# Picon: Control Flow Integrity on LLVM IR

Thomas Coudray, Arnaud Fontaine[1], and Pierre Chifflier[1]
{pierre.chifflier,arnaud.fontaine}@ssi.gouv.fr
thomas.coudray.fr@gmail.com

[1] Agence Nationale de la Sécurité des Systèmes d'Information

**Abstract.** Control flow integrity is a well explored field of software security for more than a decade. However, most of the proposed approaches are stalled in a proof of concept state – when the implementation is publicly available – or have been designed with minimal performance overhead as main objective, sacrificing security. Currently, none of the proposed approaches can be used to fully protect real-world programs compiled with most common compilers (e.g. GCC, Clang/LLVM). In this paper we describe a control flow integrity enforcement mechanism for LLVM IR, called Picon, whose main objective is security. Our approach is based on compile-time code instrumentation, making the program communicate with its external execution monitor. The program is terminated by the monitor as soon as a control flow integrity violation is detected. Our approach is implemented as an LLVM plugin and is working on LLVM's Intermediate Representation.

## 1 Introduction

Traditional program exploitation by an attacker often involves bypassing the size of a buffer to write to an arbitrary address in memory, and then redirecting execution to the code newly written to this address. This has lead to the introduction of protections to prevent these problems. Stack *canaries* [10] add random values between frames in the call stack, to detect stack overflows, and equivalent protections exist to prevent heap overflows. Data Execution Prevention (DEP) [2] adds a separation between data, which can be read or written, and code, which should be executable and never written. It can be enforced by the hardware, e.g. NX (No-eXecute) bit on x86, XN (eXecute-Never) on ARM.

The generalization of these protections, now widely used in modern operating systems, has changed the typical form of exploits to work around them. In addition, the separation between code and data in $W \oplus X$ is not so clear in real programs: some data are interpreted not directly as code, but as an indirect way of executing code. This is the case of the return address, which specifies the address of the instruction to be executed when returning from a function. This address, if modified, can be used by

an attacker to execute some existing code in the application, leading to code-reuse attacks.

The initial attack vector has to be provided by the attacker, usually in a data buffer. Even when these data are not directly executable, the attacker can specify a sequence of return addresses, each of them pointing to some instructions followed by a return. By choosing the effect of these instructions, the executed code can be controlled by the attacker. This technique is known as Return-Oriented Programming, or ROP [21] and has been proved Turing-complete using a set of gadgets from the standard GNU C library. Other techniques involve code-reuse such as Jump-Oriented Programming [6] which removes the reliance to the stack and the return instruction. String Oriented Programming (SOP) [20] and Signal Return Oriented Programming (SROP) [8] are also based on the same principle.

To protect the execution of a program against code-reuse attacks, a common technique is to randomize the memory layout of a program on every program execution, using Address Space Layout Randomization (ASLR). This way, memory addresses change at each program execution and are harder to predict. While powerful, this technique is not sufficient, mostly because addresses of some parts of the program may leak or be guessed, but also because of remaining problems like format-string vulnerabilities [17]. Because randomization is done for entire sections at once, the discovery of one address often means defeating the entire randomization. In some other cases, brute-force techniques or blind-ROP [4] can be used to detect the small parts of code preceding a return, also known as *gadgets* (ROP Widgets).

To protect the execution flow of a program, other techniques must be used in addition to existing protections, such as *control flow integrity*.

### 1.1 Control flow integrity

Program exploitation often subverts the intended data flow in a vulnerable program. This in turns makes it possible to hijack the control flow in order to control the program behaviour. Control Flow Integrity (CFI) provides a protection against control flow hijacking attacks. The CFI property was formalized by Abadi *et al.* in 2005 [1]. In this paper, CFI is used to enforce the program execution to follow only paths existing in the Control Flow Graph (CFG), obtained using static analysis of the binary. Then, the binary is rewritten to add checkpoints before branch instructions, along with tags to check that the target is in accord with the predicted/expected control flow.

One of the difficult parts of CFI is the extraction of the control flow graph. It can be recovered statically on the source code [28], on the binary file (using structural analysis with tools like Hex-Rays decompiler, for example), or dynamically (e.g. execution profiling [27]). Some of these reconstructions may be incomplete, when a branch cannot be predicted precisely, or when the analyzed form does not match the code actually produced by the compiler (modified by some optimizations).

A first way to classify control flow integrity methods is based on the type of input program: some of them work on the original sources [13,15,26], while some others work on the assembly, or binary form of the program [1,7,29]. Working at the source code level allows one to extract a precise control flow graph, and more information about the program. On the other hand, working at the binary level allows one to protect programs without requiring the source code, but is architecture-dependent and inherently less precise.

The second classification method is based on the type of component responsible for enforcing the security policy, called an *Execution Monitor*. The enforcement of security policies by monitoring executions was formalized by Schneider in 2000 [24], with the definition of *safety properties*, later extended by Basin *et al.* [3]. An execution monitor can be internal or external to the protected program. It can also have a different granularity depending on the enforcement policy mechanism used. An execution monitor must be tamper-proof to ensure control flow integrity, and have the ability to terminate the process in case the policy is not respected.

An inlined execution monitor integrates the verification code into the program code during compilation or *via* binary rewriting. An inline monitor shares the same address space as the monitored program, and the verification code has to be protected in the binary itself.

An execution monitor can also be externalized, *i.e.* be moved out of the program. In this case, the monitor "observes" the program execution, and checks that the expected control flow of the program is respected. On one hand, this approach is more secure than an inlined monitor as the monitor as its own logic, independent from the monitored program. It can be implemented in different locations: it can be a user process, a kernel module, an hypervisor, *etc.* The more hardened the monitor is, the harder it will be for an attacker to compromise the binary protected using CFI. On the other hand, since the monitor must be able to observe quite precisely the execution flow of some program, and to kill it in case an unexpected execution flow is detected, an external monitor is a very sensitive "process". Actually, some additional care must thus be taken to ensure the monitor

itself cannot be subverted and/or compromised. Depending on how the execution flow of the monitored program is concretely observed by the monitor, the runtime cost of the external approach is by definition more costly than the integrated/inlined one.

The main advantage of using an external monitor is to globally improve the security offered by a CFI protection: an attacker needs to control both the program and its monitor to successfully exploit a vulnerability.

### 1.2 Limitations of current approaches

Most implementations of control flow integrity make trade-offs between security and performance. This implies removing some of the checkpoints, for example by not instrumenting function calls, and only taking care of return instructions.

However, removing protections leaves the protected program vulnerable to attacks, because some gadgets are still available for an attacker. As shown in [12] and [11], most control flow integrity implementations have been tested and demonstrated to be quite permissive in still allowing an attacker to build ROP attacks.

Other techniques, like forward-edge CFI [26], or control flow guard [5,13] as implemented recently in Windows, only protect indirect forward calls, and thus only provide partial protection.

Another technique, called control flow locking, was introduced in [7] to mitigate these attacks. The method relies on a lock that is set before any control flow change, and that the next instrumented point will verify some predefined condition before unlock it and permit the execution to continue. However, this work was limited to statically linked binaries.

**Unaligned ROP gadgets** Many of the ROP gadgets found in binaries consist of unaligned instructions that have not been produced by the compiler, but that happen to be interpretable as valid instructions by the processor. This mainly concerns the `x86` architecture, due to the number and the repartition of possible opcodes, and the fact that instructions are not required to be aligned on this architecture. This limitation is also shared by most implementations of CFI on these architectures.

