Applying System Call Filtering to Real-World Binaries (Experience Paper)

SOUMYAKANT PRIYADARSHAN, Bloomberg, USA
SEYEDHAMED GHAVAMNIA, Bloomberg, USA

System call filtering restricts applications to the system calls they require, but inferring accurate syscall sets for binary-only programs remains challenging. A central difficulty lies in recovering precise control-flow information from binaries: over-approximation leads to overly permissive syscall filters, while missed control-flow edges result in unsound policies. Although many techniques have been proposed to improve control-flow recovery in binaries, their practical impact on syscall inference remains poorly understood.

In this work, we conduct an empirical study of syscall inference from binaries using multiple off-the-shelf binary analysis tools. We evaluate how different control-flow refinement techniques affect inferred syscall sets in practice. Our experiments on real-world applications show that refinements targeting individual control-flow transfers (e.g., call-site and callee argument matching) substantially improve per-call target precision but often do not reduce the overall syscall set. In contrast, techniques that reduce the global set of address-taken functions—such as leveraging relocation information to accurately identify code pointers—yield the most significant syscall reductions. Finally, we identify a practical lower bound on syscall reduction achievable via sound static binary analysis alone, and show that further improvements are likely to require configuration or workload-aware specialization.

CCS Concepts: • **Security and privacy → Software reverse engineering**.

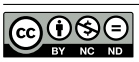
Additional Key Words and Phrases: Binary analysis, Static analysis, Binary instrumentation, System call filtering, Control-flow graph recovery, indirect call resolution

ACM Reference Format:
Soumyakant Priyadarshan and Seyedhamed Ghavamnia. 2026. Applying System Call Filtering to Real-World Binaries (Experience Paper). *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA122 (October 2026), 22 pages. https://doi.org/10.1145/3832213

1 Introduction

Applications rely on system calls to access operating system services such as file I/O, memory management, and networking. Modern operating systems like Linux expose a large system call interface, with over 300 distinct system calls across architectures. While most applications require only a small subset of these services, the default execution environment permits all system calls, substantially increasing the attack surface. Classic exploitation techniques such as ROP [9, 11, 13, 14, 44] and return-to-libc [47] leverage control-flow corruption to invoke unintended library functions, effectively enabling unauthorized system call invocations. System call filtering [15, 23, 25, 41] enforces a least-privilege policy by restricting applications to only the system calls they require. Mechanisms such as seccomp [1] enable efficient runtime enforcement of these policies, mitigating the impact of control-flow hijacking and post-exploitation activities. However, deploying syscall

Authors' Contact Information: Soumyakant Priyadarshan, Bloomberg, New York, USA, spriyadars15@bloomberg.net; Seyedhamed Ghavamnia, Bloomberg, New York, USA, sghavamnia@bloomberg.net.



ACM 2994-970X/2026/10-ARTISSTA122
https://doi.org/10.1145/3832213

filtering in practice remains challenging: filters must be precise enough to preserve functionality while avoiding unnecessary system call exposure.

Ability to infer system calls for closed-source software and legacy applications without vendor support can be beneficial. Production software is typically distributed and deployed in binary form. Operating directly on binaries removes dependence on vendor cooperation or proprietary build environments, enabling users to exercise greater control over the security of deployed software. Moreover, binary-level analysis can be more complete than source-based approaches. Real-world programs often include proprietary third-party components and handwritten assembly code that are invisible to source-level analyses but may introduce additional system calls. Analyzing the binary ensures that syscall inference reflects all code executed at runtime.

However, deriving syscall filters from binaries is challenging due to the loss of high-level program structure, including explicit function boundaries and type information. Despite extensive prior work on binary analysis [8, 16, 17, 35, 37, 40, 45, 50], recovering accurate control-flow graphs (CFGs) from binaries remains an open problem. Sysfilter [15] represents a state-of-the-art approach that infers syscall policies from binaries using context-sensitive control-flow refinement to reduce over-approximation in the inferred syscall set. While its techniques are effective in principle, its practical deployment faces limitations. Sysfilter is built on Egalito [50], which currently supports only position independent executables and imposes environmental constraints on deployment, thereby limiting its applicability in real-world settings.[1]

Other widely used binary analysis tools [8, 26, 35, 37, 40, 42, 49, 50, 54] attempt to recover control-flow information using static analysis, but their precision varies substantially in practice. While prior studies [5, 36] have evaluated control-flow recovery in isolation, the practical implications of these differences for downstream security mechanisms–particularly syscall filtering remain poorly understood. Motivated by this gap, we present an empirical study of syscall inference from binaries using off-the-shelf binary analysis tools. Our goal is to understand how existing tools behave in practice, what limits their precision and soundness, and which refinements meaningfully improve syscall filtering in real-world settings. We structure our study around the following research questions:

- **RQ1 (Feasibility across real-world binaries):** To what extent can off-the-shelf binary analysis tools handle real-world programs in practice—including stripped and unstripped binaries and different compiler optimization levels when inferring system call filters?
- **RQ2 (Precision and soundness):** How precise and sound are the syscall filters inferred by different tools, and how do they compare in terms of syscall over-approximation and correctness?
- **RQ3 (Impact of control-flow refinement):** To what extent do control-flow and indirect-call refinement techniques reduce inferred syscall sets, and which refinements provide the most practical benefit?
- **RQ4 (Limits of static syscall reduction):** What limits further syscall reduction after aggressive control-flow refinement using static binary analysis?

1.1 Contributions

This paper presents an empirical study of system call inference from binaries using contemporary binary analysis tools. Specifically, we make the following contributions:

- **Comprehensive evaluation of binary analysis tools.** We empirically evaluate five widely used binary analysis platforms: IDA-Pro [26], Ghidra [42], BinaryNinja [49], Angr [45], and

[1]Egalito is tightly coupled to a specific system environment and depends on `glibc` 2.27. In our experiments, we were unable to build or run Egalito on Ubuntu 22.04, which ships with `glibc` 2.35.

SAFER [37]—on real-world applications, analyzing robustness, precision, and soundness across different compiler optimization levels and stripped/unstripped binaries. We also reimplement and evaluate refinement techniques from prior work, including relocation-based code-pointer identification [17, 50], call argument matching [48], and Sysfilter's context-sensitive control-flow graph refinement [15].
We also introduce two new refinements—*data-section splitting* and *escape analysis*—to further reduce syscall over-approximation.

- **Practical limits of static syscall inference.** By analyzing the provenance of residual syscalls after aggressive refinement, we identify a practical lower bound on syscall reduction achievable via sound static binary analysis. Our results indicate that further reductions require configuration- or workload-aware techniques (e.g., profile-guided analysis or deployment-specific specialization), which trade generality for tighter syscall filters.
- **A modular framework for syscall inference from binaries.** We design and implement a backend-agnostic framework for syscall inference that supports multiple binary analysis tools and provides a configurable interface for enabling or disabling individual refinement techniques. This framework enables easy integration and evaluation of future analysis tools.

2 Background

Conceptually, syscall filtering is analogous to control-flow integrity (CFI) [3, 53, 55]: both techniques require statically approximating a program's permissible behavior and enforcing that approximation at runtime. While CFI enforces valid control-flow transfers, syscall filtering enforces valid kernel interactions. In the binary-only setting, syscall filter must be derived directly from executables and shared libraries. This requires recovering control-flow information from binaries and identifying all code paths that may reach system call instructions, either directly or through library wrappers. Due to the absence of type information and high-level program semantics, accurate control-flow graph recovery is challenging [5, 36].

