Fine-Grained Call-Graph Analysis for Forward-Edge Control-Flow Integrity in LLVM

Fin
Cofibrant

Abstract

Coarse-grained forward-edge control-flow integrity schemes – type-tag matching, as used by kCFI, Control Flow Guard, and IFCC – constrain an indirect call to "any function with a compatible signature." That constraint is far weaker than it looks: any two functions taking the same number and kind of pointer arguments are interchangeable under it, which is enough to hide `execve`-shaped targets behind an innocuous-looking callback signature. We present a static, whole-module analysis that resolves, per indirect call site, the *exact* set of functions reachable through data flow – not merely type-compatible with it – and a code-generation scheme, implemented as an LLVM pass, that enforces that set at runtime with a per-call-site read-only jump table, a packed pointer encoding that removes the added dependency chain, and a branch-predictor-friendly guard. The typical call site benefits substantially: the modal resolved target-set size across our corpus is roughly 40× smaller than the modal set size under type-flow filtering alone. Evaluated on FFmpeg, TinyCC, and OpenSSL, a liveness pass followed by a greedy-plus-local-search index assignment also reduces jump-table memory by roughly 8.5× (17,435 to 2,053 pointer-widths) relative to a naively encoded baseline, and liveness alone cuts the tracked function set by 4.8× (9,594 live functions to 2,000 actually used as indirect-call targets). The transform passes the full LLVM test suite unmodified (100% correctness) and is performance- neutral at the macro level; a gem5 microarchitectural study isolating the indirect-call path shows branch-predictor behavior at parity with, or better than, the unprotected baseline once the packed-pointer optimization removes the dependent-load chain.

1 Introduction

Consider a call through a function pointer sitting on the stack next to an unbounded buffer:

```
typedef void f_ptr(void);
int good(void) { /* ... */ }
void bad(void) { /* ... */ }

int main(void) {
    char buf[32];
    f_ptr f = good;
    gets(buf); // no bound on input length
    f();
    return 0;
}
```

`gets` will write past `buf` into whatever the compiler placed adjacent to it on the stack. If that happens to be `f`, the attacker now controls where `f()` transfers control – to `bad`, or to anywhere else a crafted 64-bit value can name. This is not a new observation [1]; it is the motivating case for forward-edge control-flow integrity as a class of defense.

The standard mitigation replaces the raw pointer with a checked index:

```
static f_ptr *const table[8] = { good, 0,0,0,0,0,0,0 };

int main(void) {
    char buf[32];
    int idx = 0;
    gets(buf);
    idx &= 7; // mask into table bounds
    table[idx]();
    return 0;
}
```

`idx` is corrupted exactly the way `f` was, but the *consequence* of that corruption is now bounded: the mask confines control to the table, and the table at this call site holds exactly the functions the call site can legitimately reach – here, only `good`. An attacker can no longer redirect control to an arbitrary address; at worst they select among a small, statically known, per-call-site set.

Two questions follow immediately. First, what should populate that table – in particular, can we do better than "every function with a matching type," which is what every existing type-tag scheme settles for? Second, can the added indirection be made cheap enough to be worth deploying? This paper answers both: Section 4 presents a data-flow analysis that resolves call-site target sets far more tightly than type matching alone, Section 5 shows how to keep the resulting tables small, and Section 6 shows the runtime cost is not measurable at the macro level and favorable at the microarchitectural level once a packing optimization removes an added dependency chain.

This is a *forward*-edge mechanism – it protects calls, not returns. An analogous backward-edge scheme (replacing a return address with a checked index into a table of legal return sites) is sketched in Section 9, but LLVM IR's control-flow instructions cannot express a branch to an arbitrary point inside another function; that requires operating after instruction selection, where function boundaries are no longer opaque. This paper implements and evaluates the forward-edge mechanism only.