It is possible, however, to analyze the instructions from the binary file, and apply translations or randomization of instructions, and insertion of neutral instructions automatically, to ensure that the unaligned gadgets are replaced by other sequences, as described in [18].

However, even when the unaligned gadgets are removed from a binary, some gadgets still remains, because of the control flow, especially the

function returns. As a complement of the elimination of unaligned gadgets, the program still needs to be protected so that an attacker will not be able to use any of the two kinds of gadgets.

### 1.3 Control flow integrity on LLVM IR

In this paper, we present a practical approach to control flow integrity, with some similarities to control flow locking, but with different properties.

Our work has several key properties:

— **external monitor**: while the communication with the monitor reduces performances, the isolation increases the protection;
— **complete**: the protection is not partial, and not limited to a few points (e.g. function returns);
— **automatic**: the protection must be automatic, well-integrated with existing build tools, and not be a burden for the developer;
— **portable**: the protection works on different architectures, and does not rely on a disassembly of specific binaries;
— **support of shared libraries**: the protection supports programs that are linked with shared libraries, and this does not break the chain of verifications.

This paper is organized as follows. In Section 2, we describe a formalization of our model of control flow integrity on LLVM IR, based on a pushdown automaton. In Section 3, we describe an implementation of the proposed model, called Picon, as a plugin for the LLVM compiler, and a separate process for the execution monitor. In Section 4, we analyze the security impact on binaries protected by Picon, and discuss the results.

## 2 Theoretical foundations

The goal of this section is to formalize our model of control flow integrity on LLVM IR, and to show that it can enforce strong protection against most common attack patterns. One of the main advantages of this model is to permit easier debugging and *a posteriori* analysis of a program execution whose control flow integrity has been compromised. The last part of this section shows how to tackle a forthcoming implementation issue of the model, with identical security guarantees and reasonable debugging and *a posteriori* analysis features.

### 2.1 Control flow integrity model

Roughly, the control flow integrity model proposed is based on an *execution monitor* [24] whose goal is to enforce a security policy described

by a pushdown automaton (*i.e.* a state machine equipped with a stack). In order to formally define the *control flow integrity policy* enforced, some notations and definitions need to be introduced.

First of all, the class of programs considered needs to be defined. Basically, a program is a set of functions, one of which is the single entrypoint of the program, *i.e.* the *main* function.

**Definition 1 (Program).** *A* program $\mathcal{P}$ *is defined as a pair* $(\mathcal{F}, f_{\mathcal{P}})$ *where* $\mathcal{F}$ *is a finite non-empty set of functions defined and/or called in* $\mathcal{P}$ *(directly or transitively) and* $f_{\mathcal{P}} \in \mathcal{F}$ *is the function corresponding to the main and unique entrypoint of the program.*

In LLVM IR, a function is a set of basic blocks, each of which is a sequence of LLVM IR instructions. Only a very small subset of these instructions has to be considered to enforce control flow integrity property. Actually, it is useless to modelize the behaviour of instructions that cannot alter the control flow execution such as arithmetic/logic operations. Only instructions that influence the control flow are thus modelized. To keep the model as simple as possible, LLVM IR instructions with equivalent semantic according to the control flow definition are grouped and an abstract instruction is introduced to represent each group.

**Definition 2 (Basic block).** *A* basic block *in LLVM IR is a finite non-empty sequence of instructions. Among the set of all possible LLVM IR instructions, only the four following abstract instructions are considered:* `call`*,* `ret`*,* `unreachable` *and* `branch`*.*

*A* `call` *instruction interrupts the execution of its enclosing basic block to execute some function, and so recursively. It corresponds to the LLVM instructions* `call` *and* `invoke`*.*

*A* `ret` *instruction can only occur as the last instruction of a basic block. Its execution stops definitively the execution of the current function (and thus of its enclosing basic block), and the execution restarts with the instruction immediately following the last executed* `call` *instruction. The LLVM corresponding instructions are* `ret` *and* `resume`*.*

*A* `branch` *instruction targets a finite non-empty set of basic blocks. Its execution selects one of the targeted blocks as the next one to be executed based on some condition. If only one basic block is targeted, the branch is unconditional to the single targeted block. It corresponds to the LLVM instructions* `br`*,* `indirectbr`*, and* `switch`*.*

*A* `unreachable` *instruction is a special instruction to indicate an unreachable statement and its execution is equivalent to a "do nothing". It corresponds to the LLVM instruction* `unreachable`*.*

*The last instruction of a basic block is systematically a* `ret`*, a* `branch` *or a* `unreachable` *instruction.*

The organization of basic blocks within a function is constrained in LLVM IR. Each function has a single entry block which has no predecessors (*i.e.* it is not possible to jump on the entry block from within the function). Basic blocks with no successors are terminated by a `ret` instruction, when the function returns to its caller, or a `unreachable` instruction which is a special LLVM IR instruction to indicate an unreachable statement.

**Definition 3 (Control flow graph (CFG) of a function).** *The* control flow graph *of a function $f$ is a directed graph $G_f = (V_f, E_f)$ where $V_f$ is a finite non-empty set of vertices consisting in the set of basic blocks of $f$, and $E_f \subseteq V_f \times V_f$ is a finite set of edges. An edge $(b_1, b_2)$ exists in $E_f$ iff the basic block $b_1$ ends with a* `branch` *instruction targeting a set of basic blocks containing $b_2$.*

*The main and unique entry of a function $f$ is a basic block $b \in V_f$ denoted* $\mathsf{entry}(f)$ *such that* $\forall b \in V_f\ (b, b') \in E_f \implies b' \neq \mathsf{entry}(f)$.

*The finite set* $\mathsf{exit}(f) \subseteq V_f$ *denotes the set of basic blocks with no successors and terminated by a* `ret` *instruction. The finite set* $\mathsf{unreachable}(f) \subseteq V_f$ *denotes the set of basic blocks with no successors and terminated by a* `unreachable` *instruction. There exists no basic block* $b \in V_f \setminus (\mathsf{exit}(f) \cup \mathsf{unreachable}(f))$ *containing an instruction* `ret` *or* `unreachable`*, or without successors.*

Interactions between functions defined and called/used in the program must be defined. These interactions have to be completely and statically known at the model level in order to enforce a control flow integrity property. This assumption may seem strong, but any missing interaction will be detected as a control flow integrity violation, so that integrity is not compromised.

**Definition 4 (Call graph of a program).** *Let $\mathcal{P} = (\mathcal{F}, f_{\mathcal{P}})$ be a program. The* call graph *of $\mathcal{P}$ is a directed graph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, B_{\mathcal{P}})$ where $V_{\mathcal{P}} = \mathcal{F}$ is its set of vertices, $E_{\mathcal{P}} \subseteq V_{\mathcal{P}} \times V_{\mathcal{P}}$ is its set of edges, and $B_{\mathcal{P}}$ is its edge labeling function.*

*An edge $(f, g) \in E_{\mathcal{P}}$ denotes that function $f$ is calling function $g$. This edge is attached a label denoted $B_{\mathcal{P}}(f, g) \subseteq V_f$ consisting in the subset of basic blocks of $f$ in which $g$ is called, with $G_f = (V_f, E_f)$ the CFG of $f$.*

Some sequences of instructions have to be inserted in the program either to report an upcoming execution flow change to an external entity

able to kill the program, or to enforce the control flow policy directly within the program which will terminates if compromised. At the model level, both alternatives are equivalent. Each kind of sequence to be inserted is called a *hook*, and insertion of these hooks is called *instrumentation*. The instrumentation step is crucial for later definition of control flow integrity policy enforcement since a control flow integrity violation will be detectable only where a hook is inserted.

