TRACE: Enterprise-Wide Provenance Tracking for Real-Time APT Detection

Hassaan Irshad, Gabriela Ciocarlie, Ashish Gehani, Somesh Jha, Yonghwi Kwon, Vinod Yegneswaran, Kyu Hyung Lee, Jignesh Patel, Dongyan Xu, Xiangyu Zhang

Abstract—We present TRACE, a comprehensive provenance tracking system for scalable, real-time, enterprise-wide APT detection. TRACE uses static analysis to identify program unit structures and inter-unit dependences, such that the provenance of an output event includes the input events within the same unit. Provenance collected from individual hosts are integrated to facilitate construction of a distributed enterprise-wide causal graph. We describe the evolution of TRACE over a four-year period, during which our improvements to the system focused on performance, scalability, and fidelity. In this time span, the system call coverage increased (from 47 to 66) while the time and space overhead reduced by over one and two orders of magnitude, respectively. We also provide results from five adversarial engagements where an independent team of system evaluators conducted APT attacks and assessed system performance. The input from our system was used by three other teams to implement real-time APT detection logic. Retrospective analysis revealed that TRACE provided sufficient evidence to detect over 80% of the attack stages across all evaluations. By the last engagement, temporal and spatial overhead had been reduced significantly to 18% and 10%, respectively.

Index Terms—Computer Security, Information Security, Intrusion Detection

I. INTRODUCTION

TARGTED APT attacks represent an ever-present threat to nation-states and enterprise networks, and no silver bullet will likely ever be successful in completely countering this menace. To date, the most promising approach to detecting and mitigating the majority of such threats involves provenance tracking. Provenance captures multiple aspects of information regarding an entity: what is the origin of the entity; how the entity is derived; and when it originated [1], [2]. While the what provenance facilitates root-cause analysis by specifying the set of artifacts (i.e., objects such as files, sockets, etc.) that are affected, the how comprises a set of events (i.e., syscalls) with timestamps that reflect when the events occurred in the causal ordering.

There are two broad classes of techniques for provenance tracking: audit logging [3]–[7] and taint propagation [8]–[15]. Both approaches have pros and cons, and neither is a complete solution for enterprise-wide APT detection/forensics. While audit logging captures both what- and how-provenance; propagation can only capture the former. Audit logging has low runtime overhead because it does not require expensive per-instruction set operations; however, it suffers from low accuracy and high space overhead (for storing event logs). While the propagation-based approaches are more accurate in certain attacks, such as information leak, they suffer from high runtime overhead and difficulty in handling program control dependencies.

In this paper, we develop a practical enterprise-wide provenance-tracking framework that captures both what- and how-provenance by leveraging the advantages of both approaches and overcoming their respective limitations. We design our system to support the execution of both what-provenance and how-provenance queries. For example, given a corrupted dataset \( x \), two what-provenance queries are: (1) “What is the source/entry point of \( x \)?” and (2) “which other files in the enterprise were derived from (and corrupted by) \( x \)?” A sample how-provenance query is: “Construct a causal graph showing the events/entities that led to the corruption of \( x \) and those that have been further corrupted by \( x \).”

Specifically, we describe the design and implementation of a highly scalable, distributed, enterprise-wide provenance tracking and collection system called TRACE. The system provides both logging and taint propagation primitives. Its host-level provenance tracking component (Section III-A) monitors host execution and collects both what- and how-provenance for individual host systems at the granularity of program execution units [5]. While this framework is inspired by a prior system called BEEP, there are notable differences. First, unlike BEEP which uses dynamic analysis and binary instrumentation, TRACE’s unit-based selective instrumentation (UBSI) scheme implements a static analysis technique based on data structure identification. In addition, it implements several performance optimizations that are crucial to scale provenance analysis on a modern browser. These optimizations led to multiple orders of reduction in time and space overheads. The enterprise-wide provenance tracking component (Section III-B) builds upon the SPADE engine [16] to gather provenance from individual entities such as hosts and construct the distributed enterprise-wide causal graph. We integrate this system with UBSI, implement a Linux kernel module to support cross-host causality analysis and extend the system with a new CDM storage module that provides the underlying distributed storage and processing infrastructure allowing issuance of APT-related queries spanning the whole enterprise.

We report on our experience building, refining, and evolving this integrated system. Our evaluation results offer confidence that instrumentation provided by TRACE enables practical collection and querying of enterprise-wide provenance for accurate and real-time threat detection. The specific contributions of our paper include the following:

- Streaming-units framework for scalable and distributed online provenance tracking;
- Static-analysis based technique for identifying unit dependencies from shared data structures;
- New UBSI approach for the Firefox browser\(^1\);

\(^1\)The original BEEP approach performed such dependency identification outside the application during post analysis of the audit logs. Our new approach performs in-application unit-dependency identification.
• Strategies to generate concise provenance graphs for forensic analysis;
• Robust generation of network artifacts that facilitate cross-host causality tracking;
• Details of system evolution during five different red-team engagements with three analysis teams over a four-year period;
• Evaluation and analysis of the TRACE system in a controlled environment that led to multiple orders of magnitude improvements in time and space overheads for unit instrumentation;
• The software [17], [18] and engagement datasets [19] have been publicly released.

II. RELATED WORK

The objective of TRACE is to apply scalable and fine-grained information-flow tracking techniques for real-time, enterprise-wide APT detection. Although prior research has laid the foundations for information-flow tracking and provenance analysis [8]–[12], [15], and their applications for malware detection at the enterprise-level [3], [4], [6], [7], [20]–[28], such systems have not been widely deployed due to their intractability and imprecision at scale.

Host-Level Audit Logging. Solutions in this category [3], [4], [6], [7] record system-level events (e.g., syscalls) during execution and causally connect events during attack investigation. They treat processes as subjects; files, sockets, and other passive entities as objects; and assume causality between subjects and objects involved in the same syscall event (e.g., a process reading a file). Audit logging is a built-in function in Linux that incurs much lower overhead than per-instruction provenance propagation. Other techniques [20]–[28] focus on the analysis of causal relationships between the captured events. While they propose various optimizations in building causal graphs or building data structures representing information flow to improve the effectiveness of attack investigation, they still suffer from dependence explosion and storage overhead.

