AMPLE: Fine-grained File Access Policies for Server Applications

Seyedhamed Ghavamnia
Bloomberg
sghavamnia@bloomberg.net

Julien Vanegue
Bloomberg
jvanegue@bloomberg.net

***Abstract*—Userspace programs depend heavily on operating system resources to execute correctly, with file access being one of the most common and critical use cases. Modern Linux distributions include a vast number of files, many of which are unnecessary for the operation of most programs. However, existing access control mechanisms typically enforce coarse-grained policies that allow programs to access far more files than they actually require. This over-permissiveness significantly increases the system's attack surface, exposing sensitive resources to potential exploitation.**

In this paper, we introduce AMPLE (Automated MAC PoLicy Extraction), a versatile tool that integrates both static and dynamic analysis to identify the files required by server applications. Ample accomplishes this by leveraging the distinct phases of server application execution, extracting runtime-dependent file paths by executing only the program's initialization phase. This novel approach addresses the limitations of relying exclusively on static analysis, which fails to identify runtime-dependent file paths, as well as the shortcomings of purely dynamic analysis, which overlooks file paths accessed in non-executed code paths. To demonstrate its effectiveness, we evaluated Ample on ten widely-used server applications. The results show that Ample significantly reduces the number of accessible files, achieving an average reduction of over 99%, and limiting access to an average of fewer than 254 files per application. This substantial reduction helps restrict access to numerous security-critical files and mitigates 13 Linux kernel CVEs.

I. Introduction

Userspace programs rely on access to underlying operating system resources to function correctly. These resources include files, network sockets, pipes, and other similar entities, which are accessed through system calls provided by the kernel. Among these various resources, files are of particular significance, as they may contain sensitive data (e.g., `/etc/passwd`) or, more critically, provide a mechanism for executing code and performing arbitrary operations (e.g., `bash`).

Linux distributions ship with many files, some of which control system behavior (e.g., network configuration files), yet most programs require only a small subset. Existing access control policies are coarse-grained, typically based on file ownership, and thus grant programs access to far more files than needed. For example, on Ubuntu 24.04 an unprivileged process can read more than 76K of 86K files, including sensitive ones like `/etc/passwd`. Vulnerabilities such as Dirty COW [16] or Dirty Pipe [18] can escalate such access to unauthorized writes, while privileged processes bypass restrictions entirely. This violates the *Principle of Least Privilege (PoLP)* [50] and enlarges the attack surface, underscoring the need to restrict programs from accessing unnecessary files.

To prevent programs from accessing unnecessary files, two key steps are required: 1) accurately identifying the files each program needs to access; and 2) enforcing fine-grained access control policies. The latter is already supported by the Linux kernel through the Linux Security Module (LSM) framework, which underpins several security tools such as AppArmor, SELinux, and LandLock. However, the former—identifying the required files—is a significantly more challenging task, primarily for two reasons.

First, static determination of file paths in programs written in C is complicated by the pervasive use of data pointers, which obscure the memory locations and values being manipulated. Second, and more critically, many file paths are not known until runtime, as they are derived from the specific configuration of the production environment. For example, Nginx relies on several file paths that are specified only at runtime via its configuration file, such as the root directory of the hosted content. As a result, static analysis alone is insufficient for deriving restrictive access control policies.

Given these challenges, prior research efforts [57], [40], [38], [63], [64] and state-of-the-art tools—such as `aa-genprof` [17] and `aa-easyprof` [1]—have adopted profiling-based techniques (i.e., dynamic analysis) to generate restrictive file access policies for programs. These approaches execute the target application under test conditions while monitoring the accessed file paths. The logged files are further supplemented with commonly used directories deemed necessary for typical operation to generate the final policy. This dynamic approach is fundamentally imprecise—missing paths exercised only in untested code (underapproximation) while granting blanket access to large directories that the program never actually needs (overapproximation).

While prior works have applied static analysis to perform argument-level system call filtering [43], [61], [59], [49], they are limited to flag-type arguments (e.g., `int`). Hence, these approaches do not offer protection for file path arguments. To the best of our knowledge, no existing work leverages static analysis to derive file path policies. In this work, we aim to bridge this gap by combining static and dynamic analysis to derive accurate and restrictive file access policies.

In this paper, we present Ample, the *first* tool that combines static and dynamic analysis to pinpoint the files a server

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00111

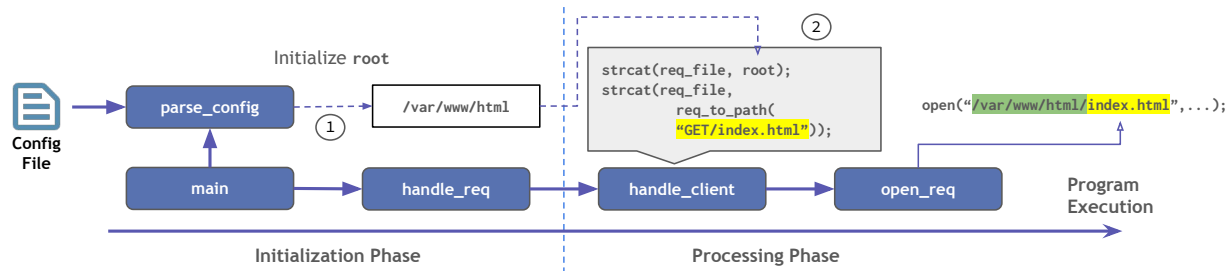

Fig. 1. Server applications parse configuration directives during initialization and use them when serving requests. In ①, the application sets the `root` path from its configuration; in ②, this root is combined with client input to form the final path for the open call. The green segment marks the initialization-determined prefix, while the yellow segment marks the client-dependent suffix.

application needs. Its key insight is that, although files are accessed throughout program execution, their paths (or parent directories) are fixed upon *initialization*. This aligns with the two-phase execution model established in previous syscall-filtering work [28], [48], where server applications parse runtime settings during their *initialization phase* and service client requests during the subsequent *processing phase*.

We develop Ample based on this key insight. Specifically, Ample's static analysis combines data flow analysis with a nested type-based analysis to derive concrete values for the hardcoded file paths; and to identify and instrument instructions that initialize memory objects containing file paths that are only available at runtime. These instrumented instructions extract the corresponding path values as soon as the program completes its initialization phase. Unlike pure profiling, which learns paths only when a system call finally touches a file, Ample captures paths as soon as they materialize in memory—typically during the program's initialization phase. Our evaluation confirms that most runtime-derived paths surface this early, allowing Ample to expose essential file dependencies well before traditional profiling methods would detect them.

Ample identifies the files required by a program and generates an AppArmor policy to enforce least-privilege access. While we use AppArmor [2] for its user-friendly format, the approach can also target SELinux [13] or LandLock [10]. Our LLVM-based prototype was evaluated on ten widely deployed server applications (Nginx, Memcached, Redis, Exim, Lighttpd, Netsnmpd, Proftpd, OpenSMTPD, Monkey Server, and Bind), five of which typically run with root privileges. Ample significantly reduces their filesystem exposure, limiting accessible files to fewer than 254 on average and producing policies up to an order of magnitude more restrictive than `aa-genprof`.

In summary, our work makes these main contributions:

(1) We present Ample, a novel hybrid analysis technique that leverages the two-phase execution model of server applications to generate restrictive file access policies.

(2) We present a novel combination of data flow, nested type-based, and pointer analysis to identify program file needs.

(3) We evaluate our prototype implementation with ten real-world and widely-used server applications, and demonstrate how Ample can identify excessive file resources available to these programs, which they do not require.

Our prototype implementation is publicly available as an open-source prototype at https://github.com/bloomberg/ample.

