Forensic Analysis of Configuration-based Attacks

Muhammad Adil Inam[◇*], Wajih Ul Hassan[◇*], Ali Ahad[‡], Adam Bates[◇],
Rashid Tahir[†], Tianyin Xu[◇], Fareed Zaffar[∓]

[◇] *University of Illinois at Urbana-Champaign*
{mainam2,whassan3,batesa,tyxu}@illinois.edu

[‡] *University of Virginia*
aa5rn@virginia.edu

[†] *University of Prince Mugrin*
r.tahir@upm.edu.sa

[∓] *Lahore University of Management Sciences*
fareed.zaffar@lums.edu.pk

***Abstract*—Causality analysis is an effective technique for investigating and detecting cyber attacks. However, by focusing on auditing at the Operating System level, existing causal analysis techniques lack visibility into important application-level semantics, such as configuration changes that control application runtime behavior. This leads to incorrect attack attribution and half-baked tracebacks.**

In this work, we propose Dossier, a specialized provenance tracker that enhances the visibility of the Linux auditing infrastructure. By providing additional hooks into the system, Dossier can generate a holistic view of the target application's event history and causal chains, particularly those pertaining to configuration changes that are among the most common attack vectors observed in the real world. The extra vantage points in Dossier enable forensic investigators to bridge the semantic gap and correctly piece together attack fragments. Dossier leverages the versatility of information flow tracking and system call introspection to track all configuration changes, including both dynamic modifications that directly update configuration-related program variables and revisions to configuration files on disk with negligible runtime overhead (less than 7%). Evaluation on realistic workloads and real-world attack scenarios shows that Dossier can effectively reason about configuration-based attacks and accurately reconstruct the whole attack stories.

I. Introduction

Cyber attacks resulting from improperly configured software have recently gained significant traction in the cyber security community. For instance, the OWASP Top-10 list, a well-known index of web application vulnerabilities, has consistently ranked security misconfigurations as one of the primary reasons for data breaches [22], [32], including several high-profile incidents that have recently been in the limelight [18], [1], [31], [27]. Similarly, CAPEC, which manages a taxonomy database of attack patterns and classifications, has categorized configuration-based attacks as one of the few high-severity classes that can considerably weaken the security posture of an organization [35]. Due to their prevalence and severity, configuration-based attacks have become major security threats, gaining increasing notoriety in hacking communities with every successful breach.

Configuration-based attacks lead to disastrous financial and business consequences, such as data breaches and system compromises. For example, in 2015, 35,000 Internet-facing instances of MongoDB were discovered to be publicly accessible without any form of authentication [36]. The total amount of data that was exposed as a result of this misconfiguration totaled roughly 680TB. Even though MongoDB provides ample security mechanisms to mitigate this problem, broken authentication was nonetheless widespread due to misconfiguration. Whether these misconfigurations were a result of oversight (e.g., default setup) or changed dynamically later on by an attacker is not known—this is because *configuration changes are not typically audited and thus often go unnoticed*. More recently, in July 2019, a configuration-based vulnerability in Capital One's firewall allowed hackers to access the credit card information of more than 100 million customers [7]. The breach cost the company a projected sum of 100 to 150 million dollars. As is evident, there is a growing need to systematically track and audit configuration changes in the face of such devastating attacks.

Provenance tracking is known to be effective in analyzing malicious system modifications. A variety of provenance-tracking techniques have been proposed in the literature [42], [62], [66], [72], [77], [63], [52]. These techniques use audit logging to record important events during system execution and then derive causal relationships between events during the analysis phase. Under the hood, these techniques leverage frameworks, such as Linux Audit [39] or Windows Event Tracing [16], which are based on system call interception to monitor accesses between system subjects (e.g., processes) and objects (e.g., files, sockets, pipes, etc.). The audit logs are then parsed into a causal graph for improved analysis and readability. If an Indicator of Compromise (IoC) is observed, a security analyst can use provenance graphs for root cause analysis to understand the chain of events that led to the flagged event. Furthermore, security analysts can also find all of the ramifications of the attack using the provenance graph.

Unfortunately, existing provenance-tracking techniques primarily collect audit traces exclusively at the operating sys-

*Equal contribution

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23057
www.ndss-symposium.org

tem (OS) level. Even though this focused approach allows trackers to log a wide variety of low-level OS events, they remain completely or partially oblivious to application semantics, resulting in incomplete coverage. Such semantic gaps undermine the forensic investigation of certain classes of attacks where information from the application layer is necessary to reconstruct the attack and correctly understand the complete chain of events. To close the semantic gap, a few provenance trackers [53], [71] attempt to integrate application-level information by leveraging applications' built-in event logging features. However, when surveying the open-source applications used to evaluate prior work [53], [71], we are surprised to discover that very few leading open-source software projects record configuration changes in log messages, rendering existing techniques inapplicable when investigating configuration-based attacks. Hence, in light of the increasing prevalence of configuration-based attacks, we advocate for a novel approach to enhance existing OS-level provenance-tracking techniques by effectively tracking configuration changes and transparently integrating OS-level audit traces with application-level information.

This paper presents Dossier, a system that audits configuration changes in a target application and encodes the necessary information into OS-level audit logs. Dossier supports the main interfaces of configuration changes: changing configuration-related program variables in memory (typically through the user/admin interfaces) and changing configuration files on disk. To track memory-based configuration changes, Dossier performs novel static analysis on the application to identify configuration variables and then instruments part of the application to track changes to those variables. For file-based configuration changes, Dossier loads a custom kernel module to track when a specific set of system calls (`write`, `pwrite`, `writev`, `pwritev`, etc.) are used to modify an important configuration file. We designed a novel file delta computation algorithm that efficiently tracks changes to the configuration files and removes redundant log entries to significantly reduce space overhead. Finally, Dossier collects whole-system provenance logs via Linux Audit based on a widely agreed-upon set of rules for forensic analysis [50], [42]. Taken together, Dossier generates holistic provenance graphs that can not only reason about attacks that are captured by existing provenance trackers [50], [42], but also configuration-based attacks.

We demonstrate the effectiveness of Dossier through real-world case studies of configuration-based attacks and show how the system can be used in a forensic investigation to enable complete reconstruction of cyber attacks. We measure the runtime overhead Dossier incurs using a combination of microbenchmarks and representative workloads. Our results show that Dossier has a negligible runtime overhead on average ($<$7%) for moderate file-based configuration changes, whereas it adds 7.96% maximum runtime overhead for memory-based configuration changes over an insecure baseline that employs application-oblivious system-call logging.

This paper makes the following main contributions:

- **An analysis of configuration-based attacks in the real world (Section II):** We present the first characteristic study of real-world configuration-based attacks to understand their impact, spread, and permeation of these attacks. Our study shows the importance of defending against configuration-based attacks and highlights the motivation of tracking configuration changes.
- **Tracking file-based configuration changes (Section V):** We develop a light-weight kernel module that intercepts `write`-related system calls associated with configuration files and logs any changes of the file content over time. To address scalability issues, we designed an algorithm that computes file differences to determine which part of the file is changed and incrementally store the deltas.
- **Tracking memory-based configuration changes (Section IV):** We build a novel static analysis tool based on the LLVM compiler infrastructure to locate and instrument code snippets responsible for configuration updates in the target application. During application execution, the instrumentation enables the underlying audit subsystem to observe modification to configuration values.
- **Realistic evaluation (Section VII):** We evaluate the effectiveness of Dossier against 5 real-world application-based attack scenarios. Dossier captures all important, relevant configuration changes. To show the low overheads of Dossier, we measure its performance under representative workloads and benchmarks.

II. Background & Motivation

In this section, we first introduce several formal definitions which are required to understand the Dossier system, and then we provide a classification for configuration-based vulnerabilities. Finally, we present an attack scenario to highlight the limitations of existing work and motivate our work.

A. Definitions

Application configuration. We use the term *configuration* to refer to the inputs of an application program that controls the application program's runtime behavior, including feature sets, environments (e.g., data directory, key file, bind address, and remote hosts), policies (e.g., permissions and access control), resource allocation, etc. Therefore, configuration is critical, sensitive information of any application.

Configurations can be set in application's configuration files or be set through the application interfaces (e.g., CLIs) for users or sysadmins. Configuration files are typically loaded by the application at the startup time to initialize the application configuration state, while user/admin interfaces allow the application configuration to be changed dynamically at application runtime. Essentially, configuration values are loaded into the corresponding program variables in the application and are used during the program execution. We term those program variables *configuration variables*. Typically, each configuration variable has a default value defined in the application program. The default value will be overwritten by the values specified in the configuration file upon the application startup. Those values will be further updated when users or sysadmins set the configuration through application interface (e.g., CLIs) during the application runtime.

Configuration-based vulnerabilities. We use the term *configuration-based vulnerabilities* to refer to vulnerabilities rooted in or triggered by the configuration settings of the target

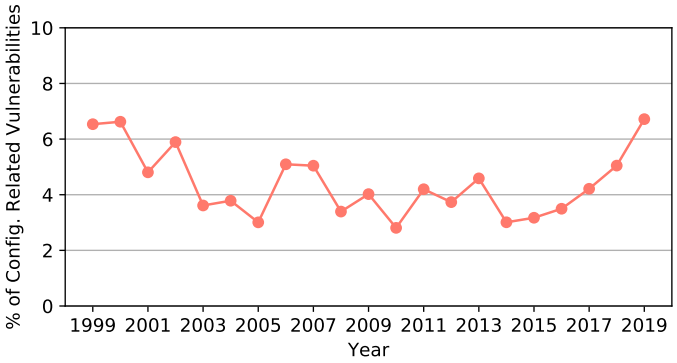

Fig. 1: The percentage of configuration-based vulnerabilities in the CVE dataset from 1999 to 2019.

applications. From a high level perspective, the implementation of security has two parts: the code and the configuration [60]. The code is the programs that security depends on, while the configuration is all the data that controls the operations of these programs. Even correctly-coded programs could be vulnerable when misconfigured; while code is written once, configuration is different for every installation.

By exploiting configuration-based vulnerabilities, attackers can launch devastating and disruptive attacks, termed *configuration-based attacks*. For example, in April 2019, a misconfiguration in Inmediata Health Group's website allowed leakage of 1.5 million individuals' data [18]. Even worse, the voting profiles of 154 million US citizens were leaked from a misconfigured instance of CouchDB running on Google Cloud services in 2016, exposing data that could have been abused for influencing elections and spreading targeted disinformation. Unsettlingly, network logs from this incident confirm that a suspicious IP address from Serbia had indeed accessed the database in the months prior to the misconfiguration's discovery. Numerous other recently reported data leaks can also be attributed to misconfigured servers, firewalls, and databases affecting a large number of individuals [29], [14], [21], [31], [23], [27], [2], [1], [82]. The potential impacts of configuration-based attacks on organizations underscore the importance of developing effective tools for their investigation.