(1) Dependence explosion [5] is a major limitation of audit logging. For a long-running process, an output event is assumed to be causally dependent on all preceding input events, and an input event is assumed to have causal influence on all subsequent output events. Such conservative assumptions create false-positive causal relations, making it difficult to reveal the true causality.

(2) High storage overhead. A preliminary study [29] shows that audit logging easily generates gigabytes of log data per host every day. This is particularly problematic for APT defense, as APT malware tends to lurk in the victim host for a long period of time.

(3) Offline analysis. Prior work that introduced [5] and refined [30] the notion of an execution unit requires both a forward and backward pass over the logs. This precludes their use in the streaming setting for real-time detection.

Per-Instruction Provenance Propagation/Tracking (I-PT). I-PT techniques [8]–[12], [15] involve fine-grained causality tracking by monitoring the execution of individual instructions. Their application to host-wide causality tracking in production environments has been hindered by the following limitations.

(1) Runtime overhead. Because they track the execution of individual instructions and propagate large provenance sets, I-PT techniques usually incur substantial runtime overhead. State-of-the-art implementations without hardware support incur a slow-down [9], which is undesirable for production environments.

(2) Lack of implicit flow handling. Many I-PT techniques have difficulty handling implicit flow, which is information flow through control dependences [31] (usually induced by program predicates). A naive solution that propagates provenance via all control dependences may lead to a high false-positive rate (up to 97% for SPEC2000INT programs [32]), with the consequence that each output is causally related to almost all inputs.

(3) Inflexible provenance granularity. I-PT techniques [8]–[12], [15] often assume one default provenance granularity (e.g., each input byte, input syscall, or file). However, APT defense may require multiple provenance sources with varying granularity (e.g., individual emails instead of an entire inbox file).

Network-level Provenance. There is a line of work in network-level provenance [33]–[36] that focuses on distributed systems. [36] monitors and records causal dependencies in the software defined networks environment. Our system supports both host-level and network-level provenance, providing comprehensive causal relationships across systems. As an infrastructure, the proposed system and existing provenance tracking tools are complementary. In other words, one may leverage existing provenance tracking techniques to enhance the capabilities of the proposed system.

Provenance Frameworks. There are a number of available provenance frameworks which range in granularity from whole system provenance capturing to capturing provenance of individual applications.

(1) Whole system. Provenance frameworks in this category [37]–[40] capture provenance at the level of kernel data structure modifications by using Linux Security Modules, Linux Provenance Modules [38] or by instrumenting the kernel itself. While the provenance captured by such frameworks is at a fine-grained level, it makes the task of mapping low-level kernel events to application-level events much harder for analysis which is also exacerbated by the high volume of provenance generated by numerous low-level events. In contrast, TRACE captures provenance at system call granularity by consuming Linux Audit logs. The efficacy of using Linux Audit logs for APT detection is already proven by work in [20]–[22], [24], [25], [27], [28]. Furthermore, we

2 The engagements were conducted as part of a broader research program focused on provenance collection for APT detection. An important aspect of this paper, that is under reported in academic papers, is the significance of objective third-party “one-shot” evaluations. This is a noteworthy departure from developer-controlled prototype evaluations, where systems are continually refined until a desired performance metric is reached. We believe that carefully documenting and sharing this unique experience benefits the academic and security research community.
selected a subset of system calls in the Linux Audit log based on the following criteria: (a) The system call must meet a minimum occurrence threshold in profiled workloads, and (b) The system call must be directly or indirectly responsible for data-flow between processes, files, pipes, file descriptors, and sockets. This choice of criteria is supported by APT detection approaches [24], [27] which use a narrower criterion for selecting system calls (only processes, files, and sockets) and yet sufficiently show the success of their selection in APT detection. The use of Linux Audit logs is not without its drawbacks. One drawback is that the local network endpoint information is not reported in Linux Audit logs. This deficiency was resolved by the addition of a kernel module to gather, and report the local network endpoint information in the Linux Audit stream. Another drawback is the need for reconstructing kernel-space semantics in user-space for generating comprehensible provenance. For example, maintaining a mirror image of the kernel file descriptor table in TRACE to generate provenance for file writes. This is required because Linux Audit logs do not provide direct information about the file path that was manipulated whereas the frameworks, mentioned above, are able to extract that information directly by traversing the kernel data structures. This approach comes with its overhead but as we show in the rest of the paper, TRACE is a scalable framework.

(2) Application. The frameworks in this category [5], [30], [41], [42] seek to do fixed execution partitions of an application to reduce dependence explosion problem. These frameworks either rely on instrumenting the application manually/automatically, or analyzing the application binary for execution models or log messages to create execution partitions. In all cases, the application-specific provenance is used in conjunction with Linux Audit logs for provenance analysis. [5], [30], [41] suffer from the drawbacks that their provenance cannot be analyzed in a streaming fashion, they report provenance for a handful of system calls, and they do not generate provenance of an application which includes execution partition metadata. Whereas, [42] streams provenance, and reports execution partition metadata but suffers from the drawback that the correlation (w.r.t. time) between application logging and the actual event is a programmer’s choice and not necessarily a related event as implied by the resulting provenance. TRACE resolves above mentioned drawbacks by using UBSI for creating execution partitions in an application automatically, streaming provenance, including execution partition metadata in provenance, and performing provenance transformations on the fly. Specifically, a dependency between threads because of a memory read event and a memory write event of the same memory location is transformed into one single thread dependency event. Thus, reducing the high volume of data generated by tracking provenance of process memory in general. Also, UBSI allows for a non-fixed size of execution partition between executions i.e. controlling dependence explosion as needed. The smaller the execution partition is, the higher the volume of generated provenance is, and vice versa.

In Table I, we compare novel features of our provenance framework with existing frameworks. The features selected describe the ease of deployment, ease of use, and features provided for facilitation of provenance analysis by each framework. Following is a description of each feature:

Requirements. System requirements for being able to generate provenance.

Provenance streaming. Whether or not the provenance generated by the framework can be analysed at runtime. If, no, then all analysis is done offline.

Application Execution Partitioning. If execution partitioning is used then how is it used to reduce dependence explosion.

