing forward-edge recognition approaches by considering the type information of function parameters, which are directly extracted from the application binaries. Therefore, τCFI can be used to harden legacy applications for which source code may not be available. We have evaluated τCFI on real-world binaries including Nginx, NodeJS, Lighttpd, MySql and the SPEC CPU2006 benchmark and demonstrate that τCFI is able to effectively protect these applications from forward- and backward-edge corruptions with low runtime overhead. In direct comparison with state-of-the-art tools, τCFI achieves higher forward-edge caller-callee matching precision.

Keywords: C++ object dispatch, indirect control flow transfer, code-reuse attack.

1 Introduction

The C++ programming language has been extensively used to build many large, complex, and efficient software systems over the last decades. A key concept of the C++ language is polymorphism. This concept is based on C++ virtual functions. Virtual functions enable late binding and allow programmers to overwrite a virtual function of the base-class with their own implementation. In order to implement virtual functions, the compiler needs to generate virtual table metadata structures for all virtual functions and provide to each instance (object) of such a class a (virtual) pointer (the value of which is computed during runtime) to the aforementioned table. Unfortunately, this approach represents a main source for exploitable program indirection (*i.e.,* forward edges) along function

returns (*i.e.,* backward edges), as the `C/C++` language provides no intrinsic security guarantees (*i.e.,* we consider Clang-CFI [29] and Clang's SafeStack [28] optional).

In this paper, we present a new control flow integrity (CFI) tool called τCFI used to secure `C++` binaries by considering the type information from application binaries. Our work targets applications, whose source code is unavailable and that contain at least one exploitable memory corruption bug (*e.g.,* a buffer overflow bug). We assume such bugs can be used to enable the execution of sophisticated Code-Reuse Attacks (CRAs) such as the COOP attack [39] and its extensions [7, 14, 24, 26], violating the program's intended control flow graph (CFG) through forward edges in the CFG and/or through attacks, that violate backward edges such as Control Jujutsu [18]. A potential prerequisite for violating forward-edge control flow transfers is the corruption of an object's virtual pointer. In contrast, backward edges can be corrupted by loading fake return addresses on the stack.

To address such object dispatch corruptions, and in general any type of indirect program control flow transfer violations, CFI [1, 2] was originally developed to secure indirect control flow transfers, by adding runtime checks before forward-edge and backward-edge control transfers. CFI-based techniques, that rely on the construction of a precise CFG, are effective [11], if CFGs are carefully constructed and sound [40]. However, these techniques still allow CRAs that do not violate the enforced CFG. For example, the COOP family of CRAs bypasses most deployed CFI-based enforcement policies, since these attacks do not exploit indirect backward edges (*i.e.,* function returns), but rather imprecision in forward edges (*i.e.,* object dispatches, indirect control flow transfers), which in general cannot be statically (before runtime) and precisely determined as alias analysis in program binaries is undecidable [38]. Source code based tools such as: SafeDispatch [22], MCFI [32, 33], ShrinkWrap [21], VTI [8], and IFCC/VTV [41] can protect against forward-edge violations. However, they rely on source code availability limiting their applicability (*e.g.,* proprietary libraries cannot be recompiled). In contrast, binary-based forward-edge protection tools, including binCFI [45], vfGuard [37], vTint [44], VCI [17], Marx [35] and TypeArmor [43], typically protect only forward edges through a CFI-based policy, and most of the tools assume that a shadow stack [23] technique is used to protect backward edges.

Unfortunately, the currently most precise binary-based forward-edge protection tools w.r.t. calltarget reduction, VCI and Marx, suffer from forward-edge imprecision, since both are based on an approximated program class hierarchy obtained through the usage of heuristics and assumptions. TypeArmor enforces a forward-edge policy, which only takes into account the number of parameters of caller-callee pairs without imposing any constraint on the parameters' types. Thus, these forward-edge protection tools are generally too permissive. CFI-based forward-edge protection techniques without backward-edge protection are broken [13], thus these tools assume that a shadow stack protection policy is in place. Unfortunately, shadow stack based techniques (backward-edge protection) were recently bypassed [20] and add, on average, up to 10% runtime overhead [15].

In this paper, we present τCFI, which is a fine-grained forward-edge and backward-edge binary-level CFI protection mechanism, that neither relies on shadow stack based techniques to protect backward edges, nor any runtime-type information (RTTI) (*i.e.,* metadata emitted by the compiler, which is most of the time stripped in production binaries). Note that, in general, variable type reconstruction on production binaries is a difficult task, as the required program semantics are mostly removed through compilation.

At a high level, there are a number of analyses τCFI performs in order to achieve its protection objective. In particular, it (1) uses its register width (ABI dependent) as the type of the parameter for each function parameter, (2) when determining whether an indirect call can target a function, it checks whether the call and the target function use the same number of parameters and whether the types (register width) match, (3) based on the provided forward-edge caller-callee mapping it builds a mapping, back from each callee to the legitimate addresses, located next to each caller. τCFI's backward-edge policy is based on the observation that backward edges of a program can be efficiently protected, if there is a precise forward-edge mapping available between callers and callees.

We have implemented τCFI on top of DynInst [6], which is a binary rewriting framework, that allows program binary instrumentation during loading or runtime. Note that τCFI preserves the original code copy of an executable by instrumenting all code of an executable shadow copy, which is later mapped to the original binary after it was loaded and τCFI's analysis finished. τCFI works with legacy programs and can be used to protect both executables and libraries. τCFI performs per-file analysis; as such each file is protected individually. We have evaluated τCFI with several real-world open source programs (*i.e.,* NodeJS, Lighttpd, MySql, etc.), as well as the SPEC CPU2006 benchmarks and demonstrated that our forward-edge policy is more precise than state-of-the-art tools. τCFI is applicable to program binaries for which we assume source code is not available. τCFI significantly reduces the number of valid forward edges compared to previous work and thus, we are able to build a precise backward-edge policy, which represents an efficient alternative to shadow stack based techniques.

In summary, we make the following contributions:

- We present τCFI, a new CFI system that improves the state-of-the-art CFI with more precise forward-edge identification by using type information reverse-engineered from stripped x86-64 binaries.
- We have implemented τCFI with a binary instrumentation framework to enforce a fine-grained forward-edge and backward-edge protection.
- We have conducted a thorough evaluation, through which we show that τCFI is more precise and effective than other state-of-the-art techniques.

2 Background

In this section, we provide the needed technical background to set the stage for the remainder of this paper.