2.1 Binary Analysis and Control-Flow Recovery

Binary analysis aims to recover program structure from compiled executables, including functions, control-flow graphs (CFGs), and call graphs. A central challenge in this process is handling *indirect control-flow transfers*. Source-level constructs such as virtual functions, switch-case blocks and code involving function pointers are compiled into indirect calls/jumps. Unlike direct calls/jumps whose targets are embedded in the instruction and can be retrieved easily, retrieving the targets of indirect calls is hard.

To conservatively model indirect control flow, binary analysis tools identify *address-taken functions*—functions whose addresses may be stored in registers or memory and later invoked indirectly. Unresolved indirect calls are typically linked to all address-taken functions, which can significantly over-approximate the CFG. As a result, inaccuracies in CFG recovery propagate directly to downstream analyses such as syscall inference, often leading to overly permissive syscall filters.

The goal of this paper is to empirically evaluate how limitations in binary control-flow recovery translate to imprecision in syscall inference. By studying existing tools and refinement techniques on real-world binaries, we aim to identify which refinements meaningfully reduce over-approximation and where static analysis reaches its practical limits for further syscall reduction.

2.2 Assumptions

Our experiments target x86-64 Linux binaries. We do not assume the availability of symbol or debug information in program binaries. Our analysis is global in scope: all components of a program, including statically and dynamically linked third-party libraries, are analyzed when available at

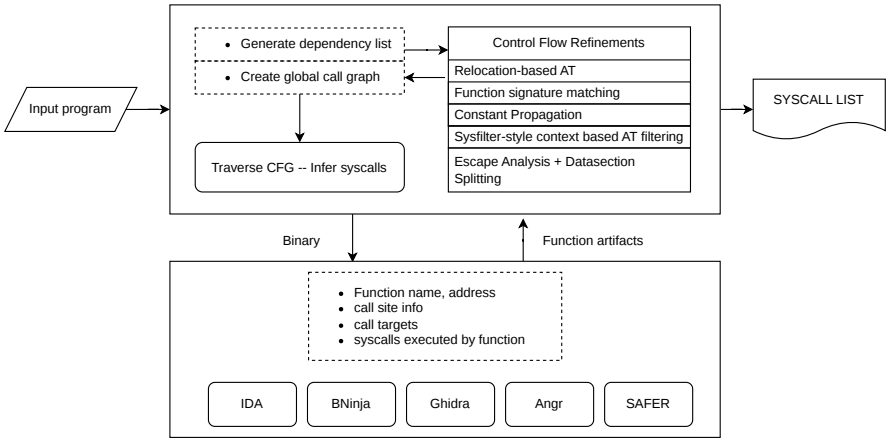


Fig. 1. Architecture of the syscall inference framework. The design decouples back-end-specific binary analysis from global call graph reasoning and syscall set generation.

analysis time. Although we build the target applications in our evaluation set, we do not rebuild their dependent libraries; instead, we analyze the default system libraries present on the evaluation host. Libraries loaded dynamically at runtime or programs executed via `execve` are considered outside the static analysis scope unless explicitly included.

3 Framework Implementation

3.1 Design Goals

Our framework is designed to support a systematic and empirical evaluation of system call inference from binaries using existing analysis tools. To this end, it is guided by two primary design goals.

First, the framework is *back-end-agnostic*. It decouples syscall inference logic from the underlying binary analysis engine, allowing multiple off-the-shelf binary analysis tools to be used interchangeably. This design enables direct comparison of tools that employ different analysis strategies and assumptions. Second, the framework supports *experimentation with different precision-improving techniques*. It provides well-defined interfaces for incorporating and selectively enabling analysis refinements, enabling controlled ablation studies to assess their impact on control-flow precision and inferred syscall sets.

3.2 Architecture

Figure 1 illustrates the high-level architecture of the framework. The framework is organized into two layers: a back-end-specific analysis layer and a back-end-independent global reasoning layer.

The back-end-specific layer interfaces with off-the-shelf binary analysis tools and is responsible for extracting function-level analysis artifacts. In our study, we use IDA Pro [26], Ghidra [42], Binary Ninja [49], angr [45], and SAFER [37] as back-end analysis engines. Each back-end performs disassembly and function-local analysis according to its own policies, while exporting a uniform set of function-level artifacts to the framework.

The global reasoning layer consumes the function-level information produced by the back-end and performs back-end-independent analysis. It constructs a global call graph, links unresolved indirect calls with potential targets, and traverses the call graph starting from the program entry point to determine reachability. When a reachable function contains a system call, the corresponding

syscall number is added to the inferred syscall set. This separation of function-local extraction and global reasoning allows syscall inference logic and control-flow refinements to be applied uniformly across different back-ends.

In addition to the back-end analysis tools, our framework supports a collection of control-flow refinement techniques that operate in the global reasoning layer. These include relocation-based address-taken (AT) identification, Sysfilter's context-sensitive AT refinement [15], function signature matching, global constant propagation, data-section splitting, and escape analysis. All of these refinements are implemented on top of SAFER to provide a consistent and controlled evaluation environment. Since the original Sysfilter is built on Egalito [50], which is tightly coupled to Ubuntu 18.04 and does not operate reliably outside this environment, we reimplemented only its context-sensitive AT refinement within our framework. This design allows us to evaluate Sysfilter's core idea alongside other refinement techniques while remaining independent of the original implementation.

4 Recovering Function-Level Artifacts From Binary

The first step in inferring system calls is to recover functions and their control flow. Specifically, the calling relationships between functions—and to identify `syscall` instructions within each function along with their corresponding system call numbers. In this section, we examine how each binary analysis tool evaluated in our study implements these tasks, and we discuss the sources of imprecision inherent in each tool.

4.1 Disassembly and Function Recovery

Disassembly and function recovery form the foundation of syscall inference from binaries. Errors at this stage—such as missed functions or spurious function boundaries—propagate to downstream analyses and can lead to unsound syscall filters or overly conservative syscall sets.

IDA-Pro and Ghidra primarily rely on recursive descent disassembly [36], starting from known function entry points and following control-flow edges to discover reachable code. These entry points include the program entry, addresses listed in export tables (e.g., entries in the `.dynsym` section of ELF binaries), and program initialization and cleanup routines (e.g., functions in the `.init` and `.fini` sections). Direct call targets encountered during traversal are also treated as candidate function entry points. Recursive descent may leave gaps when control-flow edges cannot be resolved; to mitigate this, both tools leverage exception handling metadata from the `.eh_frame` section to recover additional function boundaries [36]. Prior work has shown that exception handling metadata significantly improves function recovery accuracy [38].

SAFER adopts a static and statistical disassembly approach based on DASSA [40], which has been shown to achieve substantially higher accuracy—up to three times better than several contemporary disassemblers such as angr, Ghidra, and Ddisasm [19]. Recent versions of SAFER also support the use of exception handling metadata via the `.eh_frame` section, which we enable in our framework to further improve function recovery.

BinaryNinja and Angr follow a similar recursive descent strategy but complement it with linear scanning of remaining gaps to increase code coverage. Instructions discovered through linear scanning are subsequently grouped into functions based on direct call targets and local control-flow structure [36]. While linear scanning improves coverage, it may result in false negatives in function identification when functions are reachable only through indirect calls and pointer analysis is imprecise, potentially leading to incomplete syscall inference.

4.2 Control Flow Recovery

After disassembly and function identification, the next step is to recover control flow within each function and record inter-function call targets. This information is used to determine reachability to functions that may invoke system calls. Direct control-flow transfers, such as direct calls and jumps, can be resolved syntactically from instruction operands. In addition, we explicitly recover inter-module call targets mediated through the Procedure Linkage Table (PLT). For dynamically linked binaries, we resolve PLT entries by analyzing the Global Offset Table (GOT) and the corresponding relocation entries, enabling accurate identification of external function call targets across module boundaries.

In contrast, indirect calls and jumps whose targets are computed at runtime are substantially harder to resolve statically and require additional analysis, which we discuss next.