Surprisingly, there is little analysis on configuration-based vulnerabilities, despite the grim headlines and postmortems. To mitigate that, we present our analysis of configuration-based vulnerabilities collected from public vulnerability databases. We first inspect the Common Vulnerabilities and Exposures (CVE) vulnerability dataset, which is composed of vulnerability records for the last 21 years [15]. We identified various configuration-related vulnerabilities using relevant keywords (e.g., `config`, `configuration`, etc.), then manually verified that each of the returned entries were configuration-based. Once we identified entries of interest, we calculated the percentage of configuration-related vulnerabilities every year for each category. The results are shown in Figure 1. On average, configuration vulnerabilities constitute 4.09% of reports in the past decade. The latter part of the trendline seems to indicate an upward rise in the percentage of cases observed with 6.8% (≈1500) configuration-based attacks in 2019 alone. Given their potential severity, the prevalence of configuration-based vulnerabilities warrants immediate and serious attention by the security community to design a holistic auditing framework that can reason about configuration-based attacks along with other attacks.

B. Categorizing Configuration-based Vulnerabilities

To gain a better sense of how and why configuration-based vulnerabilities occur, we conduct an additional survey of the National Vulnerability Database (NVD) [19]. The NVD is a public data source that maintains standardized information about reported software vulnerabilities managed by National Institute of Standards and Technology (NIST). Searching the keyword `config` returns 4418 entries from 1992 to 2018. Selecting the 100 most recent entries for investigation, we find that 65 out of the 100 are truly related to configuration issues, while the remaining 35 happened to have the keyword "config" as a software name, class name, etc. Analyzing the 65 shortlisted entries, we arrive at the following four broad categories for configuration-based vulnerabilities:

C1: The vulnerability is insecure default configuration. In this category, the vulnerability comes from insecure configurations which are set as the default values by the developers. As a result, the deployment of the program will be vulnerable if those default configurations are not changed. For example, the default `vhost` configuration in Puppet before v3.6.2 does not include the `SSLCARevocationCheck` directive, allowing remote attackers to obtain sensitive information via a revoked certificate when a Puppet master runs with Apache 2.4 [8].

C2: The vulnerability is exposed by legitimate configuration changes. In this category, the vulnerabilities are not in the configuration values but are caused by flaws in the software code. The vulnerabilities are accidentally exposed by legitimate configuration changes for various purposes, such as feature release and behavior changes. This class of vulnerabilities can be temporarily mitigated by changing back to the original configuration. However, for a more permanent and robust solution, the underlying software flaws must be fixed in a manner so that the program is secure for *all* possible configurations. For example, the `mod_http2` module in the Apache HTTP server 2.4.17 through 2.4.23 does not restrict `request-header` length when the protocol configuration includes `h2` or `h2c`, both of which are correct configuration values and legitimate options. However, in both these settings, the Apache server allows remote attackers to cause a denial of service (memory consumption) attack via crafted continuation frames in an HTTP/2 request [11].

C3: The vulnerability allows unauthorized configuration changes as a part of the attack vector. In this category, a vulnerability allows unauthorized modification of configuration parameters as part of the attack vector to compromise the target application. For example, several previous versions of MySQL allow local users to create arbitrary configurations and bypass certain protection mechanisms by setting `general_log_file` to a `my.cnf` configuration [9]. A detailed case study of this attack is presented in Section VII-B1

C4: The vulnerability exists because configuration inputs are not sanitized or properly parsed. In this category, a vulnerability leads to software compromises due to processing

TABLE I: The distribution of the 65 most recently reported configuration-based vulnerabilities in the NVD database We determine that categories marked with * can be effectively audited, and are thus the focus of our work.

Category of vulnerabilities	Distribution
C1 : Insecure default configuration	10 (14.9%)
C2*: Exposed by legitimate configuration changes	21 (31.3%)
C3*: Exploited via unauth. configuration changes	21 (31.3%)
C4*: Exploited via malcrafted configuration inputs	16 (23.9%)

of malcrafted configuration inputs. In most cases, this happens because of incorrect parsing, translation or validation of configuration inputs. For example, in Exponent CMS 2.x prior to 2.3.7, a bug exists in `/install/index.php`'s handling of configuration data (passed via the `sc` HTTP POST parameter) that allows an unauthenticated remote attacker to permanently inject arbitrary PHP code into `/framework/conf/config.php` and execute it with the privileges of the web server [26]. This attack is discussed in more detail in Section II-C.

Table I presents the distribution of these categories (C1–C4). Each category is characterized by its relationship to the software's configuration subsystem. C1 does not involve dynamic modifications of the configuration of deployed software, making auditing an indirect solution for defending against them; these issues could be addressed more effectively through static and dynamic software analysis and verification. In contrast, C2, C3, and C4 are all associated with discrete runtime events that can be conceivably audited and are hence, within the scope of this work (study and analysis of C1 is orthogonal to our work). For C2-based vulnerabilities, our work does not focus on pinpointing the root cause vulnerability inside the code, instead, we audit the configuration changes that serve as an indicator that the vulnerability was exploited. Together, C2, C3, and C4 constitute over 85% of the surveyed NVD vulnerabilities and make our analysis and findings applicable to the common case of configuration-based attacks.

C. Motivating Attack Scenario

We now consider an attack scenario consisting of both file-based and memory-based configuration changes, and then discuss the limitations of existing causality analysis techniques for investigating this intrusion. A provenance graph for the described attack is visualized in Figure 2.

Scenario: A company uses Exponent CMS [17] (a content management system) to manage its web content, allowing multiple contributors to create, edit, and publish. It also connects a Redis database to the content management system at the backend. The company relies on Linux Audit to collect system logs at the kernel level. Given the scale of the company, the configuration files of the CMS are updated and modified frequently. Unfortunately, an attacker discovers a critical file-based configuration vulnerability (CVE-2016-7790 [10]) in this version of Exponent CMS that allows them to inject malicious code into the program's configuration file [26]. The vulnerability exists within the sample `/install/index.php` script, which is not automatically deleted after the installation of the web application. The script, when processing user-input data passed via the "`sc`" HTTP POST parameter, allows the attacker to permanently inject malicious PHP code into `/framework/conf/config.php` with the following exploit:

```
<form
action="http://[host]/install/index.php"
method="post" name="main">
<input type="hidden" name='sc[","");
  passthru($_GET[]' value='"]); // '>
<input value="submit" id="btn" type="submit" />
</form>
```

After a successful PHP code injection attack, the attacker can execute arbitrary system commands via the web shell to gain control of the website, its databases, and the entire web server. Next, the attacker escalates their privilege by exploiting a vulnerability in the Redis database [25], which allows dynamic updates of program variables that store configuration values (stored in global struct instance named `server` with type `redisServer`). The ability of dynamically modifying Redis configuration values in memory enables the attacker to change the Redis database location to the `.ssh` directory using the `CONFIG SET DIR` command. After the attacker writes their own SSH keys into the new database location using the `CONFIG SET DB` command, they are able to remotely log in to the Redis server using their SSH key, which leads to privilege escalation. The incident is eventually detected and security analysts initiate the investigation.

Limitations of System Logs. We first consider the case in which the analyst makes exclusive use of the Linux Audit logs, enabling causal analysis at the syscall level as shown in past work, e.g., [42], [52], [59], [62], [63], [66], [72], [77]. These techniques are fairly useful in that they offer a broad view of the system activity. However, they suffer from a notable semantic gap – the captured logs lack descriptions of higher-level application behaviors that are often pivotal to attack reconstruction. In this particular case, analysts are able to discover the web shell that resulted from PHP code injection, and also that the Redis database was compromised. However, because the system logs do not specify the configuration fields that were modified in the file-based attack, the intelligence provided to the analysts is extremely coarse-grained. Additionally, these configuration files are subject to frequent legitimate updates, making it difficult to discern malicious updates from authorized ones. Finally, to make matters worse, in the dynamic memory-based attack on Redis, no events in the log describe the malicious dynamic configuration change, widening the semantic gap even further. In short, the system-level logs paint an incomplete picture of attacks that leverage configuration-based vulnerabilities.

Limitations of Application Event Logs. We now consider the possibility that, in addition to standard logging, the security team has deployed a framework that integrates syscall logs with application-layer events, as done in OmegaLog [53], Hercule [71] or UIScope [87]. OmegaLog [53] and Hercule [71] make use of applications' existing event-logging statements (e.g.,debug messages) to gain insight into higher-level runtime semantics without any instrumentation. Unfortunately, we surveyed the 18 applications used in [53] along with all the applications used in our work and found that none creates records of file-based or memory-based configuration change events. As a result, neither OmegaLog nor Hercule are able to merge the log streams together in such a way that sheds light on the attack. As UIScope's purpose is to associate user

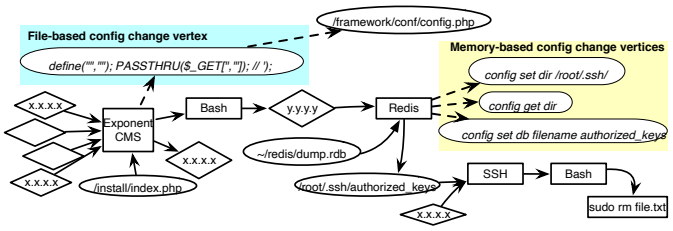

Fig. 2: The provenance graph of the motivating attack scenario described in Section II-C. In this graph we use boxes to represent processes, diamonds to represent sockets, oval nodes to represent files. Additionally, squashed rectangles with dashed edges denote configuration-specific threat intelligence that is exclusively provided by the Dossier system.

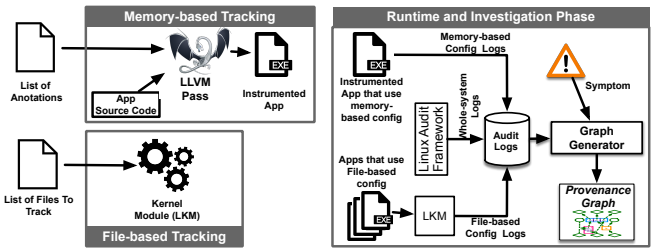

Fig. 3: Dossier architecture overview (Section III-C).

interface events with system logs, it could only assist if the configuration files were modified by a GUI, which is not common for server applications. As a result, application event analysis provides no added insight into the nature of this attack, specifically for the memory-based Redis compromise, which is still invisible to the analysts.