II. Background and Motivation

The Linux kernel treats all resources as files, and hence, many files significantly affect the system integrity. Given the numerous files accessible to each program and their potential to be exploited, it is essential to accurately identify the files needed by each application and restrict access to others.

A. Split-phase Policy Inference

Many files required by server applications are determined at runtime based on 1) runtime configuration settings; and 2) user requests. Therefore, policy inference through static analysis becomes extremely challenging and in most cases impossible. After looking into multiple server applications we observed that the files accessed by the server are limited based on the runtime settings. As shown by previous work [28], these settings are parsed and utilized during the server application's *initialization phase*. This phase operates without user interaction and concludes when the server is launched and ready to accept client requests. The server then transitions to the processing phase, during which it handles client requests [27], [48]. As shown in Figure 1, Ample extracts the environment-defined path `/var/www/html` using its hybrid approach, whereas purely static analysis misses it entirely; and purely dynamic analysis can only recover it after processing a client request.

B. Threat Model

Our threat model assumes an adversary who exploits a user-level vulnerability to gain arbitrary code execution. Ample operates without relying on other defenses: by restricting file access, it prevents attackers from leveraging sensitive files (e.g., cronjobs) for privilege escalation. Prior work shows that crafted files can also facilitate kernel exploitation [37]; by disallowing access to nonessential files, Ample reduces this surface. We further assume the attacker lacks root privileges before compromise, ensuring AppArmor cannot be disabled.

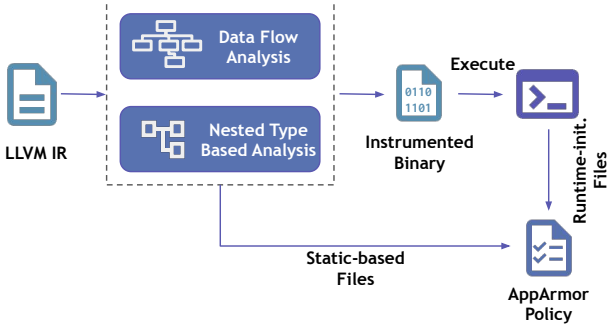

Fig. 2. Overview of Ample's process for generating AppArmor policies.

III. Design

The goal of our system is to enforce the principle of least privilege by deriving and enforcing fine-grained file access policies. To that end, Ample leverages static and dynamic analysis to identify files needed by a program. While some file paths are hardcoded within the program, many others depend on the runtime environment, and therefore, can only be determined at runtime. Hence, Ample integrates static and dynamic analysis to determine their values. We employ static analysis to identify and extract concrete values directly available in the code. For values that cannot be resolved statically, we instrument their initialization instructions to facilitate runtime extraction. Dynamic analysis is then applied to capture these values at the instrumented points, limiting execution to the program's initialization phase. Figure 2 presents an overview of how our system handles both statically concrete file path values, and runtime-initialized file path values.

A. File Path Extraction through Static Analysis

We first attempt to extract all file paths of the target program during compilation by performing static analysis. Our analysis uses three techniques: 1) data flow analysis; 2) nested type-based analysis; and 3) pointer-based value flow analysis. Since many file paths are defined outside the function in which they are used—specifically, where the file-related system call is invoked—an interprocedural analysis is required across all three techniques to accurately extract their values.

Therefore, we first generate a sound callgraph for the entire application and its libraries. Due to the abundance of indirect function call usage in C/C++ programs, we need to perform pointer analysis to resolve the targets of all indirect function call sites. Hence, we use the well-known Andersen's pointer analysis algorithm [20] to resolve the targets of all indirect function calls, and use them to generate a sound callgraph. The callgraph is sound under the assumptions that no implementation errors are present, no inline assembly is employed, and the analysis encompasses all relevant code.

a) Data Flow Analysis (DFA): Userspace programs access files through invoking *file-related system calls*, and hence, we first perform a one-time analysis of all system calls provided by the Linux kernel (v5.4) to identify them. We consider any system call which takes a path to a file or directory as an argument (e.g. `open`), a *file-related system call*. This argument typically has a character pointer type (`char*`). Ample starts from each system call invocation site (and its libc call sites) and performs interprocedural backward (and forward) analysis to extract the source of the values of interest. We combine the *use-def* chains to reach the point of definition for each value. In case the value is defined by an argument passed to the current function we recursively continue our analysis at *all* caller functions.

Ample's data flow analysis is capable of extracting a concrete file path value if it is explicitly present in the source code and defined within the same control flow path in which it is utilized. If our analysis fails to determine a concrete value, it continues applying data flow analysis to identify all initialization instructions associated with that value. These instructions are then instrumented to extract the value at runtime. If our data flow analysis reaches a point where the file path value is copied from a data struct field, we leverage our nested type-based analysis and pointer analysis to handle it. These are discussed in the following paragraphs.

b) Nested Type-based Analysis: Many programs utilize struct types to store file paths that are accessed throughout the application. For example, Nginx defines a struct named `ngx_conf_file_t` to hold core server configuration settings, including the file paths of directories served by the web server. Consequently, accurately identifying struct fields used for this purpose and extracting their associated file paths is essential. Ample employs a nested type-based analysis to facilitate this process. Once Ample's data flow analysis determines that a file path is stored within a struct field, it locates all instructions that initialize memory objects referenced by that struct field. It then applies data flow analysis at the initialization points to statically extract concrete values—when available in the source code. If a static value cannot be resolved, Ample instruments the relevant initialization instructions to enable extraction of the file path at runtime.

A key challenge is that many programs employ general-purpose structs to represent strings, including file paths. For instance, Nginx defines all strings with the `ngx_str_t` struct, which stores a length field and a pointer to the underlying memory. Because this type is used broadly—for log messages, HTTP headers, and file paths—naively tracking all initializations yields numerous irrelevant cases. Ample addresses this by analyzing chains of nested struct types to precisely identify initializations that correspond to file paths.

Our nested type-based analysis is similar to previous work [39] which propose multi-layer type analysis. However, MLTA focuses on control pointers, whereas Ample focuses on data pointers which are more widely used and add extra complexity. More specifically, since MLTA only focuses on function pointers they have a clear starting point which are address-taking operations. However, in most cases, Ample needs to first perform DFA from the API call site to reach the point where it can extract the nested struct types. This is shown in Figure 3, where Ample's DFA first determines that

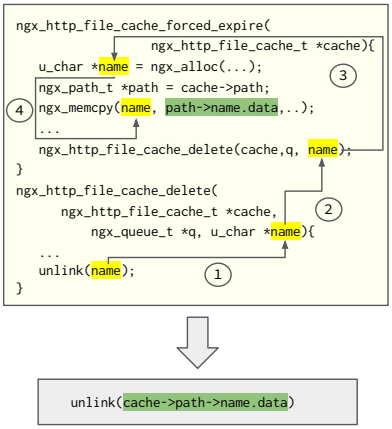

Fig. 3. Code example from Nginx, showing how Ample identifies file paths stored in nested types through interprocedural backward and forward data flow analysis. In the first three steps we perform backwards data flow analysis interprocedurally to reach the local variable defined on the stack (`u_char *name`), then we perform forward data flow analysis interprocedurally to identify any memory writes to that address (step ④). At this point our analysis determines that the file path is stored in a nested type.

the file path passed to the open call is actually stored in a nested struct type. Therefore, we have expanded on the idea on multi-layer struct types to design a more complete solution.

c) Type Escaping: Ample relies on struct type information to detect writes to file paths stored as struct fields; however, this approach may miss cases where memory is modified through generic pointer types (e.g., `void*`) (i.e., *type escaping*). This can occur when a struct's address is passed to another function and cast to a different or generic type, allowing modifications that bypass the original type annotations. This similarly applies to global variables passed to other functions through a generic pointer.