4.2.1 Handling Indirect Control Flow Transfers. Modern binary analysis tools provide limited support for resolving indirect calls and jumps using intra-function analysis. As observed by Pang *et al.* [36], tools such as Ghidra, Angr, and BinaryNinja employ intra-procedural techniques—including value-set analysis and constant propagation—to resolve certain classes of indirect control flow, most notably jump tables generated by switch-case constructs. In addition to jump tables, IDA Pro and Binary Ninja can identify trivial cases of function pointer table based dispatch where the table is loaded and dereferenced within the same function. SAFER primarily focuses on jump-table recovery and leverages the SJA framework [35], which also detects similar trivial function-pointer dispatch patterns. Indirect calls arising from virtual dispatch, callbacks, and other cases where pointers flow across multiple functions or memory objects remain particularly challenging for function-local analysis. A common conservative fallback is to associate unresolved indirect transfers with all address-taken functions, which substantially over-approximates the call graph and can inflate inferred syscall sets.

4.2.2 Identifying Address-Taken (AT) Functions. Identifying functions whose addresses are stored in registers or memory is a key step in controlling over-approximation during syscall inference. In our framework, unresolved indirect calls are conservatively linked to address-taken (AT) functions; consequently, the size of the AT set directly determines the extent of call-graph expansion and the resulting syscall set.

IDA Pro, Ghidra, and Binary Ninja employ simple heuristics to identify address-taken functions, typically by scanning for immediate operands or constants in instructions and data sections that coincide with known function entry addresses [36]. SAFER relies on a similar strategy and additionally uses DASSA's [40] analyses to validate that candidate pointers refer to valid function entry points, thereby reducing certain classes of false positives. These approaches involve a fundamental trade-off. They may introduce false positives, as not all constants resembling function addresses are used as function pointers, and false negatives, as function pointers derived through arithmetic or constructed dynamically remain invisible to direct scans. A fully sound alternative of treating all functions as address-taken would eliminate false negatives but is impractical due to the resulting explosion in call-graph size. Consequently, existing tools implicitly assume that function addresses are referenced directly and restrict their analysis to these simple cases.

Angr applies additional heuristics to reduce false positives by filtering constants that are likely floating-point values or strings. While this improves precision, it can further increase false negatives by discarding legitimate function pointers that do not match expected patterns. This limitation is reflected in our evaluation, where a portion of the system calls missed by angr can be attributed to false negatives in address-taken function identification.

We also observed subtle but impactful differences in address-taken function policies across tools. IDA-Pro and Ghidra conservatively treat entries in the dynamic symbol table as address-taken functions, whereas Safer, Angr, and BinaryNinja do not. This policy choice leads to noticeably larger AT sets—and corresponding syscall over-approximation for IDA-Pro and Ghidra. Safer treats certain disassembly gap start addresses present in the `.eh_frame` section as address-taken functions, introducing a distinct source of over-approximation.

4.2.3 Non-Returning Calls. Correctly identifying non-returning calls (e.g., calls to functions such as `exit` that do not return) is critical for accurate function boundary identification and control-flow graph recovery. Recursive disassembly typically assumes that control returns to the fall-through instruction after a call; this assumption does not hold for non-returning functions. Failure to identify such calls lead to spurious fall-through edges and over-approximation in the recovered CFG, which can in turn inflate the inferred syscall set. Prior work by Pang et al. [36] shows that tools such as IDA Pro and Ghidra struggle to accurately identify non-returning functions, a limitation that is also reflected in our results, where a portion of the over-approximated syscalls can be attributed to misidentified non-returning calls.

4.3 Syscall Identification

The final step is to identify syscall instructions in each disassembled function and resolve the corresponding system call numbers. On x86-64 systems, the syscall number is typically placed in the `RAX` register prior to execution, and recovering its value requires data-flow analysis such as constant propagation.

For both Ghidra and Angr, we leverage built-in reaching-definition analysis to identify the instruction(s) that define the syscall-number register prior to each syscall instruction. This analysis yields candidate definition sites but does not directly provide the concrete syscall number, necessitating an additional value-recovery step. In angr, we recover register values by executing a single instruction from each reaching definition site using a blank symbolic state and reading back the resulting register value. If the value becomes concrete, the syscall number is extracted via the solver interface; otherwise, dependencies on other registers are recursively tracked until a concrete value is recovered or resolution fails. In contrast, the Ghidra back-end explicitly decodes defining instructions by recursively interpreting Pcode operations, including register assignments and arithmetic and bitwise operations. If the value depends on unsupported operations or memory loads, the syscall number remains unresolved.

IDA Pro and Binary Ninja rely on built-in register-value analysis that performs local constant propagation and constant folding. When the syscall-number register is resolved to a concrete constant, the value is obtained directly via the analysis API; values derived from memory loads or otherwise unresolved locally are treated as unknown. Note that we did not encounter any case in our evaluation where the call graph reaches an unresolved syscall.

In Safer, syscall numbers are recovered using its abstract interpretation framework based on the BaseLH abstract domain, which represents values as intervals of the form `(base, [low, high])`. Safer performs a forward data-flow analysis from the function entry to each syscall instruction and computes the abstract value of the syscall-number register at that point. When the interval collapses to a single value (i.e., `low == high`), the syscall number is resolved precisely; otherwise, the analysis conservatively enumerates all syscall numbers within the interval `[low, high]`.

In addition to direct syscall instructions, we handle *syscall wrapper functions*, which invoke syscalls through regular function calls and pass the syscall number as an argument rather than loading it directly into the syscall-number register. We identify calls to known wrapper functions and recover the syscall number from the corresponding argument register at the call site. Programs may

also define local syscall wrappers. We identify these by detecting functions containing unresolved syscall instructions whose syscall-number register depends on an argument register and that are invoked from multiple call sites.

5 Global Call Graph and Syscall Set Generation

The global reasoning plane consumes function-level artifacts produced by the back-end analysis to construct a whole-program view of control flow. Its primary goal is to determine which system calls are reachable from program entry points while avoiding unnecessary over-approximation.

For each function, the back-end exports a uniform set of artifacts, including the function entry address and name, all call sites along with their resolved targets, and any unresolved call or jump sites. For inter-module calls mediated via the PLT, the back-end additionally provides the referenced function name and, when available, the corresponding library name obtained from relocation metadata. If a function contains syscall instructions or syscall-wrapper invocations, the back-end also exports the set of resolved syscall numbers associated with those sites.

Using these artifacts, we implement a single on-the-fly algorithm that incrementally constructs the global call graph while simultaneously collecting reachable system calls. The algorithm proceeds as follows:

(1) Resolve program dependencies (dynamically linked libraries) and perform function-local analysis for every module (main binary and each dependency). Each back-end exports function-level artifacts (function boundaries, call sites, resolved call targets, unresolved indirect sites, address-taken function set, PLT entries, and per-site syscall information).
(2) Initialize a work queue with the program entry points (the `main` function and initialization/-cleanup routines).
(3) While the work queue is non-empty, pop the next function and process its outgoing control-flow:
    - For direct calls to known callees, enqueue the callee if not already visited.
    - For PLT-mediated inter-module calls, resolve the target by matching the exported function name (and library name when available) against the program's dependency modules, and enqueue the resolved function if found.
    - For unresolved indirect calls/jumps, conservatively link the call to the set of address-taken functions exported by the back-end and enqueue any newly reachable AT functions.
    - If the current function contains a syscall (or a resolved syscall-wrapper invocation), add the syscall number(s) determined by the function-local analysis to the inferred syscall set. If a syscall number is unresolved, treat it conservatively per the policy (e.g., mark as "unknown" or include the full syscall universe). We reiterate that we did not encounter any case in our evaluation where the call graph reaches an unresolved syscall.
(4) Continue until the queue is exhausted; the union of recorded syscall numbers is the inferred (conservative) syscall set.

