AFRL-RY-WP-TR-2019-0115

CASUAL, ADAPTIVE, DISTRIBUTED, AND EFFICIENT TRACING SYSTEM (CADETS)

Amanda Strnad and Quy Messiter

BAE Systems

Robert Watson and Lucian Carata

University of Cambridge

Jonathan Anderson and Brian Kidney

Memorial University of Newfoundland

SEPTEMBER 2019
Final Report

STINFO COPY

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2019-0115 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

TOD J. REINHART
Program Manager
Resilient and Agile Avionics Branch
Spectrum Warfare Division

//Signature//

DAVE G. HAGSTROM, Chief
Resilient and Agile Avionics Branch
Spectrum Warfare Division

//Signature//

NEERAJ PUJARA
Spectrum Warfare Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2019	Final	9 June 2015 – 5 June 2019

4. TITLE AND SUBTITLE
CASUAL, ADAPTIVE, DISTRIBUTED, AND EFFICIENT TRACING SYSTEM (CADETS)

5a. CONTRACT NUMBER
FA8650-15-C-7558

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Amanda Strnad and Quy Messiter (BAE Systems)
Robert Watson and Lucian Carata (University of Cambridge)
Jonathan Anderson and Brian Kidney (Memorial University of Newfoundland)

5d. PROJECT NUMBER
DARPA

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
Y1AY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

BAE Systems
600 District Ave.
Burlington, MA 01803

University of Cambridge

Memorial University of Newfoundland

8. PERFORMING ORGANIZATION REPORT NUMBER
AFRL-RY-WP-TR-2019-0115

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency (DARPA/I2O)
675 North Randolph St.
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2019-0115

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The U.S. Government has for itself and others acting on its behalf an unlimited, paid-up, nonexclusive, irrevocable worldwide license to use, modify, reproduce, release, perform, display, or disclose the work by or on behalf of the U.S. Government. Report contains color.

14. ABSTRACT
CADETS unifies kernel and userspace tracing to provide provenance information that interconnects events across multiple nodes in a distributed system. CADETS combines and improves on the FreeBSD Audit and DTrace subsystems, and extends the original DTrace design to support distributed tracing across multiple hosts. Its Loom and LLVM-prov frameworks leverage the LLVM compiler for software instrumentation and for bridging userspace and kernel space information flow with precision.

15. SUBJECT TERMS
software instrumentation, provenance, information flow, FreeBSD, DTrace, LLVM, hypervisor

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Tod Reinhart
Unclassified	Unclassified	Unclassified	SAR	57	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

TABLE OF CONTENTS

List of Figures

List of Tables

1 SUMMARY

Computers today face sophisticated attacks, which can remain hidden on the compromised machine for long periods of time, slowly gathering information to exfiltrate. Some spread from machine to machine across the network. Unfortunately, over the past 30 years, tools haven't changed enough to keep up with the innovations in attacks.

As part of the DARPA Transparent Computing (TC) program, we created a Causal, Adaptive, Distributed, and Efficient Tracing System (CADETS). CADETS improves on existing audit and information flow tracking systems that exist today, and brings them into the 21st century.

Initially, CADETS worked to improve the tracing available on FreeBSD. We did this by expanding the sources of information available to DTrace to include data from audit, a security logging program that meets the Common Criteria (CC) Common Access Protection Profile (CAPP), along with adding persistent universally unique identifiers (UUIDs) to kernel objects.

CADETS has greatly improved the capability of FreeBSD to produce detailed traces across machines. We have learned first-hand the difficulties of tracing distributed systems and started work to allow distributed tracing across heterogeneous machines in our Distributed DTrace (DDTrace) capability.

Application developers are focused on the interesting behavior of their own programs. While they may add debugging features to help debug their program, these may not be the same information useful for understanding compromised systems. By modifying the LLVM compiler, we were able to add information including provenance data to traces without manually making changes to each individual application. This is accomplished using our Loom/LLVM-Prov frameworks.

2 INTRODUCTION

2.1 Problem Description

Contemporary computer systems face attackers who are sophisticated, pervasive, and persistent, and who are also able to exploit information asymmetries that allow them to “hide in plain sight” due to the opacity of current software designs. Security and systems researchers have little to boast about; current auditing and forensic tools are trapped in the 1980s and 1990s as: (1) they focus on symptomatic events that are easy to log (e.g., system calls that were the focus of the Orange Book or Common Criteria) rather than causal relationships (e.g., relating application-internal security behavior to system-wide events); (2) they remain fundamentally local as our most critical systems (and hence attacks on them) have become distributed, scaling poorly with respect to both performance and analysis capabilities; (3) they are unresponsive to changes in analyst requirements, especially as forensic investigators shift their focus within live systems; and (4) they fail to support a virtuous cycle in which analysts and software authors improve the self-descriptive capabilities of deployed systems as experience is gained, to make them more responsive to analysis over time.