This limitation renders type-based analysis potentially unsound for identifying initialization points of struct fields and global variables discovered during data flow analysis. To address this, Ample performs a preprocessing step in which it identifies all struct types and global variables whose addresses are either passed to other functions or stored in other variables. During data flow analysis, if any of these identified struct types or global variables are used to store the targeted file path, Ample bypasses type-based analysis for those cases. Instead, it leverages Andersen's pointer analysis [20] to conservatively determine *all* initialization instructions that may write to the memory locations referenced by the final file path pointer at the API call site. Although this approach incurs overapproximation, it ensures that the analysis preserves correctness with respect to the behaviors captured.

Algorithm 1 summarizes the steps Ample performs to extract file paths available in the code and to instrument runtime-based file path initialization instructions. Lines 2-7 show preliminary analyses performed by Ample to identify initialization instructions for different memory objects (e.g., global variables, struct types). Line 9 calls the backward data flow analysis function Ample performs for each call site argument (shown in Algorithm 2). Line 10 calls the nested type-based initialization extraction for file paths stored in struct fields (shown in Algorithm 3). Line 11 extracts the initialization instructions for global variables identified as sources of file paths in our backwards DFA. Lines 12-25 represent how we improve soundness by falling back to extracting initialization instructions based on the points-to set results if needed.

B. Runtime-dependent File Paths

Some file paths are dependent on runtime behavior and cannot be fully resolved through static analysis alone. We refer to these file paths as *runtime-dependent* (or *runtime-initialized*) file paths. For such cases, Ample instruments the corresponding initialization instructions to extract their values dynamically. A limitation of this approach is its reliance on code execution, which can compromise soundness if the relevant initialization instructions reside in code paths that are not exercised—particularly those triggered by specific user inputs. However, our observation is that, in many cases, file paths—or at least their main directories—are initialized during the program's initialization phase. This insight enables Ample to capture critical path components early in execution, thereby mitigating the risk of incomplete coverage and preserving correctness in practice. For example, Nginx parses and stores its web root during initialization, but the full file path for a client request is determined only during processing.

We determine whether an initialization instruction will be executed during the program's initialization phase or processing phase by considering the transition point on the callgraph. Similar to previous works [27], [28], we expect the developer to provide the transition point between the initialization phase and the processing phase. This is the only information needed from the developer, the rest of the analysis is done automatically by Ample. This is discussed in Section VII.

C. Policy Enforcement

Ample combines the resources identified through its static analysis and runtime-based analysis to generate the final AppArmor policy for an application. When generating the final policy, it takes into account the API type to determine 1) the permissions; and 2) whether the extracted paths need wildcards. For example, the `dlopen` API is used to open a file for execution and therefore its identified file would need the execute permission. The `chdir` API changes the current working directory of the process, and therefore Ample's policy would allow access to any files under that directory.

Ample currently generates single-file AppArmor policies, listing each required file and its permissions on a separate line. While it does not yet output policies in the formats required by other LSM frameworks (e.g., SELinux, LandLock), it identifies all resources accessed by the program—a fundamental requirement for any LSM tool—and can thus be readily extended to support additional backends.

Algorithm 1 Access Policy Inference: Statically extract concrete paths values (partial access policy) and store instructions to be instrumented. Then, execute the instrumented program until the transition point and synthesize the complete policy.

Require: $ir = i_1 \dots i_n$ // IR instr list
Ensure: $(ppol, iir)$ // Partial Policy + Instrumented IR

procedure StaticPolicyInference
  // Perform Andersen's pointer analysis algorithm
  $(cg, pts\_to\_sets) \leftarrow$ AndersenPtrAnalysis(ir)
  // Extract pointer initialization instructions
  $(pts\_to\_inits) \leftarrow$ ExtractInitialPtrInstrs(ir)
  // Identify file access-related API call sites
  $(cs\_args) \leftarrow$ ExtractFileAccessCallSites(ir)
  $iir \leftarrow ir$
  $(ppol, gvars, stfields) \leftarrow$ BackwardsDFA(cs_args)
  $(stInits) \leftarrow$ ExtractTypeInitInstr($ir, stfields$)
  $(gvarInits) \leftarrow$ ExtractInitInstr($ir, gvars$)
  // Select Pointers To Instrument
  foreach var in $gvars \cup stfields$ **do**
    if addr_is_taken(var)
      $PtrRequired[var] \leftarrow$ true
  // Instrument Selected Pointer Fields Accesses
  foreach $stInit$ in $stInits$ **do**
    $iir \leftarrow iir \cup$ InstrumentInstr($ir, stInit$)
  // Instrument Selected Global Pointers Accesses
  foreach $gvarInit$ in $gvarInits$ **do**
    $iir \leftarrow iir \cup$ InstrumentInstr($ir, gvarInit$)
  // Instrument Points-to Set Initializations
  foreach ptr in $PtrRequired$ **do**
    foreach ptr_init in $pts\_to\_inits[ptr]$ **do**
      $iir \leftarrow iir \cup$ InstrumentInstr(ir, ptr_init)
  return $(ppol, iir)$

Require: $ir = i_1 \dots i_n$ // IR instr list
Ensure: pol // Inferred Policy

procedure HybridPolicyInference
  $(ppol, iir)$ = StaticPolicyInference(ir)
  // Run the instrumented program until the transition point
  $rpol \leftarrow$ ExecuteInstrumented(iir)
  return $ppol \cup rpol$

Algorithm 2 Backwards data flow analysis algorithm. `PathArg`: file path argument passed at call site, `DefKind`: switch based on definition type of value

Require: cs_args // File path call site arguments
Ensure: $(ppol, gvars, stfields)$ // Partial Policy + Global Variables + Struct Fields

procedure BackwardsDFA(cs_args)
  $ppol \leftarrow \emptyset$, $gvars \leftarrow \emptyset$, $stfields \leftarrow \emptyset$
  foreach cs_arg in cs_args **do**
    $Seen \leftarrow \emptyset$; $Origins \leftarrow \emptyset$; $v_0 \leftarrow$ PathArg(cs_arg)
    $W \leftarrow \{(\text{Func}(cs), v_0)\}$
    while $W \neq \emptyset$ **do**
      $(f, v) \leftarrow$ Pop(W)
      if $(f, v) \in Seen$
        continue
      $Seen \leftarrow Seen \cup \{(f, v)\}$
      switch DefKind(v):
        case Argument $farg_i$: // Reached arg, analyze callers
          foreach $c \in$ Callers(f) **do**
            Push(W, (Func(c), $c.arg_i$))
        case CallInst cs: // Defined by function call
          if RetVal of cs // Defined by return value
            foreach $ce \in$ CalleeOf(cs) **do**
              Push(W, (ce, RET_VAL(ce))) // Analyze callee
          if arg_i of cs // Defined by argument
            foreach $ce \in$ CalleeOf(cs) **do**
              Push(W, (ce, $ce.arg_i$)) // Analyzing callee arg
        case Alloca / MemObj / DeRef:
          foreach $s \in$ ReachingStores(f, v) **do**
            Push(W, (f, StoredValue(s)))
        case GetElementPtrInst and isStField(v):
          $stfields \leftarrow stfields \cup \{v\}$
        case Global:
          $gvars \leftarrow gvars \cup \{v\}$
        case Constant:
          $ppol \leftarrow ppol \cup \{v\}$
      end switch
  return $ppol, gvars, stfields$

IV. Implementation

This section outlines the implementation of Ample, which consists of two primary components: a compile-time and runtime analysis. The compile-time analysis is implemented as an LLVM pass that statically analyzes the target program. Following static analysis, Ample executes the initialization phase of the instrumented binary to collect runtime-initialized file path values. Finally, it combines both static and runtime file paths to construct the complete AppArmor policy.