2.1 Exploiting Object Dispatches in C++

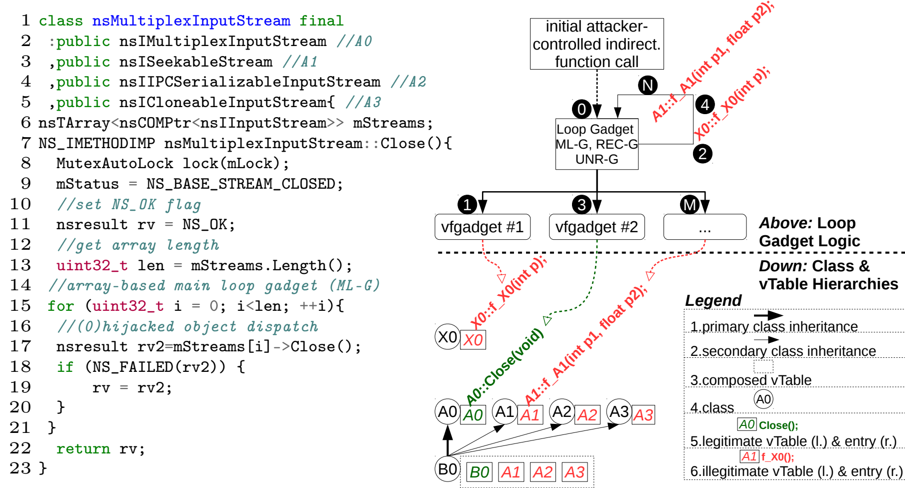

Fig. 1: COOP main loop gadget (`ML-G`) operation with the associated C++ code.

Figure 1 depicts a `C++` code example (left) and how a COOP main-loop gadget (right) (*i.e.,* based either on ML-G (main-loop), REC-G (recursive) or UNR-G (unrolled) COOP gadgets, see [14] for more details) is used to sequentially call COOP gadgets by iterating through a loop (REC-G excluded) controlled by the attacker.

First, the object dispatch (see line 17 depicted in Figure 1) is exploited by the attacker in order to call different functions in the whole program by iterating on an array of fake objects previously inserted in the array through, for example, a buffer overflow. Second, in order to achieve this, the attacker previously exploits an existing program memory corruption (*e.g.,* buffer overflow), which is further used to corrupt an object dispatch, ❶, by inserting fake objects into the array and by changing the number of initial loop iterations. Next she invokes gadgets, ❶ and ❸ up to Ⓜ, through the calls, ❷ and ❹ up to Ⓝ, contained in the loop. As it can be observed in Figure 1, the attacker can invoke from the same callsite legitimate functions (in total Ⓝ) residing in the virtual table (vTable) inheritance path (*i.e.,* at the time of writing this paper this type of information is particularly hard to recuperate from program binaries) for this particular callsite, indicated with green color vTable entries. However, a real COOP attack invokes illegitimate vTable entries residing in the entire initial program class hierarchy (or the extended one) with little or no relationship to the initial callsite, indicated with red-color vTable entries. Third, in this way different addresses contained in the program (1) (vTable) hierarchy (contains only virtual members), (2) class hierarchy (contains both virtual and non-virtual members) and (or) the whole program address space can be called. For example, the attacker can call any entry in the: (1) class

hierarchy of the whole program, (2) class hierarchy containing only legitimate targets for this callsite, (3) virtual table hierarchy of the whole program, (4) virtual table hierarchy containing only legitimate targets for this callsite, (5) virtual table hierarchy and class hierarchy containing only legitimate targets for this callsite, and (6) virtual table hierarchy and class hierarchy of the whole program. Finally, because there are no intrinsic language semantics—such as object cast checks—in the `C++` programming language for object dispatches, the loop gadget indicated in Figure 1 can be used without constraint to call any possible entry in the whole program. Thus, making any program address the start of a potential usable gadget.

2.2 Type-Inference on Executables

Recovering variable types from executable programs is generally considered difficult for two main reasons. First, the quality of the disassembly can vary considerably from one used underlying binary analysis framework to another and w.r.t. the compiler flags which were used to compile the binary. Note that production binaries can be more or less stripped (*i.e.,* RTTI or other debugging symbols may or may not be available etc.) from useful information, which can be used during a type-recovering analysis. τCFI is based on DynInst and the quality of the executable disassembly is sufficient for our needs. In contrast to other approaches, the register width based type recuperation of τCFI is based on a relatively simple analysis compared to other tools and provides similar results. For a more comprehensive review on the capabilities of DynInst and other tools, we advice the reader to review Andriesse *et al.* [3]. Second, if the type inference analysis requires alias analysis, it is well known that alias analysis in binaries is undecidable [38] in theory and intractable in practice [31]. Further, there are several highly promising tools such as: Rewards [27], BAP [10], SmartDec [19], and Divine [5]. These tools try more or less successfully to recover (or infer) type information from binary programs with different goals. Typical goals are: (1) full program reconstruction (*i.e.,* binary to code conversion, reversing, etc.), (2) checking for buffer overflows, and (3) checking for integer overflows and other types of memory corruptions. For a comprehensive review of type inference recovering tools in the context of binaries, we suggest consulting Caballero *et al.* [12]. Finally, it is interesting to note that the code from only a few of the tools mentioned in the previous review are actually available as open source.

2.3 Security Implications of Indirect Transfers

Indirect Forward-Edge Transfers. Illegal forward-edge indirect calls may result from a virtual pointer (vPointer) corruption. A vPointer corruption is not a vulnerability but rather a capability, which can be the result of a spatial or temporal memory corruption triggered by: (1) bad-casting [25] of `C++` objects, (2) buffer overflow in a buffer adjacent to a `C++` object, or (3) a use-after-free condition [39]. A vPointer corruption can be exploited in several ways. A manipulated vPointer can be exploited to make it point to any existing or added

program virtual table entry or to a fake virtual table added by the attacker. For example, an attacker can use the corruption to hijack the control flow of the program and start a COOP attack [39]. vPointer corruptions are a real security threat that can be exploited in many ways as for example if there is a memory corruption (*e.g.,* buffer overflow, use-after-free condition), which is adjacent in memory to the `C++` object. As a consequence, each memory corruption, which can be used to reach the memory layout of an object (*e.g.,* object type confusion), can be potentially used to change the program control flow.

Indirect Backward-Edge Transfers. Program backward edges (*i.e.,* `jump`, `ret`, etc.) can be corrupted to assemble gadget chains such as follows. (1) No CFI protection technique was applied: In this case, the binary is not protected by any CFI policy. Obviously, the attacker can then hijack backward edges to *jump* virtually anywhere in the binary in order to chain gadgets together. (2) Coarse-grained CFI protected scenarios: In this scenario, if the attacker is aware of what addresses are protected, the attacker may deviate the application flow to legitimate locations in order to link gadgets together. (3) Fine-grained CFI protection scenarios: In this case, the legitimate target set is stricter than in (2). But, assuming that the attacker knows which addresses are protected and which are not, she may be able to call legitimate targets through control flow bending. (4) Fully precise CFI protected scenarios (*i.e.,* SafeStack [23] based): In this scenario, the legitimate target set is stricter than in (3). Even though we have a one-to-one mapping between calltargets and legitimate return sites, the attacker could use this one-to-one mapping to assemble gadget chains if at the legitimate calltarget return site there is a useful gadget [13].

3 Threat Model

We follow the same basic assumptions stated in [43] w.r.t. forward edges. More precisely, we assume a resourceful attacker that has read and write access to the data sections of the attacked program binary. We assume that the protected binary does not contain self-modifying code or any kind of obfuscation. We also consider pages to be either writable or executable, but not both at the same time. Further, we assume that the attacker has the ability to exploit an existing memory corruption in order to hijack the program control flow. As such, we consider a powerful yet realistic adversary model that is consistent with previous work on CRAs and their mitigations [23]. The adversary is aware of the applied defenses and has access to the source and non-hardened binary of the target application. She can exploit (bend) any backward-edge based indirect program transfer and has the capability to make arbitrary memory writes.