6 Precision Improvements

In this section, we explore techniques for refining the targets of indirect calls and jumps. As discussed in Section 4, contemporary binary analysis tools primarily resolve intra-procedural indirect control-flow transfers, such as jump tables, leaving most indirect calls involving virtual dispatch or pointer propagation unresolved.

Prior work [15, 21, 28, 48, 50, 53] has proposed a variety of techniques to improve indirect-call precision. Some techniques focus on pruning infeasible targets at individual indirect call sites [21, 28, 48], thereby reducing the average number of targets per call. Other techniques aim

to reduce the global set of address-taken functions [15, 50], which indirectly constrains the target sets of unresolved calls. Although both approaches improve control-flow precision, their impact on syscall inference differs significantly.

For syscall inference, improvements limited to individual indirect calls are often insufficient: as long as any indirect call remains unresolved, it must be conservatively linked to the full address-taken set, resulting in no reduction in the inferred syscall set. In contrast, techniques that reduce the overall address-taken set can directly shrink this conservative approximation and therefore have a meaningful impact on syscall filtering.

In the remainder of this section, we discuss representative techniques along both dimensions and evaluate their effectiveness in practice. We additionally present new approaches aimed at reducing the address-taken set, including heuristics for data-section splitting and a static pointer escape analysis.

6.1 Function Signature Matching

Function signature matching aims to refine indirect call targets by exploiting compatibility between call sites and potential callees. TypeArmor [48] is the first work to explore this idea in the context of enforcing control-flow integrity (CFI) [3, 53, 55] on binary executables. The core intuition is that, even when an indirect call cannot be resolved precisely, many candidate targets can be ruled out if their function signatures are incompatible with how arguments are prepared at the call site.

In binaries, explicit type information is unavailable, making precise reconstruction of function prototypes difficult. TypeArmor [48] therefore adopts a conservative, count-based many-to-many matching strategy based on argument counts and return-value usage. At each indirect call site, it computes an upper bound *max* on the number of arguments prepared, while for each potential callee it computes a lower bound *min* on the number of arguments consumed. A call site cs_{max} may target a callee f_{min} if $min \leq max$, reflecting the fact that binaries may legally pass more arguments than a callee uses. TypeArmor additionally enforces return compatibility by disallowing call sites that expect a return value from targeting void functions. We follow the same conservative argument-count and return-usage matching policy in our implementation.

We further extend signature matching with coarse-grained argument type inference. Each argument is classified as one of four abstract types: *pointer*, *integer* or *unknown*. Pointers are identified via memory dereferences or offsets from `RSP` or `RIP`, integers are constants outside the binary's address range, and *both* denotes constants within the address range. Arguments whose types cannot be inferred are marked *unknown*. During matching, *unknown* is allowed to match any type. *Integer* and *pointer* are treated as incompatible.

Overall, function signature matching refines targets at individual indirect call sites by reducing the average number of feasible callees per call. However, its impact on syscall inference is limited. Due to inherent imprecision in binary analysis, there typically exists at least one indirect call site with a maximally permissive signature—i.e., the highest inferred argument count and all arguments classified as *unknown*. Such a call site must be conservatively linked to the entire address-taken function set during global call-graph construction, resulting in no reduction in the inferred syscall set.

6.2 Global Constant Propagation

Most binary analysis tools, including IDA-Pro, Ghidra, BinaryNinja, and Angr, perform intra-function (local) constant propagation by default to resolve certain indirect control-flow targets. This is effective for simple cases where constant values are assigned and consumed within the same function. Recent work such as SysPart [41] also leverages local constant propagation as part of its analysis.

Our goal is to address cases where constant values—particularly function pointers—are passed across function boundaries. A common example is library routines such as `qsort`, which accept a comparison function as an argument and invoke it indirectly within the callee. In such cases, the indirect call target cannot be resolved using function-local analysis alone, as the constant value originates at the call site of the library function rather than within the callee.

To enable inter-procedural propagation of such constants, we augment our function-local analysis to export additional information. Specifically, for each call site, we record argument value information, including both the coarse-grained type information described in Section 6.1 and any concrete constant values when available. In addition, for each indirect call or jump, we record whether the target is read directly from a register and, if so, which argument register it depends on. Using this information, we extend the global control flow graph generation process to perform a form of global constant propagation across the call graph, allowing constants passed as arguments to be propagated to indirect call sites in callees.

Note that we do not apply similar inter-procedural propagation to return values. In practice, return values are less frequently used as indirect call targets, and propagating them soundly across multiple return paths and call sites would introduce significant complexity while offering limited benefit for syscall inference.

Despite these extensions, global constant propagation remains inherently limited. Many indirect calls and jumps do not directly use constant values or argument registers, or their values are overwritten, merged across control-flow paths, or loaded from memory. Such cases cannot be resolved by constant propagation and must still be conservatively linked to the address-taken function set. As a result, while global constant propagation can resolve a small number of additional indirect calls, its overall impact on reducing the inferred syscall set is limited, as we show in our evaluation.

6.3 Relocation-Based Address-Taken Identification

A major source of over-approximation in identifying address-taken (AT) functions is the difficulty of distinguishing genuine code pointers from integer constants that merely resemble function addresses. Simple heuristics based on scanning instructions and data sections for address-like values often misclassify integers as function pointers, unnecessarily increasing the global AT set.

Many binary rewriting systems such as CCFIR [53], SBR [39], Retrowrite [17], Egalito [50], and Sysfilter [15] instead leverage relocation metadata to accurately identify code pointers. Position-independent executables (PIEs) cannot contain absolute code or data addresses because the operating system may load the executable at an arbitrary base address as part of address space layout randomization (ASLR). Consequently, the linker emits relocation entries identifying locations that require address fixups during program loading. These entries precisely identify values that are intended to represent code or data pointers, allowing them to be distinguished from ordinary integer constants.

PIE has become the default compilation mode in modern compiler toolchains (e.g., GCC and Clang) on most Linux distributions to support ASLR-based defenses. As a result, relocation metadata is widely available in contemporary binaries, making relocation-based code-pointer identification practical for real-world deployments.

In our framework, this relocation-based AT identification serves as the baseline refinement upon which all subsequent refinements are applied.

6.4 Sysfilter-Style Context-Sensitive Address-Taken Set Refinement

Sysfilter [15] is a state-of-the-art static approach for inferring syscall filters from binaries. Beyond relocation-based filtering, Sysfilter applies an iterative, context-sensitive process to further refine

Algorithm 1: Context-based pointer refinement (Sysfilter-style)

Input : Entry points; function artifacts; optional symbol/object metadata
Output : Discovered pointers Ptrs; reachable functions Reach

```
1  Ptrs ← ∅
2  Reach ← {entry points}
3  repeat
4      NewPtrs ← ∅
5      NewReach ← ∅
6      foreach function f in Reach do
7          NewPtrs ← NewPtrs ∪ CodeOriginPtrs(f)
8          foreach data object reference o in f do
9              if metadata for o available then
10                 NewPtrs ← NewPtrs ∪ CodePtrs(o)
11             else
12                 NewPtrs ← NewPtrs ∪ AllPtrsInDataSection
13         Resolved ← ResolveIndirectCalls(f, Ptrs ∪ NewPtrs)
14         NewReach ← NewReach ∪ Resolved
15     Ptrs ← Ptrs ∪ NewPtrs
16     Reach ← Reach ∪ NewReach
17 until no change in Ptrs and Reach
18 return Ptrs, Reach
```

indirect call targets. We implement and evaluate an equivalent method, summarized in Algorithm 1. The algorithm maintains two evolving sets: the set of reachable functions (`Reach`) and the set of discovered function pointers (`Ptrs`). Starting from program entry points, it iteratively traverses functions reachable via direct control-flow edges. During each iteration, pointers originating directly in reachable functions (e.g., immediate operands or instruction-pointer–relative constants) are added via `CodeOriginPtrs` (line 7). When a reachable function references a data object, the algorithm handles pointers stored in data conservatively (lines 9–13): if symbol or object metadata is available, only code pointers embedded in that object are added; otherwise, function pointers present in the entire data section are included upfront. The accumulated pointer set is then used to resolve unresolved indirect calls in reachable functions, potentially expanding the set of reachable functions (lines 15–16). This process repeats until a fixpoint is reached, ensuring that only pointers reachable under valid execution contexts contribute to the final address-taken set.