Limitations of Application Instrumentation. An alternate approach to bridging the semantic gap is to instrument applications to make high-level semantics visible to the underlying system log [62], [66]. For example, BEEP [62] and ProTracer [66] analyze and instrument programs to report the beginning of individual autonomous execution units, mitigating dependency explosion in long-lived programs. ProTracer [66] uses BEEP [62] as the basis for its execution partitioning, otherwise, it does not log any application-layer information. Because BEEP is event loop-based, it is poorly positioned to track file-based or memory-based changes to the application state. Rather than focusing exclusively on tagging event-handling loops, MPI [65] supports execution partitioning for a broader range of applications (e.g., browser tabs) through limited developer code annotations. However, MPI's analysis uses those annotations to instrument the high-level tasks present in the application to achieve execution partitioning and does not instrument or log any configuration updates during application execution. In short, these techniques have focused exclusively on resolving semantic gaps related to execution units and are not suited to application state tracking in particular configuration changes. While a broader range of analysis tools have been proposed to aid in application security [73], [46], their applicability to configuration auditing is unclear.

III. Dossier Overview

A. Threat Model and Assumptions

Based on our study of configuration-based vulnerabilities in Section II, this work, similar to previous work [42], [62], [66], [72], [77], [63], [52] on forensic analysis, considers an adversary attempting to exploit a software vulnerability, take control of the host, and/or maintain access to the system by malicious modification of software configurations. Beside these standard capabilities of a remote attacker, we also, focus on three capabilities permitted by configuration-based vulnerabilities: 1) The attacker exploits a software vulnerability after it is exposed by a configuration change made by the system administrators (C2). 2) The attacker bypasses or negates preventative security mechanisms by exploiting a vulnerability that allows them to modify software configurations and render the defenses useless or less effective (C3). 3) The attacker compromises software by providing malcrafted configuration data to vulnerable input handlers (C4).

We do not consider abuse of insecure default configurations (C1) in this work because such configuration vulnerabilities do not involve dynamic modifications of the configuration of an application (details in Section II). Additionally, we make the following assumptions: 1) We assume that a system-level causality tracker is running on the host. Like other work in this space (e.g., [52], [49], [42], [68]), we assume that the causality tracker is not compromised and that the audit logs are correct at the time of analysis. Audit framework compromises can be mitigated through system hardening [42] or detected through tamper-evident logging techniques [57], [69], [70]. 2) For detecting file-based configuration changes, we assume that the sysadmins are able to locate the configuration files of the application under analysis. This assumption is reasonable as prior work [82] suggests that sysadmins frequently change configuration files and are aware of the configuration file paths. Moreover, sysadmins can use automated configuration file mining tools [54] to identify the config files' locations. 3) For detecting memory-based configuration changes, we assume that the sysadmins have access to the source code of the target applications, which is required for the static LLVM analysis.

We do not consider hardware and kernel-level attacks or backdoors in this paper. Furthermore, we assume only explicit attacks, i.e., side or covert channel attacks are out of the scope of this paper.

B. Problem Statement and Design Goals

Our survey of prior work leads us to the problem statement: *Existing provenance collection and analysis systems are unable to encode static and dynamic configuration changes of target applications into the underlying whole-system provenance graph, preventing the investigation of configuration-based attacks in a causally-correct manner.* Addressing this problem is challenging for the following reasons:

- Configuration changes often happen dynamically (i.e., in memory) and are thus not visible to system-level provenances tracker that can only monitor writes to on-disk application files. To log modifications in memory, we require a generic means of tracking changes from inside of the application's internal state.
- Even when configuration changes cause disk operations, existing audit and provenance frameworks are still insufficient

because the data buffer argument of the syscall is typically not logged. Thus, while it may be possible to determine that a file containing configuration was changed, the log will not reveal the specific fields modified. To make matters worse, due to the high frequency of write syscalls, it is also common to avoid overheads by disabling their auditing altogether.

- Even after logging the configuration changes on the system, it is challenging to integrate this information with existing auditing streams in a semantically- and causally-valid way. In particular, it is unclear how to associate configuration update events with system-layer audit records in a manner that preserves the correctness of causal graphs.

With these challenges in mind, we set out to design a system that provides: 1) precise identification of changes to configuration state at the field-granularity; 2) correct association of configuration changes to external system events, such as attributing the change to a specific user or process; 3) support for fine-grained detection of configuration changes regardless of the update mechanism, be it file- or memory-based; 4) highly efficient configuration auditing that avoids prohibitively costly runtime or log storage overheads.

C. Our Approach

To achieve these goals, we present Dossier, a framework for configuration auditing. An overview of the Dossier architecture is shown in Figure 3. For file-based configuration tracking (Section V), Dossier includes a kernel module that takes as input a list of sensitive configuration files to monitor for content changes, which facilitates selective interception of `write` and other relevant system calls. To allow for memory-based configuration tracking (Section IV), the user first annotates configuration variables in the program. Then, our LLVM-based analysis uses the annotations to determine the placement of instrumentation hooks to emit configuration change log events. Finally (Section VI), Dossier combines the configuration logs with the underlying system logs to create holistic provenance graphs that link configuration events to system events in a causally-correct manner, facilitating precise investigation of configuration-based attacks. A provenance graph generated by Dossier is shown in Figure 2; squashed rectangle vertices provide evidence that would not have been available using previous approaches. The integration of configuration-related application telemetry with system logs allows holistic provenance generation and enables forensic investigators to bridge the semantic gap by effectively piecing different attack fragments together. Prior research has also shown the benefit of integrating system logs and application telemetry (e.g., [53], [71], [87]), and even execution partitioning solutions [65], [66] extract coarse-grained application telemetry to improve investigations. While we motivate our approach through configuration-based attacks, we note that Dossier is a generic tool for auditing application state information maintained in program variables. Dossier is among the first attempts to capture application state change information and consider them in provenance tracking. We hope that this work will spur further research into application state auditing.

IV. Tracking Memory-based Config Changes

As discussed in Section II-C, certain applications such as Redis, NGINX Unit[1], and Bind [5] dynamically update configuration values at the memory locations during runtime without invoking any system calls. Thus, to track memory-based configuration changes, Dossier needs to log the target value whenever it is updated in the memory locations. To track such configuration changes, we designed an automated toolchain that first statically analyzes the target application to identify and mark instructions that update *configuration variables*—program variables that store configuration values—in the entire application. After that our toolchain instruments the application at those marked locations with the logging statements to disclose configuration changes to the underlying audit subsystem.

Upon a dyanmic update, our logging statements record the changed configuration values along with other important meta-information (e.g., timestamps and PID) required for provenance graph integration. We build the instrumentation using the LLVM compiler [61] in which our instrumentation is a *transformation pass* on LLVM bitcode of the target application. We generate target application bitcode through WLLVM [38]. WLLVM provides tools for building whole-program (or whole-library) LLVM bitcode files from an unmodified C/C++ source package. Our toolchain consists of three phases, which we describe in the next few subsections.

A. Annotating Application Configuration Variables

During the first phase, Dossier requires the sysadmins to annotate configuration variables that store configuration values. Optionally, Dossier also allows the sysadmins to specify variable types as annotations. Dossier takes those types as inputs and annotates all the variables that match the specified types. Following prior studies [85], [83], [78], [44] on configuration analysis, we assume that mature software projects use unified data structures or APIs for managing configurations. Such APIs make it easier for end-users to locate and annotate all the configuration variables present in the project. We confirmed this observation in our work by analyzing Redis, Bind9, and NGINX Unit. It took 2–8 lines to annotate those three applications. A large-scale analysis can be found in [85], [78] for C/C++ and Java programs, respectively. Overall, it takes on average seven lines of annotation across the projects in prior studies. Our numbers are consistent with those. Similarly, locating configuration-related structures/APIs is also straightforward. It took one of our authors around 20 minutes to identify the data structures in the Redis code. Note that this annotation is a one-time effort.

Fully automatic solution for identifying configuration variables is an area of research [48], [90]. We note that those studies are orthogonal to our work. We will incorporate the proposed automated approaches as our future work.

B. Static Analysis

Using sysadmins provided annotations, Dossier identifies all the locations in the application where the configuration variables are defined or updated. For such identification, Dossier

[1]NGINX Unit is a recent version of NGINX webserver that introduced dynamic updates and many other new features.

locates all the STORE instructions in the bitcode and gets all the possible locations in the program where variables are updated. Next we find out if those STORE instructions update annotated configuration variables, so that we can mark those instructions for instrumentation in the final phase of our toolchain. To figure out that, we extract destination operands from those STORE instructions. We have three different types of destination operands in the STORE instructions and we handle each one differently, as described below.

1. Variable Destination Operand: When the destination of the STORE instruction is a variable, it is fairly straightforward to figure out if that destination belongs to an annotated configuration variable by simply comparing the name (Lines 6–7 in Algorithm 1). The following bitcode snippet, is an example of such an instruction,

```
1  store i32 90, i32* @port, align 4, !tbaa !1
```

In this example, the port variable is annotated as a configuration variable; Dossier identifies the destination operand of the STORE instruction to be port. Therefore, it marks this STORE instruction for further instrumentation in the final phase.

2. GetElementPtr Instruction Destination Operand: GEP instructions are pointers to the memory locations of certain variables [28]. When the STORE instruction destination is a GEP instruction, Dossier queries the Program Assignment Graph (PAG) of the target application to figure out if those destinations belong to annotated configuration variables. PAG is a graph representation of LLVM bitcode where each value and instruction in the bitcode is mapped into a PAGNode or PAGEdge. These PAGEdges represent constraints between pointers. We use SVF tool [37], [76] to generate PAG from the LLVM bitcode. A GEP instruction is represented as a GEP edge in the PAG, where it flows from a source to a destination (the source is the memory location that it points to). Dossier generates a PAGNode using the destination value of the STORE instruction (Line 5 – Line 4 in Algorithm 1). If the destination value is a GEP instruction, a GEP edge will flow into the PAGNode (Line 10 in Algorithm 1). Next, Dossier checks if the source of the GEP edge is an annotated configuration variable. If so, the STORE instruction updates the value of a configuration variable and should be marked for further instrumentation in the next phase (Line 18 in Algorithm 1). Since the PAG is field-sensitive, it also includes the offset of the GEP instruction, which helps mark and identify specific offsets of the configuration variables.

Additionally, Dossier uses the offset value recorded in the GEP edge to calculate the precise location of the configuration variable (Lines 14–16 in Algorithm 1) to achieve field sensitivity. For example, consider the following source code:

```
1  struct server { int port; };
2  struct server config;
3  config.port = 90;
```

The above code snippet leads to the following bitcode:

```
1  store i32 90, i32* getelementptr inbounds
2  (%struct.server,%struct.server* @conf,i64 0,i32 0),
3  align 4, !tbaa !1
```

The annotated configuration variable is conf with the struct type server. When the port offset of conf is updated, it is done via GEP instruction in bitcode. The STORE instruction above, uses the GEP instruction to directly update the offset port of config. Our approach marks such a case using the PAGNode of the GEP destination operand and its GEP edge in the PAG for further instrumentation in the final phase of our toolchain to log configuration changes.