A. Compile-time Analysis

At compile time Ample (i) resolves file paths that are already concrete, and (ii) injects instrumentation at the instructions that initialize paths whose value must later be captured. It walks the *use-def* chain backward from every API call site, applying its nested type-based analysis when paths are stored in structs. For cases where we need to fall back to pointer analysis (as discussed in Section III), our tool utilizes SVF's [53], [54] implementation of Andersen's points-to analysis algorithm [20], enabling the extraction of the points-to sets for each pointer in the program. This analysis also resolves the targets of function pointers, and Ample uses SVF's callgraph to ensure sound callgraph construction.

a) Nested Type-based Analysis: When a file path pointer is stored as a nested struct field, Ample augments its data-flow analysis with a nested type-based analysis. In LLVM, struct and array accesses are represented by `GetElementPtrInst` instructions [11]. Ample analyzes these instructions to recon-

Algorithm 3 Type-based initialization extraction. `isWrite`(I): I is a memory-writing instruction (e.g., Store, memcpy), `isStField`(G): GEP whose pointer operand is a struct type, `PtrOp`(I): the pointer-typed operand written-to by I, `StructOfPtr`(G): the struct type of G's pointer operand, `DefKind`: switch based on definition type of value

Require: $(ir, stfields)$ // Instr list + targeted struct fields
Ensure: $InitInstr$ // Struct fields initialization instructions

procedure EXTRACTTYPEINITINSTR($ir, stfields$)
  $Chains \leftarrow \emptyset$
  foreach $g \in stfields$ **do**
    $Chains \leftarrow Chains \cup$ BACKWDTYPECHAINS(g, $\langle\ \rangle$)
  $InitInstr \leftarrow \emptyset$
  foreach $I \in$ INSTRUCTIONS(ir) **do**
    if ISWRITE(I)
      $WChains \leftarrow$ BACKWDTYPECHAINS(PTROP(I), $\langle\ \rangle$)
      if $WChains \cap Chains \neq \emptyset$
        $InitInstr \leftarrow InitInstr \cup \{I\}$
  return $InitInstr$

Require: (v, τ) // Struct field value + prefix type chain
Ensure: Res // Set of all computed type chains for v

procedure BACKWDTYPECHAINS(v, τ)
  $chain\_size \leftarrow len(\tau)$
  $Seen \leftarrow \emptyset$, $Res \leftarrow \emptyset$
  $W \leftarrow \{$FUNCOF$(v), v, \tau)\}$
  while $W \neq \emptyset$ **do**
    $(f, x, \tau) \leftarrow$ POP(W)
    if $(f, x, \tau) \in Seen$
      continue
    $Seen \leftarrow Seen \cup \{(f, x, \tau)\}$
    switch DEFKIND(x):
      case GetElementPtrInst and isStField(x):
        $\tau \leftarrow \tau \cdot \langle$ STRUCTOFPTR(x) $\rangle$
        PUSH(W, (f, PTROP(x), τ))
      case Cast/PtrToInt/IntToPtr/Load:
        PUSH(W, (f, OPERAND(x), τ))
      case Argument $farg_i$:
        foreach $c \in$ CALLERS(f) **do**
          $Res \leftarrow Res \cup$ BACKWDTYPECHAINS($c.arg_i, \tau$)
    end switch
  if $len(\tau) \neq chain\_size$
    $Res \leftarrow Res \cup \{\tau\}$
  return Res

struct the chain of nested struct types by recursively traversing backward through the `GetElementPtrInst` chain until the complete hierarchy is recovered.

After identifying the nested type chain, Ample locates the corresponding initialization instructions. It scans all `StoreInst` and `CallInst` operations, checks whether the target address type matches the innermost struct in a recovered chain, and then verifies the entire chain. When the match is confirmed, the instruction is marked for instrumentation (as shown in Algorithm 3).

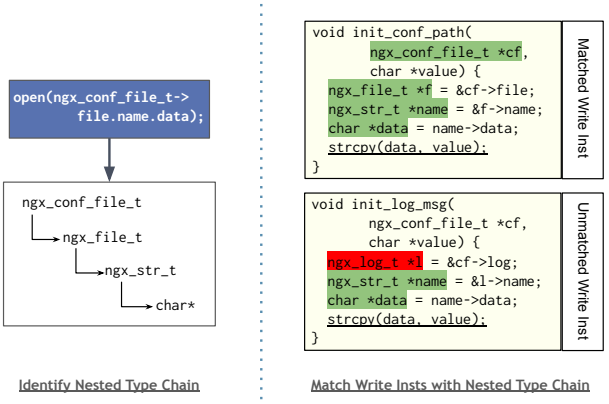

Fig. 4. Improved precision in matching path-writing instructions using nested type chains. Green: included matches; Red: excluded

This process is illustrated in Figure 4. On the left, the identified struct chain is shown, while on the right, two candidate write instructions are presented. Both call the `strcpy` function (`CallInst`). The write instruction in the `init_conf_path` function matches the identified struct chain and is therefore recognized as an initialization instruction for the file path passed to the open call. In contrast, the instruction in the `init_log_msg` function does not match and is excluded.

b) Instrumentation: Ample instruments three categories of write instructions: 1) modify file-related fields within a nested struct chain; 2) modify global variables serving as sources for file-related API arguments; and 3) instructions that write to memory objects identified—via pointer analysis—as locations that store file paths.

Handling differs by instruction type. For `StoreInst`, writes typically occur character-by-character within loops performing pointer arithmetic; our instrumentation records both the written character and the target address, and the runtime aggregates successive writes into complete file paths. To our knowledge, this is the first application of pointer tracking for dynamic string reconstruction. For `CallInst`, typically invoking string-copy routines (e.g., strcpy), the instrumentation instead logs the source pointer. Finally, Ample instruments the boundary between initialization and processing phases, identified by manual source-code inspection as in prior work [26], [27], to terminate extraction of initialization-dependent file paths.

B. Runtime Analysis

The compile-time analysis component of Ample produces an instrumented binary that must be executed in order to extract file paths initialized at runtime. Many applications support a broad range of configuration options specified by the user at launch time, some of which refer to files that the application must access. Common sources of runtime-initialized file paths include environment variables, command-line arguments, and configuration files.

When file paths are specified via environment variables, they are often retrieved through invocations of functions such as

getenv or secure_getenv. Our tool statically identifies these instructions and instruments them to extract the environment variable values at runtime. These values are typically available during the program's initialization phase, allowing extraction without requiring execution of the full application. In addition, environment variables may also be passed as arguments to the main function, which are similarly handled.

For applications that utilize command-line options or configuration files to specify file paths, Ample's instrumentation captures these values during runtime execution. These dynamically obtained file paths are incorporated into the final policy.

To perform the runtime phase, Ample launches the instrumented binary and monitors its execution until it transitions into the processing phase.

V. Evaluation

In this section, we evaluate the correctness and usability of Ample in generating restrictive policies by generating AppArmor profiles for ten popular server applications and comparing them with aa-genprof's profiles. We use aa-genprof as the baseline because it is natively supported on Ubuntu and generates appropriately restrictive policies. Other prior tools are less suitable: aa-easyprof produces overly permissive profiles [1], ASPGen [36] focuses on log parsing already covered by aa-genprof, and the remaining tools [40], [63], [38] are either not open source or no longer maintained and incompatible with recent distributions.

For each application, we generate the LLVM IR and run Ample's static analysis against it. Ample's analysis extracts any file paths available in the source code and creates an instrumented binary by identifying and instrumenting file path initialization instructions.