2 Background

2.1 Existing Forward-Edge Controls

Table 1 summarizes the deployed alternatives. Software mitigations (DEP, ASLR, stack canaries) reduce the exploitability of the underlying memory-safety bug

but do not constrain *where* a successfully hijacked indirect call can go. Hardware mitigations narrow that further: Intel CET's indirect branch tracking permits landing anywhere prefixed with `endbr64` [2] – i.e., any valid function entry in the program, which is coarser than even type-based filtering. ARM MTE tags allocations and traps cross-tag accesses, at the cost of address-space bits, and is a memory-safety control rather than a CFI one specifically [3].

Control	Constrains target to
DEP	non-executable data (no constraint on valid-code targets)
ASLR	unknown base address (probabilistic, not a bound)
Stack canaries	detects stack overwrite *after* corruption, before use
Intel CET	any `endbr64`-prefixed function entry
kCFI / CFG / IFCC	any function of compatible type
This work	**the resolved, per-call-site target set**

Table 1: Forward-control-flow constraints of existing mechanisms.

2.2 Why Attackers Still Chain Gadgets

Buffer overflows that survive DEP and ASLR typically proceed by chaining existing code – return-oriented programming (ROP) [4], its jump- and call-oriented variants (JOP, COP), and signal-frame-based SROP [5] – rather than injecting new code. CET narrows JOP and COP by restricting valid indirect-branch landing sites, but, as above, only down to "any function start," not to the targets a given call site can actually reach. This gap is exactly where the analysis in Section 4 adds value over CET and over type-tag schemes alike.

2.3 LLVM Function Pointers

LLVM IR treats modifying a function pointer's bit pattern via arithmetic as undefined behavior; the only defined operations are equality, assignment, and calling. This matters for the transform: since no valid program can rely on the address of a function pointer beyond identity, it is legal to globally rename every function value in a module to a packed index/pointer pair (Section 3) without breaking any program that does not already rely on undefined behavior.

3 Design

3.1 Per-Call-Site Jump Tables

For a call site whose resolved target set is $\{f_0, \ldots, f_n\}$ (Section 4), the indirect call is rewritten to index a read-only global table containing exactly those n functions, padded with null entries out to the next power of two minus one so the index mask is a single bitwise AND, instead of calling through the raw pointer. There is no requirement that each call site own a distinct table – Section 5 shows call sites with overlapping target sets can share one.

3.2 Removing the Dependent Load

A direct implementation replaces one indirect call with two *serially dependent* loads: compute the index, then load the table entry at that index, then call it – a longer critical path than the original code had. Since x86-64 and AArch64 both leave the upper 16 of a 64-bit pointer's bits unused in practice, the transform instead packs a per-target index into those bits at the point each function value is created, so the index and the pointer travel together as one value with no load in between:

$$\texttt{packed} = (\texttt{index} \ll 48) \mid \texttt{function\_pointer}$$

At the call site, `packed` splits into `index` (top 16 bits) and `pointer` (bottom 48 bits) via two independent masks – independent, so they can execute in parallel – and the call proceeds only if the table entry at `index` equals `pointer`:

```
idx = top16(x);
ptr = bottom48(x);
if (table[idx] == ptr) call ptr();
else { report(); exit(1); }
```

On mismatch the process exits before the call executes. This is the same security property as the naive table lookup – a value diverted from its legitimate target is caught before use – with one dependent load removed from the hot path.

3.3 Recovering the Branch Cost

The added equality check is a conditional branch on a path that was previously unconditional. Because a mismatch occurs only under active attack, the branch is annotated `llvm.expect` to always predict taken, letting the CPU speculatively execute the call before the comparison resolves. Section 6 shows this removes essentially all of the transform's remaining measurable cost.