3. Pointer Alias Destination Operand: Different variables could be stored at the address of the configuration variable, which is later dereferenced to use the value that it points to. To handle these cases, we leverage Andersen Pointer Analysis [40] from SVF. Andersen's pointer analysis is a context-insensitive, inter-procedural analysis for obtaining pointers to variables. We used the MemorySSA (Static Single Assignment form of a program's variables) generated by the Andersen Pointer Analysis, which allows us to reason about the interactions between various memory operations and find all PAGNodes in the PAG that are aliases to another variable. The discovered PAGNodes in this step, subsequently allow us to find the aliases to annotated configuration variables. Dossier marks those instructions which update these pointer aliases by checking if the destination value is an alias of configuration variables (Lines 8–9 in Algorithm 1). This provides completeness as Dossier marks and identifies all STORE instruction with the destination as the pointer to the address of configuration variables as well as values from configuration variables. Consider the example:

```
1  int * conn_ptr = &curr_conn;
2  (*conn_ptr)++;
```

The corresponding bitcode uses pointer alias as destination operand in STORE instruction:

```
1  %1 = load i32, i32* @curr_conn, align 4, !tbaa !1
2  %2 = add nsw i32 %1, 1
3  store i32 %2, i32* @curr_conn, align 4, !tbaa !1
```

conn_ptr points to an annotated configuration variable curr_conn (Line 1) and updates it's value on the next line. Within the bitcode the address of curr_conn will be loaded into a register (Line 1) and the update to the pointed address is performed through the STORE instruction (Line 3). Pointer aliasing will cater to such a case and mark the STORE instruction which indirectly updates the annotated configuration variable for further instrumentation in the next phase.

C. Logging Instrumentation

Dossier inserts a logging statement after each Dossier-marked STORE instruction from the static analysis phase. The logging statement records the value of the changed configuration. Dossier includes utility functions for type casting and uses Linux auditd logging function audit_log_user_message. This function uses the running audit daemon, which avoids additional overhead and writes configuration change logs along with the whole-system logs.

Function Specialization Dossier's instrumentation is pre-equipped with an optional function specification optimization to minimize runtime overhead of tracking configuration changes, while maintaining precision. This is based on the observation that mature systems such as NGINX Unit, Redis and Bind all maintain dedicated functions for handling

Algorithm 1: Mark `STORE` instructions

Global Variables:
`AnderPTA`: Anderson Pointer Analysis
PAG: PAG obtained from `AnderPTA`
Input :
`allStoreInst`: Set of all `STORE` instructions in the program
`configVars`: Known configuration variables
`offset`: Offset of the structure of `configVars` that points to the field
Output:
`S`: Set of `STORE` instructions that update the target configuration variables

```
1  Function FilterStores():
2      S ← ∅;
3      foreach STORE instruction i ∈ allStoreInst do
4          dest ← i.getpointerOparand(); ;
5          PAGNode ← PAG.getPAGNode(dest); ;
6          if dest ∈ configVars then
7              S.insert(i);
8          else if AnderPTA.isAlias(dest,configVars)==MAY_ALIAS
             then
9              S.insert(i);
10         else if PAGNode has an incoming normalGEPEdge then
11             foreach edge ∈ PAGNode.incomingEdges() do
12                 if edge == normalGEPEdge then
13                     if edge.source ∈ configVars then
14                         if offSet is specified then
15                             if edge.getOffset() == offset
                                 then
16                                 S.insert(i);
17                         else
18                             S.insert(i);
19     return S;
```

dynamic updates. Given a user-provided function, this specialization pass only considers `STORE` instructions that are part of provided function or any function in the call tree of the provided function. The approach minimizes overhead by reducing the number of identified `STORE` instructions as configuration updates. For instance, function specialization for `nxt_router_conf_create` function in NGINX Unit server reduces the identified configuration update `STORE` instructions (with pointer aliasing) from 3054 to 90, which decreases the performance overhead significantly.

D. Putting It All Together

We demonstrate our static analysis technique using two open-source applications, Redis and NGNIX Unit, that provide interfaces for updating configuration dynamically. We discuss another application in Appendix A. We select these applications for their relevance, popularity, and wide-scale usage.

Redis: Redis [24] is an in-memory data store. The configuration values for Redis are stored in a global struct instance named `server` with type `redisServer`. Redis allows dynamic updates of these configuration values at runtime, as discussed in Section II-C. For example, configuration commands `CONFIG SET DIR` and `CONFIG SET DB`, respectively update the directory and DB name dynamically. In the underlying code, these commands update the values of variables `rdb_filename` and `rdb_save_time_start`, present in the struct `server` at offsets 168 and 174 respectively. To evaluate Dossier, we annotated the global variable `server` and the struct offsets, 168 and 174. We then ran our static analysis. Our static analysis results for Redis are discussed in Table V. We were able to successfully mark the relevant `STORE` instructions. We also simulated the Redis attack explained in Section II-C and successfully logged the configuration updates on the target variables (annotation provided by the sysadmins), `rdb_filename` and `rdb_save_time_start` in the `server` struct.

TABLE II: The arguments extracted from different intercepted system calls, writing to the target configuration files.

System Call	Arguments Extracted	Description
`write`	`buffer`	The contents of the buffer are extracted from the system call arguments.
`pwrite`	`buffer & offset`	The contents of the buffer as well as the offset is extracted from the system call arguments
`writev`	`iovec vec`	The iov array present in the system call arguments is iterated in order and buffer from each entry is extracted.
`pwritev`	`iovec vector & offset`	The buffer in each iovec entry is extracted in order along with the offset present in the system call arguments.
`sendfile`	`src_fd & offset`	The contents of source file are extracted given the file descriptor and the starting offset from the system call arguments.

NGINX Unit: NGINX Unit [20] also allows dynamic configuration changes; however, NGINX Unit configuration change handling methodology is much more complex than Redis. Each new request to NGINX Unit is assigned a configuration object. Upon a configuration change, the NGINX Unit uses a new configuration object for all the subsequent requests. Due to this added complexity, Dossier uses the structure type `nxt_socket_conf_t` instead of variable names to annotate configuration variables. Once the annotations are provided, Dossier uses static analysis to mark and instrument configuration variables. To filter redundant instructions, Dossier applied function specialization for the function `nxt_router_conf_create`. The results of our static analysis under different configurations for NGINX Unit are discussed in Table V.

V. Tracking File-based Config Changes

One way to update the application's configuration is to change the configuration files on the disk. To keep track of such changes, we need to record all the content written to each configuration file. To this end, we propose a light-weight Linux kernel module (LKM) that keeps track of content changes to configuration files. Our LKM intercepts all the file-related syscalls capable of updating configuration files on disk.[2]

Our workflow of capturing file-based configuration changes is as follows: 1) Our LKM takes a list of paths to the configuration files as an input. 2) Our LKM hooks the following syscalls: `write`, `pwrite`, `writev`, `pwritev`, and `sendfile`. 3) Every time one of those syscalls is invoked, our LKM checks the file handler argument to see if it is writing to a configuration file (from step 1). If yes, it proceeds to the next step for logging. Otherwise, it returns control to the syscall handler. 4) Our LKM then extracts the contents of the file update using the syscall arguments. Table II shows what types of syscalls we intercept and how we extract content from the arguments of those system calls. Our LKM also extracts relevant process information (PIDs, PPID, etc.) from the current `task_struct`, which is later used to stitch the config logs to the system audit logs at the causally-correct places, facilitating attack investigations. 6) Finally, our LKM writes

[2]We intercept a range of different `write`-related calls because certain programs make use of these to update configuration files. For example, SQLite3 has an option to replace `seek+write` with `pwrite` as it reduces the overhead by replacing two syscalls with one. Similarly, `writev`, `pwritev`, and `sendfile` have their own advantages over `write` in certain cases.

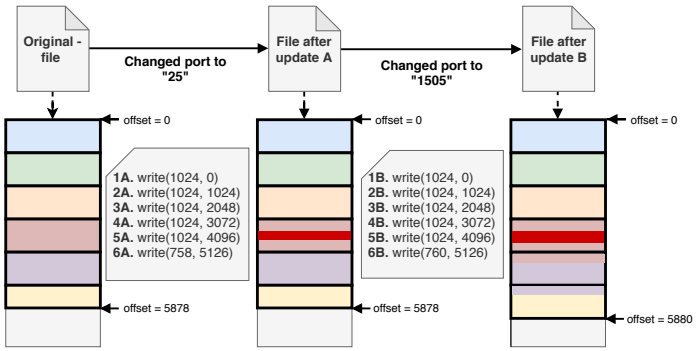

Fig. 4: Visualisation of port value update in config file and corresponding write syscalls in Exponent CMS

this information to audit logs, using Linux Audit API function call `audit_log()`.

For brevity, in this section, we will only discuss how we handle `write` syscall since we handle other syscalls in a similar way. Note that in Section VII, we evaluate and benchmark Dossier, enabling all the five syscall hooks in our LKM.

A. Computation of File Deltas

During our experiments, we were able to log configuration value changes using `write` syscall interception. However, we found out that some applications update the entire configuration file even if they are changing a small part of the file. The entire file is rewritten in fixed-size chunks during the update and the size of these fixed chunks varies for different applications, typically ranging from 1kB to 16kB. Due to this problem, our initial approach of intercepting `write` syscall and extracting buffer will wrongly assert that the entire file was changed even though only part of file was updated, leading to incorrect results during forensic analysis. Further, this approach incurs unnecessary space overhead by storing the entire file's contents rather than just the changed field.

To elaborate on this problem, consider a scenario shown in Figure 4, where the port value was changed two times in the configuration file of Exponent CMS [17]. We call the first configuration update `A` and the second update `B`. The initial size of the file is approximately 5.74kB and this file is rewritten in chunks of 1kB during update. In the figure, the first argument of the `write` syscall denotes the buffer size, and the second argument denotes the starting offset. During update `A`, the port is changed from `"62"` to `"25"` and the application rewrites the entire file using a set of 6 `write` syscalls (`1A` – `6A`). Each syscall overwrites a specific chunk within the file. Since the new port value uses the same number of characters as in the previous port value, the size of the file remains the same, with the actual port update occurring in `4A`. Note that all `write` syscalls for update `A` overwrote the old data with the same contents except write number `4A`.