Application Execution Partitioning Metadata. If execution partitioning is used then how is the metadata for execution partitions is reported. For example, the metadata for an execution partition in TRACE is unit id reported in provenance for each loop as well as the dependency events between loops.

Agent Reporting. Whether or not the agent provenance (user and group credentials) is reported to identify privileged and unprivileged system activity.

Indirect Agent & Permission Update. Whether or not indirect updates to process’ user and group credentials and file’s permissions are reported as first class provenance events. An indirect update (i.e. without a system call) strongly implies malicious activity.

Entity Resolution. Whether or not, file descriptor manipulations are resolved to the entity that they refer to. This makes analysis intuitive by searching for path manipulations rather than indirectly through file descriptors. Also, this significantly reduces the volume of the provenance generated.

Process Filtering. Whether or not, a process can be white-listed so that its provenance is not generated to reduce volume of provenance generated.

Entity Versioning. Whether or not, entities can be versioned in provenance to allow for excluding false dependencies through irrelevant versions of an entity in simple provenance graph traversal.

Self-protection. Whether or not, the framework provides resilience to targeted attacks against the monitoring framework.

As shown in Table I, TRACE provides system-level as well as application-level provenance, and the two are merged into a unified provenance stream so that further analysis is not required to generate a coherent provenance graph. Also, we focus on representing system events in an intuitive way to facilitate provenance analysis by providing richer provenance metadata, automatically inferring events based on the TRACE internal state changes, and representing multiple low-level events as one high level event.

III. SYSTEM OVERVIEW

We describe below the two key components of TRACE: host-level provenance tracking with streaming units and enterprise-wide provenance tracking.
TABLE I
COMPARISON OF PROVENANCE FRAMEWORKS: AA: APPLICATION ANALYSIS; AI: APPLICATION INSTRUMENTATION; LA: LINUX AUDIT LOGS; DS: DATA STRUCTURES; UP: USER PERSPECTIVES; LKM: LINUX KERNEL MODULE; * NO, BUT INFERABLE FROM PROVENANCE;

<table>
<thead>
<tr>
<th>Provenance Framework</th>
<th>Requirements</th>
<th>Provenance Streaming</th>
<th>Application Execution Partitioning</th>
<th>Application Execution Partitioning Metadata</th>
<th>Agent Reporting</th>
<th>Indirect Agent &amp; Permission Update</th>
<th>Entity Resolution</th>
<th>Process Filtering</th>
<th>Entity Versioning</th>
<th>Self-Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi-Fi [37]</td>
<td>OS kernel build</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No*</td>
<td>No</td>
<td>Yes</td>
<td>Enforced by OS kernel</td>
</tr>
<tr>
<td>LPM-based [38]</td>
<td>Modified OS kernel</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No*</td>
<td>No</td>
<td>Yes</td>
<td>Enforced by OS kernel</td>
</tr>
<tr>
<td>CamelFlow [40]</td>
<td>Modified OS kernel</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No*</td>
<td>Yes</td>
<td>Yes</td>
<td>Enforced by OS kernel</td>
</tr>
<tr>
<td>OmegaLog [42]</td>
<td>AA, LA, LKM</td>
<td>Yes</td>
<td>Automatic log analysis</td>
<td>Reported in provenance</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MCI [41]</td>
<td>AA, LA</td>
<td>No</td>
<td>Automatic execution analysis</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MPI [30]</td>
<td>AI, LA</td>
<td>No</td>
<td>Manual UP instrumentation</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BEEP [5]</td>
<td>AI, LA</td>
<td>No</td>
<td>Automatic DS instrumentation</td>
<td>Reported in provenance</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TRACE</td>
<td>AI, LA, LKM</td>
<td>Yes</td>
<td>Automatic execution</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

A. Streaming Provenance Execution Units

Many long-running programs share a common property: their execution is dominated by an event-handling loop, which usually handles an independent request on each loop interaction. This is confirmed by our study of more than 100 open-source applications, including 51 server applications (e.g., web, mail, media, ssh-daemon, remote desktop, version control) and 43 client-side applications (e.g., browser, email client, multimedia, messenger). They are written in various languages (C, C++, Java, Python) and may involve threads/processes. Hence, our host monitoring component incorporates a new technique called unit-based selective instrumentation (UBSI) that builds on the notion of a unit [5]. Specifically, it identifies borders and dependences between such loop iterations – a unit – so that causal graphs may be constructed that contain only causally related entities/events.

First, static analysis is performed offline to recognize the execution and data unit structures of a program. Next, inter-unit dependences are identified. Selective program points (e.g., unit boundaries or unit dependence-inducing instructions) are then instrumented. During execution, system calls and unit-related events are captured by the provenance tracking runtime.

Reverse Engineering Unit-Inducing Loops. To detect the unit-inducing loops, UBSI leverages three observations: (1) such loops tend to be at the top level; (2) their loop bodies must make some I/O system calls; and (3) their loop bodies dominate the execution time. We have developed dynamic analysis techniques using PIN tool [43] to pinpoint unit-inducing loops in applications [5]. We then use source code and binary instrumentation techniques [44] to instrument the loop entry and exit points such that special log entries are generated to indicate unit boundaries.

Reverse Engineering Inter-Unit Dependencies. In some cases, a unit by itself may not fully cover the sub-execution that handles an independent input. Instead, a few interdependent units together constitute a semantically independent sub-execution. In practice, there are memory dependencies across unit boundaries. However, only some of them – called workflow dependencies – are helpful in connecting units that belong to the same sub-execution. Examples include the dependencies caused by the enqueue and dequeue operations of a task queue. In previous work [5], inter-unit dependencies were identified via a number of training runs of the target binary, which suffers from incomplete discovery of dependencies.