6.5 Data Section Splitting

Data section splitting aims to reduce over-approximation introduced by treating all data-derived pointers uniformly. The goal is to approximate the benefits of Sysfilter's symbol information based analysis even when full symbol information is unavailable, as is common in stripped binaries.

We begin by identifying a set of *known data objects* through static analysis. These include jump tables, statically identifiable function pointer tables, and objects referenced via symbols in dynamic symbol tables or export tables. In addition, we apply layout-based heuristics to identify likely virtual tables in data sections by detecting regions that exhibit vtable-like structure—namely, a

small constant offset field followed by a pointer to type metadata and one or more consecutive code pointers aligned at word boundaries.

These known objects form an initial partitioning of the data section. The remaining gaps between identified objects are conservatively treated as separate data objects. This produces a complete partitioning of the data section into disjoint candidate objects.

Given a pointer into the data section, we determine its corresponding points-to set by first identifying the object(s) it may refer to. For the candidate data object, we scan its contents to identify embedded pointers. If a code pointer is found, it is added to the points-to set. If a data pointer is found, we recursively resolve the corresponding data object and include its points-to set. This recursive resolution continues until a fixpoint is reached.

6.6 Escape Analysis

While data section splitting improves the precision of points-to sets, not all data or code pointers are relevant to indirect control flow. Escape analysis aims to determine whether a pointer can safely be ignored for the purposes of indirect call resolution. Similar analysis was first introduced by Priyadarshan et al. [37] where escape analysis was used in the context of identifying and recreating jump tables in binaries.

Given a pointer created or taken within a function, we perform an intra-procedural taint analysis to track its flows. A pointer is considered *escaping* and therefore relevant if it flows into an indirect call or jump target. In addition, a pointer is conservatively treated as escaping if it flows to a function argument, a return value, global memory, or heap or stack storage, as such flows may later influence indirect control flow outside the current function.

If none of these escape conditions are met, the pointer is considered non-escaping and is ignored during indirect call resolution. This filtering step eliminates a large number of spurious data and code pointers that are local to a function and do not affect control-flow behavior.

In our framework, escape analysis is applied in conjunction with data section splitting: only pointers that escape are subjected to object-based resolution, while non-escaping pointers are discarded. Together, these techniques significantly reduce the global address-taken set without sacrificing soundness.

7 Evaluation

7.1 Experimental Setup

Evaluated Binaries. We evaluate our framework on a set of widely used, real-world applications: *Apache HTTP server (Apache-httpd)*, *Bind*, *BlazingMQ*, *Comdb2*, *lighttpd*, *Memcached*, *Mysql*, *Nginx*, *Proftpd*, *Postgresql*, and *Redis*. These applications span diverse domains such as web serving, database systems, and network services. Each program is compiled using GCC at multiple optimization levels (O0, O1, and O2) and evaluated in both stripped and unstripped configurations. We note that *MySQL* could only be successfully built and evaluated at optimization level O2.

Host Machine. All experiments are conducted on an x86-64 machine running Ubuntu 22.04.4 LTS with a stock Linux kernel. The system is equipped with an AMD EPYC 7551P processor featuring 32 cores and 64 hardware threads, and 1 TB of RAM. We use the system-provided `glibc` and GCC 11.4.0 toolchain for building and evaluating all binaries. All analyses and test executions are performed on this dedicated host to ensure consistent experimental conditions.

7.2 Baseline Evaluation of Binary Analysis Frameworks

7.2.1 Evaluating RQ1 (Feasibility). The goal of RQ1 is to evaluate whether contemporary binary analysis tools can reliably analyze real-world binaries compiled under different optimization

Table 1. Summary of backend robustness across optimization levels and stripped binaries. "Optimization effect" and "Stripping effect" list the applications whose inferred syscall counts differ across configurations.

Tool	Failed	Optimization Effect	Stripping Effect
IDA Pro	None	None	None
Ghidra	None	None	None
BNinja	None	None	None
SAFER	None	None	None
Angr	None	Bind, Comdb2, Lighttpd Nginx, Proftpd, Redis	Lighttpd (O2), Nginx (O2) Proftpd (O2)

Table 2. Inferred syscall count and soundness for O2 stripped binaries. Since inferred syscall counts remain unchanged across optimization levels and stripping configurations for all evaluated tools except Angr, which exhibits only minor variations, we report O2 stripped binaries as a representative configuration. GT denotes the dynamically observed syscall count obtained using `strace` and `perf`. Inf., Miss, and Pass (%) denote the inferred syscall count, observed missed syscalls, and test pass rate, respectively. A * indicates misses caused by dynamically executed programs outside the static analysis scope.

		IDA			Ghidra			BNinja			SAFER			Angr		
Program	GT	Inf.	Miss	Pass	Inf.	Miss	Pass	Inf.	Miss	Pass	Inf.	Miss	Pass	Inf.	Miss	Pass
Apache-httpd	85	283	0	100	282	0	100	165	1	100*	212	0	100	122	4	4.4
Bind	80	286	0	100	286	0	100	180	0	100	228	0	100	138	0	100
BlazingMQ	76	280	0	100	279	0	100	150	1	100*	204	0	100	98	7	16
Comdb2	93	280	0	100	279	0	100	163	1	100*	209	0	100	111	8	1.9
Lighttpd	71	280	0	100	279	0	100	150	1	100*	202	1	100*	111	4	83.3
Memcached	71	280	0	100	279	0	100	152	0	100	204	0	100	107	4	29.5
MySQL	109	294	0	100	293	0	100	188	1	100*	230	1	100*	132	10	0.0
Nginx	69	281	0	100	280	0	100	150	0	100	203	0	100	105	5	0.0
Postgresql	84	290	0	100	289	0	100	189	1	100*	227	0	100	141	4	0.0
Proftpd	91	283	0	100	282	0	100	168	0	100	216	0	100	129	3	21.5
Redis	82	285	0	100	284	0	100	176	0	100	222	0	100	131	2	0.2

levels and in both stripped and unstripped configurations. Table 1 summarizes the robustness of the evaluated backends. We observe that all tools successfully analyze every benchmark in our evaluation. Furthermore, inferred syscall sets remain identical across optimization levels and stripping configurations for all tools except ANGR. Even for ANGR, the observed differences are minor: only a handful of applications exhibit variations, and in every case the inferred syscall count differs by at most one system call.

These observations indicate that compiler optimization has little impact on syscall inference. While optimization may influence the recovery of individual indirect control-flow transfers (e.g., jump tables), the overall inferred syscall set remains largely unaffected. This is because even a single unresolved indirect call must be conservatively linked to the global set of address-taken functions, resulting in similar syscall over-approximation irrespective of improvements in local control-flow recovery.

Table 3. Coverage achieved by the benchmark test suites.

