Transparent DIFC: Harnessing Innate Application Event Logging for Fine-Grained Decentralized Information Flow Control

Jason Liu
University of Illinois at Urbana-Champaign
Champaign, USA
jdliu2@illinois.edu

Anant Kandikuppa
University of Illinois at Urbana-Champaign
Champaign, USA
anantk3@illinois.edu

Adam Bates
University of Illinois at Urbana-Champaign
Champaign, USA
batesa@illinois.edu

***Abstract*—Information flow control is a canonical approach to access control in systems, allowing administrators to assure confidentiality and integrity through restricting the flow of data. Decentralized Information Flow Control (DIFC) harnesses application-layer semantics to allow more precise and accurate mediation of data. Unfortunately, past approaches to DIFC have depended on dedicated instrumentation efforts or developer buy-in. Thus, while DIFC has existed for decades, it has seen little-to-no adoption in commodity systems; the requirement for complete redesign or retrofitting of programs has proven too high a barrier.**

In this work, we make the surprising observation that developers have already unwittingly performed the instrumentation efforts required for DIFC — application event logging, a software development best practice used for telemetry and debugging, often contains the information needed to identify application-layer event processes that DIFC mediates. We present T-DIFC, a kernel-layer reference monitor framework that leverages the insights of application event logs to perform precise decentralized flow control. T-DIFC identifies and extracts these application events as they are created by monitoring application I/O to log files, then references an administrator-specified security policy to assign data labels and mediate the flow of data through the system. To our knowledge, T-DIFC is the first approach to DIFC that does not require developer support or custom instrumentation. In a survey of 15 popular open source applications, we demonstrate that T-DIFC works seamlessly on a variety of popular open source programs while imposing negligible runtime overhead on realistic policies and workloads. Thus, T-DIFC demonstrates a transparent and non-invasive path forward for the dissemination of decentralized information flow controls.

1. Introduction

Modern applications are growing increasingly complex, to the point where security vulnerabilities are inevitable despite large, dedicated security development teams [40]. While identifying and patching such vulnerabilities are important, access control plays an important role in mitigating the harmful effects of these vulnerabilities by restricting the resources available to a compromised application. In particular, the notion of Information Flow Control (IFC) has received considerable attention over the years [5], [6], [35], [9], [2], [3], [23], [10]. In an IFC system, administrators can enforce confidentiality or integrity guarantees by restricting the flow of sensitive data. Today, IFC is supported by a variety of popular security models, including SELinux [24], [19], [17]. A key feature of such frameworks is that security policies can be specified by system administrators *without* explicit support from developers or software vendors. This is possible because the policies describe generic system-level abstractions that can be mediated at the operating system interface.

However, the generalizability of traditional IFC comes at a cost — because the protection state is defined over system objects, the security model is oblivious to application-layer semantics and is unable to mediate accesses at finer granularities. IFC views each program as an opaque black box, making it impossible to reason about higher-level abstractions such as website user accounts or key-value pairs in a datastore. For example, consider a web server that uses an event handling loop to accept new connections. Each time a new client connects, they authenticate to the web application using a site-specific user account and are granted access to client-specific resources on the machine. However, from the perspective of IFC, the protection domain of the process remains unchanged regardless of which client is connected. As a result, if the application is compromised, the attacker has immediate access to all clients' data, creating a lucrative opportunity for data exfiltration [33].

Decades after the advent of information flow control, Decentralized Information Flow Control (DIFC) [30] was proposed as a means of addressing this shortcoming. In a DIFC system, programs can emit their own additional labels that can be appended to the security context of data objects, allowing individual programs to specify a finer-grained data flow policy. Such an approach is "decentralized" because it enables many stakeholders in the system to contribute to the policy; further, since these additional labels only further restrict the protection state of the system, it is possible to achieve classic information flow control assurances at the operating system layer while still enjoying the fine-grained protections of DIFC. Initial implementations of DIFC restricted information flow at the language level [30], [32], with proposals for OS-level support following shortly thereafter [8], [48], [21], [34].

Unfortunately, despite its ability to produce expressive access control policies, DIFC's adoption in real systems remains extremely limited. Why hasn't DIFC become a prominent approach to securing complex systems? A major factor limiting the proliferation of DIFC is the tremendous development costs of its adoption — to date, all DIFC implementations have assumed that programs are rewritten or instrumented in order to add and remove labels from data objects. If programs are working in concert to achieve more complex DIFC policies, this requires multiple independent development teams to coordinate these changes, to say nothing of the modifications required to the operating system or language runtime.

We argue that this burden has prevented the further proliferation of DIFC technologies; thus, a DIFC approach that was fully *transparent* to the applications it mediates would be highly useful. However, the design of such a system poses several obstacles: how can an application emit custom security labels or declassify data without changes to source code, and how could the operating system detect that the program has changed context to avoid overpropagating existing labels (i.e., taint explosion)?

In this work, we make the surprising observation that developers have already provided cues to the active security context of their applications in the form of application event logs. For example, consider a server that is receiving a new request from client u. Common best practices dictate that the developer should create a record of this event to facilitate debugging and fault detection. This log event is likely to contain the client name u, which is precisely the information needed to meet classic DIFC goals — while handling a request from client u, the server should propagate the label u to data it writes to, and it should not be able to read data from non-u clients. Further, when a log event marks the handling of a new request for client v, this could serve as a cue to the DIFC system to stop propagating label u and to start propagating v. Put another way, the information needed to support *event processes* [8], a method of compartmentalizing application data flows that is a prerequisite to performing DIFC at the operating system layer, is already present in the application event logs [14].

Building on these insights, this work presents the design of T-DIFC, an operating system extension that permits administrators to transparently define DIFC policies based on applications' innate logging behaviors. To operate T-DIFC, administrators define a simple security policy through inspecting the log events of target applications. At runtime, T-DIFC interposes on `write` events associated with applications logs and is activated if a new event trigger matches a regular expression in the security policy. On a match, T-DIFC updates security labels based on a classic DIFC tagging model. Finally, T-DIFC mediates accesses to labeled data to enforce the administrator's security goals.

Our contributions can thus be summarized as follows:

Transparent Decentralized Information Flow. We present the design of a novel DIFC operating system extension that leverages innate application logging instead of custom instrumentation. To our knowledge, T-DIFC is the first DIFC system to boast *transparent* specification and enforcement of application-layer information flows. We implement T-DIFC as a stackable Linux Security Module that is compatible with SELinux, further simplifying its operation.

No-Hassle Policy Specification. Through a series of application case studies, we demonstrate the ease with which an administrator can specify a T-DIFC policy without developer cooperation. For example, we provide a policy for ProFTPD that enforces access control for arbitrarily many application-level users stored in LDAP or SQL databases, which is otherwise infeasible with standard UNIX access control primitives.

Efficient Mediation of Information Flow. We exhaustively evaluate the performance of T-DIFC through standard operating system benchmarks and representative application workloads. When benchmarking the performance of ProFTPD with a realistic and useful security policy loaded, we were unable to observe *any* overhead of T-DIFC under a variety of different workloads. Such a feat is possible due to the efficient design of T-DIFC's label propagation mechanism, as well as the infrequency with which new application event messages trigger a change in security context (relative to other application activity).

2. DIFC Preliminaries

In traditional IFC, a reference monitor mediates data flow according to a pre-defined confidentiality or integrity lattice of security classes. The IFC policy restricts flow of information from higher-level security class to lower-level security class (confidentiality), or vice versa (integrity). However, traditional IFC policies do not consider the semantics of data accesses by the underlying programs. For example, consider a web server that uses an event handling loop to accept new connections. Each time a new client connects, they authenticate to the web application using a site-specific user account and are granted access to client-specific resources on the machine. However, from the perspective of IFC, the protection domain of the process remains unchanged regardless of which client is connected. Thus, a compromised application grants the attacker immediate access to all clients' data [33].

DIFC addresses this limitation by allowing each program to specify its own finer-grained data flow policy by creating additional security labels for data handling within the program. The reference monitor then enforces all applications' flow policies as information propagates through the system. Myers and Liskov's original formalization of DIFC [30] defines the label of each data object as a set of owner-readers pairs. Each owner of the data could specify a set of allowed readers. For example, let L represent labels, $\mathrm{O}(L)$ represent the set of all owners in L, and $\mathrm{R}(L, o)$ represent the readers allowed by owner o in L.