3.4 Where in the Pipeline

Forward-edge protection is fully expressible at the LLVM IR level: IR still has function calls as first-class constructs, so indices can be attached to values and checked before a `call` instruction is emitted. Backward-edge protection cannot be expressed this early – IR's `br` has no notion of jumping into the middle of another function – and must wait until instruction selection has lowered functions into concrete addresses (Section 9). Section placement for the read-only tables themselves also matters: IR-level `constant` globals are a hint the backend usually, but is not obligated to, honor by placing them in `.rodata`; marking the section explicitly and referencing it via `llvm.used` prevents the optimizer from eliminating an apparently-unread table.

3.5 Crossing Module Boundaries

A function pointer that reaches an externally visible parameter can no longer be resolved by this module's analysis alone – the caller could be anywhere. Two options apply:

1. **Duplicate for precision.** If `f(f_ptr x)` is called both internally, with a fully resolved `x`, and externally, split it into an `internal` variant reached only from resolved call sites (keeping the tight table) and an `external` ABI-visible entry point that degrades to a mask-only check. More generally, for

a chain of wrapper calls $f(x) \to g(x) \to h(x)$, only the call sites whose full argument history is internal can be duplicated this way; the question of *which* wrappers are worth duplicating, when a function is reached by both fully- and partially-resolved paths, is an optimization problem left to future work.

2. **Remove the boundary.** Whole-program compilation via LTO, or languages whose compilers already emit one module per program – Rust's compiler does this by construction, since it has no stable ABI to preserve across a module boundary – eliminates the problem entirely, since every call site is then internal to the analysis.

4 Resolving Call-Site Target Sets

4.1 Type-Flow Filtering

LLVM's own ABI compatibility rules already bound which callees are legal at a given call site: calling convention must match; for non-`tailcc` conventions, ABI-impacting attributes and full prototypes must match, with pointer parameters required only to agree on address space, not pointee type. This is the *entire* precision budget available to kCFI, CFG, and IFCC.

4.2 Data-Flow Resolution

Type filtering alone cannot exclude a same-signature function that is provably unreachable from a specific call site. We add a backward fixpoint over the values that can reach a call's called operand: stores and loads (including one level of indirection through a GEP, to catch writes into a computed array slot); PHI nodes; global variable initializers; arguments threaded across other calls, including one level of return-value tracking through resolved direct callees; and, for indirect calls whose target set is itself only partially known, the union over every currently-candidate callee's own parameter flow. Each traced value becomes a set-equation variable; an unhandled construct (a volatile access, or a value with no further traceable definition) marks that variable *unbounded*, which propagates and forces the call site back to type filtering alone. The system is solved to a fixpoint, since a value's resolved set can only grow, and is bounded above by the number of live functions in the module.

At the module level, this produces one constraint per call site of the form

$$A \supseteq A_0 \cup B_1 \cup \cdots \cup B_n$$

where A_0 is the statically resolved part and each B_i is another constraint variable (an argument, a global, a return value) contributing to it. We solve this system directly rather than via a general points-to solver such as Andersen's [6]: general points-to analysis solves points-to for every pointer in a program, whereas the restriction to function-pointer flow, motivated in Section 2.3, admits a narrower and faster fixpoint. We also considered SVF, which solves closer to the intended problem, but it targets whole-program statically linked binaries and is not built for the per-module, incremental setting this transform runs in.

4.3 A Concrete Precision Gap

The gap this analysis closes is not abstract:

```
int square(int x) { return x*x; }
int cube(int x) { return x*x*x; }
int increment(int x) { return x+1; }
int potentially_evil(int x) { /* ... */ }

const fp funcs[3] = {square, cube, increment};
// dispatch via funcs[i % 3] in a loop
```

Type filtering alone cannot exclude `potentially_evil`: it shares the signature `int(int)` with the other three, and signature compatibility is the only information a type-tag scheme has. Every pointer-typed parameter can additionally be reached from a more general signature such as `int f(void*, void*, void*)`, which is exactly the shape of `execve(const char*, char *const[], char *const[])` – so "a function of compatible type exists somewhere in the binary" is a materially weaker guarantee than it sounds. Our analysis produces

```
[ptr @square, ptr @cube, ptr @increment, ptr null, ...]
```

for this call site, against the type-filtered table a kCFI/CFG/IFCC-style scheme would allow:

```
[ptr @square, ptr @cube, ptr @increment,
 ptr @potentially_evil, ptr null, ...]
```

This is not an isolated example: across our corpus, the *modal* resolved target-set size – the set size occurring most frequently across all indirect call sites – is roughly 40× smaller under this data-flow analysis than the modal set size under type-flow filtering alone. The typical call site is not merely helped by data-flow resolution; type filtering alone is typically a poor approximation of it.