Program	Tests	Line Cov. (%)	Func. Cov. (%)
Apache-httpd	830	56.7	61.6
Bind	362	79.3	86.6
BlazingMQ	1343	74.2	66.3
Comdb2	1866	54.1	62.6
Lighttpd	6	70.4	70.4
Memcached	112	76.9	88.5
Mysql	8439	61.9	61.0
Nginx	428	71.6	83.4
Postgresql	228	72.5	74.9
Proftpd	1813	25.7	33.9
Redis	3727	66.2	68.8
Average	–	**64.5**	**68.9**

Similarly, we observe negligible impact from stripping binaries. Although symbol and debug information can improve function recovery and address-taken function identification, this provides little benefit in our evaluation because the third-party libraries used by the benchmark applications are the default system libraries, which are themselves stripped. Since these libraries are responsible for the majority of system call invocations, the over-approximated control-flow graphs within them dominate the final inferred syscall set. Consequently, keeping only the main binary unstripped does not significantly change the inferred syscall counts. The small differences observed for Angr arise from its incomplete function recovery in stripped binaries, where symbol information enables the recovery of a few additional functions. As shown in the next section, however, these minor differences do not alter the broader observation that Angr's inferred syscall sets remain incomplete and result in test-suite failures.

7.2.2 Evaluating RQ2 (Precision & Soundness). The goal of RQ2 is to evaluate both the *precision* and *soundness* of the syscall sets inferred by different binary analysis tools. We evaluate soundness by enforcing the inferred syscall filter at runtime using seccomp-BPF [1]. Specifically, we instrument each target binary's `main` function using Safer's instrumentation API so that the program installs the seccomp filter immediately before entering `main`. The program is then executed using its corresponding test suite, and we report the percentage of test cases that complete successfully.

Additionally, we identify system calls missed by each binary analysis tool. First, we approximate the set of system calls required by each application by executing the test suites under `strace` and `perf` and treating the union of all observed system calls as the *ground truth*. Table 3 summarizes the coverage achieved by the benchmark test suites. On average, the test suites achieve 64.5% line coverage and ≈ 69% function coverage. Although this represents substantial program coverage, the observed syscall set is inherently incomplete, since some valid execution paths may not be exercised. Consequently, the reported missed syscalls correspond to definite false negatives with respect to the observed executions, whereas additional inferred syscalls should be interpreted as conservative over-approximation rather than definitive false positives.

Table 2 summarizes the inferred syscall count and soundness of syscall inference. Among the evaluated tools, **Binary Ninja** produces the least over-approximated syscall sets while remaining largely sound. In contrast, **IDA Pro** and **Ghidra** yield nearly identical results and infer substantially larger syscall sets, often permitting almost the entire Linux syscall interface. Through manual

Table 4. Incremental impact of precision-improvement techniques on inferred syscall counts and average indirect control-flow targets (AICT). Each column cumulatively adds the indicated technique(s).

	+Relocation		+SF-AT		+Split+Escape		+ArgCnt+Ret		+ArgType		+ConstProp	
Program	**Sys**	**AICT**	**Sys**	**AICT**	**Sys**	**AICT**	**Sys**	**AICT**	**Sys**	**AICT**	**Sys**	**AICT**
Apache-httpd	137	2197	108	2016	105	939	105	839	105	798	105	790
Bind	152	9798	128	9370	124	6624	124	5554	124	5149	124	4980
BlazingMQ	122	12580	109	12401	105	9917	105	9269	105	8762	105	8749
Comdb2	134	8168	132	7791	128	5884	128	5428	128	4938	128	4843
Lighttpd	120	1074	113	795	111	523	111	485	111	421	111	402
Memcached	118	1013	112	816	107	505	107	472	107	419	107	394
Mysql	164	54121	156	36846	152	34397	152	31000	152	29600	152	29401
Nginx	124	1567	111	1424	108	1128	108	1043	108	957	108	938
Postgresql	154	16668	148	15364	137	14521	137	13012	137	12111	137	10658
Proftpd	141	2381	126	1242	123	966	123	774	123	744	123	707
Redis	135	4612	126	3721	104	2755	104	2339	104	2201	104	2552

inspection of representative cases, we identify two primary sources of this over-approximation: (i) conservatively treating exported functions as address-taken even when no pointers to them exist in code or data (Section 4.2.2), and (ii) failing to identify non-returning calls, which introduces spurious fall-through paths (Section 4.2.3).

With respect to soundness, **angr** produces incomplete syscall sets for most applications, resulting in multiple missed syscalls and substantial test failures. These misses primarily stem from angr's heuristic filtering during address-taken function identification, particularly its exclusion of constants inferred as floating-point values or strings. Such heuristics eliminate legitimate function pointers and therefore miss valid indirect-call targets.

Among the remaining tools, **IDA Pro** and **Ghidra** achieve complete syscall sets across all evaluated applications, resulting in 100% test pass rates. SAFER also remains complete for nearly all programs, missing only a single syscall in *lighttpd*, which originates from a runtime-executed `bash` script outside our static analysis scope. Similarly, **Binary Ninja** misses a single syscall for *apache*, *bmq*, *comdb2*, *lighttpd*, *postgresql*, and *mysql*. In every case, the missing syscall originates from a program executed at runtime (e.g., Apache invoking Python CGI scripts), which is intentionally excluded from our analysis.

Overall, these results highlight a clear trade-off between precision and soundness in syscall inference. Conservative analyses such as IDA Pro and Ghidra preserve soundness by aggressively over-approximating control flow, whereas more aggressive heuristics, as employed by **angr**, improve precision at the cost of completeness. Binary Ninja provides the best trade-off among the evaluated general-purpose binary analysis tools, producing the smallest syscall sets while remaining sound for the analyzed binaries.

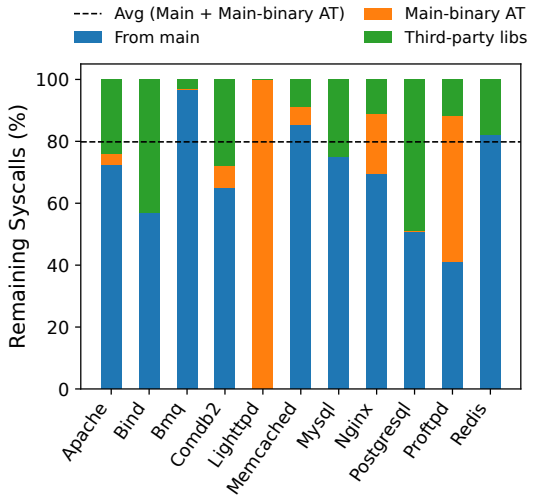
7.3 Evaluating Control Flow Refinement Techniques

7.3.1 Evaluating RQ3 (Impact of Control-Flow Refinement). Table 4 evaluates the incremental impact of each control-flow refinement technique on syscall inference. All refinement techniques—including those borrowed from Sysfilter—are implemented on top of the SAFER backend. We use SAFER with relocation-based address-taken (AT) identification as the baseline, since Sysfilter's context-sensitive refinement inherently relies on relocation metadata.

Table 5. Soundness of syscall filtering for Sysfilter's context sensitive AT refinement (+SF-AT) and all control-flow refinements (+Improvements). We report test pass rate (% Pass) and number of missed syscalls (Miss). The ∗ in the Pass column indicate that the missing system calls in those cases have been manually verified to be coming from a dynamically executed program via `system` or `execv` function call

	+SF-AT		+Improvements	
Program	Pass (%)	Miss	Pass (%)	Miss
Apache-httpd	100*	6	100*	6
Bind	100	0	100	0
BlazingMQ	100*	1	100*	1
Comdb2	100*	1	100*	1
Lighttpd	100*	1	100*	1
Memcached	100	0	100	0
Mysql	100*	1	100*	1
Nginx	100	0	100	0
Postgresql	100*	1	100*	1
Proftpd	100	0	100	0
Redis	100	0	100	0

Fig. 2. Provenance of remaining syscalls after all control-flow refinements. `From main` denotes syscalls reachable via direct control-flow from the program entry, `Main-binary AT` denotes syscalls reachable from address-taken functions within the main binary, and `Third-party libs` denotes syscalls originating from address-taken functions in third-party libraries.