Let us consider a confidentiality lattice for this original DIFC formalization. Data flows from label L_1 to L_2 require that L_1 is at least as permissive as L_2 ($L_1 \sqsubseteq L_2$); if not, L_2 could either be updated to be less permissive, or the flow could be rejected. To be as permissive, L_2 must at least have the all the owners in L_1, and each owner in L_2 cannot designate extra readers relative to L_1. This *restriction* relationship is defined as

$$\begin{aligned} L_1 \sqsubseteq L_2 \iff & \mathrm{O}(L_1) \subseteq \mathrm{O}(L_2) \wedge \\ & \forall o \in \mathrm{O}(L_1) : \mathrm{R}(L_1, o) \supseteq \mathrm{R}(L_2, o) \end{aligned}$$

If the data flow is allowed, we must also ensure that L_2 reflects the potential new owners in L_1. This can be achieved by setting the label L_2 to be the *join*, or least upper bound, of labels L_1 and L_2, $L_1 \sqcup L_2$. The join is the lowest-level label that is at least as permissive as both L_1 and L_2; i.e., $L_1 \sqsubseteq L_1 \sqcup L_2$ and $L_2 \sqsubseteq L_1 \sqcup L_2$. The join is defined as

$$\mathrm{O}(L_1 \sqcup L_2) = \mathrm{O}(L_1) \cup \mathrm{O}(L_2)$$
$$\mathrm{R}(L_1 \sqcup L_2, o) = \mathrm{R}(L_1, o) \cap \mathrm{R}(L_2, o)$$

3. Motivation

Of course, in order for the above formulation to work, it is necessary for applications to participate in the DIFC protocol. As OS-based approaches to DIFC began to emerge, a central challenge became reconciling operating system abstractions with the notion of finer-than-process principals. Efstathopoulos et al.'s Asbestos system presented a solution in the form of "Event Processes" [8], which decompose a long-lived traditional processes into a set of autonomous sequential subprocesses.[1] By associating security labels with event processes, rather than processes, it became possible for the reference monitor to track information flow more precisely and avoid problems with overtainting. While Asbestos defined a custom syscall interface that was incompatible with existing systems, later systems (e.g., HiStar [48], Flume [21]) adapt these ideas to UNIX environments, allowing the system to run existing programs and utilities.

However, while event processes solve the problem of finer-than-process flow tracing, they have traditionally exacerbated of invasiveness of program instrumentation required by DIFC. In addition to defining a security policy and performing runtime labeling, application developers also needed to explicitly identify event processes in source code and insert the appropriate logic. Thus, while OS-based approaches made it possible for DIFC-aware programs to interact with traditional UNIX programs, the difficulty of creating DIFC-aware programs remained comparable to language-based approaches.

The present study is motivated by the observation that, often, a dedicated instrumentation effort is not necessary to extract DIFC security context from programs. For example, consider the MoinMoin Wiki service that is instrumented by Krohn et al. in evaluation of the Flume system [21]. The "flumed" version of MoinMoin defines an extended security policy that assures integrity against application plug-ins. To achieve this, the authors treat `httpd` as an endpoint where all data is declassified and wrap `wiki.py` in a custom security module that retrieves the client's security context from the reference monitor and applies labels to the core application logic.

Interestingly, however, the vanilla version of MoinMoin has its own way of tracking a client's context, shown in the following application log event:

```
[Wed Sep 23 11:59:59.061504 2022]...
  [pid 66913:tid 139754678122240]...
   MoinMoin.util.abuse INFO :...
   auth/login (moin): status success:
   username "Alice": ip 127.0.0.1...
```

Here, it can see that MoinMoin has recorded the successful login attempt for a subject with username Alice at `127.0.0.1`, and that Alice's session is being managed by process id `66913` on thread `139754678122240`. We make several observations about this log event. First, MoinMoin has implicitly defined a security context for this web session (username Alice) and associated with a system process (pid 66913). Second, whenever an application event of this format appears, it implicitly marks the boundary of a new event process in the MoinMoin service; this is evidenced by the fact that application event logs can be used to retroactively perform execution partitioning of system log events [14]. Third, because MoinMoin writes this application event to disk via a system call, it is actually the case that MoinMoin is already implicitly communicating its current security context to the reference monitor (i.e., the OS kernel). Fourth, we can configure `httpd` in pre-forked mode to prevent server threads from handling multiple clients' data. Finally, given that the reference monitor already knows the application's security context and associated process identity, it is not necessary for MoinMoin to have a dedicated security module because the reference monitor can perform the labeling itself.

In the present study, we attempt to operationalize these insights in the form of a transparent OS-based DIFC system. While it would appear that the necessary security context is innately exposed by MoinMoin, several challenges stand in the way of harnessing this information: *How can we identify and extract security contexts from innate application log events?*, *How can a reference monitor simulate DIFC labeling and event process decomposition without the cooperation of the application?*, *How can a reference monitor access the information contained in application log events?*, and *How can a machine operator define a DIFC security policy without the cooperation of the application developers?* In what follows, we explore the answers to these questions.

4. Threat Model

This work considers an adversary that is attempting to access system resources through a network-facing program. The program handles data from many clients, but the distinction between these clients and data objects is opaque to the operating system. Thus, when considering application semantics, the system-layer security policy permits the program to run in an overprivileged state.

We make the following assumptions about this environment. First, we assume the kernel and associated system utilities are correct and cannot be compromised by the attacker. This is a standard assumption for operating system access control (e.g., Linux Security Modules [7]) and is made more reasonable using known platform hardening techniques [19]. Further, we also make a standard assumption from the DIFC literature, namely, that the runtime integrity of the target application is also assured. This is necessary because programs can specify their own security policies in DIFC models; if the target application is compromised, it logically follows that any policies it is allowed to write to are also compromised. However, this does not rule the possibility that the application contains vulnerabilities that permit the attacker to trick the program into acting on their behalf (e.g., command injection). Further, it is not necessary for the target application to trust other programs; therefore, other applications are still

1. The notion of event processes has re-emerged more recently in the system auditing literature, where process activities are decomposed via various "execution partitioning" strategies, e.g., [22], [25], [14], [26].

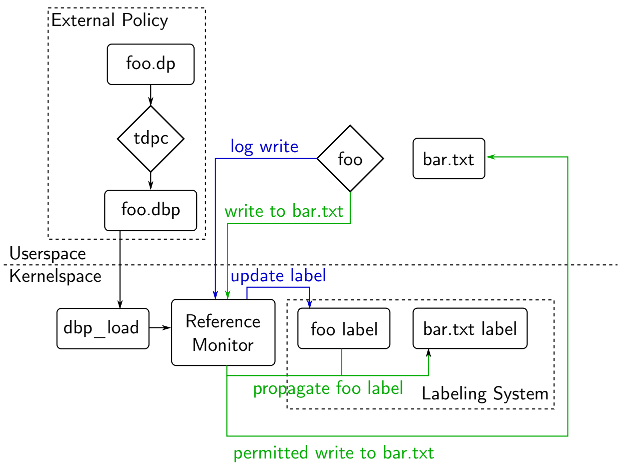


Figure 1: Overview of T-DIFC's design. External policies (e.g., for program `foo`) are created in userspace, compiled with `tdpc`, the T-DIFC policy compiler, and loaded into the kernel at runtime via the `dbp_load` interface. The reference monitor updates process labels on log writes (shown in blue) and allows or denies information flows according to external policies, propagating labels on allowed flows (shown in green).

permitted to be under full attacker control. Lastly, and as discussed in Section 8.1, we assume that the target application is emitting standard event logs that describe its activities.

5. Design

Our design is driven by the following goals:

Decentralized information flow control. Our solution must be able to express classical decentralized information flow guarantees (Sec. 2) and ensure their enforcement at the operating system layer.

Finer-than-process granularity. While DIFC does not strictly require event processes, issues of overtainting and taint explosion quickly emerge without it. To fully enable the advantages of DIFC beyond what is possible with non-decentralized flow control, our solution must be able to track information flows at a finer-than-process granularity.

Transparency. Unlike existing DIFC systems, our solution must be deployable on a target application without instrumentation or the developers' assistance. Specifically, our solution must permit a machine operator to specify external DIFC policies on behalf of the target application.

Reference monitor guarantees. Our solution must reasonably satisfy the reference monitor concept [1]. In other words, our solution's enforcement mechanism must be tamperproof, be able to offer complete mediation of relevant system activities, and be verifiable.

5.1. Overview

We present T-DIFC, a transparent approach to decentralized information flow control within operating systems. An overview of T-DIFC is displayed in Figure 1. T-DIFC consists of three major components: the kernelspace labeling system, the kernelspace reference monitor, and the userspace external policy language and compiler.

Labeling system. This component maintains the labels, comprised of a set of tags, associated with data on the system. While the semantics of tags ultimately depend on their logical use according to external policies, tags generally represent the sources of data used by programs. All objects in the system that manipulate data require a corresponding tag; this includes both processes and files.

Reference monitor. The T-DIFC reference monitor is responsible for mediating information flows by propagating labels and performing enforcement checks as specified by the security policy. Whenever an information flow occurs between two objects, the reference monitor will either permit or deny the flow depending on the types of object (files, processes, sockets), the labels of the object, and the relevant policies statements. When a flow is permitted, the reference monitor propagates any labels carried by source of the flow to its destination.

External policy language and compiler. The external policy language allows machine operators to specify DIFC policies for target applications without requiring modifications to program source code. External policies specify which logs the labeling system should intercept to modify a process's labeling state, and which information flows involving the policy's tags the reference monitor should permit. These policies can then be compiled in userspace before being loaded by T-DIFC.