Roughly, a hook is inserted before any bifurcation of execution flow occurs, that is just before `call` instructions with a hook called `cfiCall`, `ret` and `unreachable` instructions with a hook called `cfiExit`, and `branch` instructions with a hook called `cfiBeforeJump`. Although it is required to control *before* the execution flow is modified, it is also important to insert some control *after* the execution flow is modified, that is at the entry of a function with a hook called `cfiEnter`, at the entry of a basic block with a hook called `cfiAfterJump`, and at the return of function calls with a hook called `cfiReturned`.

**Definition 5 (Program instrumentation).** *An* instrumented program *is a program $\mathcal{P} = (\mathcal{F}, f_{\mathcal{P}})$, denoted $\overline{\mathcal{P}}$, verifying all the following properties for each function $f \in \mathcal{F}$ and each basic block $b \in V_f$ appearing in its CFG $G_f = (V_f, E_f)$:*

- *each* `call` *instruction to a function $f' \in \mathcal{F}$ in $b$ is immediately preceded by the sequence of instructions corresponding to the* `cfiCall` *$f'$ hook, and immediately followed by the sequence of instructions corresponding to the* `cfiReturned` *$f'$ hook;*
- *if $b \in \mathsf{exit}(f) \cup \mathsf{unreachable}(f)$ then the last instruction of the block is immediately preceded by the sequence of instructions corresponding to* `cfiExit` *$f$ hook, otherwise $b$ ends with a* `branch` *instruction immediately preceded by the sequence of instructions corresponding to* `cfiBeforeJump` *$(f, b)$ hook;*
- *if $b = \mathsf{entry}(f)$ then $b$ starts with the sequence of instructions corresponding to the* `cfiEnter` *$f$ hook, otherwise $b$ starts with the sequence of instructions corresponding to the* `cfiAfterJump` *$(f, b)$.*

An instrumented program contains sufficient hooks to protect its control flow integrity.

**Definition 6 (Control flow integrity policy).** *Let $\overline{\mathcal{P}} = (\mathcal{F}, f_{\mathcal{P}})$ be an instrumented program and $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, B_{\mathcal{P}})$ its call graph. The* control flow integrity policy *for program $\overline{\mathcal{P}}$ is described by a deterministic pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where $Q = \{q_e, q_c, q_r, q_b\}$ is its*

*set of states,* $\Sigma = \{\texttt{cfiCall}\ f, \texttt{cfiEnter}\ f, \texttt{cfiExit}\ f, \texttt{cfiReturned}\ f \mid f \in V_{\mathcal{P}}\} \cup \{\texttt{cfiBeforeJump}\ (f,b), \texttt{cfiAfterJump}\ (f,b) \mid f \in V_{\mathcal{P}}, b \in V_f\}$ *is its set of inputs,* $\Gamma = \{\langle f, \sigma\rangle \mid f \in V_{\mathcal{P}}, \sigma \in V_f^*\}$ *is its finite set of stack symbols,* $q_0 = q_c$ *is its initial state,* $Z_0 = \langle f_{\mathcal{P}}, \mathsf{entry}(f_{\mathcal{P}})\rangle$ *is its initial stack symbol,* $F = \{q_e\}$ *is its set of accepting states, and* $\delta \in (Q \times \Sigma \times \Gamma) \to \wp(Q \times \Gamma^*)$ *is its transition function such that*

$$\delta(q_e, \texttt{cfiCall}\ f', \langle f, b\sigma\rangle) = \begin{cases} \{(q_c, \langle f', \mathsf{entry}(f')\rangle\langle f, b\sigma\rangle) \mid (f, f') \in E_{\mathcal{P}}\} & \textit{iff } b \in B_{\mathcal{P}}(f, f') \\ \emptyset & \textit{otherwise} \end{cases} \tag{1}$$

$$\delta(q_r, \texttt{cfiReturned}\ f', \langle f, b\sigma\rangle) = \begin{cases} \{(q_e, \langle f, b\sigma\rangle) \mid (f, f') \in E_{\mathcal{P}}\} & \textit{iff } b \in B_{\mathcal{P}}(f, f') \\ \emptyset & \textit{otherwise} \end{cases} \tag{2}$$

$$\delta(q_e, \texttt{cfiExit}\ f, \langle f, b\rangle) = \begin{cases} \{(q_r, \epsilon)\} & \textit{iff } b \in \mathsf{exit}(f) \\ \emptyset & \textit{otherwise} \end{cases} \tag{3}$$

$$\delta(q_c, \texttt{cfiEnter}\ f, \langle f, b\sigma\rangle) = \begin{cases} \{(q_e, \langle f, b\sigma\rangle)\} & \textit{iff } b = \mathsf{entry}(f) \\ \emptyset & \textit{otherwise} \end{cases} \tag{4}$$

$$\delta(q_e, \texttt{cfiBeforeJump}\ (f,b), \langle f, b\sigma\rangle) = \{(q_b, \langle f, b\sigma\rangle)\} \tag{5}$$

$$\delta(q_b, \texttt{cfiAfterJump}\ (f,b'), \langle f, b\sigma\rangle) = \{(q_e, \langle f, b'b\sigma\rangle) \mid (b, b') \in E_f\} \tag{6}$$

*where* $G_f = (V_f, E_f)$ *is the CFG of* $f$ *and* $\epsilon$ *denotes an empty sequence.*

*An* instantaneous description *of* $\mathcal{M}$ *is a triple* $(q, \omega, \beta) \in Q \times \Sigma^* \times \Gamma^*$ *describing a situation of* $\mathcal{M}$ *where* $q$ *is a state of the automaton,* $\omega$ *is a sequence of inputs to treat, and* $\beta$ *is a stack.*

When no transition exists in the automaton for a given input according to its current state and stack, it indicates that the control flow integrity of the program has been compromised. As a consequence, the compromised program must be immediately terminated.

**Proposition 1 (Detection of compromised CFI).** *If the control flow integrity of an instrumented program* $\overline{\mathcal{P}}$ *is compromised while it is protected by the control flow integrity policy given in Definition 6, then* $\overline{\mathcal{P}}$ *is terminated at the first hook encountered in the resulting execution flow or by the end of* $\overline{\mathcal{P}}$ *itself.*

*Proof.* Let $\overline{\mathcal{P}} = (\mathcal{F}, f_{\mathcal{P}})$ be an instrumented program and $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ the automaton describing the control flow integrity policy enforced on $\overline{\mathcal{P}}$. Let $(q, \omega, \beta)$ be the instantaneous description of $\mathcal{M}$ just after the exploited instruction $i$ in basic block $b$ of $f$ is executed.

By definition, instruction $i$ can only be a `call`, a `ret`, a `branch` or a `unreachable` instruction as only those instructions can influence the control flow.

If $i$ is a `call`, then it is immediately preceded by a `cfiCall` $f'$ hook. So, according to the definition of the transition function $\delta$, the only valid values for $q$ and $\beta$ are $q_c$ and $\langle f', \mathsf{entry}(f') \rangle$, respectively. From this instantaneous description, the only valid next input is `cfiEnter` $f'$, which can be emitted only when $f'$ is called, by definition.