During the update `B`, the port value was changed from `"25"` to `"1505"` using a set of 6 `write` syscalls (`1B` – `6B`). In this update, since we added two new characters in the port value, the final file size was increased by 2 Bytes – all the `write` syscalls rewrite the previous data until the `4B` syscall. The `4B` syscall updates the port value to `1505`. Due to an addition of 2 bytes, a ripple effect takes places and 2 Bytes from the end

Algorithm 2: Compute deltas

Input :
`rawLogs`: Linux `auditd` logs
`initFile`: Initial file snapshot before any file updates
`clusterInterval`: Interval of `audit_timestamp` to cluster logs entries
Output:
`outputLogs`: Final logs with entries containing file deltas

```
1  Function ComputeDeltas():
2      List dataLogs ← rawLogs Audit log entries logged by Dossier ;
3      Set alreadyClustered ← ∅;
4      List clusters ← [] ;
5      foreach l1 ∈ dataLogs do
6          if l1 ∈ alreadyClustered then
7              continue; // Ignore l1, as it is already clustered
8          List curCluster ← [];
9          curCluster.append(l1);
10         foreach l2 ∈ dataLogs ∧ l2 ∉ alreadyClustered do
11             if timestamp(l2)-timestamp(l1) == clusterInterval
                 then
12                 curCluster.append(l2);
13                 alreadyClustered.append(l2);
14         clusters.append(curCluster) ;
15     sort(clusters); // Sort cluster entries based on audit_timestamps;
16     List newEntries ← [];
17     foreach c ∈ clusters do
18         // Take clusters and get new log entry for each cluster;
19         newEntry,newFile ← getNewLogEntry(initFile, c);
20         newEntries.append(newEntry) ;
21         initFile ← newFile ;
22     List otherLogs ← rawLogs // Log entries not collected by Dossier ;
23     List outputLogs ← otherLogs+newEntries ;
24     return outputLogs;
```

of the 4th chunk move to the start of the 5th chunk. Similarly, 2 bytes from the end of the 5th chunk move to the start of the 6th chunk as shown in Figure 4. Hence, the `5B` and the `6B` syscalls do not rewrite the same data anymore.

We observed two things in this example: 1) We can not compute the correct file delta by comparing the buffer of two `write` syscalls writing on the same file offsets. For instance, if we compare `5A` with `5B` because they write to the same file offsets, we will reach the incorrect conclusion that 2 bytes were added to the front of the data and 2 bytes were removed from the end following the `5B` update. In reality, neither conclusion is true. 2) We also can not compute the correct file delta by comparing the complete file state between two subsequent `write` syscalls. Returning to update `B`, if we compute the file delta between the `4B` and the `5B` syscalls, we will again reach the incorrect conclusion that 2 new bytes were added and two new bytes were removed from the 5th chunk of the file. Both of these problems arise due to the ripple effect. From the examples above, we can see that we only need to compare the first and the last state of the complete file during an update to compute the correct file delta. Based on this observation, we developed an algorithm that clusters syscalls together to compute accurate file deltas and remove unnecessary logs.

Algorithm. We assume an initial snapshot of the configuration files before loading our LKM. Once our LKM is loaded, it extracts the file offset along with the buffer for each `write` syscall on the target configuration files, and stores it as a write syscall entry in the audit logs. We make use of both the audit logs and the initial configuration files to compute the file deltas. Our file delta approach is outlined in Algorithms 2 and 3.

First, we parse the audit logs in an offline fashion and find all the write syscall entries logged by Dossier for a given configuration file (Algorithm 2, Line 2). Next, we cluster

Algorithm 3: Get a new log entry from a log cluster

```
Input :
cluster: List of clustered audit log entries
initFile: Initial file snapshot before any cluster update
Output:
outputEntry: Final log entry with the correct file delta
newFile: New file version created after writes from cluster
1  Function GetNewLogEntry():
2      newFile ← initFile;
3      foreach c ∈ cluster do
4          offset ← c; // "offset" field value ;
5          buffer ← c; // "buffer" field value ;
6          // Update newFile by inserting buffer at specific offset and
             overwriting previous data;
7          newFile ←
           newFile[:offset]+buffer+newFile[len(buf)+offset:] ;
8      // Compute delta between newFile and initFile and store in newBuf;
9      newBuf ← delta(initFile, newFile) ;
10     // Create new log entry with newBuf for the buffer field ;
11     outputEntry ← cluster[0] with buffer=newBuf;
12     return (outputEntry, newFile);
```

all the syscall entries for a single file update together. We observed that all the consecutive `write` syscalls for a single file update exhibit similar timestamps, so we cluster the all the syscall entries with timestamps that fall within a time period (Algorithm 2, Lines 4– 14). This time period is a configuration parameter in our system and defined as the `ClusterInterval`. In our implementation, we use `0` for `ClusterInterval` value because in our experiments all the consecutive write syscalls for a single update exhibited the same audit timestamp.

Once we have reduced the syscall entries to a list of clusters, we sort all the clusters (Algorithm 2, Line 15) and process them according to the Algorithm 3. Given a cluster, we generate the corresponding new log entry containing only the file delta. For this purpose, we service all `write` syscalls present within the cluster, in order, on the initial snapshot of the configuration file (Algorithm 3, Lines 3– 7). This provides us with a new file containing all the writes for a given file update. The simulation of write syscalls is possible as each syscall entry contains both the buffer and the offset. We then use python's `difflib` library to compute the delta between the initial file and the newly updated file (Algorithm 3, Line 9). Finally, this delta is used to create a new syscall entry with buffer equal to computed delta and timestamp equal to the first cluster entry. The new configuration file of the processed cluster servers as an initial file for the next cluster and so on until all the clusters are processed (Algorithm 2, Line 21). In this way, we effectively reduced all the write syscall entries into a handful of new syscall entries containing the exact file deltas. Lastlt, we remove the previously logged syscall entries from the audit logs and replace them with the new syscall entries (Algorithm 2, Lines 22– 23).

The computation overhead of our technique is not a concern because the deltas computation is an offline process. A sysadmin can compute the file deltas using our technique on demand (e.g., before conducting an investigation) or periodically to reduce storage overhead of the logs. The algorithm's runtime complexity is O(N), which is acceptable for offline use cases.

Example. To better understand the Algorithm 3, consider an example where we perform two updates at separate intervals on a 32kB file. During each update, the file is rewritten in chunks of 4kB using 8 write syscalls. This results in a total of 16 write syscalls. During the offline parsing of logs, we will generate 2 clusters of 8 syscalls each based on the audit timestamps. For each cluster, we perform all the 8 `write` syscalls on the initial file version. We then compute the differences between the initial and the updated file version of each cluster. As a result, we effectively replaced the initial 16 write syscalls with 2 syscalls containing the exact file deltas. These 2 entries will contain the exact changes in file content after an update without any repititions. Therefore, computation of accurate deltas significantly reduces the space overhead of our logging. Here, the reduction of storage overhead is an added benefit but not the goal of our file-delta computation algorithm.

VI. Runtime and Investigation Phases

A deployment of Dossier includes both the kernel module for tracking file-based configuration changes (Section V) and application instrumentation for tracking memory-based configuration changes (Section IV). The kernel module is deployed once in the OS and works for all the target applications running on the OS. The instrumentation needs to be applied to every target application. At runtime, Dossier logs both file- and memory-based configuration changes along with the whole-system provenance logs.

During the investigation phase, the analyst can use Dossier to generate holistic provenance graphs given an Indicator of Compromise (IoC). This holistic graph is generated by parsing logs that contains both whole-system provenance and configuration change logs collected by Dossier. We modified SPADE's provenance graph generation system [50] to generate holistic provenance graphs. SPADE can parse whole-system provenance logs generated by Linux Auditd into provenance graphs. We added capability in the SPADE system to also handle configuration change logs. As described in Section V and Section IV, Dossier also collects meta-information, such PID and timestamp and appends this meta-information to configuration change logs during runtime. Such meta-information allows us to figure out which process vertex is responsible for changing the configuration values. Thus, during holistic provenance graph generation, Dossier uses this meta-information to determine the candidate *process vertex* to attach the *configuration change vertex*.

VII. Evaluation

In this section, we focus on evaluating Dossier's performance and efficacy as a specialized provenance tracker. In particular, we first investigated the runtime overhead of Dossier's LKM. Second, we evaluated the runtime overhead of the application instrumentation introduced by Dossier. Finally, we measured the effectiveness of Dossier in reasoning about real-world configuration-based attacks. We collected whole-system provenance logs using the Linux Auditd framework. All experiments were performed on an Ubuntu 18.04 LTS VM with 32Gb memory and eight CPU cores.

A. Runtime Overhead of Dossier

The overall runtime overhead of Dossier comprises of delays from two components: 1) overhead due to our LKM to track file-based configuration changes and 2) overhead due

to application instrumentation to track memory-based configuration changes. From the design of the system, it is evident that the file-based overhead is a function of two parameters: the overhead incurred due to the hooking mechanism to trap *all* write system calls (irrespective of the file type) and the additional overhead caused by the processing required to service a few system calls specific to configuration files (we call these "configuration file writes"). To quantify the former, we use custom workloads and stress tests that allow us to measure the overhead with various frequencies of file writes with and without our LKM. To measure the overhead of the latter, we again used our custom workload along with three popular microbenchmarks from various domains. Similarly, the memory-based overhead is a function of another two parameters: the overhead caused due to pointer aliasing and the degree of function specialization.

1) Overhead of file-based tracking: To evaluate the impact of Dossier's hooking mechanism, we implemented a customized stress test that initiates write syscalls on different files. We measured the time overhead with and without using our LKM. We also varied the number of write syscalls (100K–900K) to account for different workloads. For each run, we initiate write syscalls and then we take the average to calculate overhead per syscall. A comparison between the blue and the green lines in Figure 5 shows the hooking overhead results for this experiment. Our performance overhead for syscall hooking mechanism is approximately 6-10 ns/op (an approximate 3% increase). The main takeaway from the stress test is that the overhead of the hooking mechanism stays under 5% under stress and is acceptable.

We also measured the overhead of making write syscalls specifically on configuration files (on which Dossier performs some extra processing). We repeated the customized stress testing, but this time we made 1000 write syscalls to configuration files. Those 1000 write syscalls went through the extra processing and were eventually recorded in the Linux audit logs. We evaluated against 1000 write syscall to configuration files because previous study [82] has shown that 99.3% of configuration file updates of popular applications happen less than 1000 times over their lifetime. Thus, for each run, we initiated 1000 write syscalls to configuration files and varied the number of write syscalls to non-configuration files. After that we take the average to calculate overhead per syscall. Again, we also varied the number of write syscalls (100K–900K) to account for different workloads. A comparison between the red and blue lines in Figure 5 shows the configuration writes overhead results for our experiment. We noticed that increasing the percentage of configuration writes also increases the runtime overhead with the maximum overhead of 7% for 100K syscalls (10% configuration writes). Note that for most workloads, the majority of write syscalls are for non-configuration files. Hence, for an application with 500K+ writes, Dossier incurred a maximum configuration write percentage overhead of 4% for the common case. For 900K writes, this percentage overhead was further reduced to 1%.

To further evaluate the overhead, we used the LMBench microbenchmark [34]. We ran LMBench five times without loading our LKM. We took an average of the five runs to get a baseline to compare our overhead against. We then loaded Dossier's LKM and enabled all the hooks pertaining

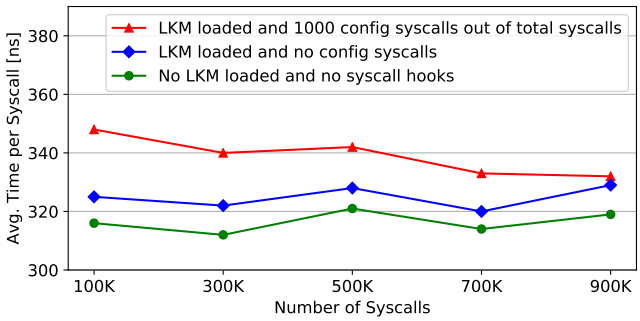

Fig. 5: Runtime overhead of Dossier's file-based configuration tracking approach under different scenarios.