4.4 Aggregates and Concurrency

The analysis as described treats a struct's fields as one merged set – a call through `x.f1` is checked against the union of every field ever assigned in `x`, not just `f1`'s own assignments. This is sound but not maximally precise; per-field tracking is possible at the IR level (structs and arrays are both just indexed lists of operands) but complicated by aliasing through arithmetic on element pointers, and is left as future work. Values passed across threads or processes, and volatile accesses, are treated as unresolvable by construction: a thread argument from another module is already unresolvable by the module-boundary rule above, and volatile accesses are permitted by the LLVM Language Reference to expose state not otherwise reachable via ordinary loads and stores, so no assumption about their value survives.

5 Reducing Table Memory

A distinct table per call site, each sized to the number of live functions, costs $O(nm)$ for n functions and m call sites – the naive baseline measured below.

5.1 Liveness Filtering

Only functions ever observed *as a value* – taken as a pointer, not merely called by name – need an index at all. Table 2 shows this alone cuts the tracked set from

9,594 live functions to 2,000 actually used at an indirect call site, across our test corpus.

	Count
Live functions (taken as a value)	9,594
Used at an indirect call site	2,000

Table 2: Liveness filtering removes over three-quarters of tracked functions.

5.2 Optimal Index Assignment

Given the resolved target sets, the remaining freedom is how to assign indices to functions so each table's addressed span – $\max(f(S_i)) - \min(f(S_i))$ for target set S_i under assignment f – is as small as possible; a table only needs to store the span it is addressed over, not the full function count. Formally: find a bijection $f : S \to \{1, \ldots, |S|\}$ minimizing

$$\sum_i \big(\max(f(S_i)) - \min(f(S_i))\big).$$

This is NP-hard: restricting every S_i to size 2 reduces it directly to the minimum linear arrangement problem, a known NP-hard problem, by taking $S_i = \{a, b\}$ for each edge (a, b) of the MinLA input graph. We therefore use a two-stage heuristic.

Algorithm 1: Greedy ordering

```
GreedyOrder(sets S_1..S_n):
  order S_1..S_n ascending by size
  L = []; seen = {}
  for S in ordered sets:
    for a in S:
      if a not in seen:
        append a to L; add a to seen
  return L # index of a is its position in L
```

Ordering sets from smallest to largest and assigning free indices greedily is optimal whenever sets don't overlap much – the common case in practice, confirmed below. A local-search refinement pass then tries, for each set, swapping a member's index with the span's current endpoints, keeping the swap only if it reduces total cost, iterated to a fixpoint:

Algorithm 2: Pairwise refinement

```
Refine(sets S_1..S_n, f):
  for S in sets:
    l = min(S, f); u = max(S, f)
    for a in S with l <= f(a) <= u:
      if swapping f(a) <-> l improves cost:
        keep it; continue
      if swapping f(a) <-> u improves cost:
        keep it; continue
```

Refinement is a local search over transpositions; since every permutation decomposes into transpositions, repeated application can in principle reach any permutation, though nothing guarantees a monotonically improving path exists from a given start. In practice we run it to a fixpoint (cost stops decreasing) rather than for a fixed iteration count.

5.3 Empirical Results

Table 3 measures jump-table memory, in pointer-widths, on files drawn from FFmpeg, TinyCC, and OpenSSL.