2.2 Research Goals

To address flaws in current systems, we developed CADETS, a solution grounded in fundamental improvements in dynamic instrumentation, scalable distributed tracing, and programming-language support for compiler driven instrumentation. CADETS enables analysts to explore both historic and live events by providing provenance data well suited for causal backtracking and temporal pattern matching. To support the causal analyses required for in-field forensics, CADETS introduces *local and distributed information-flow* to be accomplished through new *static and dynamic software instrumentation techniques* and a *distributed tracing model*. Local and distributed information flow tracking provides fine-grained inspection of local systems as well as holistic visibility into and correlation of suspicious activities across nodes in a network. New static analyses are needed to automate instrumentation of userspace applications without requiring modification of source code. Furthermore, static interprocedural analysis helps to define precise source/sink relationships, avoiding data explosion in provenance tracking. New dynamic software instrumentation techniques are required to extend the types of events and data that can be captured in a consistent and reliable manner. Finally, a distributed tracing model offers a cohesive and scalable information tracking solution that spans multiple systems. Our target platform is FreeBSD, although many of the technologies we will be using and developing could apply to other deployed systems. Intrinsic to the CADETS design is consideration that performance is a critical aspect with goals to minimize memory, compute, and network overhead.

The instrumentation of userspace software without modifying the source code allows for many parts of the operating system to be instrumented with minimal intervention by the end user. DTrace provided some of this functionality through DTrace providers (e.g. function boundary tracing) but instrumentation using these tools often lacked semantics. In a previous DARPA project, CRASH, members of the CADETS team had developed a tool for verifying temporal assertions in software called

Temporally-Enhanced Security Logic Assertions (TESLA). It was recognized that the static analysis and instrumentation portion of this tool could be extracted and modified to be used as a general instrumentation framework. The results of this work is Loom. In order to maintain the same level of visibility as information flows from userspace to the operating system (OS) kernel, advancement in the integration of userspace and kernel space tracing is required. LLVM-Prov leverages Loom to address this need and explores static analysis methods to enhance provenance tracking.

CADETS exploits recent advances in OS and compiler instrumentation and tracing technologies that offer new insight into whole-system behavior. Part of our work is refocusing existing frameworks for forensic investigation rather than debugging or performance measurement – among other things, improving integrity, confidentiality, and reliability, namely with the FreeBSD Audit and DTrace systems. In addition, we also introduced support for distributed causal analysis at scale through the development of DDTrace which leverages DTrace to provide tracing across nodes networked together as part of a distributed system. We also implemented a hypervisor tracing capability called HyperTrace that allows tracing of guest virtual machines (VMs) from a single host.

While DTrace and Loom/LLVM-Prov are the crux of our research, we've also explored other research areas to bolster the operation use of our system. This includes new graph data format specifications and graph database optimizations.

2.3 Evolution of Approach

Initially, we sought to achieve our research goals based on the concept of the following components:

- Event Query (EQ) is a query and instrumentation language for use by both human analysts and automated systems. It would allow analysts to specify queries that would direct instrumentation across local and distributed systems.
- WATCHMAN is a framework to instrument and trace operating system and application behavior within local systems. It is responsible for taking queries from the EQ front end and compiling to a variety of tracing, temporal tracking, and information flow back ends.
- DEQUE: DISTRIBUTED EVENT QUERIES FOR EQ links both a front-end analytics system fielding EQ with WATCHMAN agents on nodes across the heterogeneous distributed system, and coordinates tracing between nodes in the system.

Figure 1 depicts these concepts.