Relocation yields a consistent set of address-taken functions by accurately identifying genuine code pointers from ELF relocation entries, thereby eliminating a major source of imprecision that is independent of the underlying binary analysis backend. Consequently, the subsequent refinement techniques operate on a common set of address-taken functions and are largely backend-independent. We therefore evaluate them using a single backend to isolate their individual impact while avoiding the substantial engineering effort of reimplementing each refinement across all tools. We choose Safer because it (i) provides direct access to ELF relocation metadata, (ii) is open source, (iii) achieves baseline precision comparable to commercial binary analysis tools, and (iv) provides the best balance of precision and soundness among the open-source backends evaluated (Table 1).

Each subsequent column in Table 4 cumulatively applies the refinement indicated in its header. We report both the inferred syscall count (*Sys*) and the average number of targets per unresolved indirect control-flow transfer (*AICT*), allowing us to distinguish reductions in the global syscall set from improvements in local control-flow precision.

We first evaluate **relocation-based address-taken (AT) identification**. By leveraging relocation metadata to distinguish genuine code pointers from address-like constants, this refinement substantially reduces the global AT set, lowering inferred syscall counts by approximately 40% across all benchmarks.

Building on relocation, **Sysfilter's context-sensitive AT refinement** further prunes the global AT set by retaining only pointers reachable under valid execution contexts. This refinement consistently reduces the inferred syscall set by a further 10–20% while simultaneously reducing AICT. As shown in Table 5, these reductions are achieved without affecting soundness. The few

remaining misses originate from dynamically executed programs (e.g., `execve`/`system`) that lie outside our static analysis scope.

Adding **data-section splitting and escape analysis** on top of Sysfilter produces the largest additional reduction in syscall counts across almost all applications. These techniques partition the global data section into bounded objects and discard pointers that do not escape to indirect control-flow contexts, thereby eliminating many spurious code pointers that would otherwise be conservatively included. On average, this refinement reduces syscall counts by approximately 3–20% (e.g., *redis* from 126 to 104 and *comdb2* from 132 to 128), while also substantially reducing AICT. Table 5 further shows that these additional reductions preserve the same level of soundness as Sysfilter, indicating that the discarded pointers do not correspond to feasible program behaviors in our evaluation. This confirms that techniques that shrink the *global* address-taken function set directly reduce syscall over-approximation.

8 Findings

(1) **Feasibility across real-world binaries.** The binary analysis tools evaluated in this paper are robust and can analyze large, real-world programs compiled at different optimization levels and in both stripped and unstripped forms to infer syscall filters.
(2) **Trade-offs between precision and soundness.** Precision and soundness vary significantly across tools. Conservative analyses (e.g., IDA Pro and Ghidra) consistently produce sound but highly over-approximated syscall sets, whereas more aggressive tools such as angr infer smaller syscall sets at the cost of missing required system calls under real workloads. In default (off-the-shelf) configurations, Binary Ninja produces the least over-approximated syscall filters while remaining sound.
(3) **Limited impact of per-call refinements.** Control-flow refinement techniques that improve precision on a per–control-flow-transfer basis—such as function signature matching and global constant propagation—substantially reduce the average number of targets per indirect call, but have little to no impact on the inferred syscall set.
(4) **Importance of reducing the global address-taken set.** Techniques that reduce the global set of address-taken functions are substantially more effective for syscall reduction. In particular, using relocation information to identify genuine code pointers reduces syscall over-approximation by approximately 40%. Sysfilter-style context-sensitive address-taken function pruning further reduces inferred syscall sets by at least 10%.
(5) **Effectiveness of data-section splitting and escape analysis.** We introduce additional heuristics, including data-section splitting and escape analysis, which further reduce syscall sets by 3–20%. While data-section splitting is not theoretically sound in the general case, it proved to be sound for all programs in our evaluation corpus.
(6) **Practical limits of static syscall reduction.** After aggressive control-flow refinement, most remaining syscalls originate from core program logic reachable from `main` or from legitimate callbacks within the main binary, revealing a practical lower bound on syscall reduction achievable via sound static binary analysis alone. Further reductions are likely to require configuration- or workload-aware techniques, such as profile-guided analysis or deployment-specific specialization, which trade generality for tighter syscall filters.

9 Related Work

This paper investigates binary analysis frameworks for system call filtering. This section reviews prior work in two areas: binary analysis and system call filtering.

9.1 Binary Analysis

Binary analysis platforms can be divided into two broad categories: (i) Dynamic analysis platforms [2, 10] and (ii) Static analysis platforms [17, 26, 37, 42, 49, 50, 53, 54]. Dynamic analysis observes concrete executions and can achieve high precision for exercised paths, but is inherently limited by workload coverage and requires runtime instrumentation. Static analysis reasons about all statically reachable code paths and is therefore better suited for generating a more complete syscall filter offline. Recent static techniques leverage relocation and loader metadata (e.g., Retrowrite [17], Egalito [50]) to precisely identify code pointers, improving over earlier binary analysis platforms such as PSI [54] and Multiverse [7] that rely on heuristics such as scanning the code and data sections for constants that look like code pointers.

9.2 Binary Points-To Analysis

Points-to analysis determines the set of memory locations or functions that a pointer may reference and is commonly used to resolve indirect calls. Classic source-level approaches, including Andersen's inclusion-based analysis [4], Steensgaard's unification-based analysis [46], and data-structure analysis (DSA) [31], rely on type information and explicit variable semantics, which are unavailable in binaries.

Early binary analyses such as value-set analysis (VSA) [6] track register values using strided intervals but suffer from severe over-approximation due to merges at control-flow joins, often leading to scalability and precision issues. More recent works, including BPA [28] and BinPointer [29], mitigate this by partitioning memory into abstract blocks and performing points-to analysis over these blocks. BinPointer employs a coarse, region-based model (e.g., stack, heap, globals) that allows overlapping regions and quickly loses precision as pointers flow through memory. BPA improves precision by introducing a global, non-overlapping memory partitioning derived from memory access patterns, yielding more precise indirect-call target sets. However, BPA prioritizes scalability and precision and does not guarantee full recall. BinDSA [21] further attempts to recover abstract data-structure information. Its evaluations show that both BinDSA and BPA have less than 100% recall in indirect-call recovery.

Our data-section splitting is closest in spirit to BPA's global memory partitioning, but differs in scope and objective. We conservatively partition only global data objects that can be confidently recovered by static analysis (e.g., jump tables and function pointer tables), and do not partition stack or heap memory. Since syscall inference depends on approximating the global set of address-taken functions rather than resolving per-call targets, this restricted partitioning—combined with escape analysis—is sufficient and empirically sound in our evaluation.

9.3 System Call Filtering

Restricting the set of system calls that a program may invoke has long been recognized as an effective defense mechanism, originally explored in the context of host-based intrusion detection [18, 20, 22, 27, 30]. More recently, syscall filtering has gained renewed attention as a mitigation against user-level exploits and as a means of reducing the kernel attack surface by limiting reachable kernel code paths.

Several systems combine static and dynamic techniques to construct syscall filters. Confine [23] generates syscall profiles for containers by composing a source-level `glibc` call graph with dynamically observed binaries executed inside the container. Chestnut [12] similarly derives syscall filters using static CFG recovery with angr [45] and dynamic execution. Since angr is already evaluated extensively in our framework, we do not separately compare these systems.