Method	Table cost (ptrs)
Naive (one table per call site)	17,435
After deduplicating identical sets	6,509
Greedy ordering	2,053
Greedy + refinement	2,053
Random restart + refinement (baseline)	5,471

Table 3: Jump-table memory across FFmpeg, TinyCC, and OpenSSL.

Greedy ordering alone reaches the same cost as adding refinement on this corpus, meaning refinement's extra $O((mn)^2)$ cost buys nothing further here; both comfortably beat a randomly permuted-then-refined ordering, confirming that the greedy heuristic – not the refinement step – is doing the real work. Net effect: roughly 17 KB of jump-table memory per program becomes roughly 2 KB, an 8.5× reduction. Figure 1 shows the per-file distribution behind the aggregate numbers, and Figure 2 isolates greedy vs. refined cost per file to show how close to optimal greedy alone gets.

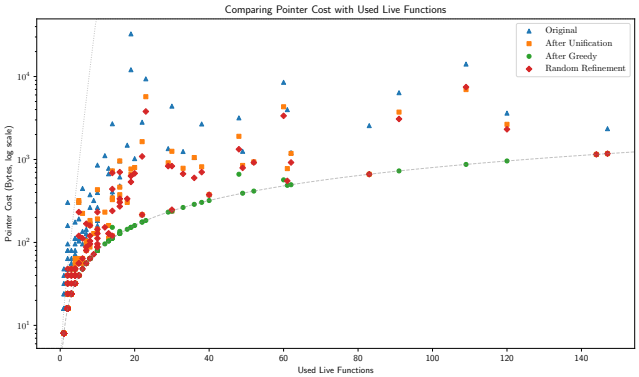

Figure 1: Per-file table cost across methods.

5.4 Further Reductions

If one call site's target set is a subset of another's under a compatible index ordering, the smaller table can be dropped entirely and its call site redirected into the larger one. In the worst case – every subset of a fixed live-used set actually occurring as some call site's target set – placing every table needed reduces to the shortest superpermutation problem, for which tight bounds are known [7], [8]; this bound is not encouraging in principle ($n! + O((n-1)!)$), but Figure 3 shows it is not the regime real programs land in: reuse of the same function pointer set across many call sites is limited in practice, so the naive-with-liveness-and-ordering result above already captures nearly all of the available reduction.

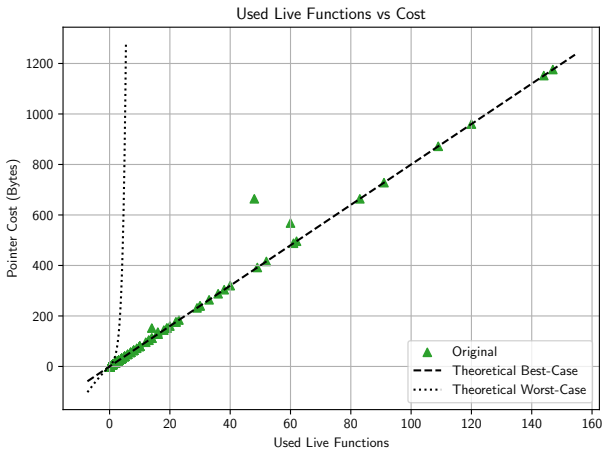

Figure 2: Greedy vs. refined cost per file: refinement rarely improves on greedy.

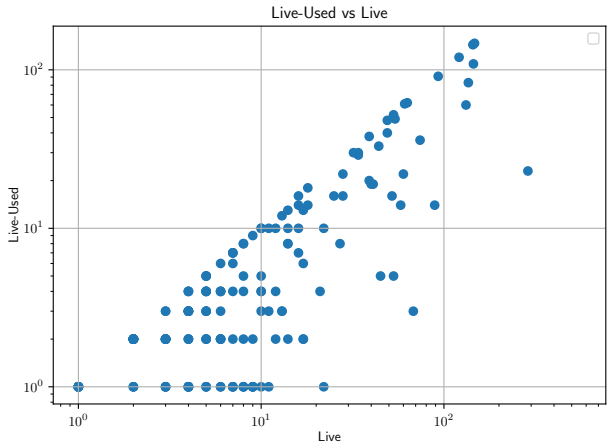

Figure 3: Live vs. used-live functions per file.