A. Application Statistics

We have used the following ten server applications for our evaluation: Nginx, Memcached, Redis, Exim, Lighttpd, Snmpd (Net-Snmp), Proftpd, OpenSMTPD, Monkey Server, and Bind. Table I reports, for each application, (i) the total number of file-related syscall invocation sites, (ii) the subset whose *every* incoming data-flow path resolves to a concrete path at compile time ("Static"), even when reached from multiple callers, and (iii) the remaining sites that require runtime instrumentation to obtain the path ("Runtime Inst"). We have provided details about the programs in our dataset in the following paragraphs.

a) Web Servers: Nginx, Lighttpd, and Monkey are all widely-used web servers, and transition into their processing phase in the ngx_worker_process_cycle, server_main_loop, and mk_server_loop functions, respectively. Each API call site in these servers is reached by multiple, sometimes runtime-only, control-flow paths, so our static analysis rarely resolves concrete paths for them.

b) Databases: Memcached and Redis are both widely-used in-memory databases. The worker_libevent and aeMain functions are the transition point between the initialization phase and the processing phase for Memcached and Redis, respectively and are annotated accordingly.

TABLE I
Static and runtime file-related syscall locations for each application along with the each program's size (LoC).

Application	Size (LoC)	APIs		
		Total	Static	Runtime
Nginx	141K	60	1	59
Memcached	32K	28	4	24
Redis	203K	157	17	140
Exim	142K	178	3	175
Lighttpd	99K	35	1	34
Snmpd	370K	103	41	62
Proftpd	692K	72	9	63
OpenSMTPD	120K	60	44	16
Monkey Server	57K	41	0	41
Bind	329K	401	62	339

c) Mail Servers: Exim and OpenSMTPD, two widely used mail servers, make extensive file accesses. Exim parses its configuration in main and then shifts to its processing phase, so we mark the basic block in main where that transition occurs. OpenSMTPD likewise needs a basic-block annotation to delineate initialization from processing.

d) Net-Snmp: Simple Network Management Protocol (SNMP) is the standard monitoring protocol for network equipment. Net-Snmp is an open source implementation of this protocol, and we evaluate Ample by generating AppArmor policies for Net-Snmp's snmpd daemon, which runs on client hosts and answers queries from SNMP managers.

e) Proftpd: Proftpd is a highly configurable open-source FTP server and can be used to serve files. Proftpd has two functions which start the processing phase, daemon_loop and fork_server, which we annotate accordingly.

f) Bind: A widely-used DNS server, providing name resolution service. Bind has a setup function which initializes the environment and then starts handling the client requests. We annotate the main function after it calls the setup function.

B. Policy Effectiveness

To analyze the effectiveness of Ample's policies in reducing each program's accessible files, and compare it against the current state-of-the-art, aa-genprof, we generated policies for each server application using both tools. To ensure a fair comparison, we stopped collecting runtime file paths once the program transitioned into its processing phase (for both tools). We ran our experiments on Ubuntu 24.04, on an AMD EPYC 7543P Server with 32 cores and 512GB of RAM.

To derive the number of files accessible by each policy, counting policy lines is misleading, because each rule can expand through variables or wildcards and aa-genprof profiles often include other policy files. For instance, Memcached's aa-genprof profile pulls in tunables/global and abstractions/base, which themselves import additional rules, greatly enlarging the effective file set. Instead of resolving every variable, wildcard, and include, we adopt an empirical test: a stub C program that simply opens a given path is confined by each policy, then executed against every

file on the system; we log which paths succeed or fail, allowing a direct comparison between Ample and `aa-genprof`. We use a vanilla Ubuntu 24.04 Docker image, supplemented with application-specific file paths for this comparison.

Figure 5 shows how Ample significantly outperforms `aa-genprof` providing access to much less files. Out of over 86K files present in the Docker image—of which approximately 76K are accessible to unprivileged processes—Ample-generated policies reduce file access significantly, permitting as few as 31 files in the most restrictive case and up to 1775 files in the least restrictive case. It is noteworthy that Ample provides better security guarantees while simultaneously providing better correctness guarantees, since `aa-genprof` is inherently limited by its strict dynamic analysis nature.

Snmpd is the only case where Ample is less restrictive, permitting 1,775 files versus 1113 with `aa-genprof`. The main reason for the high number of files accessible through Ample's policy is that Snmpd opens the `/proc` directory using the `opendir` API, and then traverses files under that directory based on its requirements. One of the use cases of this is the `sh_count_procs` function, whose goal is to extract the status and number of processes running on the system. This function opens the `/proc` folder and then traverses over all the running processes, attempting to read the `/proc/[pid]/cmdline` and `/proc/[pid]/status` files. When APIs such as `opendir` are used, Ample conservatively grants directory-wide access to reduce the risk of missing required files. However, upon examining the policy generated by `aa-genprof` we realized it only includes a subset of the files needed, and therefore, breaks the program. Ample could benefit from finer-grained data-flow analysis that tracks returned directory streams, which we leave for future work. This limitation stems from our design choice to trade some security for functionality; Section VII discusses this in detail.

Although `aa-genprof` relies on dynamic tracing, its profiles still allow broad directories such as `/usr` that were never exercised, making them simultaneously *unsound* (by missing paths needed in untested code) and *incomplete* (by admitting unnecessary files). For example, our Nginx setup hosts pages from `/var/www/html`, but `aa-genprof` permits only the configuration file path itself (`/usr/local/nginx/conf/nginx.conf`), and not paths defined within that file, because they are only accessed after a client request is received. Figure 7 shows `aa-genprof` grants access to nothing under `/var` yet always includes `/usr`, whereas Ample allows fewer than ten targeted `/var` entries. These results highlight why `aa-genprof` is unsuitable for production policies.

C. Ablation Study

To evaluate the contribution of Ample's static and dynamic analyses, we performed an ablation study deriving policies for each phase separately. Since dynamic analysis depends on instrumentation inserted by the static phase, static analysis cannot be fully disabled; in the dynamic-only setting, however, we exclude purely static results. Figure 6 shows that the two phases share many discoverable paths, such as constant strings

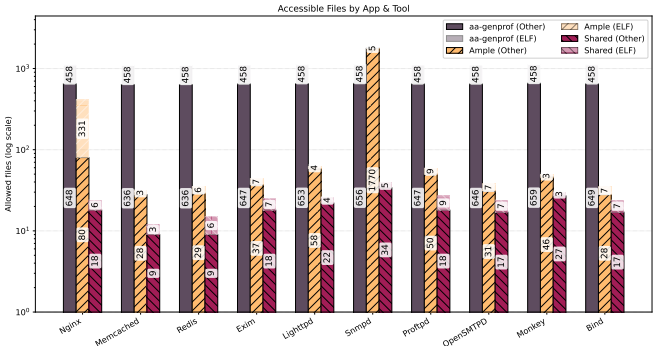

Fig. 5. Number of total and ELF files accessible after applying Ample, `aa-genprof`, and both.

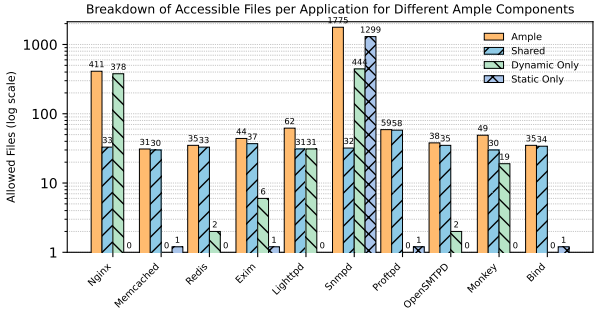

Fig. 6. Breakdown of how static analysis and dynamic contribute to deriving policies in Ample.

embedded in code. Such strings are statically observable and also recorded at runtime when written to. Also, libraries and their strings are considered in the "shared" policies. Notice how dynamic analysis contributes unique environment-dependent paths otherwise not captured statically. For example, when setting Monkey's runtime configuration directory to `/tmp/monkey/conf`, Ample identified it during initialization and generalized it with a wildcard as `/tmp/monkey/conf/*` which captured 16 of the 19 files unique to dynamic analysis. Multiple runtime-only APIs may also operate on the same file, as when Nginx both opens and deletes a cache file (e.g., `open` vs. `unlink`). Our analysis correctly distinguishes such cases explaining why runtime counts in Table I do not equal dynamic-only paths counts.