Other works emphasize *configuration or phase-aware* syscall filtering. SPEAKER [32], Temporal Specialization [24], and Syspart [41] generate distinct syscall policies for different execution phases or runtime contexts. These approaches demonstrate that incorporating workload or configuration knowledge can significantly reduce syscall sets, but they rely on assumptions about execution phases rather than improvements in static control-flow recovery. In the context of RQ4, such techniques complement our findings by illustrating how further syscall reduction may require configuration or workload-aware specialization beyond what sound static analysis alone can achieve.

Finally, several studies explore argument-level syscall filtering [33, 34, 43, 51, 52], aiming to restrict not only which syscalls are allowed but also how they are invoked. These approaches typically depend on fine-grained argument semantics or source-level information and are therefore orthogonal to our focus on syscall-set inference from binaries.

10 Conclusion

In this paper, we presented an empirical study of syscall inference from binaries using contemporary binary analysis tools. We evaluated the feasibility, precision, and limitations of inferring syscall filters for real-world, binary-only programs, and systematically examined the impact of control-flow refinement techniques on the resulting syscall sets.

Our results show that precision and soundness vary significantly across tools. Conservative analyses achieve soundness at the cost of substantial over-approximation, while more aggressive approaches reduce syscall sets but risk unsoundness under real workloads. We further demonstrate that refinements targeting individual indirect calls improve local control-flow precision but rarely translate into smaller syscall filters. In contrast, techniques that reduce the global address-taken function set—such as relocation-based filtering, context-sensitive refinement, data-section splitting, and escape analysis—are the primary drivers of syscall reduction.

Finally, our analysis reveals a practical limit to syscall reduction achievable via sound static binary analysis alone. After aggressive refinement, most remaining syscalls originate from core program logic or legitimate callback-based designs within the main binary. Further reductions are therefore likely to require configuration- or workload-aware techniques that trade generality for tighter syscall filters. We hope that the insights from this study help guide future work on practical syscall filtering and binary analysis for system hardening.

11 Data Availability

The artifact accompanying this paper, including the source code of our syscall inference framework, evaluation scripts, and instructions for reproducing the experiments, is publicly available at https://github.com/bloomberg-science/spectra.

References

[1] [n. d.]. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

[2] 2020. Pin 3.2 User Guide. https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/.

[3] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40. doi:10.1145/1609956.1609960

[4] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen.

[5] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security*.

[6] Gogul Balakrishnan and Thomas Reps. 2010. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 1–84.

[7] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *NDSS*.

[8] Andrew R. Bernat, Kevin Roundy, and Barton P. Miller. 2011. Efficient, sensitivity resistant binary instrumentation. In *ISSTA*. doi:10.1145/2001420.2001432

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ASIACCS*. 30–40. doi:10.1145/1966913.1966919

[10] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *CGO*. doi:10.1109/CGO.2003.1191551

[11] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of CCS 2008*, Paul Syverson and Somesh Jha (Eds.). ACM Press, 27–38. doi:10.1145/1455770.1455776

[12] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*. 139–151. doi:10.1145/3474123.3486762

[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security*. 559–72. doi:10.1145/1866307.1866370

[14] Dino Dai Zovi. 2010. Practical return-oriented programming. *SOURCE Boston* (2010).

[15] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. Sysfilter: Automated System Call Filtering for Commodity Software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*.

[16] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. doi:10.1145/3033019.3033028

[17] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE S&P*. doi:10.1109/SP40000.2020.00009

[18] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. 2004. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*. 194–208. doi:10.1109/SECPRI.2004.1301324

[19] Antonio Flores-Montoya and Eric Schulte. 2019. Datalog Disassembly. In *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, Santa Clara, CA, USA, 1059–1076.

[20] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*. 120–128. doi:10.1109/SECPRI.1996.502675

[21] Lian Gao and Heng Yin. 2025. BinDSA: Efficient, Precise Binary-Level Pointer Analysis with Context-Sensitive Heap Reconstruction. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1190–1211. doi:10.1145/3728928

[22] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*.

[23] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*.

[24] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Security Symposium*.

[25] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-grained Configuration-driven System Call Filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. doi:10.1145/3548606.3559366

[26] Hex-Rays. 1998. IDA Pro Disassembler. http://www.hex-rays.com/idapro/.

[27] Kapil Jain and R Sekar. 2000. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*.

[28] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining indirect call targets at the binary level. In *NDSS*. doi:10.14722/ndss.2021.24386

[29] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. BinPointer: towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 169–180. doi:10.1145/3497776.3517776

[30] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Automating mimicry attacks using static binary analysis. In *USENIX Security Symposium*, Vol. 14. 11–11.

[31] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices* 42, 6 (2007), 278–289. doi:10.1145/1250734.1250766

[32] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-Phase Execution of Application Containers. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 230–251. doi:10.1007/978-3-319-60876-1_11

[33] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. doi:10.1145/3274694.3274703

[34] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*. doi:10.1109/EuroSP48549.2020.00010

[35] Huan Nguyen, Soumyakant Priyadarshan, and R Sekar. 2024. Scalable, Sound, and Accurate Jump Table Analysis. In *ISSTA*. doi:10.1145/3650212.3680301

[36] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *IEEE S&P*. doi:10.1109/SP40001.2021.00012

[37] Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R Sekar. 2023. SAFER: Efficient and Error-Tolerant Binary Instrumentation. In *USENIX Security*.

[38] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. 2020. On the impact of exception handling compatibility on binary instrumentation. In *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*. 23–28. doi:10.1145/3411502.3418428

[39] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. 2020. Practical fine-grained binary code randomization. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 401–414. doi:10.1145/3427228.3427292

[40] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2023. Accurate Disassembly of Complex Binaries Without Use of Compiler Metadata. In *ASPLOS*. doi:10.1145/3623278.3624766

[41] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. 2023. SysPart: Automated Temporal System Call Filtering for Binaries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1979–1993. doi:10.1145/3576915.3623207

[42] Roman Rohleder. 2019. Hands-on Ghidra - A Tutorial about the Software Reverse Engineering Framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*. doi:10.1145/3338503.3357725

[43] Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2023. Confine: Fine-grained System Call Filtering for Container Attack Surface Reduction. *Computers & Security* (2023), 103325. doi:10.1016/j.cose.2023.103325

[44] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications security (CCS)*. doi:10.1145/1315245.1315313

[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP)*. doi:10.1109/SP.2016.17

[46] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41. doi:10.1145/237721.237727

[47] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*. 121–141. doi:10.1007/978-3-642-23644-0_7

[48] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953. doi:10.1109/SP.2016.60

[49] Jordan Wiens, Rusty Wagner, Peter LaFosse, and Vector 35 Inc. 2016. Binary Ninja. https://binary.ninja/. Interactive reverse–engineering platform for binary analysis.

[50] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *ASPLOS*. doi:10.1145/3373376.3378470

[51] Yunlong Xing, Jiahao Cao, Kun Sun, Fei Yan, and Shengye Wan. 2022. The devil is in the detail: Generating system call whitelist for Linux seccomp. *Future Generation Computer Systems* 135 (2022), 105–113. doi:10.1016/j.future.2022.04.016

[52] Dongyang Zhan, Zhaofeng Yu, Xiangzhan Yu, Hongli Zhang, Lin Ye, and Likun Liu. 2022. Securing Operating Systems Through Fine-grained Kernel Access Limitation for IoT Systems. *IEEE Internet of Things Journal* (2022). doi:10.1109/JIOT.2022.3222074

[53] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 Security and Privacy Symposium*. 559–573. doi:10.1109/SP.2013.44

[54] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. *ACM VEE* (2014). doi:10.1145/2576195.2576208
[55] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*.

Received 2026-01-30; accepted 2026-04-16