TABLE III: LMBench measurements for Dossier. All times are in microseconds. Percent overheads are shown in parenthesis.

Test Type	Vanilla	Dossier
File and memory latencies in μ seconds (smaller is better)		
File create (0k)	878.56	885.82 (1%)
File delete (0k)	655.72	652.56 (-0.5%)
File create (10k)	904.3	950.46 (5%)
File delete (10k)	641.82	648.76 (1%)
mmap latency	6101.8	6179.2(1%)

to write syscalls, including `write`, `pwrite`, `writev`, `pwritev`, and `sendfile`, and then ran LMBench five times. We also ran 1000 configuration file writes while kernel was loaded with all hooks enabled. The results of the two experiments are shown in Table III, which summarizes the low overhead of the hooking mechanism. The only noticeable overhead ($\approx 5\%$) observed was that, on average, the 10K `create` time increased from 904.3 microseconds to 950.46 microseconds.

We also measured the overhead using the Postmark benchmark [58] and the BLAST benchmark [6]. For both these benchmarks, each test was repeated 10 times to ensure consistency and minimize the effects of background noise. The Postmark test simulates the workload of an email server. It was configured to run 30,000 transactions with file sizes ranging from 1KB to 2MB in 10 sub-directories, with up to 3,000 simultaneous transactions. Similar to LMBench, we ran 1000 configuration file writes while kernel was loaded with all hooks enabled. The overhead incurred by Dossier for the Postmark test is shown in Table IV and only amounts to around 2%. Furthermore, to estimate Dossier's performance for scientific applications, we ran the BLAST benchmarks. The BLAST benchmark simulates typical biological sequencing workloads. As shown in Table IV, BLAST benchmarks impose almost no additional overhead implying that Dossier has little to no effect on computation-intensive jobs.

2) Overhead of memory-based tracking.: We now measure the runtime overhead of tracking memory-based configuration changes. As mentioned above, the overhead of the application instrumentation is determined by two parameters: 1) pointer aliasing – covers more `STORE` instructions due to alias capturing and 2) the degree of specialization – optimizes the performance of the instrumentation. We tested the overhead of application instrumentation using Redis and NGINX Unit server under previously mentioned parameters as shown in Table V. For Redis, we used Redis's official benchmark to measure the runtime overhead. The percentage overhead was calculated

TABLE IV: Postmark and BLAST benchmarking results for Dossier.

Test	Vanilla (s)	Dossier (s)	Overhead
Postmark	70.6	72.2	2.27%
Blast	236.21	235.49	-0.3%

based on the time taken to complete 100K query requests for the `SET` method. For annotation, we specified the global struct variable `server` along with the offsets (168, 174) to be tainted. We then ran the taint analysis with different configurations—with and without offset specification. Table V shows the results of our overhead experiments. In most cases, the runtime overhead is under 2%. The maximum runtime overhead is 7.96% in our experiments.

Similarly, we measured the runtime overhead for the NGINX Unit server using ApacheBench [3]. We initiated a total of 1 million HTTP requests with 100 concurrent requests at a given time. The percentage overhead was calculated based on the time taken in milliseconds per request. For annotation, we specified the type of structure as `nxt_socket_conf_t`, that is, we annotate all the variables with the specified type. We then ran static analysis under different configurations. Table V shows our results for NGINX Unit experiments. We can see that the percentage overhead is minimal with a maximum overhead increase of 2.22%.

We also logged the number of lines instrumented against each configuration in Table V. After a comprehensive manual review, we were able to identify the correct number of `STORES` that should have been instrumented by Dossier. This allowed us to calculate the true positives, false negatives, and false positives of the automatic instrumentation process. Based on this information, we calculated the precision and recall of main-memory based approach of Dossier against different configurations. However, for one configuration of Redis and NGINX Unit each, shown in row 5 and 6 of Table V, we were unable to manually identify the false negatives and recall, due to the large scale of the programs (16.9K and 4.25K `STORES` respectively). The true positives, false positives and precision were calculated for these case as we just had to manually evaluate instrumented `STORE` instructions.

Our results show that offset specification significantly reduces the marked `STORES` and the percentage overhead. We observe this behavior in the case of Redis as shown in Table V. We also found that recall is slightly impacted without aliasing because of missing alias updates, but we gain high precision because there are no false positives. On the other hand, with aliasing enabled, recall significantly improves, but precision is compromised because some alias instructions are incorrectly instrumented. These results highlight the inherent limitations of underlying static pointer analysis tools. Even with state-of-the-art static analysis tools utilized by Dossier (i.e., SVF [37], [76]), the correctness of the analysis is impacted due to the imprecision during the pointer analysis phase. Dossier aims to cater to this problem by providing sysadmins with optimizations, such as function specialization and offset specification, to reduce the number of false positives with aliasing enabled. Moreover, sysadmins have the option to disable pointer aliasing altogether at the cost of completeness and still get reasonable recall values with high precision as shown in Table V.

B. Effectiveness on Attack Forensics

We present three case studies to show the effectiveness of Dossier in forensics for real-world configuration-based attacks. We present another case study in Appendix B.

1) MySQL Attack: For this case study, we chose CVE-2016-6662 [9] vulnerability. This vulnerability allows attackers to remotely inject malicious settings into MySQL configuration files *(my.cnf)*, leading to serious consequences. In particular, upon server restart, some parts of the *my.cnf* configuration file are loaded with root privileges (such as the *mysqld_safe* wrapper script, which is executed as root) and an attacker can modify these values and parameters to point to their malicious code or allow them to preload a malicious shared library through the `--malloc-lib` parameter (which could have a TCP-based reverse shell payload). In such a scenario, successful exploitation can escalate an adversary's privileges giving them access to a remote shell with root permissions.[3]

We now consider a scenario where a company uses vulnerable version of MySQL server. All employees are allowed to access certain sections of the database. The server collects OS-level logs via Linux Audit. One night, an employee from this company forgets to log themselves out of their computer, leaving their device exposed. Capitalizing on this naive mistake, an attacker uses the employee's account to exploit the MySQL vulnerability. They know that the installed MySQL server has a vulnerability that allows them to modify the configuration file and load the malicious code, which has a TCP reverse shell that calls back to the attacker's machine (the vulnerability even allows the attacker to create their own configuration files as well). The attacker makes the necessary changes by updating the configuration file *(my.cnf)* such that when the MySQL server restarts, the new configurations will be loaded, and the call back function will be initiated. Now, the attacker waits patiently for the restart to happen. In fact, they can even trigger a manual reboot of the target server by launching a package or system update process or by issuing a `SHUTDOWN SQL` statement via the *mysqladmin*. Eventually, when the server restarts, the malicious configurations are loaded, and the payload is executed. As soon as the TCP call back is completed, the attacker starts exfiltrating the data from the MySQL database. Soon the admin detects the breach and terminates all the outbound connections. Later, the Incident Response (IR) team is asked to investigate this attack. The IR team immediately digs into the audit logs to reconstruct of the attack. They discover that there had been an "unauthorized" configuration update in the configuration file. However, the issue the IR team now faces is that they do not know which user or which particular write operations to the configuration file had been responsible for the attack. They may be able to filter out some suspect write syscalls from the logs (which in itself will be a daunting task); however, it will be difficult for them to find the exact configuration parameter or value that was updated (MySQL server has more than 500 configuration parameters), what were the exact contents of the write operation and who was responsible for the update (or which process did the update).

[3]Furthermore, the vulnerability can be exploited even if security modules, such as SELinux and AppArmor are installed on the target system with default active policies for the MySQL service.

TABLE V: Evaluation results of memory-based configuration tracking. We measure overhead and accuracy of our static analysis and instrumentation under two parameters: 1) applying pointer aliasing and 2) applying function specialization (TP = True Positives, FP = False Positives, FN = False Negatives).

Application	Annotation set			Pointer Alias	Specialization	Total STORES	Config STORES	Overhead (%)	Precision (%)	Recall (%)	TP, FP, FN
	Variable	Offset	Type								
Redis	server	168, 174	redisServer	No	No	16966	4	0.18	100	67	4, 0, 2
Redis	server	168, 174	redisServer	No	configSetCommand		3	0.17	100	75	3, 0, 1
Redis	server	168, 174	redisServer	Yes	configSetCommand		7	0.21	58	100	4, 3, 0
Redis	server	Any	redisServer	No	configSetCommand		68	1.24	100	87	68, 0, 10
Redis	server	Any	redisServer	No	No		522	7.96	100	-	522, 0, -
NGINX Unit	Any	Any	nxt_socket_conf_t	No	No	4025	33	0.78	100	-	33, 0, -
NGINX Unit	Any	Any	nxt_socket_conf_t	No	nxt_router_conf_create()		15	0.31	100	64	15, 0, 8
NGINX Unit	Any	Any	nxt_socket_conf_t	Yes	nxt_router_conf_create()		90	2.22	27	96	24, 66, 1

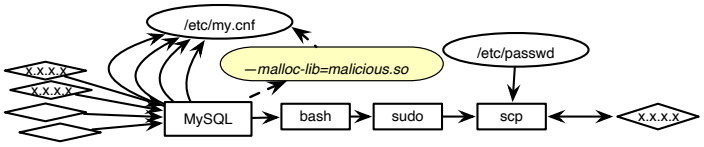

Fig. 6: Holistic provenance graph of MySQL attack presented in Section VII-B1. Squashed rectangle with dashed edges represent provenance of configuration values.

Now consider Dossier was running on the MySQL server. In the attack discussed above, Dossier will be able to log the precise location of the configuration variable/parameter that was changed along with the meta-information (e.g., PID of the config writing process). This will give the IR team enough information to reconstruct the attack, remove the malicious library from the system, and identify the user that forgot to log out of the system. We enacted the scenario described in this section. An attack was launched on a vulnerable MySQL server; however, our LKM was successfully able to log the writes to the configuration file with no FPs or FNs. We then ran our file deltas algorithm on top of the collected logs and extracted the exact file delta showcasing the malicious write causing the malicious code execution attack. The holistic graph generated by Dossier is shown in Figure 6. During the attack, the entire configuration file is rewritten after a single update, and it costs 2.1KB of log data. However, after applying the file delta algorithm, the storage overhead for this single update is reduced to only 147 Bytes. Since [82] has shown that 99.3% of configuration file updates of popular applications happen less than 1000 times over their lifetime, the storage overhead for a 1000 such updates amounts to only 147KB (after file-delta computation), which is negligible compared to the total size of audit logs.