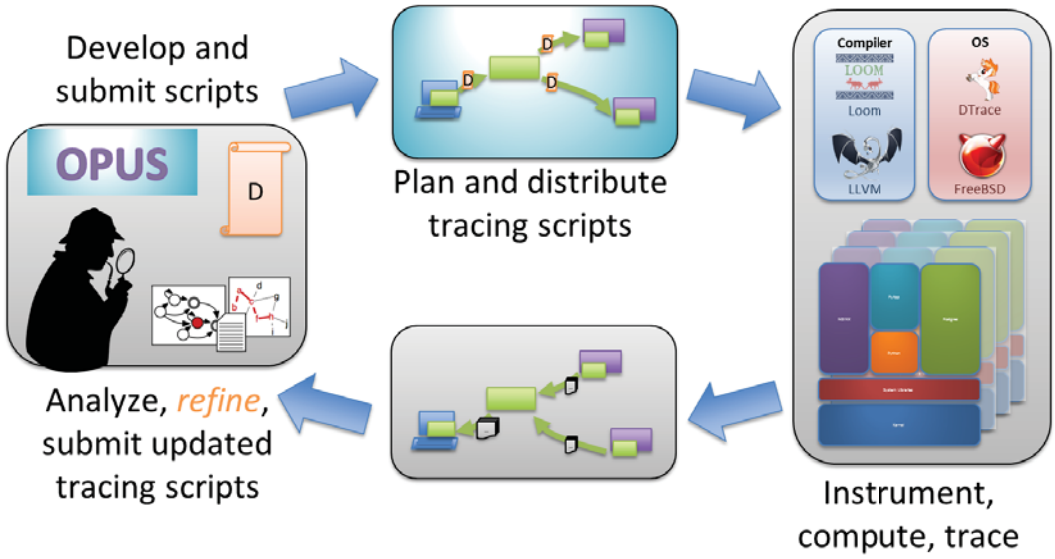

Figure 1 The CADETS workflow

This first design revolved around the idea of enabling an adaptive instrumentation approach where the CADETS system can adjust to selective levels of data focus or resolution. However, early achievements in DTrace and Loom/LLVM-Prov development, as well as the impracticality of that approach during live engagements where multiple TA2 performers are supported, lead to a different architecture.

2.3.1. EQ Simplified to D Language Improvements

In our initial efforts towards the development of an EQ language (that borrows syntax from the D language and TESLA's C-based temporal assertions), we found that the natural and pragmatic approach was to reuse the D language itself to specify the type and granularity of tracing across programs that run on one or more hosts. Only small changes were made to the D language, allowing us to avoid alienating core users from the open source community while also providing the necessary instrumentation granularity.

Furthermore, to provide higher level semantics, we have adapted the Observed Provenance in User Space (OPUS) system to allow TA2 teams or analysts to answer provenance and trust questions (e.g. what was the sequence of steps that resulted in the creation of this graph node?). OPUS uses a Portable Operating System Interface (POSIX) compliant Provenance and Versioning Model (PVM). While useful, this model is too tightly coupled to the POSIX semantics. Thus, we updated the PVM model to a second version (PVMv2) to allow us to express and explore other non-POSIX compliant models. This version is implemented in the libPVM library.

2.3.2. WATCHMAN Replaced by DTrace and Loom/LLVM-Prov

The original goal was to design and implement a framework to instrument and trace OS and application behavior within both local and distributed systems. Leveraging

existing instrumentation mechanisms, CADETS proposed to develop new techniques such as just-in-time re-instrumentation based on “fat binaries” -where LLVM intermediate representation (IR) is stored along with the native instruction stream-, extend DTrace to support information flow and temporal expressions, develop new instrumentation agents, add information flow through OS and across programs, link automata to information flow tags, and address implicit flows.

However, early successes achieved with DTrace and Loom/LLVM-Prov technologies lead to the abandonment of the WATCHMAN implementation. We instead rely on DTrace (in the kernel space) and Loom/LLVM-Prov (in the userspace) frameworks to perform instrumentation and tracing at both the local and distributed level. Our approach to implementing the instrumentation agents is motivated by the need to provide transparency which in turn facilitates not only provenance analysis, but also debugging and performance analysis. To this extent, our audit provider (dtaudit) bridges the gap between debugging and auditing. In addition to creating new providers for DTrace (i.e. dtaudit, mbuf), we have also fixed bugs and made improvements to the framework. Our changes have been upstreamed to FreeBSD and are being used by the open source community.

In the userspace, Loom/LLVM-Prov performs instrumentation and tracing. Loom automatically weaves into programs a new system call (dt_probe) that reports arbitrary event data. LLVM-Prov, an instrumentation framework based on LLVM, provides provenance tracking by performing automatic intra- and inter-procedure analysis without modifying the source code. LLVM-Prov also provides support for MetaIO by augmenting system calls to track UUID. Furthermore, to enable analysis and instrumentation across entire programs, we have provided support for LLVM IR fat libraries (i.e. all the FreeBSD libraries are compiled this way).

2.3.3. DEQUE Redefined as DDTrace

As EQ and WATCHMAN were reconceptualized, DEQUE’s role, originally to distribute EQ event queries to WATCHMAN nodes, was also redefined. However, its overall goal of providing a distributed tracing framework that reconciles the trace data across layers and nodes is served by DDTrace.

2.4 System Overview

DTrace technology and the Loom/LLVM-Prov framework form the basis of instrumentation for the current CADETS system. At the kernel level, existing and new CADETS DTrace probes record an expanded set of system behaviors. For userspace applications, Loom user-defined policy files are used to establish the instrumentation points for target applications and libraries as well as throughout the FreeBSD OS. In addition, enhanced kernel data flow tracking is provided by LLVM-Prov. These technologies are integrated to achieve whole system information tracking across multiple hosts. While DTrace and the Loom/LLVM-Prov framework are the key components of CADETS instrumentation, the system is supported by other components that contributed to the operation and workflow of CADETS as illustrated in Figure 2. A description of each component in CADETS is provided below. For more detailed information regarding the implementation of each component, please consult our Software Subsystem Design Document (SSDD).