4 Design and Implementation

In this section, we present a brief overview of τCFI followed by its design and implementation.

4.1 Approach Overview

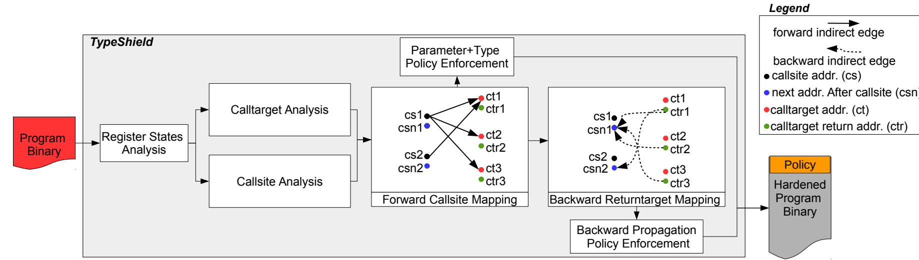

Fig. 2: Main steps performed by τCFI when hardening a program binary.

Figure 2 depicts an overview of our approach. From left to right, the program binary is analyzed by τCFI and the calltargets and callsite analysis are performed for determining how many parameters are provided, how many are consumed, and their register width. After this step, labels are inserted at each previously identified callsite and at each calltarget. The enforced policy is schematically represented by the black highlighted dots (addresses, *e.g., cs1*) in Figure 2 that are allowed to call only legitimate red highlighted dots (addresses, *e.g., ct1*). Next, to compute the set of addresses which a return instruction can target, the address set determined by each address located after each legitimate callsite is computed. This information is obtained by using the previously determined callsite forward-edge mapping to derive a function return backward map that uses return instructions as keys and return targets as values. This way, τCFI has a set of addresses for each function to which the function return site is allowed to transfer control. Finally, range or compare checks are inserted before each function return site. These checks are used during runtime to check if the address, where the function return wants to jump to, is contained in the legitimate set for each particular return site. This is represented in Figure 2 by green highlighted dots (addresses *e.g., ctr2*) that are allowed to call only legitimate blue highlighted dots (addresses *e.g., csn1*). Finally, the result is a hardened program binary (see right-hand side in Figure 2).

4.2 Parameter Count and Type Policy

Parameters can be passed through registers or the stack. In the Itanium `C++` ABI, the first six parameters are passed through registers (*i.e.,* `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`). Even when a 64-bit register is used to pass a parameter, the actual number of bits used in the register might be smaller. Therefore, we treat the used widths of parameter-storing registers as the types of the parameters. There are four types of reading and writing access on registers. Therefore, our set of possible types for parameters is $\{64, 32, 16, 8, 0\}$, where zero models the

absence of a parameter. For the Itanium ABI, our analysis tracks the 6 registers used in parameter passing and classifies callsites and calltargets according to how these registers are used.

Our analysis overapproximates at callsites and underapproximates at calltargets the parameter count and types, which is due to the general difficulty of statically determining the exact number of arguments provided by a callsite and the number of parameters required by a calltarget and w.r.t. the widths of registers used in parameter passing. Specifically, at a callsite, the analysis calculates an *upper bound* for the number of arguments and for the widths of those registers that store arguments. For instance, for a function call that passes one argument with a width of 32-bit, the analysis may estimate that there are two arguments passed and the first one's width is 64-bit. Furthermore, the analysis on a calltarget (a callee function) calculates a *lower bound* for the number of needed parameters and for the widths of those registers that store parameters.

Because of the approximations in our analysis, our policy for matching callsites and calltargets allows a callsite to transfer to a calltarget if (1) the number of estimated arguments at a callsite is greater than the number of estimated parameters at a calltarget; and (2) for each argument at the callsite and its corresponding parameter in the calltarget, the estimated width of the argument is greater than the estimated width of the parameter. Part (1) is about the parameter count and is the same as the parameter-count policy in TypeArmor [43]; part (2) is about the parameter types and enables τCFI to provide a finer-grained policy than just considering the parameter count.

4.3 Instruction Read-Write Effect

We first introduce some definitions and notation. The set $\mathcal{I}$ describes the set of possible instructions; in our case, this is based on the instruction set for x86-64 processors. An instruction $i \in \mathcal{I}$ can perform two kinds of operations on registers: (1) Read n-bit from a register with $n \in \{64, 32, 16, 8\}$ and (2) Write n-bit to a register with $n \in \{64, 32, 16, 8\}$. Note that there are instructions that can directly access the higher 8 bits of the lower 16 bits of 64-bit registers. For our purpose, we treat this access as a 16-bit access.

Next, the possible effect of an instruction on one register is described as $\delta \in \Delta$ with $\Delta = \{w64, w32, w16, w8, 0\} \times \{r64, r32, r16, r8, 0\}$. Note that 0 represents the absence of either a write or read access and $(0, 0)$ represents the absence of both. Meanwhile, wn with $n \in \{64, 32, 16, 8\}$ implies all wm with $m \in \{64, 32, 16, 8\}$ and $m < n$ (*e.g.,* $w64$ implies $w32$); the same property holds for rn. The Itanium `C++` ABI specifies 16 general purpose integer registers. Therefore, the read-write effect of an instruction on the set of registers can be described as $\delta_p \in \Delta^{16}$. Our analysis performs calculations based on the effect of each instruction $i \in \mathcal{I}$ via the function $\text{regEffect} : \mathcal{I} \mapsto \Delta^{16}$. Note that this function can be purely defined based on the semantics of instructions.

4.4 Calltarget Analysis

Our calltarget static analysis classifies calltargets according to the parameters they expect by taking into account the parameters' count and types. Given a set of address-taken functions[4], the static analysis performs an interprocedural analysis to determine the register states for the 6 argument registers.

Next, we present τCFI's analysis, followed by a discussion of optimizations and interprocedural analysis. The basic analysis determines, for each register and at a particular program location, that it is in one of the following states:

- rn, where $n \in \{64, 32, 16, 8\}$ represents that the lower n bits of the register are *read before written* along all control flow paths starting from the location.
- $*$ represents that, along some control flow path, the register is either written before read or there are no reads/writes on the register.

The basic analysis described above can be implemented as a classic backward-liveness analysis, except that it needs to track widths in read operations. For instance, for an instruction i, if the regEffect function shows that i reads the lower 16-bits of `rax`, then the state of `rax` immediately before the instruction is $r16$. For an instruction with multiple successors, the register states after the instruction are calculated based on the states at the beginnings of the successors. For instance, if an instruction has two successors, and the state of `rax` is $r64$ before the first successor and the state of `rax` is $r32$ before the second, then the state of `rax` after the instruction is $r32$, essentially indicating that all paths starting from the end of the instruction have a $r32$ read before write for `rax`. Recall that the calltarget analysis performs an underapproximation; so using $r32$ is safe even though one of the paths performs a $r64$ read.

The backward-liveness analysis, however, is inefficient. Our implementation actually follows TypeArmor [43] to perform a forward interprocedural analysis (with some modification to consider widths of read operations). We refer readers to the TypeArmor paper for details and give only a brief overview here.