2) Wordpress htaccess attack: For this case study, we chose a vulnerability in WordPress [33], a widely used website creation platform. The vulnerability, described in CVE-2020-8658 [13], results from a vulnerable anti-CSRF implementation within the htaccess plugin used by WordPress. It allows an attacker to modify the `.htaccess` configuration file of a website, take control of the website, and direct a victim to a malicious web page instead of the legitimate website. `.htaccess` is a sensitive hidden configuration file present within web servers and often serves an attack vector to hide malware, hide backdoors, inject content, and redirect search engines to malicious websites.

We now consider a scenario where a company uses an affected version of Wordpress. The use of `.htaccess` is enabled within the server since WordPress and various WordPress plugins use this file for in-directory tweaks to the web server's behavior. Moreover, the only logging implemented within the web server is at the OS level via the Auditd subsystem on the root machine. An attacker exploits the vulnerability present within the Htaccess plugin to direct the victim to a malicious web page with certain exploit code. The exploit code arbitrarily edits the contents of the `.htaccess` file such that every time the website is loaded from a search engine such as Google, Bing, etc., the users are redirected to a malicious web page instead of the legitimate company's website. The attack might go unnoticed for some time since the employees within the company use the complete website URL for accessing the website instead of getting redirected from search engines. Eventually, the system administrators discover that the website has been compromised and an Incident Response (IR) team is asked to come in and investigate. The IR dig into the audit logs and discover that there have been several updates on the `.htaccess` file, including legitimate file revisions from various WordPress plugins. However, the issue the IR team faces is that they do not know which particular write operations to the .htaccess file had been responsible for the attack. This is because existing OS based audit logs do not contain information about the exact contents of the write operation.

Now, consider the scenario that the system administrators had Dossier running on the server monitoring the sensitive .htaccess file. In the attack discussed above, Dossier will log the precise content within the .htaccess file that was changed. This will give the IR team sufficient information to reconstruct the attack, remove the malicious contents from the file, and identify the vulnerable plugin responsible for the exploit. In summary, an attack was launched on a vulnerable WordPress server; Dossier's Linux kernel module successfully logged all writes to the `.htaccess` file with no FPs or FNs. Moreover, Dossier's file diff algorithm extracted the exact file diff, revealing the malicious write causing the website compromise. Due to complete file rewrite, a single update costs around 698 Bytes during the attack. After applying the file delta algorithm, the storage overhead is reduced to only 142 Bytes. Similarly, a 1000 such updates amounts to a negligible overhead of 142KB.

3) BIND-9 Dynamic Update Attack: For this case study, we chose a vulnerability in BIND-9 [5] - a widely used open source DNS server. BIND-9 can be used both as an authoritative or as a caching DNS server and provides several features such as load balancing, notify and dynamic update, etc. The vulnerability in question, described in CVE-2017-

3143 [12], allows an attacker to forge a valid TSIG signature and issue unauthorized dynamic updates. The affected versions of BIND servers rely solely on TSIG keys with no other address-based ACL protection. An attacker who is able to send and receive messages to an authoritative DNS server, and who has knowledge of a valid TSIG key name for the zone and service being targeted, may be able to manipulate BIND-9 into accepting an unauthorized dynamic update. The attacker can leverage these capabilities to perform malicious dynamic zone content manipulation.

Consider a scenario where a company uses an affected version of BIND-9 to name their machines in their own domain. The only logging implemented within the server hosting BIND9 is at the OS level via the Auditd subsystem on the root machine. An attacker with knowledge about the TSIG key name exploits the aforementioned vulnerability to forge valid TSIG signatures and issues unauthorized dynamic updates to maliciously update the zone content within the server. Eventually, the network administrators notice that the server has been compromised and some of the DNS queries are returning incorrect responses and the users within the company are directed to wrong and potentially malicious websites. The IR team is then called in to investigate the issue. The IR team digs into the OS-level audit logs collected by the root machine. Since the malicious update on the zone takes place dynamically in-memory, no events in the log describe the malicious zone update. Eventually, due to the semantic gap that exists within the OS level logs, the IR could not conclude which particular zone update operations had been responsible for the attack.

Now, consider the scenario that the system administrators had Dossier running on the DNS server monitoring dynamic updates to sensitive structs containing information about the zone and the records enclosed within it. In the attack discussed above, Dossier will successfully log all the dynamic updates on the target zone variables annotated by the system administrators. This will give the IR team enough information to reconstruct the attack, remove the malicious contents from the zone, and identify that the attack exploited some vulnerability within the dynamic zone update implementation of the server. We enacted the scenario described in this section. We used the dynamically updateable structure type `dns_zone_t` within BIND to annotate the target zone variables at different struct offsets. Moreover, we used function specialization optimization and limited the instrumentation to the dynamic update handler function i.e., `update_action()`. After annotation, Dossier used static analysis (without pointer aliasing) to successfully mark and instrument the dynamic updates (STORES) on the target annotated variables with just one FN (100% precision, 84% recall). Resultantly, Dossier was able to log almost all relevant dynamic updates to the zone during our experimentation with an average log overhead of 126 Bytes per update. This allowed for a more holistic provenance graph generation including the dynamic updates for investigating the underlying attack.

VIII. Related work

In Section II-C, we described the limitations of existing provenance trackers that Dossier addresses, and complement the discussion on related work here.

Taint tracking approaches have been demonstrated to work effectively in concert with auditing, e.g., ProTracer [66] leverages tainting at the granularity of execution unit to avoid logging overhead. However, ProTracer does not log configuration changes. Argos [73] taints memory blocks and instruments malicious payloads to create network intrusion detection signatures, but does not provide a general solution for logging certain application-specific instructions. ShadowReplica [55] and Taintpipe [67] implement techniques to speed-up dynamic dataflow tracking (DFT) but their analysis does not taint specific application-state changes and instead focuses on generating control flow profiles. Moreover, TaintBochs [45], Rain [56], and UniSan [64] leverage taint analysis techniques to analyze sensitive data lifetime, conduct reachability analysis for inter-process DFT, and eliminate information leaks within OS kernel respectively. In short, these prior taint tracking works target different end-goals and do not seem to provide an obvious solution for auditing specific application-state changes.

Prior work has proposed techniques to tackle misconfigurations that violate correctness properties of the systems [86], including configuration validation and testing [83], [89], [78], [88], and misconfiguration troubleshooting [41], [74], [80], [81]. However, those techniques mostly target correctness issues, but cannot deal with configuration-based attacks. A few recent work has proposed solutions for detecting access-control misconfigurations [47], [43], [82], [84]. However, as discussed in Section II, configuration-based attacks are much broader than access control configurations. Furthermore, few work provides fine-grained, system-level forensic analysis.

Prior work [79], [51], [75] attempts to close the semantic gap that exists in provenance analysis through instrumenting function calls and arguments. However, these instrumentation-based systems capture every function call rather than only capturing function calls related configuration changes. Moreover, configuration semantics are usually not captured at function argument level as they can be assigned inside the function and these systems will not capture them. Finally, these systems only log information per-application basis, and do not utilize whole-system provenance graph. Thus, they do not have mechanism to connect information flow between different applications. On the other hand, Dossier inserts configuration update information into the whole-system provenance graph to reconstruct information from the whole system.

IX. Conclusion

In this work, we propose Dossier, a provenance tracker that logs changes to the configuration state of an application. Whether the change vector is file-based or memory-based, Dossier can track configuration updates during runtime. To log file-based configuration changes, Dossier features a light-weight kernel module. For memory-based Dossier uses program annotations and static analysis built on top of LLVM to capture the memory-based config updates. During the investigation phase, Dossier stitches logs related to configuration changes with the whole-system provenance graph to generate a holistic provenance graph. Evaluation on realistic workloads and attack scenarios shows that Dossier generated provenance graphs can accurately reason about configuration-based attacks with minimal runtime overhead.

ACKNOWLEDGMENT

We thank our shepherd, Dongyan Xu, and the anonymous reviewers for their comments and suggestions. Muhammad Adil Inam was partially supported by the Sohaib and Sara Abbasi Computer Science Fellowship. This work was supported in part by the NSF under contracts CNS-20-55127 and CNS-17-50024. 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] "100 million americans and 6 million canadians caught up in capital one breach," https://zd.net/3k6zVNM.

[2] "154 million us voter records exposed following hack," https://www.helpnetsecurity.com/2016/06/23/154-million-us-voter-records-exposed/.

[3] "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[4] "BEA WebLogic Server," https://docs.oracle.com/cd/E13222_01/wls/docs100/index.html.

[5] "BIND9," https://gitlab.isc.org/isc-projects/bind9.

[6] "Blast Benchmarks," https://fiehnlab.ucdavis.edu/staff/kind/collector/benchmark/blast-benchmark.

[7] "Capital one looked to the cloud for security. but its own firewall couldn't stop a hacker." https://wapo.st/37he1Uto.

[8] "CVE-2014-3250," https://nvd.nist.gov/vuln/detail/CVE-2014-3250.

[9] "CVE-2016-6662," https://nvd.nist.gov/vuln/detail/CVE-2016-6662.

[10] CVE-2016-7790. https://nvd.nist.gov/vuln/detail/CVE-2016-7790.

[11] "CVE-2016-8740," https://nvd.nist.gov/vuln/detail/CVE-2016-8740.

[12] "CVE-2017-3143," https://nvd.nist.gov/vuln/detail/CVE-2017-3143.

[13] "CVE-2020-8658," https://nvd.nist.gov/vuln/detail/CVE-2020-8658.

[14] "Database leaks data on most of ecuador's citizens, including 6.7 million children," https://zd.net/3k4orKD.

[15] "Download CVE List," https://cve.mitre.org/data/downloads/index.html.

[16] "Event Tracing," https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.

[17] "Exponent CMS," www.exponentcms.org.

[18] "Inmediata health group notifies patients of data security incident," https://prn.to/2SZs4pm.

[19] National vulnerability database. https://nvd.nist.gov/.

[20] "Nginx Unit Server," https://www.nginx.com/.

[21] "Open database exposes millions of job seekers' personal information," https://bit.ly/3due7c8.

[22] "Owasp top 10 projects," https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[23] "Personal info of 93.4 million mexicans exposed on amazon," https://www.databreaches.net/personal-info-of-93-4-million-mexicans-exposed-on-amazon/.

[24] "Redis Database," https://redis.io/.

[25] "Redis Unauthorized Access Vulnerability," https://bit.ly/3j1a9ZN.

[26] "Remote code execution in exponent - htb23290 security advisory," https://www.immuniweb.com/advisory/HTB23290, journal=ImmuniWeb.

[27] "Report: Job portal database exposed," https://www.safetydetectives.com/blog/talanton-leak-report/.

[28] "The Often Misunderstood GEP Instruction," https://llvm.org/docs/GetElementPtr.html.