6 Evaluation

6.1 Correctness

The full LLVM test suite passes at a 100% rate with the transform applied, confirming the rewrite preserves program semantics across a broad range of C and C++ code.

6.2 Macro Benchmarks

Table 4 reports execution-time ratios relative to a plain `-O2` baseline, summary statistics across the LLVM test suite.

Variant	Mean	Median	Δ mean
Table lookup (dependent)	1.3293	0.3392	−2.4%
Packed (dependence removed)	1.3253	0.3266	−2.7%
Packed + `llvm.expect`	1.3632	0.3303	+0.1%

Table 4: Exec-time ratio vs. `-O2` baseline (1.3616 mean, 0.3439 median).

These differences sit within run-to-run noise for a general-purpose suite not designed to stress indirect calls specifically – which is itself informative: indirect calls are not frequent enough in typical code for this transform to register as a macro-level regression.

6.3 Microarchitectural Behavior

To isolate the indirect-call path, we ran a synthetic benchmark dispatching through a 9-entry function-pointer array in a tight loop (loop counter marked volatile to prevent constant folding) under gem5's O3CPU model with L1/L2 caches enabled [9].

Metric	Base	Table	Packed	+Hint
Ind. lookups	167552	167538	147935	147935
Ind. hits	166978	166979	147299	147299
Ind. mispredicts	0	0	0	0
Sim. seconds	0.002948	0.003156	0.002721	0.002721

Table 5: gem5 indirect-branch-predictor stats, 10^5 iterations.

The dependence-removed variant is the fastest of the four at this iteration count – about 8% faster in simulated time than the unprotected baseline – with identical misprediction behavior across all four (zero indirect mispredicts throughout; the guard branch itself never causes a misprediction once hinted, and even before hinting the predictor learns the near-always-taken pattern quickly enough not to show up here). At 10^6 iterations with `-O2` and default codegen, one specific instruction-alignment choice made by the optimizer cost 25% – comparing the generated assembly directly showed the two binaries differing only in NOP/XCHG padding preceding the hot loop, an artifact of that particular code layout rather than of the security mechanism. Recompiling the identical source with `-march=native` (letting the optimizer size instructions for the actual target rather than a generic one) removes the artifact:

Variant	Mean (s)	Variance
`-Os` + table	2.1380	1.4×10^{-5}
`-O2` (baseline)	1.8116	3.6×10^{-4}
`-O2` + table	2.1516	3.7×10^{-4}
`-O2` + packed	1.8598	2.2×10^{-4}

Table 6: `march=native` results: the packing optimization's contribution.

With target-tuned codegen, the packed-pointer variant costs 2.7% over baseline; the pre-packing table-lookup variant costs closer to 18%. The gap between those two rows is the packing optimization's measured contribution, isolated from both the underlying security mechanism and from unrelated codegen noise.

7 Security Discussion

Type-confusion vulnerabilities that culminate in a function-pointer overwrite – a recurring pattern in JIT-compiled JavaScript engines, and in native C/C++ more generally – are constrained by this scheme wherever the corrupted pointer feeds a statically analyzable call site: the corrupting write can still happen, but the subsequent call is checked against the resolved target set before it executes, regardless of which same-signature function the attacker aimed for. This is strictly tighter than type-tag matching for the reason shown in Sec-

tion 4 – same-signature-but-unreachable targets are excluded outright. The guarantee is specific to statically compiled code: a JIT's hot-path machine code is generated at runtime and is not covered by a static LLVM pass; extending this approach there would need the JIT itself to consult and update the resolved target sets, which we leave as an open problem.