First, note that τCFI's analysis operates at the basic block level instead of the instruction level. Second, the analysis further refines the $*$ state to be either w or c, where w (write before read) refers to a register being written to before read from along some control flow path and c (clear/untouched) represents that the register is untouched along some control flow path. The reason for such a refinement is that during forward analysis, if the states of all argument registers before a basic block b are either rn or w (e.g., when b reads or writes all argument registers), then there is no need to keep analyzing the successor basic blocks since their operations would not change the state before b; this enables an early termination of the forward analysis and is thus more efficient. On the other hand, if the state of one of the argument registers is c, then the forward analysis has to continue. This is because c indicates the register is untouched so far, but it can

[4] Since an indirect call can target a function only if the function's address is taken, there is no need to analyze functions whose addresses are not taken; this is similar to TypeArmor.

be read or written in a future basic block. Further, the analysis is interprocedural and maintains a stack to match direct function calls and returns during analysis. Finally, for indirect calls, however, it does not follow to the targets, but performs an underapproximation instead.

Parameter count and types Once the analysis finishes, we can calculate a function's parameter count and parameter types based on the state before the entry basic block of the function. The argument count is determined using the highest argument register that is marked rn. The type of an argument register is directly given by the rn state of the register.

4.5 Callsite Analysis

Our callsite analysis classifies callsites according to the arguments they provide by considering the argument *count* and their *types*. For callsite analysis, overestimations are allowed: the callsite analysis overestimates the number of arguments and the widths of arguments. As such a callsite is allowed to target a calltarget that requires a smaller or equal number of parameters and that requires a smaller or equal width for each parameter.

For callsite analysis, we employ a customized reaching-definition analysis. The analysis determines the states of registers. At a particular program location, it determines whether or not a register is in one of the following states:

- sn, where $n \in \{8, 16, 32, 64\}$: this represents a state in which the register's lower n bit is set in a control-flow path ending at the program location.
- t (trashed): this represents a state in which the register is not set on all control flow paths ending at the program location.

τCFI's reaching-definition analysis is implemented as an interprocedural backward analysis similar to TypeArmor [43], the difference being that τCFI also tracks the widths in write operations to infer sn states. Once the analysis is finished, it uses the register state just before an indirect callsite to determine its argument count and types: If an argument register is in state sn, then it is considered an argument that uses n bits; the argument count is determined by the highest argument register whose state is sn.

4.6 Return Values

Knowing more information about return values of functions increases CFI precision. For instance, an indirect callsite that expects a return value should not call a function that does not return a value; similarly, an indirect callsite that expects a 64-bit return value should not call a function that returns only a 32-bit value. For calltarget analysis, τCFI traverses backwards from the return instruction of a function and searches for uses of the `RAX` register to determine if a function has a void or a non-void return type. In case there is a write operation on the `RAX`

register, τCFI infers that the function's return type is non-void; furthermore, it tracks the widths of write operations to infer the width of the return type. For calltarget return-value type estimation, overapproximations are allowed.

At a callsite, τCFI traverses forward from the callsite to search for reads before writes on the `RAX` register to determine if a callsite expects a return value or not. In case there is such a read on the `RAX` register, τCFI infers that the callsite expects a return value; furthermore, it tracks the widths of read operations to infer the width of the expected return value. For callsite return-value type estimation, underapproximation is allowed.

4.7 Backward-Edge Analysis

In order to protect backward edges, we have designed an analysis that can determine possible legitimate return target addresses for each callee function. Our algorithm used for computing the legitimate set of addresses for each callee works as follows. First, a map is obtained after running the callsite and calltarget analysis (see Section 4.4 and Section 4.5 for more details); it maps a callsite to the set of legal calltargets where forward-edge indirect control-flow transfer is allowed to jump. This map is then reversed to build a second map that maps from the return instruction of a function (callee) to a set of addresses where the return can transfer to.

The return target address set for a function return is determined by getting the next address after each callsite address that is allowed to make the forward-edge control flow transfer. The map is obtained by visiting a return instruction address in a function and assigning to it the addresses next to callsites that can call the function. At the end of the analysis, all callsites and all function returns have been visited and a set of backward-edge addresses for each function return address is obtained. Note that the function boundary address (*i.e.,* `ret`) is detected by a linear basic block search from the beginning of the function (calltarget) until the first return instruction is encountered. We are aware that other promising approaches for recovering function boundaries (*e.g.,* [4]) exist, and plan to experiment with them in future work.

4.8 Binary Instrumentation

Forward-Edge Policy Enforcement. The result of the callsite and calltarget analysis is a mapping that maps a callsite to its allowed calltargets. In order to enforce this mapping during runtime, callsites and calltargets are instrumented inside the binary program with two labels. Additionally, each callsite is instrumented with CFI checks. At a callsite, the number of provided arguments is encoded as a series of six bits. At a calltarget, the label contains six bits encoding how many parameters the calltarget expects. Additionally, at a callsite 12 bits encode the register-width types of the provided arguments (two bits for each parameter), while at the calltarget another 12 bits are used to encode the types of the parameters expected. Further, at a callsite, several bits are used to encode if the function is expecting a `void` return type or not, and the width of the return

type if it is nonvoid (similarly for a calltarget). All this information is written in labels before callsites and calltargets. During runtime before a callsite, these labels are compared by performing an XOR operation. In case the XOR operation returns false (a zero value), the transfer is allowed; otherwise, the program execution is terminated.

Backward-Edge Policy Enforcement. Based on the previously determined reverse map, before each function return a randomly generated label value is inserted. We decided to use these kinds of values as our main requirement is to map a return to a potentially large number of return sites. The same label is inserted before each legitimate target address (the next address after a legitimate callsite). In this way, a function return is allowed to jump only to the instruction that follows next to the address of a callsite.

For callsites that target a calltarget that is also allowed by another callsite, τCFI performs a search in order to detect if the callsite already has a label attached to the address after the callsite. If so, a new label is generated and multiple labels are stored for the address following the callsite. In this way, calltarget return labels are grouped together based on the reverse map. This design allows the same number of function return sites as the forward-edge policy enforces for each callsite. Finally, in case the comparison returns true, the execution continues; otherwise, it is terminated.

4.9 Implementation

We have implemented τCFI using the DynInst [6] (v.9.2.0) instrumentation framework with a total of 5,501 lines of C++ code. We currently restricted our analysis and instrumentation to x86-64 executables in the ELF format using the Itanium C++ ABI calling convention. τCFI can deal with the level of executable obfuscation with which DynInst can deal. As such, we fully delegate this responsibility to the used instrumentation framework underneath. We focused on the Itanium C++ ABI convention as most C/C++ compilers on Linux implement this ABI. However, the implementation separated the ABI-dependent code, so we expect it to be possible to support other ABIs as well. We developed the main part of our binary analysis pass in an instruction analyzer, which relies on the DynamoRIO [9] library (v.6.6.1) to decode single instructions and provide access to its information. The analyzer is then used to implement our version of the reaching-definition and liveness analysis. Further, we implemented a Clang/LLVM (v.4.0.0, trunk 283889) backend (machine instruction) pass (416 LOC) used for collecting ground truth data in order to evaluate the effectiveness and performance of our tool. The ground truth data is then used to verify the output of our tool for several test targets. This is accomplished with the help of our Python-based evaluation and test environment implemented in 3,239 lines of Python code.