5.2. Labeling System

We adapt the labeling system introduced by Flume [21]. *Tags* are basic primitives used to denote ownership or information sources of data. The set of all tags, or the *label*, associated with some data indicates all sources of this data. To control where information may flow, application policies control which *capabilities* associated with each tag other applications may possess. For tag t, the $+$ capability t^+ allows an application to add t to its label, roughly corresponding to reading data owned by t. The $-$ capability t^- allows an application to remove t from its label, declassifying the data.

As tags are created dynamically, the first program to create a tag t gets ownership of the tag, and controls which of t's capabilities other programs have. The first program defines a default capability set that all other programs begin with; other programs may voluntarily relinquish any capabilities from this set using a *capability mask*, but they can never add capabilities that are not granted in the default set. Specifically, if the default capability set is C and a program's capability mask is M_p, then the program's effective capability set $C_p = C \setminus M_p$. Regardless of the default capabilities specified, the owner of a tag always has all capabilities by default unless they are explicitly masked.

5.3. Reference Monitor

In the T-DIFC labeling system, the classical restriction and join relations are simply $\sqsubseteq \; = \; \subseteq$ and $\sqcup \; = \; \cup$. However, there are additional constraints on whether information flows are allowed, as label propagation may require adding or removing tags from labels, which requires the corresponding capabilities as described above. Specifically, any data with label L flowing to a process p

```
⟨string⟩ ::= '"' [^"]* '"'

⟨capture-group⟩ ::= '<' [1-9] '>'

⟨tag-fragment⟩ ::= ⟨string⟩ | ⟨capture-group⟩

⟨tag⟩ ::= 'tag' '(' ⟨tag-fragment⟩+ ')'

⟨cap⟩ ::= '+' | '-'

⟨tag-cap⟩ ::= ⟨cap⟩* ⟨tag⟩

⟨statement⟩ ::= 'settags' ⟨tag-cap⟩* (* set process label tags *)
  | 'addtags' ⟨tag-cap⟩+ (* add tags to process label *)
  | 'deltags' ⟨tag⟩+ (* remove tags from process label *)
  | 'setcaps' ⟨tag-cap⟩* (* set default tag caps *)
  | 'addcaps' ⟨tag-cap⟩+ (* add tag cap to default *)
  | 'delcaps' ⟨tag-cap⟩+ (* remove tag cap from default *)
  | 'setmask' ⟨tag-cap⟩* (* set process cap mask *)
  | 'addmask' ⟨tag-cap⟩+ (* add cap to process cap mask *)
  | 'delmask' ⟨tag-cap⟩+ (* remove cap from process cap mask *)

⟨pid⟩ ::= 'self' | 'parent' | 'children' | ⟨capture-group⟩

⟨process-block⟩ ::= 'process' ⟨pid⟩+ '{' (⟨statement⟩ ';')+ '}'

⟨init-block⟩ ::= 'init' '{' ⟨process-block⟩+ '}'

⟨match-block⟩ ::= 'match' ⟨regex⟩ '{' ⟨process-block⟩+ '}'

⟨log-location⟩ ::= 'stdout' | 'stderr' | ⟨string⟩

⟨config⟩ ::= 'id' [0-9]+ (* policy ID *)
  | 'logfile' ⟨log-location⟩ (* writes to these files are logs *)
  | 'max_process_label' [0-9]+ (* max process label size *)
  | 'max_socket_label' [0-9]+ (* max socket label size *)

⟨top-statement⟩ ::= ⟨config⟩ | ⟨init-block⟩ | ⟨match-block⟩

⟨policy⟩ ::= ⟨top-statement⟩*
```

Figure 2: Grammar for the T-DIFC policy language. Some productions, such as <*regex*>, are omitted for brevity.

with current label L_p results in the process label updating to $L'_p = L_p \sqcup L$ if and only if

$$\forall t \in L'_p : t \notin L_p \implies t^+ \in C_p;$$

otherwise, the flow is denied. Because files have no associated capabilities, flows from a process p to a file f are always permitted, with the file label updating to $L_f \sqcup L_p$.

To enforce all these requirements, we must insert checks and label propagation logic at any information flow mediated by the kernel. Our reference monitor design is influenced by the hooks made available for Linux Security Modules (see Section 6). Kernel-traceable information flows include any mechanisms for file I/O, network I/O and interprocess communication (IPC).

Socket writes are an exception to the general rule above, because flows to sockets leave the system and are thus no longer enforceable by T-DIFC thereafter. We adopt the same policy as existing DIFC systems, and require sockets to always have an empty label, which forces applications to explicitly declassify all data that is sent out a socket. However, because T-DIFC is instrumentation-free, we cannot guarantee that policies can explicitly declassify data prior to socket writes if there is no log to intercept. As an alternative, we also allow a process p to write to a socket if L_p has at most n tags, where n is configurable by p's policy, and $\forall t \in L_p : t^- \in C_p$; essentially, p can specify that labels below a certain size may be implicitly declassified before socket writes, so long as p is actually allowed to declassify all the tags in L_p. Even if this implicit declassification occurs and the write is allowed, L_p remains unchanged; it is not updated to an empty label.

5.4. External Policy Language

A key feature of T-DIFC is the ability for machine operators or administrators to define *external* DIFC policies such that the application does not need to be modified to cooperate with the DIFC system. The grammar for the external policy language is given in Figure 2. At a high level, each policy is defined by a configuration preamble (`config`) and blocks of actions to modify the labeling state when initializing a program (in the `init-block`) or upon matching a log write (in a `match-block`).

Configuration. The mandatory `id` option declares a unique integer policy ID used to associate applications with their policies at runtime. The `logfile` option, which can be specified multiple times, defines which write syscalls will be considered log writes by listing specific output locations as log locations; they can either name file descriptors (e.g., `stderr`) or file paths (e.g., `/var/log/foo.log`). The `namespace` option defines how the policy will interact with other policies on the same system; policies within the same namespace will see the same tags. The "unique" namespace ensures a policy will have sole ownership over its own tags; other policies will be unable to reference these tags, although the reference monitor will still enforce the same propagation rules. Shared namespaces allow policies to be cooperatively designed; e.g., an upstream program could initially tag data, while a downstream program can enforce some dataflow policy according to the incoming tagged data. The policy can also specify a process maximum label size with `max_process_label` as a mechanism to mitigate taint explosion by restricting access to certain data; e.g., a maximum label size of 1 could prevent a client from reading files other than their own. The maximum socket label size, as described in Section 5.3, is given by `max_socket_label`.

Modifying labeling state. The policy state can be modified by adding, removing, or setting the current label's tags. Any statement that could add tags needs to additionally define the default capabilities for the tag in case the policy becomes the tag owner; it is also possible to later adjust the default capabilities for owned tags. The current capability mask can be modified in a similar way by specifing tag capabilities to change.

Where and when to modify the labeling state. Statements to modify the current labeling state can either occur when a program is launched in an `init-block`, or when matching a specific log message identified by a `match-block`. We envision the `init-block` being potentially useful for dropping unneeded capabilities at the start of a program. For match blocks, it is further possible to extract substrings from the matched log using "capture groups" denoted by angled brackets ("<>").

The process whose state is modified must additionally be specified as either the current process (`self`), its parent (`parent`), its children (`children`), or a specific process identified by PID. In match blocks, the PID may be a captured substring of the matched log. These options exist to allow T-DIFC to correctly operate regardless of where a log is emitted; i.e., if a process forks a handler that needs to be labeled, the parent process is still able to correctly label the forked child.

6. Implementation

We implement the T-DIFC reference monitor as a non-exclusive Linux Security Module (LSM) [44] for Linux v5.4 in 3142 lines of C. T-DIFC is a non-exclusive LSM: it can be stacked with other LSMs, so T-DIFC can focus purely on its own DIFC policy.[2] Thus, if an administrator also requires a traditional MAC system like SELinux [38], it is possible to run T-DIFC concurrently with other Linux LSMs. We note that it is important for T-DIFC to be the final LSM in the stack when performing permission checks; otherwise, if T-DIFC permits an operation (e.g., a read) but some other LSM later denies the operation, then T-DIFC's label state will become inconsistent, as it assumes the operation actually occurred and may update file or process labels according to the DIFC policy.

We implement the external policy compiler in 1301 lines of OCaml 4.08.1. Having a userspace compilation step minimizes the amount of complex parsing and validation code that must run within the kernel. Furthermore, we can more easily iterate on the frontend language while keeping the same binary format later read by the kernel.

Associating Programs and Policies. At runtime, the system must load all external policies via a `sysfs` interface into T-DIFC. At runtime, we use the policy ID to associate programs with their loaded policies: any program that should be associated with policy i must have an extended attribute (`xattr`) set on its binary file containing i. Processes without any active policy (i.e., either the binary had no policy ID `xattr`, or no loaded policy was found) will still propagate labels as specified in Section 5.3.