To remedy the aforementioned problem and support streaming provenance, we developed a static analysis method to detect shared data structures accessed by multiple threads. It identifies and instruments the instructions that induce dependences through the statically identified shared structures. The interdependent units can then be detected and clustered. In addition, UBSI was refined to reduce the volume of events generated while still capturing the dataflow between units. In Engagement 1, UBSI reported read and write events on memory by units. The mentioned events were used to identify dependencies between units. Due the nature of the Firefox application, there were too many read and write events by units. This is shown in Fig. 1(a). In Engagements 2 and 3, an abstraction of unit dependency was developed. SPADE’s Audit Reporter was extended to automatically convert read and write into unit dependencies events between units, as shown in Fig. 1(b). Although UBSI was still responsible for reporting the memory read and write events, this greatly reduced the number of final provenance events from UBSI, as shown in Table II. In Engagements 4 and 5, a final update was made. We migrated unit dependency identification upstream to UBSI instrumentation. This significantly reduced the runtime overhead. Specifically, we cached memory read and write events, and only reported confirmed dependencies between units. This eliminated unnecessary inter-process communication (IPC) between instrumented applications and the Linux Audit subsystem. Additionally, the update in Engagement 4 reports only the last dependency between any two units to further minimize the runtime and space overhead. This also simplified...
B. Enterprise-Wide Provenance Tracking

SPADE [16] is an open source software infrastructure that provides support for provenance auditing in distributed environments. TRACE uses it as an integration framework. SPADE uses a graph-based data model consisting of vertices and directed edges, each of which can be labeled with an arbitrary number of annotations. The model uses the following three vertex and five edge types (Figure 1) from the Open Provenance Model (OPM) [48] ontology:

**Two Vertex Types:** (1) Controlling Agent, executing Process (blue rectangles), and (2) Data Artifact (yellow ovals).

**Five Edge Types:** Defining (1) which process used which artifact (green arrows), (2) which artifact wasGeneratedBy which process (red arrows), (3) which process was TriggeredBy which other process (blue arrows), (4) which artifact wasDerivedFrom which other artifact, and (5) which process wasControlledBy which agent.

TRACE extends the ontology to handle the constructs described in the following sections. SPADE has been architected to decouple the collection, filtration, storage, and utilization of provenance metadata, as illustrated in Figure 2. A novel provenance kernel mediates between producers and consumers of provenance information, and handles the persistent storage of the records. The kernel handles buffering, filtering, and multiplexing incoming metadata from multiple provenance reporters via a non-blocking interface; supports multiple provenance stores; and responds to concurrent queries from provenance consumers. The kernel also supports modules that operate on the stream of provenance graph elements, allowing the aggregation, fusion, and composition of provenance elements to be customized with provenance filters [49].

The TRACE system uses two sources of information about system activity that were used to output provenance in a Common Data Model (CDM) format. The two sources include:

- **Linux Audit.** This component of the operating system kernel was configured to log all system calls specified in Table III. Calls from all processes across the host were audited except for those that are executed by the TRACE system.
- **UBSI Instrumentation.** Linux Audit does not provide information regarding the dataflows through memory in and between threads. This dataflow was captured by UBSI instrumented programs. UBSI instrumentation divides a program into individual components called units that are iterations of program loops. It then audits reads and writes by units of memory locations shared between threads.

SPADE’s Audit Reporter was extended to process the two aforementioned sources to generate provenance. A single audit event received contains the following information: system call (number, arguments, return values), process identifiers (pid, ppid), user identifiers (uid, gid, euid, egid), and supplementary information (e.g., remote network address, filesystem paths). The Audit Reporter uses the event records to build an overview of the system in terms of these provenance objects:

- **Process.** The subject or object of the system call.
- **Unit.** The subject performing read or write on a memory address.
- **Artifact.** The objects affected by the system call. The objects through which dataflow was reported were: (1) Local Filesystem Artifacts: regular file, named pipe, unix socket; (2) Memory: Address allocated in a thread or affected by a unit; (3) Network: Local or remote address; (4) IPC: unnamed pipe; and (5) Unidentified: Unknown objects whose type could not be determined.
The Audit Reporter maintains state for each of the provenance objects and internal mappings between them to generate an accurate representation that reflects the model in the operating system kernel. Our approaches abstract portions of the information flow graph, emit provenance at multiple resolutions, and use information-flow labels for various units. To handle such fine-grained provenance elements generated by USBI (Section III-A), we extended SPADE's Audit Reporter. Furthermore, in the Audit Reporter we exploited domain semantics to make performance improvements:

- Automatically garbage collecting internal data structures for provenance objects. This was done by relying on the life-cycle semantics of provenance objects in the provenance domain.
- Space efficient and time efficient retrieval and storage, and prevention of duplicate provenance objects by using provenance object identifiers guaranteed to be consistent and unique in the provenance domain. This also allowed for not needing to store repeated information for provenance objects across provenance events.

**Persistent Storage.** The kernel commits the integrated provenance (and subsequently retrieves when necessary) through a uniform provenance storage interface that allows graph elements to be inserted and retrieved. Each storage subsystem implements this interface and leverages its native functionality to best implement the required functions. We developed a Kafka Storage for SPADE that leverages the Apache Kafka [50] project's client to stream provenance records to a corresponding server. This functionality was extended in SPADE’s CDM [51] Storage to use an Avro [52] schema, defined by STARC [53], that is customized for reporting operating system activity. Each client can be configured with a host-specific Kafka producer identifier. This facilitates tracking provenance from multiple hosts in a distributed environment. The CDM Storage also periodically sends statistics about the mix of provenance records that have been published up to that point in the session.

**Network Awareness.** SPADE supports provenance queries about distributed computation. It models a network connection as a pair of network artifacts connected by used and was-GeneratedBy edges. Each endpoint of a network artifact can independently construct the same artifact without explicit coordination. This property allows for complete decentralization of provenance collection, while still ensuring that subgraphs from different hosts can be reassembled. We implement network artifacts with this property by combining the time the connection was initiated with the IP addresses and TCP or UDP ports of the two endpoints.

The system is decentralized; each computer maintains its authoritative repository of the gathered provenance. Flows between processes, files, and network connections are recorded at each host, and the resulting metadata is stored locally. Applications are oblivious to the provenance collection and metadata distribution.

**Space/Performance Optimizations.** First, a number of abstractions are applied to the events generated by USBI-instrumented application to reduce the volume of data published. Abstracted relationships were reported after all underlying events were processed. Although such abstraction can eliminate the assurance of total ordering between events, our evaluation over five adversarial engagements demonstrates that it does not impact the forensics capability of TRACE. By reporting the abstracted event in place of the last underlying event, the detection logic can be sure that all relevant underlying events have already been processed. To simplify analysis, we made updates in our system to report abstracted relations as early as possible. Second, unit-related records that were previously stored using statically allocated buffers was changed to use dynamic sizes, which dramatically reduced the memory footprint of the corresponding process. In addition, de-duplication strategies were implemented in data structures to reduce memory usage. Third, each process can be controlled by a different agent, letting its data structure be linked to a separate instance tracking its user, group, etc. In practice, the diversity of controlling agents over time is low. This was exploited by defining a new kinetic data structure (that associates agents with spans of temporal operation). This significantly reduced unit-related memory requirements.