D. Policy Correctness

To ensure that Ample's policies do not break the program, we validate each application by testing its execution under the inferred policies, thereby checking for false negatives.

Our tests should be similar to a real-world environment with workloads exploring different features and functionalities. Simply running each program with default runtime settings would not explore different code paths and their respective features, limiting the trustworthiness of our evaluation. For example, running Nginx using the default runtime settings provided with the source code does not enable SSL encryption.

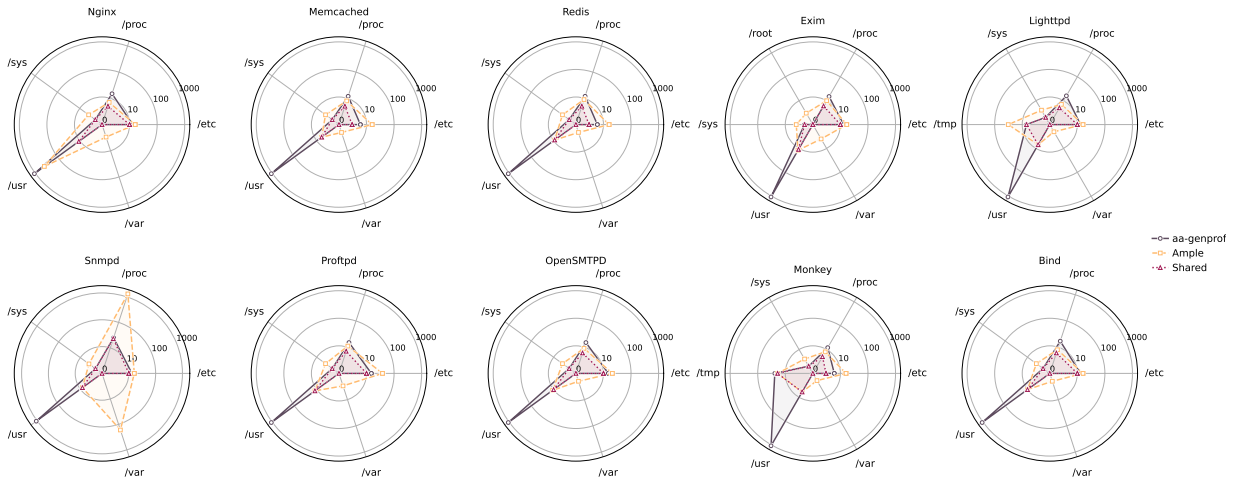

Fig. 7. Number of files accessible (on logarithmic scale) after applying Ample and `aa-genprof`, grouped by top-level directory in the Ubuntu distribution.

Therefore, for each program we collected a ready-to-use Docker image which contained a fully configured instance of the target program. This ensured we were not testing the program under non-realistic settings. Furthermore, for each program, we utilized specialized benchmarking tools tailored to its functionality. Nginx and Memcached were assessed using the Cloudsuite benchmarks [44], [3], while Redis was evaluated using the `redis-benchmark` tool [12] Lighttpd was tested using the `iipsrv` Docker image [7] which launches the IIPImage server (a high performance image server) on top of a fully configured Lighttpd instance. We tested Proftpd using the `instantlinux/proftpd` image [9], and Bind with the `conceptant/bind` Docker image [4] in conjunction with `dnsperf` [6]. Similarly, Exim and Opensmtpd were tested using the `imixs/exim4` [8] and `wodby` [14] images, respectively, both being stress tested by `xstress` [15].

For Monkey and Snmpd, no sufficiently configured Docker images were available. However, we ensured comprehensive testing by directly deploying and stress-testing these servers under various loads. This alternative approach maintained consistency in our evaluation strategy while ensuring robust validation of the policies in realistic deployment scenarios.

To evaluate our security policies, we applied the derived policy to the entire Docker container using the `--security-opts apparmor` option. If the setup process of any Docker image required additional files beyond those needed by the target program itself, we manually included them to ensure the container's correct initialization. Notably, our generated policy successfully passed all tests across all programs without any false negatives, underscoring the correctness of our approach.

E. Performance

Ample increases the compilation time of a program due to its static analysis and increases the bitcode size due to its instrumentation. We do not report the spatial overhead given it is minimal (less than 10%), and that the instrumented binary is used only to craft the AppArmor profile, after which the original, uninstrumented executable runs. In this section we report the compile-time overhead of running Ample, and the runtime overhead of enforcing AppArmor. Previous work [63] already quantified AppArmor's launch-time overhead, which does not impact request handling, so we omit that measurement here.

a) Compile-time Overhead: Ample incurs compile-time overhead due to performing static analysis. The major bottleneck in Ample's analysis is performing pointer analysis on the target program. While most programs finish in less than one hour, Bind is the only exception which takes close to ten hours. Our overhead is on par with previous works that use Andersen's [27], [28], [62]. This can be reduced by using other pointer analysis algorithms such as Steensgaard's [52] which have lower algorithmic complexity, but also less precision.

b) Runtime Overhead: We use lmbench [41] to measure the overhead of opening and closing a file with and without AppArmor enabled. To measure whether the policy file size affects the overhead we leverage Ample's largest policy in our evaluation. To do so, we use the `lat_syscall.c` benchmark along with a temporary file path and run it 100 times for three scenarios: 1) AppArmor disabled; 2) AppArmor enabled with `aa-genprof`'s Nginx policy; and 3) AppArmor enabled with Ample's Nginx policy. Figure 8 shows that AppArmor enforcement adds no measurable overhead—opening and closing a temporary file averaged 1.19µs both with and without policies from either `aa-genprof` or Ample. The same holds for `stat` and `fstat`. Thus, Ample's finer-grained policies significantly reduce file exposure without hurting performance.

F. Security Evaluation

We have evaluated the Ample's security benefit by considering both user-level attacks and kernel vulnerabilities.

a) Hindering User-level Attacks: User-level attacks are usually conducted by chaining together sequences of instructions to acquire arbitrary code execution capabilities—most

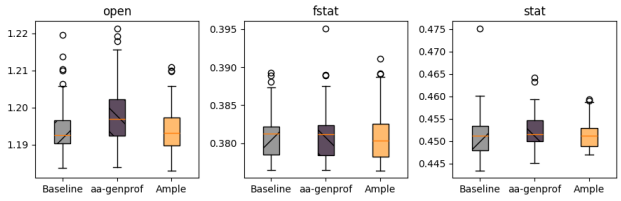

Fig. 8. Execution time of three file-related syscalls for baseline (without any AppArmor policy), with policy generated by `aa-genprof`, and Ample.

simply by invoking `execve("/bin/bash")`. To prevent this, we need to restrict *all* means of executing unauthorized binaries by the attacker. Previous works [27], [28], [48] defend against this attack by filtering the `execve` system call in its entirety. Yet many programs legitimately use this system call (i.e., cannot be filtered), and even in cases where it can be filtered there remain other functionalities that provide the same capability (e.g., cronjobs). Ample protects against these attacks by restricting access to 1) implicit code execution features; and 2) binaries available on the host.