Intercepting Log Writes. We use LSM hooks to perform both permission checks and label propagation. However, the interface of the `file_permission` hook is unfortunately too restricted to perform log regex matching, as we need access to the written string. Thus, T-DIFC also directly wraps the `write` and `writev` syscalls. The wrapper first checks if the output file is a log file according to the policy's log locations. If so, then the wrapper matches each regex stored in the policy's data with the written string. For any successful match, the wrapper extracts any captured strings from the regex and runs the corresponding bytecode, referring to the captured strings when needed. Finally, the wrapper then continues to the original `write` code that actually performs the write.

Label Implementation. T-DIFC's design logically represents tags as strings, so tags can be generated from fragments of log messages at runtime. However, T-DIFC must also frequently compare tags to check whether a tag is being added to a label or not, as adding a tag requires the corresponding capability. To mitigate the impact of frequent string comparisons, T-DIFC instead builds a hash table of integer tag IDs at runtime. Whenever a tag t is referenced by a policy, T-DIFC looks up the tag's ID by hashing the tag string. If an entry for t is found, then T-DIFC uses the existing ID, and also knows that t has already been allocated; otherwise, T-DIFC generates a unique ID for t, implemented via a counter, and makes the current process the owner of t. The tag ID is then used in the label to represent the tag, allowing T-DIFC to compare integers instead of strings at runtime.

Because the tag ID table is built at runtime, tag IDs cannot be used on labels that persist across power cycles, which includes file labels. Instead, whenever a file is labeled, T-DIFC looks up the tag string corresponding to the tag ID for each ID in the label, and serializes the set of tag strings to the file's extended attributes along with some additional metadata. We also do not want other policies to unexpectedly read the file if they are not permitted to, or allow them to take ownership of the tags in the file, so we also store each tag owner's policy ID and default capabilities. If T-DIFC encounters a new tag when reading a file, it sets the owner of the tag as specified by this stored owner ID instead of defaulting to the current process, which both prevents a different policy from stealing ownership of the tag, and allows T-DIFC to correctly set some owner of the tag if the current running process does not actually have an associated T-DIFC policy.

7. Security Analysis

In this section, we will consider how T-DIFC satisfies the design goals outlined in Section 5.

Decentralized Information Flow Control. DIFC allows any application to take the ownership of and impose access control restrictions on information handled within the application, rather than having a central authority control the flow control policy. T-DIFC fulfills this classical goal of DIFC. Tags in T-DIFC handle the ownership of information while capabilities handle enforcing access control. Any program p can allocate a tag t, as long as t has not already been allocated by some other program. p then becomes the owner of t and has complete control over the access control policy for t. p can restrict access to data tagged with t by removing t^+ from the default capability set. This ensures no program other than p will be able to read any data tagged with t. Once data has been tagged with t, the data will retain t until t is explicitly declassified, which requires the declassifying process to hold the t^- capability. Thus, if p chooses not to add t^- (i.e., capability to remove t) to the default capability set, then t can never be declassified by any program other than p itself, guaranteeing that t's access control restrictions will always hold for any data initially tagged with t, even as this data is manipulated in the system.

Finer-Than-Process Granularity. T-DIFC provides finer-than-process granularity flow tracking by inferring the boundaries of event processes. Rather than having programs explicitly transition to a new event process like Asbestos [8], T-DIFC's reference monitor updates the program's security label automatically when a new event process boundary is detected. Note that, unlike past approaches that required dedicated program instrumentation, T-DIFC's ability to trace at this granularity depends on the presence of innate application event logs that mark the boundaries

2. The desirable ability to deploy DIFC in tandem with other MAC systems motivated our decision to not build T-DIFC over top of legacy systems such as Flume [21].

of event processes. In Section 8.1, we demonstrate that this practice is commonplace in mature open source projects.

Transparency. T-DIFC achieves the goal of transparency through the use of the external policy language and compiler. Instead of instrumenting an application's source code to enforce DIFC policies, T-DIFC offers a policy language for machine operators to specify DIFC policies. This external policy can then be loaded into kernelspace by the T-DIFC reference monitor to control a program's labeling decisions and access control policy (via capabilities) transparently to the monitored applications.

Reference Monitor Guarantees. For T-DIFC to satisfy the reference monitor concept, it must be tamperproof, assure complete mediation of system activity, and be verifiable [1]. Our argument for satisfying these properties follows from our decision to implement T-DIFC as stackable module that is interoperable with SELinux.

Tamperproof. The T-DIFC enforcement mechanism runs in Ring 0 (kernelspace) and cannot be directly accessed or modified from Ring 3 (userspace) code. If an attacker is able to modify critical system resources, it may be possible for them to escalate privilege and disable the module. Running T-DIFC alongside SELinux with the SELinux reference policy activated hardens the kernels and provides some protection from integrity violations. We also rely on an SELinux policy module to protect the integrity of T-DIFC's userspace dependencies (external policy, log files). This is because expressing these kinds of restrictions using type enforcement is much simpler than through DIFC. Of course, the Linux kernel can still be compromised even with SELinux enabled, which allow an attacker to disable T-DIFC. That said, our system is at least as tamperproof as Linux's canonical reference monitor.

Complete Mediation. Similarly, our argument for satisfying complete mediation follows from our use of the Linux Security Modules framework. LSM's authorization hooks were placed to mediate all security-sensitive access to controlled data types [49], making it possible to achieve complete mediation using a security module. T-DIFC uses these hooks to assure that decentralized flow control is guaranteed both local flows and outbound flows. For local flows, T-DIFC can continue to monitor the flow from source to destination, so it is sufficient to simply ensure that the tags of all potentially written information are propagated. Because the T-DIFC reference monitor tracks information purely at file- and process-level granularity, it conservatively assumes that any information contained in the source could have flowed during I/O. For outbound flows, T-DIFC is no longer able to propagate tags. Therefore, the system guarantees that all information must be declassified prior to transmission. In practice, our implementation only defines a subset of the authorization hooks that would be necessary to satisfy this property; our design can viably provide this assurance.

Verifiable. While a formal verification of our proof-of-concept implementation is not practical or worthwhile, we consider the eligibility of an idealized version of our design for verification efforts. As noted above, T-DIFC's claim to tamperproofness and complete mediation is rooted in past verification efforts of Linux. The Linux Security Modules framework has been subject to rigorous study in order to assure the correct placement of its authorization hooks (e.g., [12], [18], [7], [49]), and SELinux policies have likewise been verified for correctness (e.g., [17], [19]). While T-DIFC policies are short enough to be candidates for verification, one obstacle is that the policy's target application also falls within the trusted computing base of T-DIFC. Thus, it would be necessary to demonstrate the correctness of the target program and its logging behaviors to formally verify the correctness of T-DIFC.

```
id 21;
namespace unique;
logfile "/var/log/proftpd/proftpd.log";

max_process_label 1;

match ".*proftpd\[<[0-9]+>\].*: USER <[^:]+>:
    Login successful.\n" {
  process <1> {
    settags tag(<2>);
  }
}
```

Figure 3: T-DIFC policy for ProFTPD. Whenever a client process is forked, the log message contains the client's PID and username. The policy extracts these values as captured groups <1> and <2>, respectively, from the log message, then sets the label of the process with PID <1> to <2>, the given username, with no default capabilities. The policy also sets the maximum label size of the running process to 1, which prevents a user's client from reading other users' data.

```
id 2;
namespace "";
logfile stderr;

max_process_label 1;

match "Logging in as <.+> \.\.\. Logged in!" {
  process self {
    settags +-tag(<1>);
  }
}
```

Figure 4: T-DIFC policy for Wget. The policy sets the current process label to the authenticated user's username <1> with both the $+$ and $-$ capability; any downloaded files will then be assigned this label.

8. T-DIFC Application Policies

We now evaluate the practical utility of T-DIFC by exploring the potential security context that is available in application logs, as well as how T-DIFC policies could be defined to meet the needs of specific applications.

8.1. Survey of Application Logging Behavior

T-DIFC's ability to achieve *finer-than-process granularity* depends on the presence of innate application event logs that mark the boundaries of event process. In other words, T-DIFC depends on applications periodically recording log events that describe the subjects and data objects being operated upon. To demonstrate the generality of our approach, we conducted a representative survey of logging behavior in popular open source applications.