5 Evaluation

We have evaluated τCFI by instrumenting various open source applications and conducting a thorough analysis in order to show its effectiveness and usefulness. Our test applications include the following real-world programs: FTP servers *Vsftpd* (v.1.1.0, `C` code), Pure-ftpd (v.1.0.36, `C` code) and *Proftpd* (v.1.3.3, `C` code); Lighttpd web server (v.1.4.28, `C` code); two database server applications *Postgresql* (v.9.0.10, `C` code) and *Mysql* (v.5.1.65, `C++` code); the memory cache application *Memcached* (v.1.4.20, `C` code); and the *Node.js* application server (v.0.12.5, `C++` code). We selected these applications to allow for a fair comparison with other similar tools. In our evaluation, we addressed the following research questions (RQs): **RQ1:** How **effective** is τCFI? (§5.1); **RQ2:** What **security protection** is offered by τCFI? (§5.2); **RQ3:** Which **attacks** are mitigated by τCFI? (§5.3) **RQ4:** Are other forward-edge tools **better** than τCFI? (§5.4)? **RQ5:** Is τCFI **effective** against COOP? (§5.5) **RQ6:** How does τCFI **compare** with Clang's Shadow Stack? (§5.6) **RQ7:** What **runtime overhead** does τCFI impose? (§5.7) Our setup is based on Kubuntu 16.04 LTS (k.v.4.4.0) using 3GB RAM and four hardware threads running on an i7-4170HQ CPU at 2.50 GHz.

5.1 Effectiveness

`O2`	**CS**	**CT**	**AT**	***count****		***count***		***type****		***type***	
Target	total	total	total	limit (mean ± σ)	median	limit (mean ± σ)	median	limit (mean ± σ)	median	limit (mean ± σ)	median
ProFTPD	157.0	1,011.0	396.0	349.3 ± 52.8	369.0	370.1 ± 43.3	382.0	338.2 ± 64.8	361.0	354.4 ± 85.1	390.0
Pure-FTPD	8.0	127.0	13.0	8.6 ± 4.6	8.0	10.1 ± 4.8	13.0	8.1 ± 4.0	5.0	10.0 ± 4.0	10.0
Vsftpd	2.0	391.0	10.0	8.0 ± 2.0	8.0	10.0 ± 0.0	10.0	6.0 ± 2.0	6.0	7.0 ± 3.0	7.0
Lighttpd	66.0	289.0	63.0	34.3 ± 15.1	21.0	43.7 ± 14.5	51.0	34.5 ± 14.7	23.0	45.4 ± 12.1	50.0
Nginx	270.0	914.0	1,111.0	316.8 ± 146.9	266.0	447.6 ± 124.0	528.0	317.5 ± 146.4	267.0	450.9 ± 110.2	528.0
MySQL	7,893.0	9,928.0	5,896.0	338.5 ± 189.5	179.0	490.6 ± 203.3	574.0	307.9 ± 163.6	186.0	519.6 ± 147.6	540.0
PostgreSQL	687.0	6,885.0	2,304.0	423.4 ± 176.7	471.0	497.0 ± 151.8	515.0	416.2 ± 188.1	541.0	476.9 ± 162.4	562.0
Memcached	48.0	134.0	14.0	12.3 ± 2.3	14.0	13.0 ± 1.4	14.0	12.7 ± 1.0	12.0	12.8 ± 1.0	12.0
NodeJS	10,215.0	20,196.0	7,230.0	763.1 ± 329.3	806.0	1,051.2 ± 293.2	1,169.0	683.2 ± 332.9	459.0	939.8 ± 314.0	1,022.0
geomean	170.1	1,104.8	259.8	89.0 ± 31.2	79.4	110.4 ± 27.0	123.1	83.7 ± 28.1	69.3	104.7 ± 27.8	111.6

Table 1: Allowed callsites per calltarget for τCFI's count and type policies.

Table 1 depicts the average number of calltargets per callsite, the standard deviation σ, and the median. In Table 1, the abbreviation CS refers to the callsites, while CT means calltargets. Note that the restriction to address-taken functions (see column AT) is present. The label *count** denotes the best possible reduction using the parameter *count* policy based on the ground truth collected by our Clang/LLVM pass, while *count* denotes the results of our implementation of the parameter *count* policy derived from binaries. The same applies to *type∗* and *type* regarding the parameter *type* policy. A lower number of calltargets per callsite indicates better results. Note that our parameter *type* policy is superior to the parameter *count* policy, as it allows for a stronger reduction of allowed calltargets. We consider this an important result, which further improves the state-of-the-art. Finally, we provide the median and the pair of mean and standard deviation to allow for a better comparison with other state-of-the-art tools.

Theoretical Limits. We explored the theoretical limits regarding the effectiveness of the *count* and *type* policies by relying on the collected ground truth data; essentially assuming perfect classification. Based on the type information collected by our Clang/LLVM pass, we derived the available number of calltargets for each callsite by applying the count and type policies. From the results, (1) the theoretical limit of the *count** policy has a geomean of 89 possible calltargets, which is around 8% of the geomean of the total available calltargets (1104), and (2) the theoretical limit of the *type** policy has a geomean of 83 possible calltargets, which is 7.5% of the geomean of the total available calltargets (1104). In comparison, the theoretical limit of the *type** policy allows about 13% less available calltargets in geomean than the limit of the *count** policy (*i.e.,* 69.3 vs. 79.4).

Calltarget per Callsite Reduction. (1) The *count* policy has a geomean of 104 calltargets, which is around 9.4% of the geomean of all available calltargets (1104). This is around 24% more than the theoretical limit of available calltargets per callsite (see *count** 89 vs. 110.4). (2) The *type* policy has a geomean of 104.7 calltargets, which is 9.48% of the geomean of total available calltargets (1104). This is approximatively 25% more than the theoretical limit of available calltargets per callsite (see *type** 83.7 vs. 104.7). τCFI's *type* policy allows around 9.4% less available calltargets in the geomean than our implementation of the *count* policy (104.7 vs. 110.4), and a total reduction of more than 94% (104.7 vs. 1104) w.r.t. the total number of calltargets (CT) available once the *count* and *type* policies are applied.

5.2 Forward-Edge Policy vs. Other Tools

Table 2 provides a comparison between τCFI, TypeArmor and IFCC w.r.t. the median count of calltargets per callsite. The values for TypeArmor [43] and IFCC [41] depicted in Table 2 have been adopted from the corresponding papers in order to ensure a fair comparison. Further, Table 2 conveys the limitations of binary-based type analysis, as the median of the possible target set size for τCFI is several times larger than the corresponding set sizes for system using source-level analysis. Note that the smaller the geomean values are, the better the technique is. AT is a technique that allows a callsite to target any address-taken functions. IFCC is a compiler-based solution and is included here as

Target	**IFCC**	**TypeArmor (CFI+CFC)**	**AT**	**τCFI (count)**	**τCFI (type)**
ProFTPD	3.0	376.0	396.0	382.0	390.0
Pure-FTPD	0.0	4.0	13.0	13.0	10.0
Vsftpd	1.0	12.0	10.0	10.0	7.0
Lighttpd	6.0	47.0	63.0	51.0	50.0
Nginx	25.0	254.0	1,111.0	528.0	528.0
MySQL	150.0	3,698.0	5,896.0	574.0	540.0
PostgreSQL	12.0	2,304.0	2,504.0	515.0	562.0
Memcached	1.0	14.0	14.0	14.0	12.0
NodeJS	341.0	4,714.0	7,230.0	1,169.0	1,022.0
geomean	8.7	170.4	259.8	123.1	111.6

Table 2: Legitimate calltargets/callsite for 5 tools.

a reference to show what is possible when the program's source code is available. TypeArmor and τCFI on the other hand are binary-based tools. τCFI reduces the number of calltargets by up to 42.9% (geomean) when compared to the AT technique, by more than seven times (7230 vs. 1022) for a single test program w.r.t. AT, and by 65.49% (170.4 vs. 111.6) in geomean when compared with TypeArmor, respectively. As such, τCFI represents a stronger improvement w.r.t. calltarget per callsite reduction in binary programs compared to other approaches.