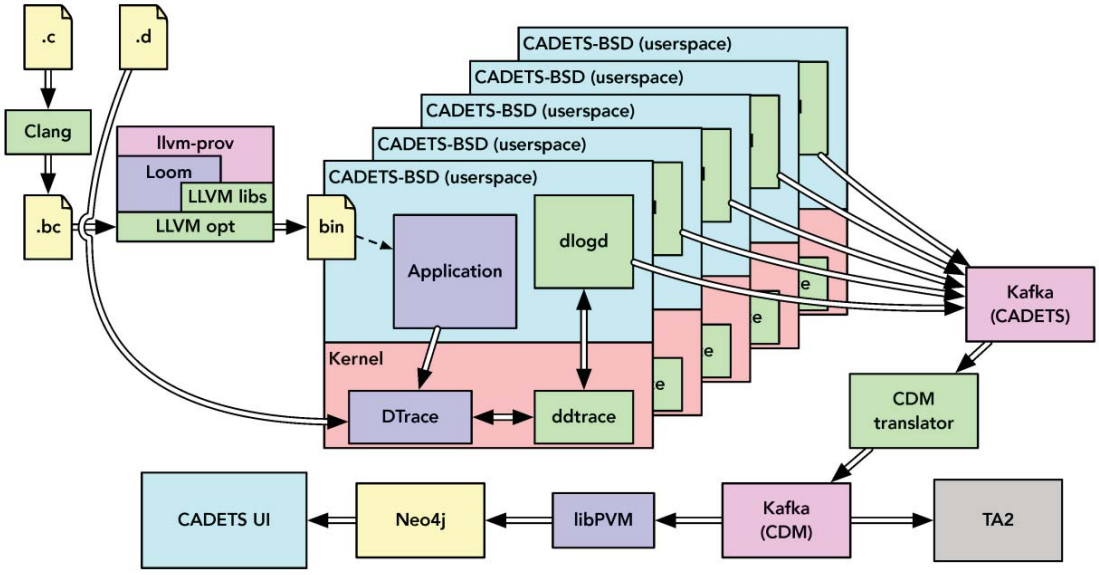

Figure 2 Overview of the CADETS data-gathering pipeline

2.4.1. FreeBSD Audit

The FreeBSD audit implementation targets the Common Criteria (CC) Controlled Access Protection Profile (CAPP), which mandates logging of security-critical system events. We have extended the audit implementation as follows to take into account the specific requirements of the DARPA Transparent Computing program, including capturing new events, associating UUIDs with key subjects and objects in the kernel, adding Message IDs to Inter-Process Communication (IPC) and network packets, and capturing additional contextual information such as system-call arguments. Many of these features have been upstreamed to FreeBSD, appearing in FreeBSD 11.x and 12.0.

2.4.2. Audit DTrace Provider

We have implemented a new Audit DTrace Provider (dtaudit) that allows DTrace to directly instrument and capture audit events. The DTrace provider observes the set of audit events available in the system, gathered at boot from the system audit configuration, and as required after boot by the audit daemon (auditd). Two DTrace probes are exposed for each possible event: a probe that fires when a record is committed on system-call completion, and later when the record is converted to basic security module (BSM) to be written to disk. DTrace scripts can specify a set of audit events to instrument, and gain access to the in-kernel data structure—and, in the case of the BSM hook, also the associated BSM-formatted record. We have upstreamed this support to FreeBSD, appearing in FreeBSD 12.0.

2.4.3. DTrace

Dynamic Tracing, or DTrace, is an operating-system tracing facility that allows programmer or administrator-provided D-language scripts to instrument system behavior, process results in kernel, and log them. We employ a CADETS-specific audit.d script that instruments key audit events using the Audit Provider, and then logs them in a JavaScript object notation (JSON) format to DTrace output.

In earlier program engagement events, we sent the JSON output to DTrace stdout, from which it was converted to the Transparent Computing Common Data Model (CDM), and then sent over Kafka. In the final two engagements, DDTrace has been integrated directly into the in-kernel DTrace implementation and is used instead.

2.4.4. Hypervisor Tracing

Hypervisor tracing allows DTrace in a FreeBSD bhyve hypervisor host to instrument DTrace probes in bhyve guests, permitting a single D script to instrument and process data from the host and multiple virtual machines. This work consists of several parts, including VirtIO drivers allowing the host and guest to asynchronously exchange probe information, a new hypercall to allow the guest to synchronously notify the host DTrace instance of probes firing, and modest DTrace extensions to allow selection between guest and host probes when writing a script.