[29] "These are the worst hacks, cyberattacks, and data breaches of 2019," https://zd.net/2SVG6Io.

[30] "thttpd," https://acme.com/software/thttpd.

[31] "The university of chicago medicine exposed 'perspective givers' database with more than a million of records," https://bit.ly/2HdTkNV.

[32] "Which of the owasp top 10 caused the world's biggest data breaches?" https://snyk.io/blog/owasp-top-10-breaches/.

[33] "Wordpress," https://wordpress.com/.

[34] "LMbench - Tools for Performance Analysis," http://www.bitmover.com/lmbench/, 2019.

[35] "Manipulating Writeable Configuration Files," https://infosec.cert-pa.it/capec-75.html, 2019.

[36] "MongoDB Databases May Be Exposed," https://ibm.co/2H6OeU1, 2019.

[37] "Static Value-Flow Analysis for C and C++ Programs," https://github.com/SVF-tools/SVF, 2019.

[38] "Whole Program LLVM," https://github.com/travitch/whole-program-llvm, 2019.

[39] "System administration utilities," man7.org/linux/man-pages/man8/auditd.8.html/.

[40] L. O. Andersen, "Program analysis and specialization for the c programming language," Tech. Rep., 1994.

[41] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis." in *OSDI*, vol. 10, no. 2010, 2010, pp. 1–14.

[42] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *USENIX Conference on Security Symposium*, 2015.

[43] L. Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 1, pp. 1–28, 2011.

[44] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, Nov. 2020.

[45] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," in *USENIX Security*, 2004.

[46] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *ACM SIGMETRICS*, 1994.

[47] T. Das, R. Bhagwan, and P. Naldurg, "Baaz: A System for Detecting Access Control Misconfigurations," in *USENIX Security*, 2010.

[48] Z. Dong, A. Andrzejak, D. Lo, and D. Costa, "ORPLocator: Identifying Read Points of Configuration Options via Static Analysis," in *ISSRE*, 2016.

[49] A. Gehani, H. Kazmi, and H. Irshad, "Scaling SPADE to "Big Provenance"," in *USENIX TaPP*, 2016.

[50] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Middleware*, 2012.

[51] E. Gessiou, V. Pappas, E. Athanasopoulos, A. D. Keromytis, and S. Ioannidis, "Towards a universal data provenance framework using dynamic instrumentation," in *Information Security and Privacy Research*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 103–114.

[52] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs," in *NDSS*, 2018.

[53] W. U. Hassan, M. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis," in *NDSS*, 2020.

[54] Z. Huang and D. Lie, "Saic: identifying configuration files for system configuration management," *arXiv preprint arXiv:1711.03397*, 2017.

[55] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *CCS*, 2013.

[56] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 377–390.

[57] V. Karande, E. Bauman, Z. Lin, and L. Khan, "SGX-Log: Securing System Logs With SGX," in *Asia CCS*, 2017.

[58] J. Katcher, "Postmark: A new file system benchmark," Technical Report TR3022, Network Appliance, Tech. Rep., 1997.

[59] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Cretu-Ciocarlie, A. Gehani, and V. Yegneswaran, "MCI : Modeling-based Causality Inference in Audit Logging for Attack Investigation," in *NDSS*, 2018.

[60] B. W. Lampson, "Computer Security in the Real World," *IEEE Computer*, vol. 37, no. 6, pp. 37–46, Jun. 2004.

[61] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *CGO*, Mar 2004.

[62] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition," in *NDSS*, 2013.

[63] ——, "LogGC: garbage collecting audit log," in *CCS*, 2013.

[64] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 920–932.

[65] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple Perspective Attack Investigation with Semantics Aware Execution Partitioning," in *USENIX Security*, 2017.

[66] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting," in *NDSS*, 2016.

[67] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined Symbolic Taint Analysis," in *USENIX Security*, 2015.

[68] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *USENIX ATC*, 2006.

[69] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian, "Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution," in *NDSS*, 2020.

[70] R. Paccagnella, K. Liao, D. J. Tian, and A. Bates, "Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks," in *CCS*, 2020.

[71] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, "Hercule: Attack story reconstruction via community discovery on correlated log graph," in *ACSAC*, 2016.

[72] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting High-Fidelity Whole-System Provenance," in *ACSAC*, 2012.

[73] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: An Emulator for Fingerprinting Zero-Day Attacks for Advertised Honeypots with Automatic Signature Generation," in *EuroSys*, 2006.

[74] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 193–202.

[75] M. Stamatogiannakis, P. Groth, and H. Bos, "Looking inside the black-box: Capturing data provenance using dynamic instrumentation," in *IPAW*. Springer-Verlag New York, Inc., 2015.

[76] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-Flow Analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[77] S. Sultana, E. Bertino, and M. Shehab, "A provenance based mechanism to identify malicious packet dropping adversaries in sensor networks," in *ICDCS*, 2011, pp. 332–338.

[78] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing Configuration Changes in Context to Prevent Production Failures," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[79] D. Tariq, M. Ali, and A. Gehani, "Towards automated collection of application-level data provenance," in *TaPP*. USENIX, 2012.

[80] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic Misconfiguration Troubleshooting with PeerPressure," in *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[81] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration Debugging as Search: Finding the Needle in the Haystack," in *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[82] C. Xiang, Y. Wu, B. Shen, M. Shen, H. Huang, T. Xu, Y. Zhou, C. Moore, X. Jin, and T. Sheng, "Towards Continuous Access Control Validation and Forensics," in *CCS*, 2019.

[83] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early Detection of Configuration Errors to Reduce Failure Damage," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Nov. 2016.

[84] T. Xu, H. M. Naing, L. Lu, and Y. Zhou, "How Do System Administrators Resolve Access-Denied Issues in the Real World?" in *CHI*, 2017.

[85] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *SOSP*, 2013.

[86] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, Jul. 2015.

[87] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "UISCOPE: Accurate, Instrumentation-free, and Visible Attack Investigation for GUI Applications," in *NDSS*, 2020.

[88] J. Zhang, R. Piskac, E. Zhai, and T. Xu, "Static Detection of Silent Misconfigurations with Deep Interaction Analysis," in *Proceedings of the 36th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'21)*, Oct. 2021.

[89] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 687–700.

[90] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "ConfMapper: Automated Variable Finding for Configuration Items in Source Code," in *IEEE International Conference on Software Quality, Reliability, and Security*, 2016.

APPENDIX

A. *thttpd Example*

We use thttpd [30], a simple and portable HTTP server, as a micro benchmark to evaluate Dossier's technique for tracking memory-based configuration changes. We modify thttpd source code just to evaluate different cases of dynamic configuration updates on one application. Specifically, we add an `update_config()` function in the main loop of the program, as shown in the following code snippet:

```
1  typedef struct {
2    int max_connections;
3    char* path_info;
4    short max_timeout;
5  } httpd_config;
6  // global configuration variables
7  httpd_config* server_conf;
8  int port;
9  // dynamic config update handler
10 void update_config(char* variable_name, char* value) {
11   if(strcmp(variable_name,"port")){
12     port = get_int_value(value);
13   }
14   if(strcmp(variable_name,"max_connections")){
15     server_conf->max_connections = \
16       get_int_value(value);
17   }
18   if(strcmp(variable_name,"path_info")){
19     server_conf->path_info = value;
20   }
21   if(strcmp(variable_name,"max_timeout")){
22     short* temp_timeout = \
23       &(server_conf->max_timeout);
24     *(temp_timeout) = get_short_value(value);
25   }
26 }
```

We annotate the global variables, `port` and the `server_conf` struct with offset. The update of `port` in Line 12 is the base case, in which the destination of the `STORE` instruction is a

variable. The update of struct variables `max_connections` and `path_info` in Lines 16 and 19 are cases when the destination of the `STORE` instruction is a GEP instruction. Furthermore, the update of struct variable `max_timeout` in Lines 23– 24 is a case of pointer aliasing, where the pointer `temp_timeout` is loaded with the address of the annotated configuration variable `max_timeout`. This pointer is then later used to refer to the address of memory space of `max_timeout` in Line 24. Dossier successfully marked and instrumented configuration variables in all the `STORE` instruction cases.

B. Oracle Weblogic Server Attack

Consider an online shopping website that uses Oracle BEA Weblogic servers [4] to manage users. One day the system administrators of the online shopping website noticed that sensitive information related to users of the website was leaked on the public forum. The administrator started the investigation to understand how the attack was performed and who was responsible. Thus, the administrator issues a root-cause analysis query on the sensitive database file to figure out who accessed and opened that database file.

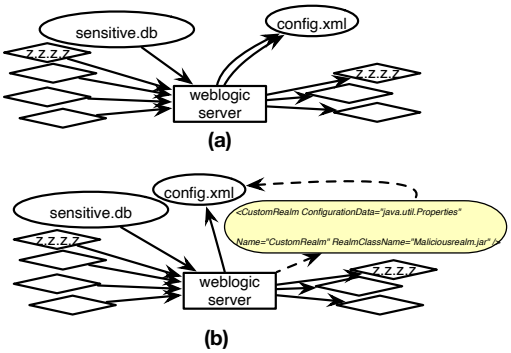

Fig. 7: Oracle weblogic server attack scenario **(a)** Provenance graph generated by tradition solutions. **(b)** Holistic provenance graph generated by Dossier. Squashed rectangle with dashed edges represent provenance of configuration values. We do not show configuration value delete event used by attacker to hide footprints for readibility.

Provenance graph generated by using traditional provenance tracker (e.g. SPADE [50]) is shown in Figure 7**(a)**. Even though, using this graph, the investigator can know that the "weblogic server" process was responsible for reading the sensitive file. However, the investigator quickly realizes that the graph does not reveal how "weblogic server" process was able to read that database file because external users are not authorized to read that file.

In this attack, the investigator was unable to understand how the "weblogic server" process reads that sensitive file because the attacker used configuration-based attack to initiate read event. Weblogic server uses `config.xml` file to store configuration data. However, if this file is not properly protected by the system-level access control, an attacker can write configuration information to redirect server output through system logs, database connections, etc. In this attack, the attacker noticed that `config.xml` was not write protected so the attacker inserts a pointer to a custom realm jar in the `config.xml`. Note that access to the Weblogic server is controlled through custom realm that manages authorization. The attacker inserted following configuration value in the `config.xml` that points to a custom realm jar file which enabled the attacker to access database file. After that attacker removed that configuration value from the `config.xml` to hide her attack footprints.

```
<CustomRealm
  ConfigurationData="java.util.Properties"
  Name="CustomRealm"
  RealmClassName="Maliciousrealm.jar"
/>
```

With Dossier, the security investigator can understand the root-cause of the attack. Holistic provenance graph generated by Dossier is shown in Figure 7**(b)**. The provenance graph pinpoints that "weblogic server" process changed configuration values to point to custom realm which authorized the attacker to access the sensitive database file.