5.3 Effectiveness Against COOP

We investigated the effectiveness of τCFI against the COOP attack by looking at the number of register arguments, which can be used to enable data flows between gadgets. In order to determine how many arguments remain unprotected after we apply the forward-edge policy of τCFI, we determined the number of parameter overestimations and compared it with the ground truth obtained during an LLVM compiler pass. Next, we used some heuristics to determine how many ML-G and REC-G callsites exist in the `C++` server applications. Finally, we compared these results with the one obtained by TypeArmor.

Table 3 presents the results obtained after counting the number of perfectly estimated and overestimated protected ML-G and REC-G gadgets. As it can be observed, τCFI obtained a 96% (184 out of 192) accuracy of perfectly protected ML-G callsites for MySQL, while TypeArmor obtained a 94% accuracy for the same program. Further, τCFI obtained a 97% (131 out of 134) accuracy for Node.js, while TypeArmor obtained 95% accuracy on the same program. Further, for the REC-G case, τCFI obtained an 94% (273 out of 289) exact-parameter accuracy for MySQL, while TypeArmor had 86%. For Node.js, τCFI obtained an accuracy of 95% (69 out of 72), while TypeArmor had 96%. Overall τCFI's forward-edge policy obtained a perfect accuracy of 95%, while TypeArmor obtained 92%. While this is not a large difference, we want to point out that the remaining overestimated parameters represent only 5% and thus do not leave much wiggle room for the attacker.

		Overestimation					
Program	#cs	0	+1	+2	+3	+4	+5
MySQL (ML-G)	192	184	3	1	0	1	3
Node.js (ML-G)	134	131	1	0	1	0	1
geomean	160	155	1	1	1	1	1
MySql (REC-G)	289	273	10	2	3	0	1
Node.js (REC-G)	72	69	2	0	0	0	1
geomean	144	137	4	1	1	1	1

Table 3: Parameter overestimation for the ML-G and REC-G gadgets.

5.4 Comparison with the Shadow-Stack

The shadow stack implementation of Abadi *et al.* [1] provides a strong security protection [11] w.r.t. backward-edge protection. However, it: (1) has a high runtime overhead ($\geq$ 21%), (2) is not open source, (3) uses a proprietary binary analysis framework (*i.e.,* Vulcan), (4) loses precision due to equivalent class merging. Hence, we propose an alternative backward-edge protection solution. In order to show the precision of τCFI's backward-edge protection, we provide

the average number of legitimate return addresses for return instructions and compare it to the total number of available addresses without any protection.

Program	Total #RA	Total #RATs	Total #RATs/RA	%RATs/RA prog. binary
MySQL	5,896.0	3,792.0	0.6	0.014%
Node.js	7,230.0	3,864.0	0.53	0.011%
geomean	6,529.0	3,827.0	0.58	0.012%

Table 4: Backward-edge policy statistics.

Table 4 presents the statistics w.r.t. the backward-edge policy legitimate return targets. More specifically, in Table 4, we use the following abbreviations: total number of return addresses (Total #RA), total (median) number of return address targets (Total #RATs), total (median) number of return address targets per return instruction (Total. # RATs/RA), percentage of legitimate return address targets per return addresses w.r.t. the total number of addresses in the program binary (% RATs/RA w.r.t. program binary). By applying τCFI's backward-edge policy, we obtain a reduction of 0.43 $(1-0.58)$ ratio (geomean) of the total number of return address targets per return address over the total number of return addresses. This means that only 43% of the total number of return addresses are actual targets for the function returns. The results indicate a percentage of 0.012% (geomean) of the total addresses in the program binaries are legitimate targets for the function returns. This means that our policy can eliminate 99.98% (100% - 0.012%) of the addresses, which an attacker can use for his attack inside the program binary. To put it differently, only 0.012% of the addresses inside the binary can be used as return addresses by the attacker. Further, we assume that the attacker cannot easily determine which addresses are still available for any given program binary, which is stripped from debug information. Note that each function return (callee) is allowed to return in geomean to around 111 legitimate addresses (MySQL 519 and NodeJS 939) in all analyzed programs. Finally, we assume that it is hard for the attacker to find out the exact set of legitimate addresses per return site once the policy was applied.

5.5 Security Analysis

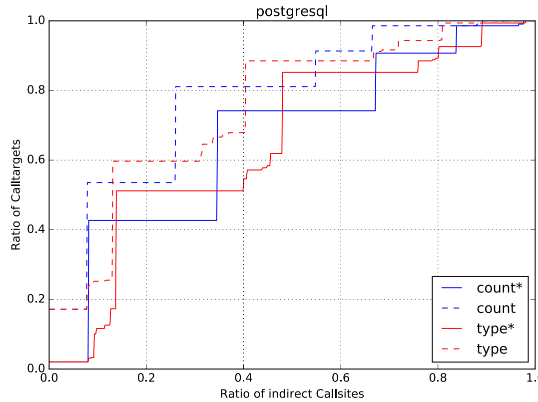

Fig. 3: CDF for the PostgreSQL program.

Figure 3 depicts the cumulative distribution function (CDF) for the PostgreSQL program compiled with the Clang `-O2` flag. We selected this program randomly from our test programs. The CDF depicts the relation between the ratio of indirect callsites and the ratio of calltargets, for the type and the count policies. While the CDFs for the count policies have only a few changes, the amount of changes for the CDFs of the type policies is vastly higher. The reason for this is fairly straightforward: the number of buckets (*i.e.,* the number of equivalence classes) that are used to classify the callsites and calltargets is simply higher for the type policies. Finally, note that the results depend on the internal structure of the particular program and may for this reason vary for other programs.

5.6 Mitigation of Advanced CRAs

Table 5 presents several attacks that can be successfully stopped by τCFI by deploying only the forward-edge or the backward-edge policy. For checking if the COOP attack can be prevented, we instrumented the Firefox library (`libxul.so`), which was used to perform the original COOP attack as presented in the original paper. We observed that due to the forward-edge policy this attack was no longer possible. For testing if backward-edge attacks are possible after applying τCFI, we used several open source ROP attacks that are explicitly violating the control flow of a C++ program through backward-edge violations. Next, we instrumented the binaries of these programs. Each attack that was using one of the protected function returns was successfully stopped.