2.4.5. DDTrace

Distributed DTrace allows DTrace output gathered on one system to be forwarded, via dlog, a reliable distributed queue based on the Kafka protocol, to other systems in a wider distributed system. dlog has multiple parts to its implementation, including a kernel component allowing DTrace to feed output into the log, and dlogd, which reliably streams logs over the network to a Kafka server. It also addresses heterogeneity issues arising from differences in instrumented nodes—e.g., as may arise due to different underlying architecture, kernel modules loaded, or operating-system version—by transmitting DTrace state such as data types over the same queue. Some further changes were also made to DTrace to permit reliable tracing during early boot and shutdown.

2.4.6. Loom

Loom is a general-purpose library for adding instrumentation to software in the LLVM IR format. It is currently capable of generating static instrumentation points that expose values (e.g., function parameters and return values) to software-defined instrumentation functions. Loom is used to add additional instrumentation to userland binaries, both programs and libraries, in FreeBSD. The resulting data is passed into DTrace to be collected into the CADETS trace.

2.4.7. LLVM-Prov

Using Loom, LLVM-Prov adds to the instrumentation by propagating precise user-space provenance. It does this with the support of new kernel system calls via a

mechanism called MetaIO. By providing insight into source-to-sink information flow at the kernel level, LLVM-Prov helps to avoid the provenance path explosion problem that occurs when the lack of visibility results in inferences about the causal dependencies of data sources/sinks.

2.4.8. CDM Translator

The CDM Translator is a program which converts the CADETS traces from the JSON format generated by the DTrace scripts to the CDM format. CDM is the standard format for all TA1s on TC to produce.

The CADETS JSON data is event-based, while CDM uses additional objects in addition to events. The CDM Translator uses semantic knowledge of the events on a CADETS system to create the objects, such as FileObjects or NetFlowObjects, as reliably as possible.

2.4.9. LibPVM

LibPVM is a library enabling the CADETS output data to be transformed from a linear system log to a provenance graph on the TA1-side. This provides a graph model of the data which accurately reflects the semantics of the underlying trace, irrespective of the final serialization format such as PVM-graph, CDM, comma-separated values (CSV). The library includes functionality for data ingestion, on-the-fly graph generation, and limited querying capabilities.

The library leverages work done on provenance modelling (PVM, the Provenance Versioning Model) and has its own serialization format (PVM-graph).

LibPVM was used as a data quality control component in CADETS, playing the role of an in-house TA2 for the purposes of understanding what information is missing from the trace and whether attack components have been captured.

2.4.10. Neo4j Database Optimization

LibPVM uses the Neo4j graph database for storing and querying the PVM-graph serialization of the data. Especially when used for real-time ingestion, the quantity of data places a significant stress on the database backend, which is not designed for write-heavy workloads.

As both libPVM and other TA2 performers were using Neo4j as a backend for their data, we have created a new storage backend for Neo4j that is optimised for the provenance-ingestion usecase (better write-concurrency support, better on-disk layout).

2.4.11. User Interface (UI)

While Neo4j is a graph visualization tool, it is not designed as an interactive tool for investigating a security attack. The CADETS UI offers practical search panes and views to enable analysts to inspect individual nodes (representing processes and I/O objects etc.) and explore PVM data to build attack graphs.

2.5 Upstreaming and Transition

Throughout the CADETS project, our goal has been to upstream improvements to open-source software, increasing the chances of industrial adoption. This has included:

- Upstreamed improvements to the FreeBSD audit implementation, appearing in FreeBSD 11.0 and later.
- The FreeBSD DTrace Audit Provider, appearing in FreeBSD 12.0 and later.
- A variety of audit-event additions and other audit-framework improvements appearing in the OpenBSM open-source GitHub repository.

We have created a new OpenDTrace distribution of DTrace, based on the FreeBSD and Mac OS X implementations, which has now been adopted by Microsoft in their Windows implementation of DTrace.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Tracing User Space

User-space tracing evolved over the life of the CADETS project, beginning with infrastructure for pure user-space tracing, evolving into a set of tools that empower better kernel-space tracing and finally growing into tools that provide both user- and kernel-space inputs into CADETS traces.

Loom was developed as a generalized instrumentation framework, based on previous work under the DARPA CRASH program called TESLA. That work, which is a part of the Clean Slate Trustworthy Secure Research and Development (CTSRD) project, used LLVM to instrument software for dynamic validation of temporal security properties. TESLA provided two forms of inspiration for the CADETS project: its temporal assertion language inspired our initial thoughts on the EQ language and its instrumentation library inspired the development of Loom.