Program	Log Level	Subject Labels	Objects	"Useful" Policy?	Sample Log
HTTP Servers					
Apache	`INFO`	App. Users (HTTP)	Files, Databases	✓	`host - user` ...
Lighttpd	Configurable	App. Users (HTTP)	Files, Databases	✓	Format `%u`: username
nginx	Configurable	App. Users (HTTP)	Files, Databases	✓	See Apache
Other Servers					
cupsd	`INFO`	App. Users	Intermediate Files	✓	... `Queued on 'printer' by 'user'.`
Postfix	`INFO`	Email Addresses	Mail Files	✓	... `from=<email>` ...
ProFTPD	`INFO`	System/App. Users	Files	✓	`USER user: Login successful.`
Load Balancing					
HAProxy	Default	IP	Backend Server Sockets	✗	`Connect from <addr>:<port> to` ...
Web Applications					
MediaWiki	`DEBUG`	App. Users	Wiki Pages (Database)	○	... `Login for user succeeded` ...
MoinMoin Wiki	`DEBUG`	App. Users	Wiki Pages, Files	○	... `authenticated user u'user' (valid)`
Databases					
PostgreSQL	Configurable	App. Users, IP	Databases	○	Format `%u`: username
Redis	Monitor	Keys	Client Processes	○	... `"get" "key"`
memcached	Verbose (`-vv`)	Keys	Client Processes	○	`<32 get key`
Web Clients					
cURL	Verbose	App. Users	Files	✓	`< 230 User <user> logged in`
Squid	Configurable	App. Users, URLs	Client Processes	✓	`%un`: username, `%ru`: request URL
Transmission	`INFO`	None	Torrent Files	✗	`Parsing .torrent file successful "<file>"`
Wget	Default	App. Users	Files	✓	`Logging in as <user> ... Logged in!`

TABLE 1: Survey of logging behavior of popular applications. Each row describes the logging behavior of an application listed under the Program column, with programs grouped into categories. The Log Level column describes the required logging configuration for the application to emit useful information about subject and objects noted in corresponding columns. The "Useful" Policy column indicates whether we were able to identify a policy given the application's logs and behavior: ✓ means that we identified a policy based on finer-than-process granularity mediation, ✗ means we did not, and ○ indicates that useful policy components were identified but the policy was incomplete. The Sample Log column provides an example of an application log message or configuration option that results in a useful subject.

Initially, our approach was to examine the top packages in the Ubuntu Popularity contest; however, even after filtering out libraries and packages without source code, we found that the most popular packages were lightweight system utilities that didn't maintain complex application state. For example, the highest ranked programs are almost system utilities such as `sed` or `coreutils`, which are relatively small with minimal logging activity. Broadly speaking, DIFC cannot provide value to these simple programs. Instead, we defined broad categories of application event types, prioritizing applications that were more likely to handle complex states for multiple clients/users. We then manually identified and reviewed representative applications from each category, carefully examining both documentation and program source code to understand their logging behaviors. No application, once identified for the survey, was omitted from our results. We decided to conclude the survey after examining 16 programs.

Table 1 shows our results. In all but one of the surveyed applications, we were able to identify application events that outlined the boundaries of event processes and provided security contexts (in the form of subject and object labels). The candidate subject labels we observed included some system objects (IP, System User) that duplicated OS level information, but mostly contained application-level semantics such as application user names. The logging configuration (Log Level) necessary for the application to emit these events varied across applications. Although some applications required high verbosity levels (`INFO`, `DEBUG`, `-vv`), many others logged the required context by default or offered a configurable interface for auditing specific events. Encouragingly, we also found that the listed log events occurred within the main event handling loop of the application, meaning that the presence of these events indicates the start of a new event process. This is consistent with past observations that application event logging can be used to perform execution partitioning on system-level audit logs [14]. *To conclude, we found that many popular applications are already logging the necessary information to express DIFC policies with* T-DIFC.

8.2. Case Study: ProFTPD

As a specific example, we consider how an administrator can leverage T-DIFC to add access control guarantees to ProFTPD, an FTP server, without any source code modifications. Our test environment uses the default ProFTPD package from Ubuntu 20.04 LTS. Suppose our administrator needs to setup a ProFTPD server to serve a large organization. The administrator leverages ProFTPD's virtual user functionality, which supports LDAP, SQL, and many other backends, to scalably integrate FTP authentication with their existing database; unfortunately, this also means that the virtual users will not be subject to system-level security policies. To avoid data leaks, the administrator must ensure that there are appropriate application-level controls in place to prevent unauthorized access.

To define a policy, the administrator first observes ProFTPD's logging behavior: how does it handle user sessions, and what kind of log messages does it emit? The administrator reviews the application logs and notices that, upon receiving a connection, ProFTPD will emit a log message containing both the client's username and client process PID. Each PID is different, indicating that ProFTPD uses a separate process per client.

Based on these observations, the administrator writes the ProFTPD policy shown in Figure 3. They first define a policy ID, label namespace, and the location of the ProFTPD log file on disk (Lines 1 – 3): the policy ID (here chosen to be 21, the standard FTP port number) is used to associate this policy with the ProFTPD binary; the namespace indicates that labels for this ProFTPD policy should be completely independent of any other policies on the system; and the log file specified is the default ProFTPD log file in Ubuntu 20.04 LTS. They then add a constraint that at most 1 label can be associated with the current event process (Line 5), guaranteeing that each user session can only access public (unlabeled) data or their own data. Finally, the administrator defines a match block over the "login successful" application event that contains the core labeling logic (Lines 7–11). The match block extracts the PID p of the client process and the user's username u, and sets the label of p to u. After defining and compiling the policy, the administrator sets the policy ID of the `proftpd` binary to activate the T-DIFC policy.

Stability of policy across program revisions. While a full-scale study of the stability of log messages in each program is outside the scope of this project, we give some insight into the stability of T-DIFC policies by examining the log messages specifically used in each policy.

We looked through ProFTPD's git history to find when the exact "Login successful." log message was first introduced. This message was committed on October 10, 1999, with commit comment "Updated logging to be more consistent, and generally be more informative." This change was later released in unstable version 1.2.0pre9 on October 27, 1999; the stable release of 1.2.0 happened on February 26, 2001, and the log message remains unchanged to the current stable version, 1.3.7.

8.3. Case Study: wget/cURL

T-DIFC is not limited to servers; it can also enforce DIFC policies for other programs, such as clients. We demonstrate how T-DIFC can use the output of Wget and cURL, which retrieve data from remote hosts, to automatically label data belonging to a particular user. Suppose a user is retrieving their file from a remote FTP or HTTP server where they must authenticate themselves with a username and password.

Once again, the administrator starts by observing the log output of these programs. They see that Wget prints status information to `stderr`, including where it connected to, the current download progress, and the user it logs in as on the remote host. Based on this application event, Figure 4 shows the Wget policy that the administrator is able to write. In Lines 7-11, the match block detects the login success message, extracts the username u, and sets Wget's label to u using the `self` keyword. As a result, any files downloaded by Wget will automatically be labeled with the username used to access the remote host. Unlike ProFTPD, Wget is a small client program that is more likely to be used in concert with other processes; to account for this, the administrator sets the namespace for this policy to an empty string, the de-facto global namespace (Line 2). Other programs can thus use the labels on the downloaded files if their policy is also in the global namespace.

It is possible to write a similar policy for cURL. cURL must be run with the verbose switch (`-v`) to output a message when authenticating to the remote host. For brevity, we omit this policy, as it is largely identical to the Wget policy aside from the regular expression used to match the username in cURL's output.

Stability of policy across program revisions. We looked for the git commit that introduced the exact matched message, i.e., "Logging in as <user> ... Logged in!" This message has been in wget since its first commit in the git repo on December 1, 1999; any prior changes on CVS are mising from the git repo's history. As this message has remained stable for at least 23 years, we believe it is unnecessary to find exactly when it originated.

8.4. General Utility of T-DIFC policies

Returning to Table 1, we now consider the space of possible T-DIFC policies we observed in the rest of survey.

HTTP Servers. Across the 3 surveyed applications, we observe that HTTP servers consistently log when HTTP-authenticated users access a site. These log statements permit T-DIFC to mediate system accesses of these users at finer-than-process granularity by labeling the client handler processes with usernames.

Other Servers. Like web servers, the other servers we survey all feature some application-level notion of user IDs, which are consistently logged. The `cupsd` server logs the authenticated user that has submitted a print job; because `cupsd` temporarily saves queued files in `/var/spool/cups/`, T-DIFC could improve confidentiality by assuring that only the user that submitted the print job has access to the file. The `postfix` mail server logs the addresses of incoming emails, potentially enabling T-DIFC to enforce different confidentiality/integrity policies to emails and email attachments based on the identity of the sender. We discuss ProFTPD in detail above.

Load Balancers. In other software that supports distributed applications, we observe less applicability for T-DIFC. For example, the `HAProxy` load balancer only handles system-level entities such as IP and port. While this information is logged, we could not identify a useful T-DIFC policy because the entities could also be mediated using standard system access controls. DIFC in general seems less applicable to these applications. We would expect these observations to hold for many other middlebox applications, e.g., firewalls.

Web Applications. We survey two wiki platforms, both of which contain log statements that link a given client handler to the authenticated user. This could allow T-DIFC to mediate system-level accesses accounting to the application-level user name. However, an important consideration is that web applications typically use databases as a backend, which appear as monolithic files from T-DIFC's perspective. As a result, T-DIFC would likely struggle to enforce meaningful security policies on this class of applications without an effective data partitioning strategy. We note that the canonical DIFC application, Flume [21], made architectural changes to the MoinMoin wiki source code in their demonstration.