Exploit	Stopped	Remark
COOP ML-G [39]		
IE 32 bit	×	Out of scope
IE 1 64-bit	✓(FP)	Arg. count mismatch
IE 2 64-bit	✓(FP)	Arg. count mismatch
Firefox	✓(FP)	Arg. count mismatch
COOP ML-REC [14]		
Chrome	✓(FP)	Void target where non-void was expected
Control Jujutsu [18]		
Apache	✓(FP)	Target function not AT
Nginx	✓(FP)	Void target where non-void was expected
All Backward edge violating attacks	✓(BP)	(1)[a] or (2)[b] or (3)[c]

[a] Jump to address $\notin$ in the $max - min$ address range.
[b] Jump to address $\neq$ then a legitimate address.
[c] Jump to address label $\neq$ the calltarget return label.

Table 5: Stopped CRAs, forward-edge policy (FP) & backward-edge policy (BP).

In summary, many forward-edge and backward-edge attacks can be successfully mitigated by τCFI as long as these attacks are not aware of the policy in place and thus cannot selectively use gadgets that have their start address in the allowed set for the legitimate forward-edge and backward-edge transfers, respectively.

5.7 Runtime Overhead

Figure 4 presents the runtime overhead obtained by applying τCFI's forward-edge policy (register type; parameter count) and backward-edge policy on all `C/C++` programs contained in SPEC CPU2006. Out of the evaluated programs: `xalancbmk`, `namd`, `omnetpp`, `dealII`, `astar`, `soplex`, and `povray` are `C++` programs, while the rest are pure `C` programs. After the programs were instrumented, we measured the runtime overhead. The geomean of the instrumented programs is around 2.89% runtime overhead. One reason for the performance drop is cache misses introduced by jumping between the old and the new executable section of the binary generated by duplicating and patching. This is necessary, because when outside of the compiler, it is difficult to relocate indirect control flow. Therefore, every time an indirect control flow occurs, jumps into the old executable section and from there back to the new executable section occur. Moreover, this is also dependent on the actual structure of the target as the overhead depends on the frequency of indirect control flow operations. Another reason for the slightly higher (yet acceptable) performance overhead is our runtime policy, which is more complex than that of other state-of-the-art tools.

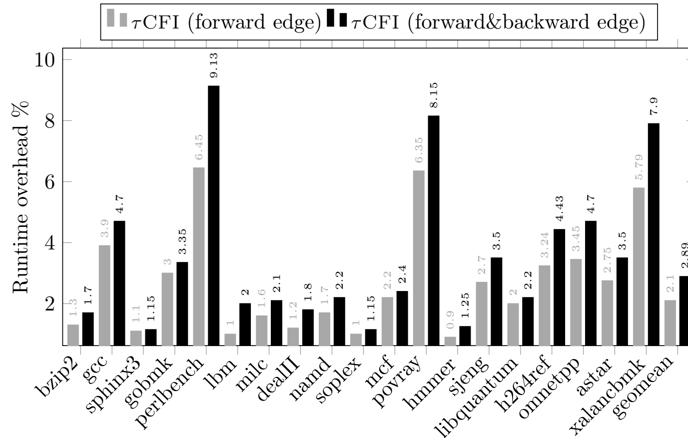

Fig. 4: Runtime overhead.

6 Discussion

Limitations. First, τCFI is limited by the capabilities of the DynInst instrumentation environment, where non-returning functions like exit are not detected reliably in some cases. As a result, we cannot test the Pure-FTP server, as it heavily relies on these functions. The problem is that those non-returning functions usually appear as a second branch within a function that occurs after the normal control flow, causing basic blocks from the following function to be attributed to the current function. This results in a malformed control flow graph and erroneous attribution of callsites and problematic misclassifications for both calltargets and callsites.

Second, parameter passing through floating point registers is currently not supported by τCFI, similar to other state-of-the-art tools. Tail calls are also

not supported for now as they lose the one-to-one matching between callers and callees. Further, τCFI does not support self-modifying code as code pages become writable at run-time. We plan to address this limitation in future work.

Third, τCFI is not intended to be more precise than source code based tools such as IFCC/VTV [41]. However, τCFI is highly useful in situations when the source code is typically not available (*e.g.,* off-the-shelf binaries), where programs rely on third-party libraries, and where the recompilation of all shared libraries is not possible.

Finally, while a major step forward, τCFI cannot thwart all possible attacks, as even solutions with access to source code are unable to protect against all possible attacks [13]. In contrast, τCFI, our binary-based tool can stop all COOP attacks published to date and significantly raises the bar for an adversary when compared to other state-of-the-art tools. Moreover, τCFI provides a strong mitigation for other types of code-reuse attacks as well as for attacks that violate the caller-callee function calling convention.

Attacker Policy Discovery Trade-offs. In general, with usage of CFI techniques, it is relatively unchallenging for an attacker to figure out where an indirect program control flow may transfer during runtime. This is because the indirect transfer targets (backward and forward) are labeled with IDs that have to satisfy certain conditions, *e.g.,* a bitwise `XOR` operation between the bits of the start and target address of indirect control flow transfer should return a one or zero in case the transfer is legal or illegal, respectively.

Thus, we note that in general it is not difficult for a resourceful attacker to figure out which callees match to which calltargets or vice versa when these are labeled with `ID`s for example. τCFI is not exempted from this. In general, if the attacker knows where an indirect transfer is allowed to jump to, he may use this wiggle space to craft his attack with the available (reachable) gadgets. The main assumption on which CFI and τCFI are built upon is that the wiggle room is sufficiently reduced for an attacker such that the likelihood for a successful attack is greatly diminished.

7 Related Work

Mitigation of Forward-Edge Based Attacks with Binary-Based Tools. τCFI is closely related to TypeArmor w.r.t. the forward edge analysis. TypeArmor [43] ($\approx$3% runtime overhead in geomean) enforces a CFI-policy based on the parameter count policy. Compared to τCFI, TypeArmor does not use function parameter types and assumes a backward-edge protection is in place. VCI [17] and Marx [35] are both based on approximated program (quasi) class hierarchies; they (1) do not recover the root class of the hierarchy, and (2) the edges between the classes are not oriented; thus both tools enforce for each callsite the same virtual table entry (*i.e.,* index based) contained in one recovered class hierarchy represented by father-child relationships between the recovered vtables. Finally, both tools use up to six heuristics and simplifying assumptions in order to make the problem of program class hierarchy reconstruction tractable. Compared to

these tools, τCFI tries not to reconstruct a high-level metadata data structure (class hierarchy), but rather performs analysis on the usage of provided and consumed parameters at the callsites and calltargets.

Mitigation of Backward-Edge Based Attacks with Binary Based Tools. According to a comprehensive survey by Burow *et al.* [11], tools that provide backward-edge protection offer low, medium, and high levels of protection w.r.t. backward edges. Further, this survey provides runtime overhead comparisons, classifies the backward-edge protection techniques into binary-based, source code based, and other types (*e.g.,* with HW support, etc.). Due to page restriction, we review only binary tools.