Loom is a general-purpose framework for instrumenting any program that is expressed in the LLVM IR. Unlike the TESLA instrumentation code that was interwoven with the semantics of temporal assertion checking, Loom can be used to add arbitrary instrumentation driven by either a policy file or by direct application programming interface (API) interaction. These interfaces evolved throughout the course of the CADETS project.

The policy file interface was added to Loom at the very beginning of the CADETS project, even before any instrumentation code had been written. Our original vision for user-space tracing was that instrumented code would submit trace events to a user-space service (WATCHMAN) that would aggregate traces from multiple sources on a host and forward them along to higher-level aggregators elsewhere in the network. Over the first year of the project we developed the basic framework, added multiple forms of logging output and implemented machine-readable output via libxo.

In the second year of the project, we began to focus on the API-driven use case for Loom as we started the development of LLVM-Prov. The LLVM-Prov use case was the primary driver for Loom in the second and third years of the project, but in the fourth year of the CADETS project we renewed our efforts to use Loom as a standalone source of trace events. Figure 3 provides a high-level overview of Loom/LLVM-prov.

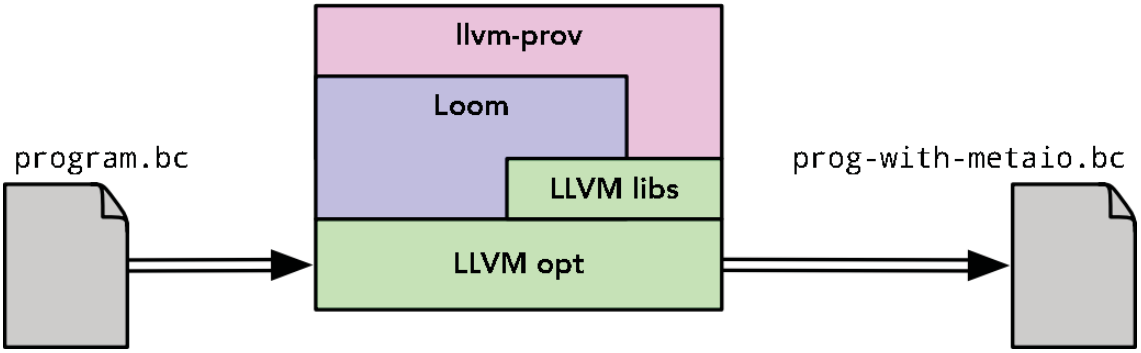

Figure 3 Loom/LLVM-prov Overview

3.1.1. Loom

Loom leverages the LLVM compiler tool change to do much of the mechanical work for instrumenting source code. LLVM compiler front-ends parse and compile the original source code to form of high level assembly code known as LLVM IR. Next, LLVM employs multiple optimization and transformation passes known as opt passes before the final machine code is produced to be run on the target platform. This workflow is shown in Figure 4. Loom is implemented as an LLVM opt pass.

In order to use Loom, the end user must provide an instrumentation policy. This policy defines the parts of the code to be instrumented and how to output the instrumentation results. This configuration is specified in a Yet-Another-Markup-Language (YAML) file, the details of which are discussed in a later section.

Using this policy, the Loom opt pass runs in two phases: instrumentation point identification; and instrumentation insertion. During the first phase static analysis is done to determine all of the points of interest within the code base. Once this list is gathered, the second phase adds the required instrumentation code to each instrumentation point.