For the first category, we specifically focus on cron jobs, which schedule the execution of programs at intervals by modifying files under the `/etc/cron.*` directories. An attacker could cause arbitrary code execution by modifying the contents of these folders, and therefore, it is crucial to protect against their access. Ample's policies for the ten applications in our dataset restrict access to *all* files under this directory. Although the default Linux permissions block unprivileged modifications of this directory, programs launched as root—e.g., proftpd—override these safeguards (see Section II).

For the second category, denying access to executables is more effective than syscall filtering: even if an attacker chains benign system calls (e.g., `open`/`read`/`mmap`) to mimic `execve`, an unreadable binary still cannot run, eliminating whole exploit classes that syscall filters may overlook. We therefore compared binaries left reachable by `aa-genprof` and by Ample (Figure 5). Ample cuts the set to less than 10 ELF files for nine of ten programs, whereas `aa-genprof` still permits 458—about half the binaries on a vanilla Ubuntu system. The sole exception is Nginx, where Ample still leaves 331 binaries reachable (yet fewer than `aa-genprof`). Here Ample falls back to pointer analysis, which overapproximates and consequently grants access to every binary listed in `$PATH`.

b) Mitigating Linux Kernel Vulnerabilities: We measure the security benefits of restricting file access at the kernel level by examining two principal classes of kernel vulnerabilities: those enabling privilege escalation from read to write on critical files (e.g., /etc/passwd), and those triggered solely by file access. By analyzing how Ample mitigates both categories, we quantify its overall impact.

Dirty COW [16] and DirtyCred [37] are examples of kernel vulnerabilities that allow non-privileged users to escalate their read permissions to write access on security-critical files. To evaluate Ample's ability to prevent these attacks, we enumerated every root-owned, world-readable file on a default Ubuntu 24.04 (20K files) and tallied how many each policy still allows. With Ample, eight programs reach less than 22 of these files, Proftpd 42, and Snmpd 1325; `aa-genprof` leaves 1040 reachable across all programs.

TABLE II
FILE-TRIGGERED LINUX KERNEL CVES BLOCKED BY AMPLE. "COUNT" = PROTECTED APPS; BASELINE HAS NO APPARMOR POLICY.

CVE	File Path	Tool Block Count		
		Base.	aa-gen.	Ample
CVE-2017-16539	/proc/scsi/scsi	0	10†	9
CVE-2017-5967	/proc/timer_list	10	10	10
CVE-2024-56678	/proc/kcore	10	10	10
CVE-2018-11964	/etc/passwd	0	3	5
CVE-2005-1041	/proc/net/route	0	3	9
CVE-2018-18955	/etc/shadow	10	10	10
CVE-2006-3626	/proc/self/environ	0	10†	9
CVE-2015-8944	/proc/iomem	0	10†	9
CVE-2016-7042	/proc/keys	0	10†	9
CVE-2010-2243	/sys/devices/syst...*	0	10	10
CVE-2009-1243	/proc/net/udp	0	10†	9
CVE-2022-48757	/proc/net/ptype	0	10†	9
CVE-2019-11811	/proc/ioports	0	10†	9

*/sys/devices/system/clocksource/clocksource0/current_clocksource
†aa-genprof outperforms Ample in Snmpd due to missed paths in `/proc`

To measure Ample's impact on kernel-space CVEs, we targeted only vulnerabilities whose exploits require file access. From the 8000 Linux-kernel CVEs in the official database [5], we used the QWEN 2.5-14B LLM to flag entries whose text suggests file interaction, trimming the set to 82. Manual vetting confirmed 62 file-dependent CVEs and extracted their path names. Because most paths lie in read-only trees such as `/proc`, we could only create 13 of them; which our analysis therefore focuses on. As Table II shows, Ample blocks every unused path without breaking execution, outperforming—or matching—`aa-genprof` in all cases except Snmpd, where `aa-genprof` omits the relevant `/proc` entries, and can break the program (as discussed in Section V-B).

VI. THREATS TO VALIDITY

a) Internal Validity: The accuracy of our results depends on the precision of static analysis and correctness of instrumentation. Static analysis may suffer from imprecision due to limitations in alias analysis, which could lead to overapproximation of accessible file paths. Similarly, instrumentation might fail to capture cases where initialization points cannot be clearly separated from subsequent execution phases.

b) External Validity: Our approach assumes that prefixes of runtime-dependent file paths are generated during initialization, which is sufficient for later-constructed paths as long as their prefixes are known. Programs that build complete paths entirely after initialization may not be fully covered. Therefore, our approach is scoped to server applications that follow the initialization/processing model, and the results may not generalize to other classes of programs.

VII. Discussion & Limitations

Ample narrows the attack surface and enforces PoLP. Though it cannot stop all exploits, it adds a lightweight layer that constrains attacker capabilities atop other defenses.

a) Security vs. Functionality Tradeoff: Our current design prioritizes soundness over completeness, occasionally yielding less restrictive policies to preserve functionality. In scenarios where stronger security is required and some functionality loss is acceptable, Ample can instead mark major overapproximated directories or paths (e.g., those arising from `opendir`) with the `audit` keyword. This enables advanced users to inspect audit logs for these cases and refine the policies as needed, providing a more flexible balance between security and functionality.

b) Soundness Limitations: We do not formally establish the soundness of the inferred policies and therefore do not provide theoretical guarantees. Rather, we provide empirical evidence by applying our technique to widely deployed software systems. As shown in Section V-B, AMPLE eliminates multiple sources of unsoundness of other tools in two major applications `Nginx` and `Snmpd`.

c) Broader Context of Access Control Policies: Access control policies are typically obtained either through explicit specification [51], [31], [33] or policy mining [23], [56], [60]. Manual specification is flexible but does not scale and often misses environment paths. Policy mining automates part of the process but relies on historical data or existing models and still demands developer input. In contrast, our hybrid program analysis approach offers a scalable middle ground. Static analysis captures developer-known paths, while limited dynamic analysis recovers environment paths that cannot be extracted statically. Together, these components allow Ample to infer restrictive AppArmor policies with minimal effort, overcoming limitations of manual and purely dynamic methods, making it a practical candidate for securing server applications.

d) Program Scope: Ample targets two-phase server applications only. Client tools like `ls` take user-specified paths at runtime, so program analysis cannot be used to infer their file needs; policies for these programs must be set by administrators. Hence, client applications are out of scope.

e) Transition Point: Similar to previous works [27], [28], the developer needs to identify the transition point between the initialization and the processing phase. More recent work [48], identify the transition point automatically, by analyzing the main loop for handling client requests. However, these would be difficult to generalize automatically for all programs.

f) Runtime Setting Modification: Although some programs (e.g., Apache) allow adjustments to runtime settings mid-execution—which may necessitate re-running the instrumented application—this feature may not be needed in production environments that rely on preconfigured containers. Hence, we believe its impact on Ample's usability is minimal.

VIII. Related Works

A. Attack Surface Reduction

Attack surface reduction through removing code [47], [19], [32], [45], [21], [25] and features [46] is a defense-in-depth approach which reduces the risk associated with attacks.

Confine [26], [49], Sysfilter [24], and Chestnut [22] filters entire system calls, while SPEAKER [35], Temporal Specialization [27], and Syspart [48] present phase-aware system call filtering. C2C [28] proposes configuration-drive syscall filters. More recent works focus on argument-level system call filtering [61], [59], [49], [42], [43], but only target primitives or flag types (e.g., `int`). Ample, on the other hand, extracts fine-grained policies based on the file path arguments used across all system calls. Deprivileging programs through removing their excessive privileges (i.e., Linux capabilities) can also be enforced through AppArmor policies. Decap [29] and LiCA [55] create mappings between system calls and capabilities, removing unneeded capabilities by analyzing their required system calls. We consider capabilities out of scope.