If $i$ is a `branch`, then it is immediately preceded by a `cfiBeforeJump` $(f, b)$ hook. So, according to the definition of the transition function $\delta$, the only valid values for $q$ and $\beta$ are $q_b$ and $\langle f, b\sigma \rangle$, respectively. From this instantaneous description, the only valid next input is `cfiAfterJump` $(f, b')$ with $(b, b') \in E_f$, which can be emitted only by jumping on an expected basic block, by definition.

If $i$ is a `ret`, then it is immediately preceded by a `cfiExit` $f$ hook. So, according to the definition of the transition function $\delta$, and because $b \in \mathsf{exit}(f)$ by definition, the only valid values for $q$ and $\beta$ are $q_r$ and $\langle f', b'\sigma \rangle$, respectively, where $(f', f) \in E_{\mathcal{P}}$ and $b' \in B_{\mathcal{P}}(f', f)$. From this instantaneous description, the only valid next input is `cfiReturned` $(f, b')$, which can be emitted only after returning from a call to $f$ occurring in basic block $b'$ of $f'$, by definition.

If $i$ is a `unreachable`, then it is immediately preceded by a `cfiExit` $f$ hook. Since $b \notin \mathsf{exit}(f)$, there exists no definition of this transition in $\delta$. So this instantaneous description does not exist as the `unreachable` instruction cannot have been executed. □

When a violation of the control flow integrity policy occurs, the stack of the monitor contains the call stack trace, *i.e.* function calls and basic blocks trace for each function called. This information is crucial for debugging purposes but also for *a posteriori* analysis of compromised program execution. However, defined as is, the sequence of basic blocks explored within a function only grows and will grow very quickly in presence of cycles in control flow graphs.

### 2.2 Partial tracing of intra-procedural executions

Keeping basic blocks sequences in the automaton stack is useless for enforcing control flow integrity policy as only the topmost (*i.e.* currently executing) basic block of the stack is used in practice. The main benefit of keeping the complete trace of executed basic blocks is to provide very precise information for *a posteriori* analysis of control flow integrity violation. If only the last basic block were kept, it would be very rough to understand how a violation occurred. As a compromise, we propose in this section to keep incomplete sequences of basic blocks in a way that guarantees a bounded size for any basic block sequences without losing too much information for *a posteriori* analysis of compromised execution.

In order to build only finite sequences of basic blocks in the transition corresponding to the input `cfiAfterJump` of the control flow integrity policy, we rely on the *domination relationship*. Computation of this relation permits to discover the set of basic blocks that will systematically be explored/executed from a given basic block in order to reach the exit basic block of a function. Consequently the sequence of explored basic blocks will be extended only if the basic block prepended to the current sequence cannot be executed anymore before the end of the function.

**Definition 7 (Post-dominator).** *Let* $G_f = (V_f, E_f)$ *be the* control flow graph *of a function* $f$ *with a single entry basic block denoted* $\mathsf{entry}(f)$ *and a single exit basic block denoted* $\mathsf{exit}(f)$.

*A basic block* $b_1 \in V_f$ post-dominates *a basic block* $b_2 \in V_f$ *such that* $b_1 \neq b_2$, *noted* $b_1 \in \mathsf{pd}(b_2)$ *iff* $b_1$ *is involved in every path from* $b_2$ *to* $\mathsf{exit}(f)$ *in the CFG.*

Knowing that a basic block $b_1$ post-dominates a basic block $b_2$ is sufficient to decide whether $b_1$ can be prepended to the current sequence starting with $b_2$ when $b_1$ is executed. However, a more efficient test can be defined than an inclusion in a set if the *immediate post-dominator* relationship is used. A basic block $b_1$ is the immediate post-dominator of a basic block $b_2$ if it is the first basic block that will be systematically explored/executed to reach the end of the function when $b_2$ is executed.

**Definition 8 (Immediate post-dominator).** *Let* $G_f = (V_f, E_f)$ *be the* control flow graph *of a function* $f$.

*A node* $b_1 \in V_f$ *is the* immediate post-dominator *of a node* $b_2 \in V_f$, *noted* $b_1 = \mathsf{ipd}(b_2)$, *iff* $b_1 \in \mathsf{pd}(b_2)$ *and* $\nexists b \in V_f \; b_1 \in \mathsf{pd}(b) \wedge b \in \mathsf{pd}(b_2)$.

Given these definitions, the transition rule associated to the `cfiAfterJump` hook in Definition 6 is modified to push/prepend the

basic block to be executed only if it is the immediate post-dominator of the last pushed/prepended one. In order to maintain the Proposition 1 proved in the previous section, the top of the stack (*i.e.* the first basic block of the sequence) must always be the currently executing one. So, when the immediate domination condition is not verified, the top of the stack is updated to always contain the currently executing basic block.

$$\delta(q_b, \texttt{cfiAfterJump}\ (f, b'), \langle f, b\sigma\rangle) = \begin{cases} \{(q_e, \langle f, b'b\sigma\rangle) \mid (b, b') \in E_f\} & \text{iff } b' = \mathsf{ipd}(b) \\ \{(q_e, \langle f, b'\sigma\rangle) \mid (b, b') \in E_f\} & \text{otherwise} \end{cases}$$

The size of the sequence of basic blocks explored is now bounded by the number of basic blocks in the CFG of the currently executing function.

## 3 Implementation

To experiment with our proposed CFI protection on LLVM IR, an implementation has been developed, called PICON [1], based on the LLVM compiler framework [16] version 3.5. This implementation supports any program compiled by the Clang frontend, which is the "LLVM native" C/C++/Objective-C compiler.

### 3.1 Overview

PICON is implemented in a two-step process, to follow the definitions given in the previous section:

1. during compilation, a plugin instruments the code;
2. at runtime, an external execution monitor implements the state automaton to enforce the control flow integrity policy of the instrumented program.

Unlike others [15,19], we have chosen not to fork the LLVM compiler, but rather to create a dynamically loaded module for the `opt` tool to implement the compilation step. Currently, compilation of an input source file (C or C++) by the Clang frontend produces a file in the LLVM Intermediate Representation (IR), which is the common code representation used throughout all target-independent phases of the LLVM compilation process. We have chosen to instrument the LLVM IR because of the following advantages:

1. Protect Integrity of CONtrol flow

— easier to handle than C or C++;
— input/source language independent;
— architecture independent;
— well structured into functions and basic blocks, each of which contains instructions in Static Single Assignment (SSA) form.

While this choice may restrict possible actions only to the APIs exported by LLVM, this also greatly reduces the dependency on LLVM internal functions, and makes maintenance easier (especially to keep the plugin up to date, LLVM being a very active project). The compilation workflow, including the PICON plugin step, from a C file to a final binary is depicted on Figure 1.

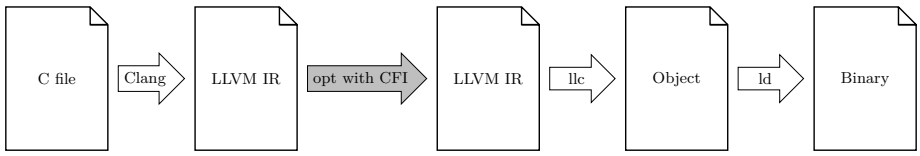


**Fig. 1.** Compilation of a single source file with the PICON plugin in gray.

The main goal of the plugin is to instrument the code to introduce communication hooks with an external execution monitor. However, it is also in charge of producing several files where essential data is stored: *identifiers* generated for functions and basic blocks to handle separate compilation units and dynamically linked libraries, and *transition tables* which contain all control flow related data that will be passed to the external execution monitor. In fact, when an instrumented program is executed, it requires the execution monitor to be running with the corresponding transition tables loaded.