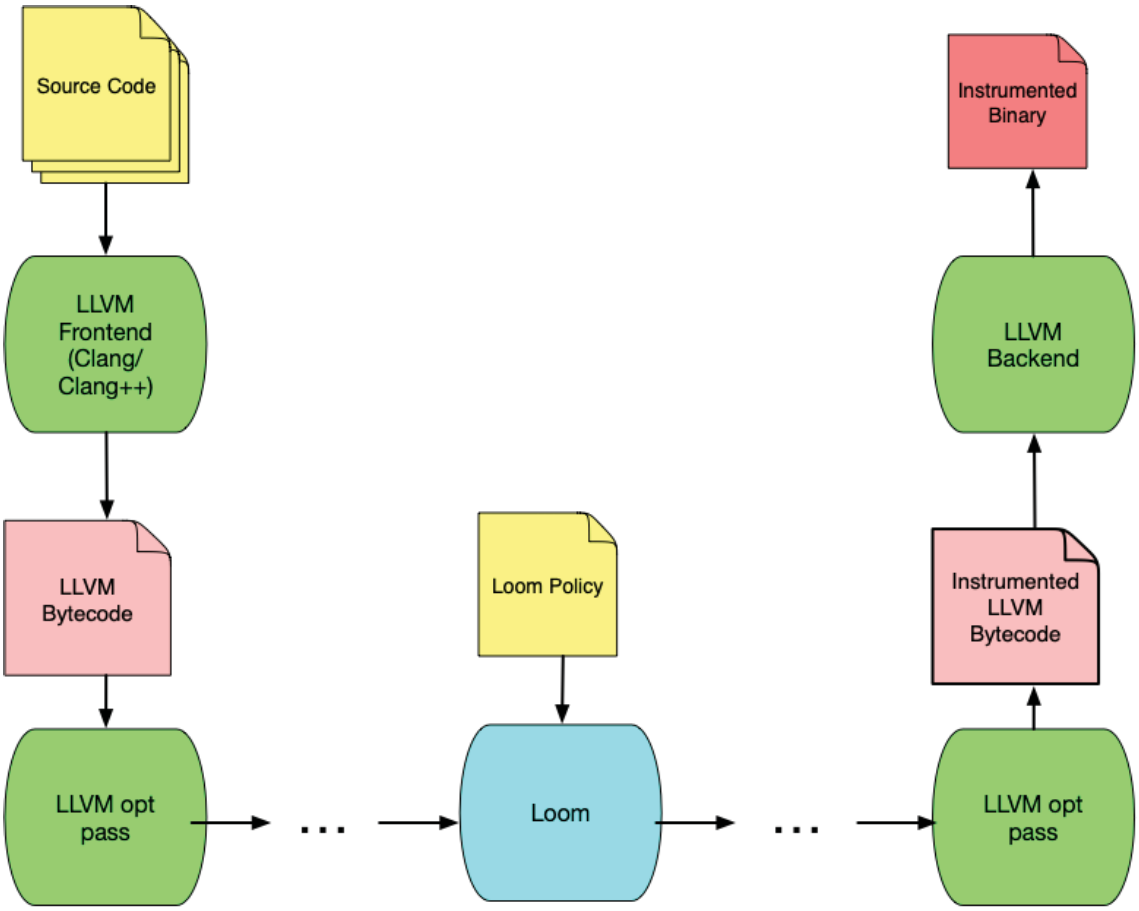

Figure 4 LLVM Compilation Flow

3.1.1.1. Initial Loom Functionality

The initial Loom implementation was limited to the functionality of instrumentation available in TESLA. This code formed the foundation upon which a general purpose tool was built. From TESLA, Loom inherited the static analysis code used to identify a limited set of instrumentation points. Additional development was required to add the policy file interface and the logging infrastructure to output results.

The initial implementation of Loom had a limited set of possible instrumentation points. It could instrument function calls either from the caller or the callee context. When instrumenting these calls, Loom had access to the call parameters. Additionally, Loom could instrument structures. For all structures it was possible to instrument reads and/or writes of each individual field.

To allow the user to define the instrumentation points and configure the output, a policy file was defined in YAML and parser for the file added to the tool. Using the policy file, the end user could define the functions and fields to instrument by name. Additionally, the conditions under which they should be instrumented could be defined (caller/callee for functions, and read/write for fields). The general configuration of Loom, including choice of logger, could also be specified. More details on the policy file format and options can be found in the CADETS SSDD.

The final piece to be added to produce a general purpose instrumentation framework was output handling. To do this, Loom defined not just a single logging output, but instead an API upon which many loggers could be defined. Initially it was possible to log the instrumentation results using simple text, extensible markup language (XML), JSON and through the FreeBSD kernel tracing tool ktrace(1). However, by defining an API for adding other loggers in the future, CADETS could use the tool as is for testing and defer selecting the appropriate output until better equipped to decide.

3.1.1.2. Userland Tracing with DTrace

From early on in the project, CADETS decided to use DTrace as the cornerstone of our tracing system. DTrace was already tightly integrated into FreeBSD, satisfying a large portion of the project's tracing requirements within the kernel. In areas where the functionality was lacking, DTrace allowed for it to be added through existing APIs. In userspace however, DTrace did not provide a mechanism to easily instrument code. This type of instrumentation was what Loom was designed to provide, but the CADETS team had to develop a mechanism to get Loom outputs into the DTrace framework.

Initially the team investigated DTrace's own mechanism for userspace tracing, Userland Statically Defined Tracing (USDT). One of the problems with using USDT directly was it required modification of the original source code. It was deemed infeasible to have to modify large amounts of userspace code, both from the initial coding effort and future maintenance cost.

Using the mechanisms employed by USDT directly in Loom was the first option investigated for the Loom/DTrace integration. However, it was quickly determined this would not be possible due to the technical limitations of Loom. USDT uses custom Executable and Linkable Format (ELF) sections that are added to a binary to define instrumentation points and the code to be run. These sections are added to a binary upon linking, in the final stage in the compiler chain. Loom runs before the linking stage and therefore does not have the ability to add ELF sections.