Databases. We survey three databases, one disk-based (`PostgreSQL`) and two in-memory key-value stores (`redis`, `memcached`). In `PostgreSQL`, T-DIFC can identify application-level subjects and label partitions of process activities accordingly; however, it is unclear whether this could lead to useful policies due to the monolithic nature of databases at the system-level. From the kernel's perspective, the database is represented as just a few files. While T-DIFC can observe that different

Test	Control	T-DIFC	Overhead
Process times (μs)			
null call	0.32 ± 0.00	0.41 ± 0.28	27.2%
null I/O	0.56 ± 0.09	0.43 ± 0.09	−23.7%
stat	0.60 ± 0.02	0.63 ± 0.02	4.5%
file open/close	1.29 ± 0.05	1.36 ± 0.05	6.0%
select TCP	4.81 ± 0.02	4.79 ± 0.01	−0.4%
signal install	0.37 ± 0.00	0.37 ± 0.00	0.0%
signal handle	0.90 ± 0.00	0.91 ± 0.01	0.4%
fork process	71.91 ± 1.16	73.24 ± 3.45	1.8%
exec process	240.4 ± 13.3	263.7 ± 23.6	9.7%
shell process	$789. \pm 241.$	$1069. \pm 436.$	35.5%
Local latency (μs)			
pipe	4.06 ± 0.30	3.64 ± 0.32	−10.3%
`AF_UNIX`	6.63 ± 0.65	6.19 ± 0.71	−6.7%
UDP	5.50 ± 0.06	5.60 ± 0.20	1.8%
TCP	7.48 ± 0.35	6.94 ± 0.41	−7.2%
TCP connect	11.28 ± 3.02	11.96 ± 3.26	6.0%
Local bandwidth (GB/s)			
pipe	6.43 ± 0.08	6.36 ± 0.12	−1.2%
`AF_UNIX`	14.30 ± 0.67	15.30 ± 0.82	7.0%
TCP	10.80 ± 0.42	11.00 ± 0.00	1.8%
file reread	10.18 ± 0.35	10.47 ± 0.09	2.8%
File latency (μs)			
create (0k)	4.64 ± 0.09	4.77 ± 0.16	2.7%
delete (0k)	3.41 ± 0.01	3.45 ± 0.04	1.1%
create (10k)	10.12 ± 0.20	9.85 ± 0.19	−2.6%
delete (10k)	5.09 ± 0.05	5.10 ± 0.02	0.1%

TABLE 2: `lmbench` results for T-DIFC, compared with the same Linux kernel running without T-DIFC. Most values are essentially equivalent within measurement error.

application users are accessing these files, our current implementation lacks a data partitioning scheme that would allow us to perform fine-grained mediation over tables, records, etc. For the in-memory stores, the application logs reveal the specific keys being accessed, but due to the limitations of our implementation we are unable to propagate these labels to memory pages or page offsets. Taken as a whole, database software highlights the limitations of the current T-DIFC system but provides interesting avenues of inquiry for future work.

Web Clients. We survey four web clients, two of which (`cURL` and `wget`) are described in Section 8.3. `Squid` is a caching proxy that can be run on either client- or server-side, but is often associated with client-side use by web browsers to reduce bandwidth consumption. However, browser caches are also associated with a number of privacy-infringing side channel attacks, e.g., [20]. Fortunately, because `Squid` records detailed logs of requested URLs, T-DIFC could be used to label cached files according to their web origin. This would allow T-DIFC to supplement the cache's existing access controls by providing system-level enforcement of same origin policies. Lastly, for the BitTorrent client `Transmission`, we are unable to identify useful subject or object labels in the log statements. Similar to the `HAProxy` case, we do not believe T-DIFC could improve security here because the client does not possess an application-level notion of subjects and operates only on system-level data objects.

9. Evaluation

We evaluate the overhead incurred by running T-DIFC relative to running an unmodified Linux kernel. We begin by quantifying the specific overheads introduced by T-DIFC in microbenchmarks, and then consider the end-to-end effect on applications. All benchmarks were done on a server with an 8-core, 4.20 GHz Intel i7-7700K processor, 64GB of RAM, and a 512GB NVMe SSD. We built T-DIFC on top of Linux v5.4.

Comparison with existing systems. Ideally, our evaluation would be able to compare T-DIFC to past DIFC systems as a baseline, most notably the infuential Flume system [21] that informs our policy language. Unfortunately, a number of factors prevented us from providing this baseline. Flume and its contemporaries were implemented for older kernels, e.g., Linux 2.6, which is past its end of life and may not be suitable for contemporary applications. In contrast, our design leverages LSM module stacking, a feature introduced in Linux 5.1, to allow T-DIFC to mediate on application logs without interfering with other access controls (e.g., SELinux). Lastly, to the best of our knowledge Flume's source code, as well as other comparable systems such as Hi-Star [48] or Asbestos [8], are not publicly available.

9.1. LMbench

We begin by using LMbench to measure T-DIFC's impact on performance without installing a policy. Without an active policy, modifications that may incur overhead include the LSM permission hooks for file and socket writes, artifacts from memory changes such as the data structures we add to process control blocks, and the modified `write` syscall used to track log messages.

The LMbench results in Table 2 show minimal overheads incurred by T-DIFC, most of which are within measurement error. While we do occasionally see negative overheads, these are most likely due to noise introduced by caching or scheduling that result in T-DIFC performing unexpectedly differently than the control. Overall, the overhead of running applications without an active policy on T-DIFC is very low, especially when the applications are working with unlabeled data, as in the LMbench measurements. However, the overhead incurred by T-DIFC may increase even for a program with no active policy if the program uses labeled data (e.g., reads from a file labeled by some other program on the system). To measure these overheads, we explicitly measure the performance of `write` syscalls in Section 9.2.

9.2. `write` Overhead

To better quantify the specific impact of T-DIFC on performance, we instrument and conduct a series of measurements on the modified `write` syscall, comparing the end-to-end results with an unmodified `write` call. Specifically, we create a series of test policies that match the message "`Benchmark: T-DIFC`", then run a varying number of statements within the match block to set the current process label to varying label sizes. By measuring the latency of such log writes, we can capture the end-to-end cost of all of T-DIFC's functionalities: log regex matching, execution of the policy code corresponding to the matched regex, and permission checks and label propagation. We tested all combinations of label sizes between 1 and 20 tags, and policy sizes between 1 and 20 statements. We seek to address the following questions:

1) What is the expected overhead of a single write call under T-DIFC?

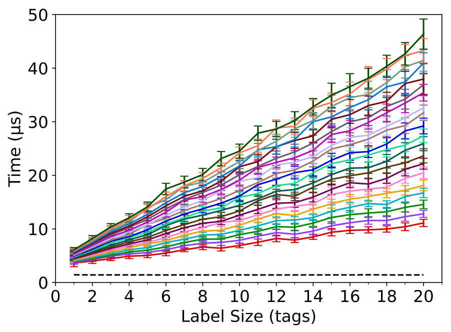
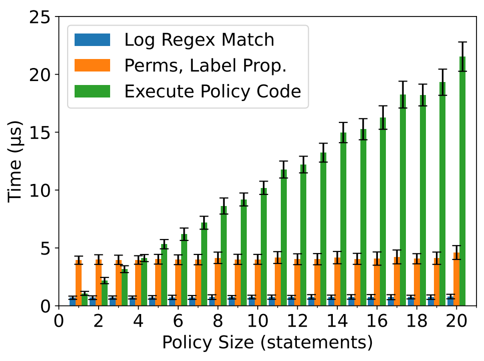


Figure 5: Effect of external policy code size on the mean write times to a monitored application log. The bottom dashed line is mean time for the write without T-DIFC. The solid lines are the time taken for the `write` syscall with T-DIFC; the bottom line represents a policy size of 1, increasing up to a policy size of 20. The runtime increases linearly with the number of policy statements.

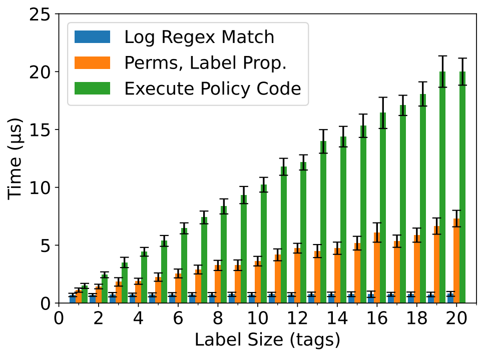
2) How is this overhead affected by the process label size and policy code size?
3) How much of this overhead is attributed to the three operations listed above per `write` call?