It is important to note that the instrumentation and the creation of transition tables is done in a completely *static* and *automatic* manner.

### 3.2 Instrumentation of the LLVM IR

The PICON plugin has two levels of granularity. One can decide to protect only inter-procedural control flow (*i.e.* function calls), or both inter- and intra-procedural (*i.e.* basic block transitions) control flow.

The instrumentation is a two-step process. First, unique identifiers are computed and assigned to each function, and each basic block if intra-procedural protection is activated. Then, the instrumentation code is

injected to communicate with the execution monitor, relying on identifiers previously computed to "name" functions and basic blocks.

**Attribution of identifiers** A unique identifier is assigned to each function and to each basic block, when appropriate; these identifiers are later denoted *idFun* and *idBB*, respectively.

Assigning unique identifiers to each function may appear trivial. However, in case of binaries created from multiple source files, some difficulties arise because function identifiers must be identical for the same functions across different compiler executions. To solve this problem, the plugin creates and maintains across compilations a file where each function already encountered is mapped to its unique identifier. When a call to a function not yet defined is encountered, the plugin assigns it a new unique identifier according to those already used and updates the file. Algorithm 1 depicts this straightforward algorithm. An analogous process is applied to compute and to assign a unique identifier to each basic block per function.

**Algorithm 1** Function Identifier Attribution at Compile Time

```
1: procedure GetFunctionIdentifier
2:     for f in all functions do
3:         if HasAlreadyBeenIdentified(f) then
4:             idFun ← GetIdentifier(f)
5:         else
6:             idFun ← GetUniqRandomID(f)
```

Once a unique identifier is assigned to each function and each basic block, the plugin creates the transition tables according to the desired level of granularity for the control flow integrity protection, *i.e.* with(out) intra-procedural control flow. The inter-procedural transition table exactly consists in the set of edges appearing in the *call graph* along with the edge labeling function (Definition 4), so if the plugin has to build this transition table, it iterates over all functions and all basic blocks of these functions of the compilation unit to build its call graph. Building the intra-procedural transition table is a completely analogous process, but applied to each function for which it has to build its *control flow graph* (Definition 3).

**Code instrumentation** Code instrumentation must be done according to the Definition 5 in the model section. That is, instrumentation consists in inserting some *hooks*, *i.e.* predefined sequences of instructions, at strategical

positions in the code, to report execution flow bifurcations to the execution monitor, as shown in the Definition 6.

There are several ways to implement these hooks: a hook can be a call to a custom function, a specific syscall, a *jump* to some inlined basic block, *etc*. It is important to note that a hook is a sensitive piece of code that must be written carefully not to introduce vulnerable code such as gadgets. In the implementation proposed, we have chosen to implement each hook by a custom function call. Figure 2 gives an example of a non-instrumented `foo` function, and Figure 3 shows the same `foo` function with injected instrumentation code (both inter- and intra-procedural related hooks).

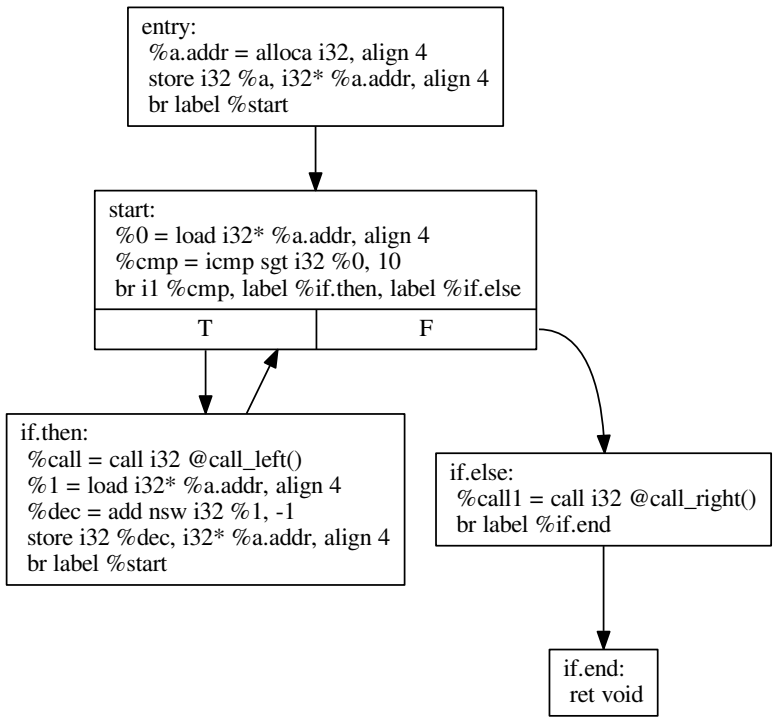


**Fig. 2.** Example CFG of a `foo` function.

According to the level of granularity set in the plugin, only a subset of the six available hooks may be inserted according to the Definition 5. For

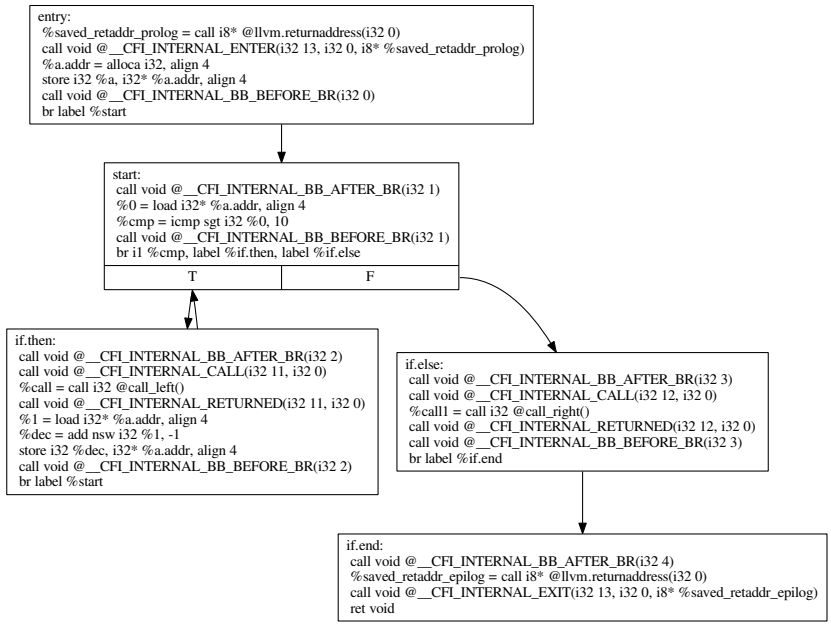


**Fig. 3.** CFG of the instrumented `foo` function of Figure 2.

the mandatory inter-procedural protection, the four following hooks are systematically inserted: `cfiCall`, `cfiReturned`, `cfiEnter` and `cfiExit`. If intra-procedural protection is activated, then the two following hooks are also inserted: `cfiAfterJump` and `cfiBeforeJump`.

### 3.3 Resolution of externals

The compilation of a source file is a local process: the LLVM compiler only has information on the file being compiled. This causes problems when handling calls to external functions. In particular, it is not possible at that point to distinguish functions that will be defined in another object file linked into the same executable from functions that will be stored in external shared libraries. In the following, the term *module* designates a single binary compilation target, for example an executable file, or a shared library.

A module identifier, later denoted *idMod*, is assigned for the complete binary target being compiled (all object files part of the same binary share the same module identifier). The module identifier is generated at compile-time, it has to be unique and deterministic. For example, it can be derived from a cryptographic hash of the binary.