**Accuracy and Fidelity Improvements.** To enable cross-host information flow tracking, we implemented network artifacts and used iptables to generate Linux Audit network filter records. When network connections are established (through application system calls), local endpoint details are not reported in the system call records of Linux Audit logs. Cross-host tracking requires this information for forensics across hosts. We developed a new Linux kernel module, called netio, that collects local endpoint details and reports them via the Linux kernel’s Audit subsystem. SPADE’s Audit Reporter was updated to (i) detect and report when the uid of a process in a system call record differs from the last known uid of that process; (ii) process the local endpoints reported by the netio kernel module; (iii) collect host configuration information from the runtime and (iv) report when file paths relate to directories, links, character devices, and block devices.

### IV. System Design Evolution

Our system was evaluated in a series of five adversarial (red-team) engagements as part of a large funded-research program.
During each engagement, attacks were launched by the red team and were analyzed by (1-3) independent performers. We describe how the system evolved over time. Summaries of the outcome of the engagements highlight the efficacy and utility of the TRACE system in uncovering APT attacks.

A. Adversarial Engagement 1

TRACE was run on Ubuntu 14.04 64-bit host. The host was set up with a vulnerable Firefox web browser (version 42) and a kernel driver to simulate a privilege escalation vulnerability. For the first adversarial engagement, our system was configured to track 47 system calls and collected 111 GB of data in a period of 4 days, capturing 1,485,144,886 events.

1) Bovia: (Figure 3). Firefox navigated to a compromised website. The compromised website was set up for memory reads and writes. We also concluded that an event-centric representation of provenance was more needlessly verbose.

2) Pandex: (Figure 4). The attacker used ssh to log into the host using stolen credentials. This connection was used to install Dropbear SSH server, which was used for future connections by the attacker. The attacker later connected to the Dropbear SSH server on the compromised host to gather system information and exit after that. Activity by the attacker to gather host information was not detected and was not present in the data along with its exfiltration. All stages of the attack were present in TRACE data except the reconnaissance and the exfiltration by the attacker. Of the three analysis teams, one was able to detect all stages in the attack. The second team failed to detect all stages in the attack. The third team did not detect any stage of the attack.

3) Bovia-Stretch: (Figure 5). Firefox navigated to a compromised website that exploited a vulnerability in Firefox and took control of the Firefox process. The hijacked process downloaded a malicious binary and then executed it. The malicious program connected to the attacker and acted as a reverse-shell. In this attack, unlike the Bovia attack, the malicious binary gained root privileges first by writing to a file. The file was created by a kernel module that was installed as benign setup for this attack. The attacker used a malicious program to read and write system files like /etc/passwd, /etc/sudoers and /etc/shadow and closed the channel. All stages of the attack were present in TRACE data. Of the three analysis teams, one was able to detect all stages in the attack, a second team was able to identify 6/7 attack stages.

**Engagement-1 Summary:** TRACE provided evidence of 15/17 attack steps across 3 attacks. At least one of the teams identified each of those 15 steps. **Lessons:** Based on results, we identified 12 critical missing system calls and determined the need to optimize UBSI output to have direct connections between units for memory reads and writes. We also concluded that an event-centric representation of provenance was more needlessly verbose.

B. Adversarial Engagement 2

TRACE monitored 59 system calls, and the total data collected for the three teams was 1.78 GB, 4.62 GB, and...
1.46 GB respectively. This was a significant reduction from Engagement 1 and a testament to the optimizations that were implemented.

**Improvements.** 12 more system calls were identified and audited to generate a richer provenance graph. TRACE was extended to include provenance tracking of UDP network traffic. An update was made in the UBSI instrumentation to report only dependencies between units rather than reporting the read of a memory address by a unit that was previously written to by a different unit. This, in theory, reduced the number of Linux Audit records generated by TRACE system at most by 50%.

Previously, the permissions of Linux files were not reported in provenance unless permissions were updated directly through a system call. From this engagement onwards, permissions of files were reported always, which enabled us to detect any surreptitious change in permissions by the red team. In addition, filesystem paths were normalized to remove special symbols, and .. to simplify the generated provenance. This reduced the effort required by the analysis teams to connect dataflows.

1) **Bovia-Simple:** This attack was the same as the Bovia attack from Engagement 1. All stages of the attack were present in TRACE data. All three analysis teams detected all 4 stages of the attack.

2) **Pandex-Dracon:** In this attack, Firefox navigated to a compromised website. The compromised website exploited a vulnerability in Firefox and took control of the Firefox process. Firefox was hijacked by allocating executable memory, writing the malicious code to it, and transferring execution to it. The hijacked Firefox process then escalated its privileges to root. The elevated Firefox process did reconnaissance and exfiltrated that information to the attacker over the network. This attack went mostly undetected despite it being a variant of the Bovia-Stretch attack from Engagement 1, which was detected by analysis teams. The
stages of the attack which were not detected by any team were also not found in the data. Retrospective analysis by the adversarial team led them to conclude that this miss was most likely due to incorrect TRACE configuration (process/user whitelisting). 2/6 stages of the attack were present in the TRACE data. Of the three analysis teams, only one team detected only the privilege escalation stage of the attack while the other two teams detected no stage of the attack.

3) Pandex-Micro: (Figure 6), Firefox navigated to a compromised website. The compromised website exploited a vulnerability in Firefox and took control of the Firefox process. The hijacked process downloaded a malicious binary and executed it. The malicious program connected to the attacker and acted as a reverse-shell. The attacker wrote python scripts to the host and executed them. The python scripts gathered network information and exfiltrated the /etc/passwd file to the attacker. All stages of the attack were present in TRACE data. Of the three analysis teams, two teams detected all 5 stages of the attack. The third team did not detect any attack stages.