To address points 1 and 2, we report the end-to-end latency (measured in user space) of varying label size and policy size in Figure 5. As can be seen, costs increase linearly with both label and policy size. This is because labels with more tags requires more iterations of the copy operation to propagate the tag to the written file. Likewise, increasing the number of policy statements means proportionally more code must be executed on each match. Because of the high level of optimization of the normal `write` call, the overheads of T-DIFC are quite high, ranging from 154% (Label and Policy Size is 1) to 3184% (Label and Policy Size is 20). However, the policy size costs only appear on `write` events to the monitored application log, not all write calls. Under normal circumstances, `write` latency would only be affected by the number of tags being propagated (i.e., label size).

As mentioned above, the runtime performance of T-DIFC is dominated by three operations: log regex matching, permission checks and label propagation, and execution of the policy code. Of these, log regex matching and policy code only impact writes to monitored application logs, while permission checks and label propagation occur on every write. Thus, we can find the expected overhead for other writes by examining the cost of permission checks and label propagation. To do so, we instrument these three operations in T-DIFC's kernel code and observe performance for all combinations of label size (1-20) and policy size (1-20). Figures 6a and 6b show how these overheads vary as either label size or policy size is held constant while the other varies. We observe that varying the label size (Fig. 6b) increases both the overhead of label propagation and policy code execution, while varying the policy size only increases the cost of policy code execution. Further, the overhead of executing policy code dominates the cost of label propagation; the overhead from permission checks and label propagation is 5 μs or lower for label sizes of up to 10 tags, as opposed to double that overhead



(a) Policy Code Size



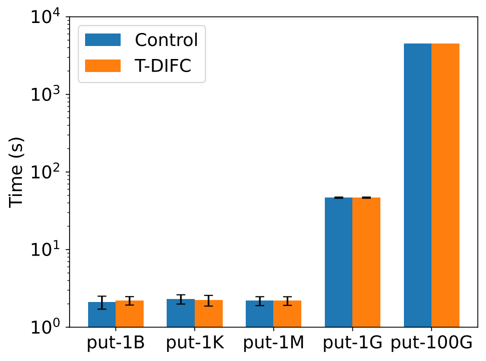
(b) Label Size

Figure 6: Mean time costs imposed by specific T-DIFC tasks under different policies. "Log Regex Match" is the time needed to match all regular expressions, "Perms, Label Prop." is the time needed to do permission checks and label propagation, and "Execute Policy Code" is the time needed to execute the policy statements within all matched blocks. In 6a label size is fixed at 10 tags, while in 6b policy size is fixed at 10 statements. The time to conduct regex matches remains fixed across both experiments.

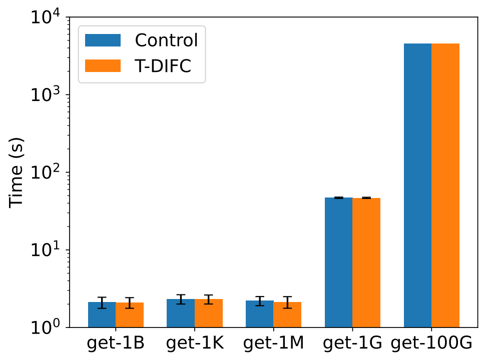
for executing policy code. This is encouraging since only the label propagation cost is imposed on every write. The time taken to match log regexes does not vary.

9.3. Application Workload Overhead

Although T-DIFC incurs relatively high overheads on log writes, log writes comprise a very small proportion of executed code in real programs. To get a better impression of expected overheads for real programs running T-DIFC policies, we measure some end-to-end runtimes of FTP transfers to a ProFTPD server running the policy introduced in Section 8.2. We use ProFTPD version 1.3.5e from Ubuntu 18.04.1 LTS, and the Linux NetKit 0.17 FTP client. The ProFTPD server is configured to use SQLite 3.22.0 as its authentication backend. We measure the total time taken for the FTP client to authenticate to the server locally, then either store or retrieve a file via the `put` or `get` command, and finally exit. The files are random bytes, varying in size from 1 byte (1B) to 1 gibibyte (1G). We also performed a test where the 1G file is transferred 100 times within the same session.



(a) FTP put



(b) FTP get

Figure 7: Mean performance costs of T-DIFC on FTP workloads. Results are averaged over 100 trials, except for 100G which was performed once. Any overhead imposed by T-DIFC falls within the measured error.

The measured times can be found in Figure 7. For this policy, the measured overhead was negligible. We note that there are many factors causing this measurement to be more favorable to T-DIFC. While the ProFTPD policy described in Section 8.2 is useful and achieves our goal of access control for ProFTPD application users, it is very simple, with only one match block and a maximum running label size of 1. As seen in 9.2, the expected overhead of write calls for small label and policy sizes is significantly lower than the expected overhead for larger label and policy sizes. Additionally, ProFTPD only emits a single log message per session when the user authenticates. Thus, the overheads reported in the prior section seem unlikely to be observed under realistic conditions.

During testing, we observed that authentication to ProFTPD using the SQL backend is relatively slow; examining the specific results in Figure 7, we see that this authentication essentially dominates the observed runtime for the smaller files (from 1B to 1M). Authentication takes around 2 seconds, which dwarfs the 2 μs overhead from a T-DIFC log writes with a label size and policy size of 1. On the other hand, for larger files, the data transfer ended up dominating the measured runtime. Presumably, if ProFTPD is parallelizing the actual file I/O and network activity, then the cost of label propagation for writing the file is only incurred for the final write of the transferred data. Once again, an overhead of around 2 μs is insignificant relative to the 45 seconds needed for actual data transfer.

10. Discussion & Limitations

General Feasibility of T-DIFC. In Section 8, we observe broad trends in the applicability and utility of T-DIFC for different classes of programs. Applications that contain application-specific notions of users (i.e., not system users) will consistently log authentication events, providing T-DIFC with the ability to infer useful subjects labels as well as partition process execution into fine-grained event processes. This includes many server applications (e.g., `proftpd`, `cupsd`, `httpd`) and some clients (e.g., `wget`, `squid`). The ability to partition processes on high-level subject labels effectively meets half of the requirements for T-DIFC to define a useful policy; the other half is whether T-DIFC is able to precisely mediate data accesses based on the kinds of objects that the application handles. At this point, application categories such as databases and and web applications highlight a notable limitation of T-DIFC's design – at present, we lack the ability to effectively data partition over objects such a databases, which appear as monolithic files from the kernel's perspective. This limitation is ultimately due to a semantic gap problem that also posed deployment challenges for classic DIFC systems. In future work, we hope to explore whether T-DIFC could be extended to support transparent data partitioning in order to overcome this problem. One possibility is that the labels emitted by T-DIFC could be propagated to low-level data structures such as memory pages and file blocks, then handled like a traditional authorization denial at the kernel level if a information flow violation is detected. Fortunately, our application survey discovered that such an intervention is often not necessary. This is because many classes of applications, most notably different kinds of network servers, handle application-specifc user semantics while handling data objects at the standard system abstraction (i.e., files). In this scenario, no further intervention is required to define useful T-DIFC policies. For this reason, we feel that network servers are the "killer application" for our current design.

Performance optimizations. While we made some effort to optimize the performance of T-DIFC, e.g., by representing tags as integers to avoid string comparisons, the overall system design is aimed to accommodate arbitrarily large labels and policies. Thus, we chose to use red-black trees to represent label sets, allowing for logarithmic complexity in the case of extremely large labels. In practice, policies may not actually require very large labels, so it may be faster to represent labels as sorted arrays rather than red-black trees, reducing the amount of memory dereferencing required at runtime.

Trusting application event logs. To achieve transparency, T-DIFC references application event logs to provide DIFC guarantees. In some ways, this is analogous to past DIFC systems' trust in the application itself; however, because these systems were able to modify the program, they often minimized the attack surface by defining guard modules that operated in a dedicated process space (e.g., [21]). Because T-DIFC operates transparently, it is not possible to achieve such privilege separation in the same way. Extending T-DIFC to include the ability to voluntarily and permanently relinquish capabilities may allow certain applications to emulate this form of privilege separation. Worse yet, recent work has demonstrated sophisticated runtime attacks that alter the contents of audit logs through control flow and data flow manipulation [45]. This attack is a concern not just for T-DIFC but also all threat detection and investigation tools. One possible way to defend against this attack is to cross-validate the application logs' contents against ground-truth control flow information [45].

Distributed support. T-DIFC treats network sockets as endpoints where any outgoing (and incoming) data must be declassified; this is the same approach as in previous systems like Flume [21]. Support for transmission of classified data would require a distributed reference monitor capable of tracking and coordinating information flows between cooperating machines. The design requirements of such a system have been explored in prior work [27] but are beyond the scope of the present study.