The original CFI implementation of Abadi *et al.* [1], MoCFI [16], kBouncer [34], CCFIR [45], bin-CFI [46], O-CFI [30], PathArmor [42], LockDown [36] mostly suffer from imprecision (high number of reused labels), have low runtime efficiency, and most of them protect either forward edges or backward edges assuming a perfect shadow stack implementation is in place. In contrast, τCFI makes no assumptions on the presence of a backward-edge protection. Further, τCFI provides a technique for protecting forward edges and does not rely on a shadow stack approach for protecting backward edges.

8 Conclusion

In this paper, we have presented τCFI, a new control flow integrity (CFI) technique, which can be used to protect program control flow graph (CFG) forward edges and backward edges in executables during runtime. For the protected stripped (*i.e.,* no RTTI information) x86-64 binaries, we do not need to make any assumptions on the presence of an auxiliary technique for protecting backward edges (*i.e.,* shadow stacks, etc.) as τCFI protects these transfers, too. We have evaluated τCFI with real world open source programs and have shown that τCFI is practical and effective when protecting program binaries. Further, our evaluation reveals that τCFI can considerably reduce the forward-edge legal calltarget set, provide high backward-edge precision, while maintaining low runtime overhead.

Acknowledgement

We thank the anonymous reviewers for their feedback, which helped to considerably improve the quality of this paper. Jens Grossklags' research is supported by DIVSI. Gang Tan is supported by US NSF grants CCF-1723571 and CNS-1624126, the Defense Advanced Research Projects Agency (DARPA) under agreement number N6600117C4052, and Office of Naval Research (ONR) under agreement number N00014-17-1-2539. Zhiqiang Lin is partially supported by US NSF grant CNS-1812553 and CNS-1834215, AFOSR award FA9550-14-1-0119, and ONR award N00014-17-1-2995.

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control Flow Integrity. In: CCS (2005)
2. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control Flow Integrity Principles, Implementations, and Applications. In: TISSEC (2009)
3. Andriesse, D., Chen, X., Veen, V.v.d., Slowinska, A., Bos, H.: An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In: USENIX Security (2016)
4. Andriesse, D., Slowinska, A., Bos, H.: Compiler-Agnostic Function Detection in Binaries. In: EuroS&P (2017)
5. Balakrishnan, G., Reps, T.: DIVINE: Discovering Variables in Executables. In: VMCAI (2007)
6. Bernat, A.R., Miller, B.P.: Anywhere, Any-Time Binary Instrumentation. In: PASTE (2011)
7. BlueLotus Team: Bctf challenge: Bypass vtable read-only checks (2015), https://goo.gl/4RYDS2
8. Bounov, D., Gökhan K., R., Lerner, S.: Protecting C++ Dynamic Dispatch Through VTable Interleaving. In: NDSS (2016)
9. Bruening, D.: DynamoRIO. http://dynamorio.org/home.html
10. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.: BAP: A Binary Analysis Platform. In: CAV (2011)
11. Burow, N., Carr, S., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-Flow Integrity: Precision, Security, and Performance. In: CSUR (2017)
12. Caballero, J., Lin, Z.: Type Inference on Executables. In: CSUR (2016)
13. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.: Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In: USENIX Security (2015)
14. Crane, S., Volckaert, S., Schuster, F., Liebchen, C., Larsen, P., Davi, L., Sadeghi, A.R., Holz, T., De Sutter, B., Franz, M.: It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In: CCS (2015)
15. Dang, T., Maniatis, P., Wagner, D.: The Performance Cost of Shadow Stacks and Stack Canaries. In: ASIACCS (2015)
16. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nurnberger, S., Sadeghi, A.R.: MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In: NDSS (2012)
17. Elsabagh, M., Fleck, D., Stavrou, A.: Strict Virtual Call Integrity Checking for C ++ Binaries. In: ASIACCS (2017)
18. Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., Sidiroglou-Douskosr, S.: Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In: CCS (2015)
19. Fokin, A., Derevenets, Y., Chernov, A., Troshina, K.: SmartDec: Approaching C++ decompilation. In: WCRE (2011)
20. Goktas, E., Oikonomopoulos, A., Gawlik, R., Kollenda, B., Athanasopoulos, E., Giuffrida, C., Portokalidis, G., Bos, H.: Bypassing Clang's SafeStack for Fun and Profit. In: Blackhat Europe (2016), https://goo.gl/zKMHzs
21. Haller, I., Goktas, E., Athanasopoulos, E., Portokalidis, G., Bos, H.: ShrinkWrap: VTable Protection Without Loose Ends. In: ACSAC (2015)
22. Jang, D., Tatlock, T., Lerner, S.: SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In: NDSS (2014)
23. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-Pointer Integrity. In: OSDI (2014)

24. Lan, B., Li, Y., Sun, H., Su, C., Liu, Y., Zeng, Q.: Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. In: IEEE Trustcom/BigDataSE/ISPA (2015)
25. Lee, B., Song, C., Kim, T., Lee, W.: Type Casting Verification: Stopping an Emerging Attack Vector. In: USENIX Security (2015)
26. Lettner, J., Kollenda, B., Homescu, A., Larsen, P., Schuster, F., Davi, L., Sadeghi, A.R., Holz, T., Franz, M.: Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In: USENIX ATC (2016)
27. Lin, Z., Zhang, X., Xu, D.: Automatic Reverse Engineering of Data Structures from Binary Execution. In: NDSS (2010)
28. LLVM: Clang's SafeStack. `https://clang.llvm.org/docs/SafeStack.html.
29. LLVM: Clang CFI (2017), `https://goo.gl/W7aMF9
30. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K.W., Franz, M.: Opaque Control-Flow Integrity. In: NDSS (2015)
31. Mycroft, A.: Lecture Notes (2007), `https://goo.gl/F7tUZj
32. Niu, B., Tan, G.: Modular Control-Flow Integrity. In: PLDI (2014)
33. Niu, B., Tan, G.: RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Inegrity. In: CCS (2014)
34. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In: USENIX Security (2013)
35. Pawlowski, A., Contag, M., van der Veen, V., Ouwehand, C., Holz, T., Bos, H., Athanasopoulos, E., Giuffrida, C.: MARX:Uncovering Class Hierarchies in C++ Programs. In: NDSS (2017)
36. Payer, M., Barresi, A., R. Gross, T.: Fine-Grained Control-Flow Integrity through Binary Hardening. In: DIMVA (2015)
37. Prakash, A., Hu, X., Yin, H.: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In: NDSS (2015)
38. Ramalingam, G.: The Undecidability of Aliasing. In: TOPLAS (1994)
39. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit Object-Oriented Programming. In: S&P (2015)
40. Tan, G., Jaeger, T.: CFG Construction Soundness in Control-Flow Integrity. In: PLAS (2017)
41. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In: USENIX Security (2014)
42. Veen, V.v.d., Andriesse, D., Göktas, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical Context-Sensiticve CFI. In: CCS (2015)
43. Veen, V.v.d., Goktas, E., Contag, M., Pawlowski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A Tough call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In: S&P (2016)
44. Zhang, C., Song, C., Zhijie, K.C., Chen, Z., Song, D.: vTint: Protecting Virtual Function Tables' Integrity. In: NDSS (2015)
45. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical Control Flow Integrity & Randomization for Binary Executables. In: S&P (2013)
46. Zhang, M., Sekar, R.: Control Flow Integrity for COTS Binaries. In: USENIX Security (2013)