4) Drakon-Netcon: Firefox navigated to a compromised website, which exploited a vulnerability in Firefox and took control of the Firefox process. Firefox was hijacked by allocating executable memory, writing the malicious code to it and transferring execution to it. The hijacked Firefox process was seen writing to a file and, as a result, the process gained root privilege. This happened because of the kernel module installed on the host as part of benign activity. The hijacked process gathered system information such as process listings and host information, allocated executable memory again, and gathered network information from the system and sent that to the attacker. Only one analysis team was able to partially detect this attack. After looking in the data provided, we were able to detect the download of the malicious binary, escalation of privilege by the Firefox process, and the exfiltration but not the source of the attack, i.e., when the attacker exploited Firefox. 4/6 attack stages were present in the TRACE data. Only one team was able to detect the reconnaissance stage of the attack, while none of the other two teams detected any stages of the attack.

C. Adversarial Engagement 3

Improvements: Five more system calls were audited. Prior to Engagement 3, dataflow through directories, links, character devices, and block devices was reported as dataflow through Linux files. The change was made to capture dataflow in a more refined way by specifying the exact type. This enabled us to see that the red team was escalating process privileges using a character device. The native tools being used in the TRACE system did not always provide complete network dataflow information. For example, the local port of a network channel is not reported in a UDP transfer. To cover this gap, a kernel module was added to SPADE’s Audit Reporter that hooked system calls related to network I/O (sendto, recvfrom, sendmsg, recvmsg, bind, connect, accept and accept4) to report complete information of a network endpoint. SPADE’s Audit Reporter did not keep any internal mapping about IPC objects through which Linux process data could be shared, e.g., Process A talking to child Process B, through a common unnamed pipe. In the resulting provenance graph, this would have been represented as Process A and Process B doing I/O on disconnected vertices in the graph. In Engagement 3, an update was made to keep the mapping and draw a direct connection between Process A and Process B in SPADE’s Audit Reporter.

1) Drakon-in-Memory: Firefox navigated to a compromised website. The compromised website exploited a vulnerability in Firefox and took control of the Firefox process. Firefox was hijacked by allocating memory, writing the malicious code to it and transferring execution to it. The hijacked...
Firefox process was seen writing to a character device and, as a result, the process gained root privileges. This happened because of the kernel module installed on the host as part of benign activity. The hijacked process downloaded a malicious binary and executed it. The malicious program gathered system information and sent it to the attacker. All 7 stages of the attack were present in TRACE data. Of the three analysis teams, one team detected all stages of the attack. The second team detected 2/7 stages.

2) Drakon-Pass-Manager: Firefox password manager extension was exploited by the attacker. The compromised extension downloaded a malicious binary and executed it. The malicious program connected to the attacker, which provided the attacker with a console. The attacker ran portscan and sent the information to the attacker. The connection which initiated the attack was not present in the data. 5/6 stages of the attack were present in TRACE data. Of the three analysis teams, one team detected all stages of the attack present in the TRACE data.

3) Phishing-Email-Link: Host fetched an email which was a phishing link. The user opened the email, navigated to the attached link, and entered confidential information, which was sent to the attacker. The connection that initiated the attack and the connection that exfiltrated the data were not present in the data. 1/3 stages of the attack were present in TRACE data. Of the three analysis teams, one team detected the reconnaissance stage of the attack.

4) Phishing-Email-Attachment: (Figure 7). Pine fetched an email from the attacker with a malicious binary attached. The attached malicious binary was downloaded and executed due to a vulnerability in the Pine client. The malicious program connected to the attacker, ran portscan, and sent the information to the attacker. The connection which initiated the attack was not present in the data. 4/5 attack stages were present in TRACE data. Of the three analysis teams, two teams detected all available stages of the attack.

**Engagement-3 Summary:** TRACE provided evidence of 17/21 attack steps across 4 attacks. At least one of the teams identified each of the 17 attack steps. **Lessons:** Firefox 54’s compartmentalized design led to significantly higher socket-based communication, motivating the need to track the socketpair syscall.

**D. Adversarial Engagement 4**

**Improvements.** One additional system call, socketpair, was identified to be audited. The socketpair system call was added to track the flow of data between host processes through unnamed socket pairs. This was a refinement in the data model; i.e., the source and sink of the dataflow were now identified as unnamed socket pairs. Prior to this change, this dataflow was still captured, but it was not reported that the dataflow was through an unnamed socket pair. In addition, flags for clone system call were reported to aid analysis teams. This enabled the analysis teams to detect the information shared between host processes like memory area, file descriptor table, etc. Prior to Engagement 4, the UBSI instrumented Firefox reported dependency between two units for each shared-memory communication. This was changed to report only the last dependency between any two units resulting in fewer unit-dependency records.

1) Azazel A/B: This attack involved two hosts: Host A and Host B. Host A was exploited first, and then Host A was used to connect to Host B to exfiltrate information from it. The attacker used ssh to log into Host A using stolen credentials. A malicious library was downloaded and added to the LD_PRELOAD shell variable. The update of the variable was not visible in TRACE output because no system call was used, but it’s effect was seen when a process loaded the library into its memory. For the attack, that program was nc. The hijacked nc process connected to the attacker without the user being aware. The attacker gathered system information and exits. Two hosts were involved in this attack. The attacker connected to Host A directly and then connected to Host B from Host A. The connection between Host A and Host B was not in the data. No team detected the execution of the malicious binary on Host A and Host B, but it was present in the data provided. The reconnaissance activity on Host B was present in the data but was not detected by any team. **All 4 stages of the attack were present in TRACE data. Of the two analysis teams, one team detected 3/4 stages and the other team detected 1/4 stages of the attack.**

On Host B, the attacker logged in using ssh and a malicious binary was executed to exfiltrate host information. **2/4 stages of the attack were present in TRACE data. Of the two analysis teams, none of the teams detected any stages of the attack.**

2) Drakon-Ptrace: Firefox navigated to a compromised website. The compromised website exploited a vulnerability
in Firefox and took control of the Firefox process. Firefox was hijacked by allocating memory, writing the malicious code to it, and transferring execution to it. The hijacked Firefox process was seen writing to a character device, and, as a result, the process gained root privileges. This happened because of the kernel module installed on the host as part of benign activity. The hijacked Firefox process used ptrace system call to inject shellcode into the ssdh process. As a result of injected code into ssdh process, it opened a connection to the attacker. The attacker gathered system information using the hijacked ssdh process and exit. In this engagement, the red team used a new code injection technique for in-memory execution, which used the ptrace system call. TRACE was not auditing the ptrace system call, and therefore it went undetected. 3/5 stages of the attack were present in the TRACE data. One analysis team detected all 3 stages.