Implicit declassification. As discussed in Section 5.3, socket writes may implicitly declassify a label L_p. First, for a tag t that is not owned by p, implicit declassification can never lead to unpermitted exfiltration, as the owner can remove t^- from the default capability set. Second, the maximum label size restriction is designed for common scenarios; for example, in the case of client handler processes that are already launched with the appropriate tag for the authenticated user u, allowing at most 1 tag when sending data ensures that the client cannot exfiltrate multiple users' data, as this would result in the process label containing at least two tags. Thus, implicit declassification accounts for one of the classic motivating scenarios for DIFC – smash-and-grab web attacks in which an attacker is able to exfiltrate many hundreds of users' data using a single client session. This said, unintended declassification may occur if the policy's tag restrictions are incorrect or if data flows into the target application in an unanticipated way. The difficulty of specifying correct security policies is a common challenge for access control systems, one that we share. Ultimately, we feel that implicit declassification is a necessary compromise in order to provide DIFC enforcement for legacy and unmodified applications.

Usability. While we have demonstrated the applicability of T-DIFC, it may be that defining complex policies for cooperating programs poses a significant cognitive burden on machine operators. The difficulty of correctly defining system-wide security policies is a well-known problem, one that we expect T-DIFC also suffers from. That said, administrators must commonly review log output when diagnosing faults or investigating threats, and log output is often well-documented. Further, T-DIFC can interoperate with SELinux such that system-wide security can be achieved without defining DIFC-aware policies for every application. Ultimately, further work is required to determine the usability of the interface exposed by T-DIFC to machine operators.

11. Related Work

Myers and Liskov originally proposed programming language constructs to enforce DIFC [30]. JFlow is a Java extension including such DIFC constructs that transpiles to Java [29]; the successor to JFlow, Jif, is also capable of enforcing integrity [31] [32]. Flow Caml is a similar extension for OCaml [37]; LIO is a Haskell library enabling DIFC [39]. While there are many benefits to programming language-level enforcement of DIFC, such as static checking for reduced overhead or more exact dataflow analysis for tainting, such an approach necessitates source code modifications, which we seek to avoid.

Another approach to DIFC is to add explicit support within the OS. Existing reference monitors for Unix-like systems apply a security policy to programs without requiring extensive source code modifications, but are unable to handle the fine-grained decisions in DIFC [28] [10] [24] [43]. Previous OS-level DIFC implementations include Asbestos [8], HiStar [48], Flume [21], and Laminar [34]. However, Asbestos and HiStar are entirely new operating systems with new interfaces, thus requiring source code changes to leverage their DIFC primitives. Although Flume extends Linux and OpenBSD and is designed to require minimal source code modifications, porting existing applications to Flume still requires some amount of development effort to modify application sources. Similarly, though Laminar is a Linux security module, it still requires source-level changes.

Even for OS approaches to DIFC, programs must somehow label and declassify data. Typically, this is done by introducing new system calls, but leveraging these system calls will necessarily require source code changes. SIESTA connects the OS DIFC mechanisms from SELinux with language-level mechanisms in Jif [16]. However, this requires the program to already be written in a DIFC-supported language.

Specific DIFC policies using prior knowledge could automatically infer labels and apply the policy as part of a language's runtime. For example, JSFlow is a JavaScript interpreter that achieves dynamic information flow tracking that automatically distinguishes user data from input fields from public data [15]. It is then possible to enforce policies restricting where user data may flow without modifying source code. However, this approach is limited to a specific language and type of policy.

Systems such as EASEAndroid and TOMOYO Linux aim to address a different aspect of usability [13], [42]. These systems aid in the analysis and generation of IFC policies for TOMOYO's own reference monitor or SEAndroid (a port of SELinux to Android), respectively. These systems focus on a different problem and are also not specifically designed for DIFC.

Many existing works focus on bringing better security support to existing applications. TightLip ports existing applications by allowing users to identify sensitive data; it then monitors this data using "doppelgangers" to track potential security breaches [47]. RIFLE translates program binaries to a new ISA that enforces information flow security with specialized hardware [41]. InfoShield is a similar architecture that instead enforces a new "information usage" security policy [36]. We seek to similarly support DIFC in legacy applications, but with only a Linux security module and per-application software policies. TaintTrace allows for data flow tracing of existing programs by rewriting the binary, but causes a significant $5.5\times$ slowdown at runtime [4]. Our overall objective is different — we seek to enforce DIFC according to some program-specific policy, not dynamic taint tracing — and we seek less performance overhead.

Our strategy to identify where the contexts of a running process may change is to watch the application's logging behavior. It is typically considered good practice to log key events during execution; around half of all logs identify key points during execution [11]. We use log writes as markers to identify when the process's contexts need to be adjusted, what data labels are appropriate, and whether declassification is allowed. Identifying key events with logs has been used for similar tasks, such as error diagnosis [46], performance profiling [50], and execution partitioning [14].

12. Conclusion

We present T-DIFC, a solution that harnesses application logs to enable transparent DIFC policies. We demonstrate that for many existing programs, application logs carry enough information to enable designing useful DIFC security policies. A major drawback of DIFC is the requirement to port existing programs to use DIFC features, which is often unfeasible due to lacking development power or closed-source programs. By mitigating this drawback, T-DIFC can be more practically deployed in current systems.

Availability

Our code is available at https://bitbucket.org/sts-lab/tdifc/src/master/.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback. We also thank Wajih Ul Hassan, Mohammed Noureddine, and Pubali Datta for their insights in application logging. This work was supported in part by the National Science Foundation under contracts CNS 17-50024 and CNS 20-55127. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

References

[1] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972.

[2] D. E. Bell and L. J. LaPadula. Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, Air Force Electronic Systems Division, 1976.

[3] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, 1977.

[4] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, page 749–754, USA, 2006. IEEE Computer Society.

[5] D. D. Clark and D. Wilson. A comparison of military and commercial security policies. In *IEEE Symposium on Security and Privacy*, 1987.

[6] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[7] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 225–234, New York, NY, USA, 2002. ACM.

[8] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. *SIGOPS Oper. Syst. Rev.*, 39(5):17–30, October 2005.

[9] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 01 1974.

[10] T. Fraser. Lomac: Low water-mark integrity protection for cots environments. In *IEEE Symposium on Security and Privacy*, pages 230–245, 2000.

[11] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 24–33, New York, NY, USA, 2014. Association for Computing Machinery.

[12] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 330–339, New York, NY, USA, 2005. ACM.

[13] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task oriented management obviates your onus on linux. In *Linux Conference*, volume 3, page 23, 2004.

[14] Wajih Ul Hassan, Mohammad Noureddine, Pubali Datta, and Adam Bates. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *27th ISOC Network and Distributed System Security Symposium*, NDSS'20, February 2020.

[15] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1663–1671, New York, NY, USA, 2014. Association for Computing Machinery.

[16] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, USA, 2007. USENIX Association.

[17] Boniface Hicks, Sandra Rueda, Luke St.Clair, Trent Jaeger, and Patrick McDaniel. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010.

[18] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. *ACM Trans. Inf. Syst. Secur.*, 7(2):175–205, May 2004.

[19] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[20] Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing*, 19(1):44–53, 2015.

[21] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, October 2007.

[22] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of NDSS '13*, February 2013.

[23] Steven B. Lipner. Non-discretionery controls for commercial applications. In *IEEE Symposium on Security and Privacy*, pages 2–2, 1982.

[24] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, page 29–42, USA, 2001. USENIX Association.

[25] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 401–410, New York, NY, USA, 2015. ACM.

[26] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th USENIX Security Symposium*, August 2017.

[27] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 23–32, 2006.

[28] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. *Softw. Pract. Exper.*, 22(8):673–694, August 1992.

[29] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 228–241, New York, NY, USA, 1999. Association for Computing Machinery.

[30] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, October 1997.

[31] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000.

[32] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.

[33] OWASP. OWASP Top 10 - The Ten Most Critical Web Application Security Risks. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2017.

[34] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 63–74, New York, NY, USA, 2009. Association for Computing Machinery.

[35] R.S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[36] W. Shi, J. B. Fryman, G. Gu, H. H.-S. Lee, Y. Zhang, and J. Yang. Infoshield: a security architecture for protecting information usage in memory. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 222–231, Feb 2006.

[37] Vincent Simonet. Flow Caml, July 2003.

[38] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1:43, 2001.

[39] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, page 95–106, New York, NY, USA, 2011. Association for Computing Machinery.

[40] The Chromium Projects. Site isolation. Last accessed January 27, 2020.

[41] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, page 243–254, USA, 2004. IEEE Computer Society.

[42] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. Ease-android: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 351–366, USA, 2015. USENIX Association.

[43] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The trustedbsd mac framework: Extensible kernel access control for freebsd 5.0. pages 285–296, 01 2003.

[44] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[45] Carter Yagemann, Mohammad Noureddine, Wajih Ul Hassan, Simon Chung, Adam Bates, and Wenke Lee. Validating the integrity of audit logs against execution repartitioning attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, 2021.

[46] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. *SIGPLAN Not.*, 45(3):143–154, March 2010.

[47] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, page 12, USA, 2007. USENIX Association.

[48] Nickolai B. Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'06, November 2006.

[49] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[50] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 603–618, Savannah, GA, November 2016. USENIX Association.