When analyzing a C file, it is not possible to know if a function, e.g. `printf`, will be defined in the same binary or in a shared library before the link step. A function defined in a shared library can also be shadowed by a function with the same name in the current binary.

We have decided not to try identifying the modules of functions during the compilation process, because it is not easy, or even feasible. Instead, function identifiers are automatically assigned. These identifiers are *relative* and unique to the current module. This method allows the compilation process to remain simple, but results in a new problem: the identifier of a function $f$ will not be the same in module $m_1$ and in module $m_2$.

The compilation process has to be modified to add new steps in order to identify the symbols and the different modules, and to add information to the created files for the monitor. The modified compilation workflow is depicted on Figure 4.

After the compilation process, there is one identifier file per resulting binary (a standalone executable, or shared library, for example), containing function identifiers. We have to bind the caller module identifier with the callee module identifier. This process is done in two steps: *symbol identification* and *symbol binding* (described in the next section). The symbol identification step resolves dynamically linked function and their identifiers. For this, a Python script analyzes the compiled binary using

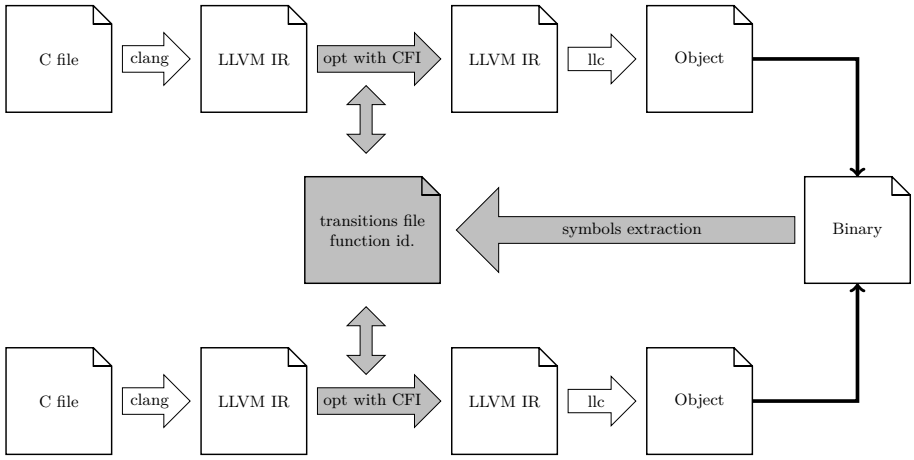


**Fig. 4.** Modified compilation process, with externals

`objdump`, and finds all symbols related to external functions. Each external symbol is then searched for recursively in each shared library dependencies of the binary, to identify in which library it is defined, and thus to find the associated module and transitions files. The transitions file of the binary is then updated to link each symbol to the identifier of the found module.

If a function is defined in several libraries, the binary instrumented with PICON will only be allowed to call the one that matched the function identification. This provides a protection against library replacement, or symbol override by one of the libraries.

The identification of symbols is described in Algorithm 2. To be correct, this algorithm must follow the resolution of symbols as done by the `ld` loader, otherwise the function that will effectively be called at runtime will not be authorized.

**Algorithm 2** Module Identifier Merging after the linker pass

```
1: procedure ResolveExternalSymbols
2:     libs ← GetAllLibrariesRecursively(binary)
3:     for sym in all symbols of binary do
4:         if IsExternalSymbol(sym) then
5:             lib ← GetLibraryContaining(sym, libs)
6:             ModuleId ← GetFileIdentifier(lib)
7:             UpdateModuleIdentifierForFunction(binary, sym, ModuleId)
```

For example, the `printf` function, is first marked as an external symbol in `a.out.cfi`. Using `ldd` and `objdump` recursively, the symbol is found in `/lib/x86_64-linux-gnu/libc.so.6`. The corresponding identifiers file is `libc.so.6.cfi`, which has been created during the compilation of `libc.so.6` with PICON enabled. Using this file, we check that an identifier exists in order to verify the transitions, but, at this point, the exact identifier of the function is not important. Finally, we update the binary's identifiers file and add the information that the `printf` function is associated with the module identifier of `libc.so.6`.

### 3.4 Execution monitor

The execution monitor is externalized from the instrumented binary; it can be implemented in different places: for example in a user process or in the kernel, as described in Section 1.1. In PICON, the execution monitor is implemented as an external process, which forks and uses the child to run the instrumented program, and uses pipes to communicate. To ease the burden of storing the transitions files, and running the monitor before each instrumented program, the following enhancements have been added:

— the transition files are embedded directly in the instrumented files, by adding custom ELF sections;
— code to re-exec the monitor is inserted, so running the instrumented binary will really run the monitor, setup the monitor, and run the instrumented program.

When the execution monitor starts, it looks in the ELF section headers of the instrumented binary for a PICON *description file*, describing the needed information for that binary. The PICON *description file* contains the unique module identifier *idMod* of the binary, its dependencies, and the transition table. The monitor must then recursively load all transition files and dependencies. If they are embedded, it is important to ensure the PICON description files nor the transition tables are modified. A simple solution is to use an asymmetric signature to sign the file headers, so that the monitor will be able to verify the integrity of the headers.

The transition table contains the allowed transitions inside the binary described in Section 3.2. Dependencies indicate the list of transition tables required for external libraries. Algorithm 3 describes how the monitor loads the transition files, and marks identifiers for the same function in different modules as equivalent.

Each time a function $f$ is used (or defined), the pair $(m_i, f_i)$ is added to the *equivalence class* of $f$. After loading all the transition files, if the

**Algorithm 3** Loading and unifying the transition files in the monitor

```
1: procedure MergeTransitionsFile
2:     for m in all modules do
3:         m_id ← GetModuleIdentifier(m)
4:         for f in all functions(m) do
5:             f_id ← GetFunctionIdentifier(f)
6:             AddEquivalence(f, (m_id, f_id))
```

function $f$ is used in different modules, its equivalence class contains a list of tuples $[(m_1, f_1), (m_2, f_2), \ldots]$.

When a function $f$ in a module $m_1$ is about to call function $g$ in module $m_2$, the PICON plugin has inserted a `cfiCall` with the identifiers $(m_1, g_1)$, that is, the identifier of $g$ as seen in module $m_1$. To verify this function call in $m_2$, the monitor will verify during `cfiEnter` of $g$ that the value $(m_2, g_2)$ as seen in module $m_2$ is equivalent to $(m_1, g_1)$.

Each time the monitored process hits a PICON hook, it notifies the execution monitor with current information about the context, as defined in the Definition 6 of the model section: the current module and function identifiers, and return address for `cfiEnter`. The monitor updates the state of the process being instrumented. Two types of unauthorized behaviours can be detected: *state mismatch*, and *identification mismatch*.

After a `cfiCall` instrumentation, if the next instrumentation is a `cfiBeforeJump`, the execution monitor triggers a state mismatch because `cfiEnter` is expected after a `cfiCall`. The list of all possible automaton states is described in Figure 5.