3) Drakon-Lib-Inject: (Figure 8). Firefox navigated to a compromised website. The compromised website exploited a vulnerability in Firefox and took control of the Firefox process. Firefox was hijacked by allocating memory, writing the malicious code to it, and transferring execution to it. The hijacked Firefox process was seen writing to a character device, and, as a result, the process gained root privileges. This happened because of the kernel module installed on the host as part of benign activity. A Linux shared object file was downloaded by the hijacked process, which was then injected into the ssdh process. The ssdh process opened a connection to the attacker, gathers system information, and exit. The initial connection was not detected by any team because it was not present in the data. In this engagement, the red team used a new code injection technique for in-memory execution, which used the ptrace system call to make the target process load a malicious library. The ptrace event was not detected because TRACE was not auditing the ptrace system call, but its effect, the loading of the malicious library, was captured. 6/7 stages of the attack were present in the TRACE data. One of the analysis teams detected 6/7 stages.

4) VNC: The attacker used vnc to log into the host using stolen credentials. A ransomware was downloaded and executed with sudo. The ransomware encrypted user files in the home directory. The connection that initiated the attack was not detected by any team because it was not present in the data. 4/5 stages of the attack were in the TRACE data. One of the two analysis teams detected all 4 available stages.

5) Metasploit: Metasploit was used to copy a malicious binary on the host. The malicious binary was executed and opened a connection to the attacker. The attacker gathered system information and exit. The reconnaissance activity by the attacker was not detected by any team because it was not present in the data. 4/5 stages of the attack were in the TRACE data. One team detected all 4 available stages. The second team detected only one attack stage.

6) Stolen-Credential-Attack: The attacker used ssh to log into the host using stolen credentials from another compromised machine. The attacker executed a malicious program called nodeup. The nodeup process used the kernel module (described previously) to elevate the privileges to root. The attacker killed SPADE and then performed system information to send to the attacker. But since SPADE was killed, no activity was recorded beyond escalation of privileges. The connection that initiated the attack was not detected by any team because it was not present in the data. The analysis teams also did not detect the copying of the malicious executable, but it was found in the data. The execution of the malicious executable, the reconnaissance, and the data exfiltration were all missed because as part of the attack the red team killed SPADE. 2/6 stages of the attack were in the TRACE data. One of the analysis teams was able to detect one of these two stages.

7) Firefox-Drakon: Firefox navigated to a compromised website. The compromised website exploited a vulnerability in Firefox and took control of the Firefox process. Firefox was hijacked by allocating memory, writing the malicious code to it, and transferring execution to it. The hijacked Firefox process was seen writing to a character device, and, as a result, gained root privileges. This happened because of the kernel module installed on the host as part of benign activity. The hijacked Firefox process used ptrace system call to inject shellcode into the ssdh process. As a result of injected code into ssdh process, it opened a connection to the attacker. The attacker gathered host information using the hijacked ssdh process and exit. The connection that initiated the attack was not detected by any team because it was not present in the data. 4/5 stages of the attack were in the TRACE data. One of the two analysis teams detected all 4 available stages.

**Engagement-4 Summary:** TRACE provided evidence of 30/41 attack steps across 8 attacks and one of the teams identified 26/30 attack steps. **Lessons:** 3/11 missed steps were attributed to targeted attacks by APT launched against TRACE. However, there was no self-protection implemented for this evaluation which made the system vulnerable to targeted attacks. Significant performance gains were realized by moving unit-dependency upstream into Firefox. We determined that tracking ptrace calls was needed to detect code injections into ssdh.

**E. Adversarial Engagement 5**

**Improvements.** The following improvements were made to the TRACE system for Engagement 5. One additional system call, ptrace, was identified to be audited, as this was one of the hijack methods used by the red team in the previous engagement to inject code into another process. When improving the performance of SPADE’s Audit Reporter, a bottleneck was identified – the serialization and deserialization of internal data structures. In SPADE’s Audit Reporter, internal data structures are backed by a persistent storage on disk as they can grow arbitrarily because of the nature of the data. The persistent storage necessitated the serialization and deserialization of data. Prior to Engagement 5, the serialization and deserialization approach being used was the default Java approach; i.e., using the Serializable interface. In Engagement 5, this was updated to use a custom serialization and deserialization approach. As a result of the new approach, on average, we observed 324% reduction in serialization time, 1436% reduction in deserialization time, and 854% reduction in the size of the serialized data. Finally, a self-protection feature was
The cause of the privilege escalation was not present in the TRACE data, but privilege escalation was detected because of change in uid of the Firefox process in the Privilege Escalation & Code Injection component. In the same component, the hijacked Firefox process is seen using ptrace system call to inject code into the sshd process. Finally, in the Reconnaissance & Exfiltration component, the hijacked sshd process connects to the attacker, does reconnaissance, and exfiltrates information to the attacker.

1) Multiple-Performers: The attacker used ssh to log into the host using stolen credentials from another compromised machine. The attacker copied /etc/passwd and /etc/hosts files and connected to another host. All attack stages (4/4) were present in the TRACE data, but none were detected by the only analysis team.

2) Firefox-Drakon: (Figure 9), Firefox navigated to a compromised website. The compromised website exploited a vulnerability in Firefox and took control of the Firefox process. Firefox was hijacked by allocating memory, writing the malicious code to it, and transferring execution to the hijacker. The hijacked Firefox process gained root privileges using a new technique, which was not detected. However, it was seen that the hijacked Firefox process' uid changed to root. The hijacked process injected shellcode into the sshd process using ptrace. The hijacked sshd process read /etc/passwd file and other system information. This attack was not detected by any team, but the attack was found partially in the data. The in-memory execution was not found along with the reconnaissance activity by the attacker. 4/6 attack stages were present in the TRACE data, but none of the stages were detected by the lone analysis team.