B. Automatic Policy Inference

Due to the challenges of generating fine-grained access control policies statically, most prior work mainly rely on profiling and dynamic analysis to generate new policies.

CASPR [58] assumes file access policies are available and maps them to minimal SELinux subject and object tags to simplify their application. EASEAndroid [57] generates SEAndroid policies for Android using semi-supervised learning. LiCShield [40] traces the execution of containers and generates AppArmor policies through logging resource accesses. Docker-Sec [38] builds an initial static profile based on container configuration parameters, and then further expands it through profiling. Lic-Sec [63] combines Docker-Sec and LiCShield to build policies which restrict resources supported by both of the previously available tools. Furthermore, Kub-Sec [64] integrates Lic-Sec into the Kubernetes infrastructure, streamlining the automatic generation of AppArmor policies for pods (containers) running in a Kubernetes cluster. ASPGen [36], generates AppArmor policies using a *strictly* dynamic analysis-based approach which suffers from unsoundness due to the inherent limitations of dynamic analysis. Also, complementary studies assess the security of existing Android policies without generating new policies [30], [34].

IX. Conclusion

We presented Ample, an automated tool for generating restrictive AppArmor policies using a novel hybrid static and dynamic analysis technique. Ample uses static analysis to extract concrete values from the program source code, and instruments the program to extract runtime-initialized file path values as well. Reducing the number of files that a program can access, removes them from the attack surface and prevents any potential attacker from taking advantage of them. We showed Ample's effectiveness by applying it to ten widely-used server applications and generating restrictive AppArmor policies that on average restrict each program to less than 254 files.

References

[1] aa-easyprof manpage. https://manpages.ubuntu.com/manpages/noble/man8/aa-easyprof.8.html.

[2] AppArmor Linux kernel security module. https://apparmor.net.

[3] Cloudsuite. https://www.cloudsuite.ch.

[4] conceptant/bind. https://hub.docker.com/r/conceptant/bind.

[5] Cve project. https://github.com/CVEProject/cvelistV5.

[6] dnsperf—dns-oarc. https://www.dns-oarc.net/tools/dnsperf.

[7] iipsrv/iipsrv. https://hub.docker.com/r/iipsrv/iipsrv.

[8] imixs/exim4. https://hub.docker.com/r/imixs/exim4.

[9] instantlinux/proftpd. https://hub.docker.com/r/instantlinux/proftpd.

[10] Landlock: unprivileged access control. https://docs.kernel.org/userspace-api/landlock.html.

[11] The often misunderstood GEP instruction. https://llvm.org/docs/GetElementPtr.html.

[12] Redis benchmark. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/.

[13] SELinux. https://www.selinuxproject.org/page/Main_Page.

[14] wodby/opensmtpd. https://hub.docker.com/r/wodby/opensmtpd.

[15] xstress version 0.40. https://github.com/xk0der/xstress.

[16] Cve-2016-5195. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-5195, 2016.

[17] Ubuntu manpages - aa-genprof. https://manpages.ubuntu.com/manpages/focal/man8/aa-genprof.8.html, 2019.

[18] Cve-2022-0847. https://nvd.nist.gov/vuln/detail/CVE-2022-0847, 2022.

[19] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 70–83, 2019.

[20] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[21] Priyam Biswas, Nathan Burow, and Mathias Payer. Code specialization through dynamic feature observation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, pages 257–268, 2021.

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

[23] Edward J Coyne. Role engineering. In *Proceedings of the first ACM Workshop on Role-based access control*, pages 4–es, 1996.

[24] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[25] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1009–1022, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 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)*, 2020.

[27] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[28] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. C2C: Fine-grained configuration-driven system call filtering. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1243–1257, 2022.

[29] Md Mehedi Hasan, Seyedhamed Ghavamnia, and Michalis Polychronakis. Decap: Deprivileging programs by reducing their capabilities. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, page 395–408, 2022.

[30] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. BigMAC: Fine-Grained policy analysis of android firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 271–287. USENIX Association, August 2020.

[31] Vincent C Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations. *NIST special publication*, 800(162):1–54, 2014.

[32] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, 2019.

[33] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[34] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, Giuseppe Petracca, and Trent Jaeger. PolyScope: Multi-Policy access control analysis to compute authorized attack operations in android systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2579–2596. USENIX Association, August 2021.

[35] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-phase execution of application containers. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 230–251, 2017.

[36] Yun Li, Chenlin Huang, Lu Yuan, Yan Ding, and Hua Cheng. Aspgen: an automatic security policy generating framework for apparmor. In *Proceedings of the 2020 IEEE International Conference on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 392–400, 2020.

[37] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1963–1976. Association for Computing Machinery, 2022.

[38] Fotis Loukidis-Andreou, Ioannis Giannakopoulos, Katerina Doka, and Nectarios Koziris. Docker-sec: A fully automated container security enhancement mechanism. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 1561–1564, 2018.

[39] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1867–1881, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Massimiliano Mattetti, Alexandra Shulman-Peleg, Yair Allouche, Antonio Corradi, Shlomi Dolev, and Luca Foschini. Securing the infrastructure and the workloads of linux containers. In *Proceedings of the 3rd IEEE Conference on Communications and Network Security (CNS)*, pages 559–567, 2015.

[41] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *Proceedings of the 2nd USENIX annual technical conference (ATC)*, pages 279–294. San Diego, CA, USA, 1996.

[42] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking exploits through API specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[43] Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization against code reuse attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[44] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 122–132. IEEE, 2016.

[45] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. Razor: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1733–1750. USENIX Association, 2019.

[46] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. Slimium: debloating the chromium browser with feature subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 461–476, 2020.

[47] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, pages 869–886, 2018.

[48] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. Syspart: Automated temporal system call filtering for binaries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1979–1993, 2023.

[49] Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis. Confine: Fine-grained system call filtering for container attack surface reduction. *Computers & Security*, page 103325, 2023.

[50] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[51] Ravi S Sandhu. Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier, 1998.

[52] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[53] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[54] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[55] Menghan Sun, Zirui Song, Xiaoxi Ren, Daoyuan Wu, and Kehuan Zhang. Lica: A fine-grained and path-sensitive linux capability analysis framework. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, page 364–379, 2022.

[56] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 175–184, 2007.

[57] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M Azab. {EASEAndroid}: Automatic policy analysis and refinement for security enhanced android via {Large-Scale}{Semi-Supervised} learning. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, pages 351–366, 2015.

[58] Lifang Xiao, Hanyu Wang, Aimin Yu, Lixin Zhao, and Dan Meng. Caspr: Context-aware security policy recommendation. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS)*, volume 2025, 2025.

[59] Yunlong Xing, Jiahao Cao, Kun Sun, Fei Yan, and Shengye Wan. The devil is in the detail: Generating system call whitelist for linux seccomp. *Future Generation Computer Systems*, 135:105–113, 2022.

[60] Zhongyuan Xu and Scott D Stoller. Mining attribute-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 12(5):533–545, 2014.

[61] Dongyang Zhan, Zhaofeng Yu, Xiangzhan Yu, Hongli Zhang, Lin Ye, and Likun Liu. Securing operating systems through fine-grained kernel access limitation for iot systems. *IEEE Internet of Things Journal*, 2022.

[62] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System call-based state-aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, 2022.

[63] Hui Zhu and Christian Gehrmann. Lic-sec: an enhanced apparmor docker security profile generator. *Journal of Information Security and Applications*, 61:102924, 2021.

[64] Hui Zhu and Christian Gehrmann. Kub-sec, an automatic kubernetes cluster apparmor profile generation engine. In *Proceedings of the 14th International Conference on COMmunication Systems NETworkS (COMSNETS)*, pages 129–137, 2022.