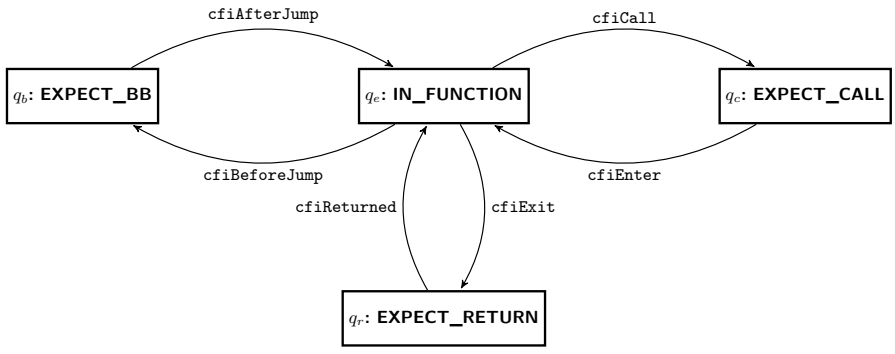


**Fig. 5.** Execution monitor's state machine.

An identification mismatch can happen, for example, when a `cfiCall` registers a function identifier to be called, and a different function identifier from the CFG allowed-transition (defined at compile time in the *transitions file*) is provided during the `cfiEnter`. Identification can also mismatch for a basic blocks transition when `cfiBeforeJump` provides a given *idBB*, and `cfiAfterJump` provides a different one from the CFG expected one.

Identification mismatch and state mismatch both trigger an alert from the execution monitor depending on the security policy used. The best action is to kill the instrumented process, but the execution monitor can also log the unexpected behaviour with precise information about the transition for debugging purposes.

## 4 Discussion and results

### 4.1 Security evaluation

The evaluation of the security of the protected program is done by comparing the number of gadgets in the original binary, and in the protected one. While it is not possible to prevent code-reuse attacks globally, our objective is to reduce the number of available ROP gadgets as much as possible, and to verify that tools cannot reconstruct a shellcode.

**Return-to-libc attacks** Return-to-libc is the perfect candidate to bypass the well known *NX protection*. With Picon protection applied to all functions dynamically linked to an instrumented binary, it is possible to prevent this type of attack by denying calls to forbidden functions of the `libc` like `system` or `execve` beyond their expected uses.

**Return-oriented programming attacks** Picon can successfully instrument all non-dynamic *call* instructions, which results in a significant decrease of the usable ROP gadgets. To successfully bypass our model, attackers have to find ROP gadgets that are not protected by Picon. However, as seen before, CFI instrumentation is widely used in a protected binary, and to fully create a reliable ROP gadgets payload, attackers have to build their entire payload while taking care not to fall in an instrumented portion of code, which will result in an execution monitor alert.

Picon, by instrumenting all return sequences in compiled programs, avoids all these potential ROP gadgets. Our current implementation does, however, keep the linked C runtime unchanged, which has 6 such gadgets in the `glibc` version of `crt1.o` used:

— `_init` which is preceded by an `add` and a `call` instruction.
— `_deregister_tm_clones`, preceded by a `pop %rbp` and a `ja` instruction.
— `register_tm_clones`, preceded by a `pop %rbp` and a `ja` instruction.
— `__do_global_dtors_aux`, preceded by a `movb`, `pop %rbp`, and `call` instruction.
— `__libc_csu_init`, preceded by a `popa` and a `call` instruction.
— and `__libc_csu_fini`.

Another source of gadgets is the PICON runtime itself, which embeds a few functions containing potential gadgets: seven functions contains potential gadgets, but five of them are strictly identical in term of sequence of instructions.

Using a standard disassembler, we measured the number of potential gadgets in a shared library, the Better String Library [2]. With standard compiler, 134 potential gadgets were found. With PICON enabled, only 9 potential gadgets on aligned instructions remain.

The following programs were also tested, looking for potential aligned gadgets in PICON protected binaries, including all their dependencies:
— `star` contains 13 gadgets, 4 protected;
— `quark` c1ntains 17 gadgets, 8 protected;
— `puzzle solver` contains 17 gadgets, 8 protected;
— `sha1sum` contains 18 gadgets, 9 protected.

This confirms that the remaining potential gadgets on aligned instructions are those previously described, *i.e.* coming from PICON and C runtimes. All return instructions in these binaries are unusable ROP gadgets.

Note that tools like ROPgadgets [23] will search gadgets in the entire program and also in unaligned instruction stream. This increase the resulting number of gadgets available, but will be greatly reduced (to the number of gadget protected by PICON minus previous gadgets from the runtime) with the In-place Code randomization [18].

**Return address attacks** Another way to hinder ROP is to replace the return address when entering a function (using `cfiEnter` hook), and restore it by the execution monitor in `cfiExit` hook. This means that, between the entry and the exit point of a function, the return address is invalid, to complicate even more the work of an attacker. We have implemented this extra feature in PICON, but it depends on some security-oriented changes to the target-specific code generator, which is target

2. http://bstring.sourceforge.net/

and architecture-dependent. The modification of LLVM to allow the modification of the return address in a portable way is ongoing work in LLVM project, and has not yet been finalized. We plan to submit it to upstream LLVM later, as there might be other uses of this feature.

**Jump-oriented programming attacks** Possibilities of Jump-Oriented attacks are reduced, since the source code must not use indirect jumps or calls. All calls or jumps are statically known during the compilation, so the attacker cannot gain any gadget to jump to a non-instrumented point. This, however, adds strong limits on input files: indirect calls are used in C++ vtables, for example.

Different solutions exist to add support of indirect calls while retaining protection of the control flow. The first solution is to use the instrumentation of forward function calls which is currently being added to Clang [25]. Forward-edge CFI could be used as a complement of our protection, and protect indirect calls using restricted jump tables. Forward-edge CFI is done during LTO and would not be integrated by Picon, so another solution is to use the information provided by LLVM to instrument indirect branches and dynamic calls in our model. This is left for future work.

### 4.2 Implementation remarks

**Compiler optimizations** In some cases, compiler optimizations introduce a change in the symmetry of enter/exit points of functions, for example the tail-call optimization. This optimization changes the instructions of a function to transform recursive function calls into iterative execution of basic blocks, heavily modifying the structure and the contents of the function.

To avoid this kind of problems, the Picon pass must be the last pass executed on the LLVM IR. Other optimizations must be applied before, especially those modifying the control flow graph.

**Execution environment** As our implementation uses an external monitor, it is critical to ensure the security of communications with the monitor. The process and the monitor should be mutually authenticated, and the integrity of the communication channel should be ensured to avoid Man-In-The-Middle (MITM) classes of attacks.

In Picon, the communication channel between the monitor and the instrumented binary is a pair of unnamed pipes. This requires, however, the monitor and the process to be created in the same process hierarchy.

Another possible attack is to preload shared libraries (for example using `LD_PRELOAD`) to override some functions, most importantly the functions used to communicate with the monitor. To avoid that, the execution environment should be restricted to prevent preloading custom libraries, for example using the `noexec` mount option to prevent the user to be able to build libraries and use them, and/or by patching the `ld` command to remove the preload feature. Another workaround is to set file capabilities on the instrumented program using SELinux or any other mechanism to disable the preload feature.

**Authorized functions whitelist** Sometimes, the program has to be linked with closed-source binary-only dynamic libraries. PICON has the abilities to handle these cases, and implements a whitelisting mechanism to permit non-instrumented calls to/returns from some given functions. When compiling a program with PICON enabled, instrumentation will not be inserted in functions present in the whitelist, and the transition will not be verified. However, it is clear that excluding dynamic libraries of the control flow integrity is insecure, and results in the addition of free ROP gadgets in the resulting executable.

A workaround, for closed-source libraries, could be to implement the same protection by disassembling the file, reconstructing the control flow graph, and adding the hooks to protect it. This has several drawbacks: notably, it is not portable, and rebuilded basic blocks is not as precise as computing basic blocks from source code.