3) Azazel: The attacker used ssh to log into the host using stolen credentials. A malicious library was downloaded and added to the LD_PRELOAD shell variable. The update of the variable was not visible in TRACE output because no system call was used, but its effect was seen when a process loaded the library into its memory. For the attack, that program was `nc`. The intention was to have the `nc` process connect to the attacker without the user being aware. However, the hijacked `nc` process failed to connect to attacker and could not gather system information. The analysis teams did detect the copying of the malicious library to the host, but they did not detect any more activity on the host by the attacker. The initial connection by the attacker and the use of the malicious library was present in the data. The attacker failed to do reconnaissance, and therefore it was neither detected nor present in the data. 3/4 stages of the attack were present in the TRACE data. The analysis team detected only one of the three available attack stages.

**V. ADVERSARIAL ENGAGEMENTS DATASETS**

In Table V, we summarize the datasets collected over the course of the five engagements to demonstrate their scale. For each dataset, the following information is provided:

- **CDM Avro Size (GB)**. Size in GigaBytes of the provenance stored by CDM storage module in Avro [52] format.
- **Deployment Time (Hours)**. Number of hours TRACE was deployed.
- **Total CDM Records**. Absolute number of CDM storage module records.
- **Percentage Of UBSI CDM Records**. Percentage of CDM storage module records generated by UBSI-instrumented applications.

Each dataset was collected over a widely varying period of time so that the attacks could be camouflaged among normal system activity. Thus the size of the datasets, and the events generated by UBSI-instrumented applications differ greatly between datasets. For example, in Engagement 5, the

<table>
<thead>
<tr>
<th>Engagement</th>
<th>Dataset Name</th>
<th>CDM Avro Size (GB)</th>
<th>Deployment Time (Hours)</th>
<th>Total CDM Records</th>
<th>Percentage Of UBSI CDM Records</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bovia</td>
<td>157.88</td>
<td>79.11</td>
<td>2,340,926,569</td>
<td>64.89</td>
</tr>
<tr>
<td></td>
<td>Pandex</td>
<td>100.11</td>
<td>79.09</td>
<td>1,485,144,884</td>
<td>66.23</td>
</tr>
<tr>
<td></td>
<td>Bovia-Stretch</td>
<td>11.34</td>
<td>7.99</td>
<td>168,341,409</td>
<td>65.77</td>
</tr>
<tr>
<td>2</td>
<td>Team-1</td>
<td>17.78</td>
<td>150.90</td>
<td>157,447,700</td>
<td>45.29</td>
</tr>
<tr>
<td></td>
<td>Team-2</td>
<td>20.26</td>
<td>150.88</td>
<td>171,306,963</td>
<td>22.40</td>
</tr>
<tr>
<td></td>
<td>Team-3</td>
<td>25.17</td>
<td>150.97</td>
<td>219,551,185</td>
<td>33.16</td>
</tr>
<tr>
<td>3</td>
<td>TRACE-0</td>
<td>131.15</td>
<td>255.88</td>
<td>1,017,052,486</td>
<td>10.32</td>
</tr>
<tr>
<td></td>
<td>TRACE-1</td>
<td>4.83</td>
<td>7.22</td>
<td>32,130,369</td>
<td>22.87</td>
</tr>
<tr>
<td></td>
<td>Multiple-Performers</td>
<td>4.37</td>
<td>6.78</td>
<td>29,767,637</td>
<td>22.01</td>
</tr>
<tr>
<td>4</td>
<td>TRACE-A</td>
<td>5.79</td>
<td>8.15</td>
<td>39,133,083</td>
<td>18.03</td>
</tr>
<tr>
<td></td>
<td>TRACE-B</td>
<td>8.16</td>
<td>8.04</td>
<td>48,869,496</td>
<td>21.15</td>
</tr>
<tr>
<td>5</td>
<td>TRACE-1</td>
<td>243.92</td>
<td>248.37</td>
<td>1,848,418,027</td>
<td>3.15</td>
</tr>
<tr>
<td></td>
<td>TRACE-2</td>
<td>250.97</td>
<td>248.37</td>
<td>1,909,729,948</td>
<td>2.59</td>
</tr>
<tr>
<td></td>
<td>TRACE-3</td>
<td>228.44</td>
<td>248.36</td>
<td>1,781,760,298</td>
<td>0.00006</td>
</tr>
</tbody>
</table>
the second engagement to the third is due to differences in makeup of system calls executed by the same benchmarking activity (due to version change in Firefox). For example, using the codebase from the third engagement resulted in 38 times more `mprotect` system calls than using the codebase from the second engagement.

The time taken by TRACE to process the Linux Audit logs also improved independently of the reduction in the UBSI-related activity. We see that even though the UBSI activity between the first and the second engagement only vary slightly, the overall TRACE processing time improved significantly.

### VII. Conclusion

This paper described the design and implementation of TRACE, a highly-scalable system for stream-based, enterprise-wide provenance tracking, that integrates three powerful capabilities: (i) unit-based instrumentation, (ii) distributed causality tracking, and (iii) efficient graph-based query analytics. We reported on our experience developing, evaluating, and refining TRACE for the explicit purpose of APT detection. Specifically, during the course of the four year project, the system was subject to a series of five adversarial engagements where an independent external team launched APT attacks. The streaming CDM output produced by TRACE was fed as input to three other analysis teams that implemented real-time APT detection logic. The TRACE instrumentation stack provided valuable forensic evidence to detect over 80% of the attack stages across all evaluations and our improvements led to multiple orders of magnitude reduction in time and space overheads for unit instrumentation. In future work, we plan to integrate an in-kernel cache-based audit logging system [54] with UBSI to improve the runtime performance of audit logging while minimizing the space overhead and incorporate instrumentation-free techniques, such as model-based causality inference [41]. Software and datasets from the engagements will be openly released to the research community.

### ACKNOWLEDGMENT

This material is based upon work supported by the United States Air Force (USAF) and Defense Advanced Research Agency (DARPA) under Contract FA8650-15-C-7562 and the National Science Foundation (NSF) under Grants CNS-1514503 and ACI-1547467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the USAF, DARPA, or NSF.

### REFERENCES