8 Related Work

kCFI tags types at call and return sites and checks tag compatibility rather than materializing jump tables, reporting roughly 2% overhead on Linux kernel benchmarks – including return-address protection, which this work does not yet implement [10]. Its precision ceiling is the type-flow filtering described in Section 4: it cannot exclude a same-signature function that is provably unreachable from a specific call site, for the same reason no type-tag scheme can. Microsoft/Clang Control Flow Guard and IFCC take a comparably coarse type-based approach. None of the three narrow a call site's target set below "everything of a compatible type" via data flow – which is this work's central contribution over all three. Liu and Ji similarly combine type and data-flow information to improve indirect-call analysis [11], targeting analysis precision for downstream consumers such as devirtualization rather than a runtime CFI enforcement mechanism. The two lines of work are complementary rather than competing: nothing prevents a hybrid that uses call-site- precise tables internally and falls back to kCFI-style type tagging only at the unresolvable module boundaries of Section 3, which is strictly at least as precise as kCFI alone, everywhere.

9 Limitations and Future Work

Module scope. The analysis resolves call sites within one LLVM module; pointers crossing a module boundary fall back to type filtering (Section 3) unless LTO removes the boundary.

Aggregate precision. Struct and array fields holding function pointers are unioned rather than tracked per field (Section 4); per-field tracking is a precision improvement with an unresolved performance/complexity cost.

Backward-edge CFI. A return-address analogue of Section 3 replaces `push return-address; ...; ret` with `push index; ...; pop index; jump table[index]`, checked the same way as a forward-edge call. This removes the hardware return-address-stack's caching benefit for predicting returns, since the actual return address is no longer pushed by a `call` instruction; recovering it would need a dedicated `pushras`/`popras` instruction pair that pushes an index to memory while still updating the RAS, which is a hardware proposal, not something implementable purely in a compiler pass. This scheme is sketched but not implemented or evaluated here, since it requires operating after instruction selection (Section 3); it is the natural next extension of this work.

Concurrency and IPC. Function pointers passed across threads within a module are analyzed the same as any other value; pointers crossing processes are out of scope, since a function's address is meaningless in another process's address space under normal (non-shared- memory, non-kernel) execution, and C leaves such transfer undefined regardless.

10 Conclusion

Static data-flow resolution of indirect-call target sets, combined with a packed-pointer jump-table encoding and a branch-prediction hint, yields forward-edge CFI that is measurably more precise than type-tag matching – a 40× smaller modal target-set size across the call sites in our corpus – without a measurable performance cost – validated against the LLVM test suite for correctness and against a gem5 microarchitectural model for the indirect-call path specifically. A liveness pass and a greedy index-ordering heuristic keep the memory cost of per-call-site tables within a small constant factor of the best case achievable by the naive scheme, reducing it by roughly 8.5× on real-world C code drawn from FFmpeg, TinyCC, and OpenSSL.

References

[1] Elias Levy (Aleph One), "Smashing the stack for fun and profit," *Phrack*, 1996.

[2] Intel Corporation, "A technical look at intel's control-flow enforcement technology," 2020.

[3] ARM, "Armv8.5-a memory tagging extension," 2020.

[4] H. Shacham, "Return-oriented programming: Exploits without code injection," 2007.

[5] E. Bosman and H. Bos, "Framing signals—a return to portable shellcode," 2014.

[6] L. O. Andersen, *Program analysis and specialization for the c programming language*, 1994.

[7] Anonymous 4chan Poster, R. Houston, J. Pantone, and V. Vatter, *A lower bound on the length of the shortest superpattern*, 2018.

[8] G. Egan, *Superpermutations*, 2019.

[9] The gem5 Project, *Gem5 simulator*, 2025.

[10] The kCFI Project, *Drop the rop*, 2019.

[11] D. Liu and S. Ji, "Improving indirect-call analysis in llvm with type and data-flow co-analysis," 2024.