### 4.3 Limitations

**Multi-programming** Our implementation is currently not able to handle program with parallel/concurrent programming, *i.e.* multiple threads/processes. One straightforward way to override this limit is to instrument concurrency-related system calls (`clone(2)` and `fork(2)` on POSIX systems, for instances).

By injecting specific instrumentation code for these calls, it has been possible to successfully detect the creation of new processes, and to instantiate a dedicated execution monitor for each process. Each monitor had its own dedicated state machine, and was able to monitor one process. However, more work is required to fully implement multi-programming support in PICON, but also to support multi-threading.

**Parallel compilation** When compiling different source files of a program with PICON enabled, each file requires information about other files, for

example function identifiers in the transitions file as shown in Figure 4. This requires a sequential compilation because of a race condition on the access to the transition file. A solution could be to implement locking on the transition file, to ensure only one instance of the compiler process can modify it at the same time.

**Dynamic code** To implement the CFI protection, we rely on the construction of the control flow graph statically, and thus are not able to track dynamic function calls as used in just-in-time (JIT) compilation, exceptions, pointer arithmetic on function addresses or dynamic loading of shared libraries. This limitation is shared by most CFI approaches.

### 4.4 Future work

**Binary instrumentation** One fundamental limitation of our approach is the requirement of the source code, and the need to compile them. When the source code of a program is not available, as it is usually the case for commercial software, or it is not supported by Clang, instrumentation of a program is not possible with our current compile-time instrumentation.

To instrument binary files, one solution is to decompile the executable into LLVM IR, apply the Picon pass on the resulting LLVM IR, and then compile it back to an executable. However, the binary translation environment provides some additional challenges. Static translation has some fundamental limitations, due to its equivalence to the halting problem [14] making it undecidable. One such limitation is the presence of indirect branches and calls, which do not have statically discoverable targets. In practice, indirect branches are usually caused by the following constructions:

— indirect gotos (a rarely used feature of the C and other languages);
— switch lowering to jump-tables (a compiler optimization).

In the case of an executable that has not been specially crafted, indirect calls targets are expected to be in the set of all function symbols available in the binary, and can usually be recovered by static or dynamic analysis.

Projects such as Dagger [9] have been successfully tested, and provide a straightforward method to instrument executable files without requiring the source code.

**Link-time optimization** Modern compilers such as LLVM support a feature called Link-Time Optimization (LTO), which defers code generation to link-time, and keeps the intermediate object files in LLVM IR.

Traditionally, this was used with great success to enable optimizations otherwise impossible on isolated file (such as cross-object function inlining). The PICON pass could be implemented as a part of LTO IR optimizations. This would solve the problems related to cross-object transition table and identification uniquing, by having all functions visible in a single IR module. Also, LTO improves precision by making the program's complete control flow graph available. Finally, since LTO passes run as part of the `ld` linker, it is also possible to directly use the linker for resolving external function symbols in linked shared libraries, thus avoiding the need of symbols identification at compile-time.

**Picon and obfuscation mechanisms** PICON protects the binary for execution integrity, but does not hide the instrumentation, or the control flow graph of the binary. Other protection mechanisms, especially obfuscation techniques such as o-llvm [19] at the LLVM IR layer, or a Protector Packer [22] like *UPX* at the binary layer, could be used in addition to CFI protection.

However, obfuscation and CFI might interfere. The obfuscation must not break the CFI protection by altering the semantics of the program. The obfuscation step must not create dynamic-code/self modifying code (sometimes used in virtualized packer) nor add gadgets for the obfuscation step. Although CFI and obfuscation could be used together, the CFI adds extra information about the CFG that could help an attacker to reconstruct the logic of the program.

## 5 Conclusion

In this paper, we have discussed a model for robust control flow integrity protection, and the security properties of programs protected by this model. A proof-of-concept implementation has been proposed, based on the LLVM compiler framework, and an external monitor. The result is a plugin for the LLVM compiler called PICON, which does not complicate the compilation process. The plugin allows a global protection of the program, including shared libraries, without having to sacrifice parts of the protection.

As complementary, simpler protections like prevention of execution of the stack and randomization become commonplace, we believe that control flow integrity will become more systematic in the future as it is a key part of the protection against ROP attacks. The protection of control

flow integrity must be complete to be powerful, and thus must not be weakened for the sake of performances.

## References


1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. pages 340–353, 2005.
2. Starr Andersen and Vincent Abella. Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, Data Execution Prevention. `https://technet.microsoft.com/en-us/library/bb457155.aspx`. Accessed: 2015-01-21.
3. David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable Security Policies Revisited. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, volume 7215 of *Lecture Notes in Computer Science*, pages 309–328. Springer Berlin Heidelberg, 2012.
4. Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. pages 227–242, 2014.
5. R.J. Black, T.W. Burrell, M.O.T. de Castro, M.S. Da Silva Costa, K. Johnson, and M.R. Miller. Control flow integrity enforcement at scale, October 24 2013. US Patent App. 13/450,487.
6. Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. pages 30–40, 2011.
7. Tyler K. Bletsch, Xuxian Jiang, and Vincent W. Freeh. Mitigating code-reuse attacks with control-flow locking. pages 353–362, 2011.
8. Erik Bosman and Herbert Bos. Framing Signals - A Return to Portable Shellcode. pages 243–258, 2014.
9. Ahmed Bougacha, Geoffroy Aubey, Pierre Collet, Thomas Coudray, Amaury de la Vieuville, and Jonathan Salwan. Dagger: decompiling to LLVM IR. *LLVM Europe*, 2013.
10. Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. pages 5–5, 1998.
11. Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. May 2015.
12. Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. May 2014.
13. Jim Hogg. Visual Studio 2015 Preview: Work-in-Progress Security Feature. `http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx`. Accessed: 2015-01-21.
14. R. Nigel Horspool and Nenad Marovac. An Approach to the Problem of Detranslation of Computer Programs. *Comput. J.*, 23(3):223–229, 1980.

15. Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer Integrity. pages 147–163, 2014.
16. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. pages 75–88, Mar 2004.
17. Kyung-Suk Lhee and Steve J. Chapin. Buffer Overflow and Format String Overflow Vulnerabilities. *Softw. Pract. Exper.*, 33(5):423–460, April 2003.
18. Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. pages 601–615, 2012.
19. Grégory Ruch Pascal Junod, Julien Rinaldini. Obfuscator-LLVM. `http://www.o-llvm.org`. Accessed: 2015-01-21.
20. Mathias Payer and Thomas R. Gross. String Oriented Programming: When ASLR is Not Enough. pages 2:1–2:9, 2013.
21. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.
22. Kevin A. Roundy and Barton P. Miller. Binary-code Obfuscations in Prevalent Packer Tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013.
23. Jonathan Salwan. ROPgadget. `https://github.com/JonathanSalwan/ROPgadget`. Accessed: 2015-01-21.
24. Fred B. Schneider. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.*, 3(1), February 2000.
25. The Clang Team. Clang 3.7 documentation: Control Flow Integrity. `http://clang.llvm.org/docs/ControlFlowIntegrity.html`. Accessed: 2015-04-08.
26. Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. pages 941–955, 2014.
27. Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. pages 1–12, 2012.
28. Zhongxing Xu, Ted Kremenek, and Jian Zhang. A Memory Model for Static Analysis of C Programs. pages 535–548, 2010.
29. Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. pages 337–352, 2013.