In rethinking the DTrace integration problem, it was decided that Loom could use Statically Defined Tracing (SDT) probes that exist in the kernel. These probes are usually used to add additional tracing point in kernel code and do not rely on the mechanisms used in USDT. In order to pass data from userspace to the kernel a system call, dt_probe(2), was added to FreeBSD.

3.1.1.3. dt_probe(2) System Call

When instrumenting software with Loom and DTrace, Loom adds calls to the dt_probe(2) system call. dt_probe(2) is a simple system call implemented in the CADETS fork of FreeBSD that takes pointers to userspace data and passes them to the DTrace kernel module with a SDT(...) function provided by the DTrace SDT provider. These pointers can then be used to access the data values for inclusion in the CADETS trace.

Since its initial implementation for the project, the system call definition for dt_probe(2) has not changed but its semantics have. Initially, the call parameters were expected to be up to six pointers to data that was being instrumented. However, due to how DTrace probes are identified, using the SDT provider did not allow for unique identification of the problem site.

To solve this issue, it was decided to use the first parameter of dt_probe(2) as an identifier, leaving the other five parameters for instrumentation data. This design change required the addition of metadata to the Loom policy file. In the new policy format, the end user could specify a metadata id for each instrumentation point making them uniquely distinguishable.

3.1.1.4. Loom Extensions

Additional functionality was added to Loom throughout the project to extend its capabilities and ease of instrumentation. The ability to add instrument points to global variables and variadic functions was added. However, variadic function instrumentation is limited and can only be instrumented on the caller side where the number of arguments to the function is known.

As part of their Transparent Computing project, the Tracking and Analysis of Causality at Enterprise level (TRACE) team also used Loom to instrument software. They added the functionality to instrument pointers as needed for their work. This code was provided to the CADETS team and integrated into the main Loom repository.

Parameters were added to the Loom policy files to help identify instrumentation sites. Wildcards were added to allow matching against multiple named arguments. The ability to limit the scope of instrumentation to a single file was also added. This is useful in projects with large amounts of variable name reuse.

3.1.1.5. Data Transformation

Since the Transparent Computing CDM relies heavily on UUIDs, it was necessary to transform some userspace values before adding them to the CADETS trace. A proof-of-concept transforms framework was added to Loom for this purpose. The framework allows arbitrary functions to be defined and called using instrumented variables, logging the result. The policy file syntax was updated to allow for the specification of transforms to be performed in individual variables. The initial version of this functionality includes the necessary function to retrieve UUIDs for file or process descriptors in FreeBSD. This performs as expected but the mechanism for adding functions remains a bit cumbersome. Future work will be done to make the addition of arbitrary functions more seamless.

3.1.1.6. Build System Integration

Finally, in order to automate the use of Loom within FreeBSD, the operating system’s build system was modified to allow for instrumentation. Rules were added to build LLVM intermediate representation for programs and libraries to make it

easier for testing on individual parts of the OS. Policy files can also be defined for the entire operating system and passed into the full build process.

3.1.2. LLVM-Prov

LLVM-Prov was developed to address the *n* × *m problem:* the explosion of complexity that provenance algorithms face when presented with nodes that contain many sources and many sinks with many possible provenance paths. The development of LLVM-Prov began midway through the second year of the CADETS project. At that time, we identified two complementary needs within the program:

1. the need for userspace and kernel tracing to refer to the same objects with identifiers that are not subject to data races (e.g., file descriptors), and

2. the need for kernel-based tracing to have increased visibility into userspace information flows to avoid a combinatorial explosion in provenance-graph processing (the *n* × *m problem*).

We designed the MetaIO mechanism to address these two needs. We then spent the remainder of the program improving the mechanism and its static information-flow analyses, to improve trace results. We also invested significant effort in integration activities to ensure that both our team and the BBN team would be able to automatically build our LLVM-based toolchain, use that toolchain to build the CADETS variant of FreeBSD, build CADETS FreeBSD images and then test or run those images. Compiler modifications are high-risk activities in the context of a large project, so we invested very significant amounts of time in continuous integration (CI) development before engagements and then, sometimes, in CI debugging in the lead-up to engagements.

3.1.2.1. Key Hypothesis

The central hypothesis of our LLVM-Prov work was that the fusion of user- and kernel-space provenance information via the system-call layer can provide provenance information that is more reliable than pure userspace tracing and more precise than pure kernel-space. We explored this hypothesis by:

- developing compiler-based tools to analyze information flow within userspace software (see Tools for Information Flow Analysis section),
- augmenting the FreeBSD system call interface to pass provenance information (“MetaIO”) across system calls without data races (see MetaIO section),
- developing a Loom-based transformation to translate applications’ use of POSIX system calls into MetaIO calls (see Loom-based MetaIO transformation section) and
- augmenting the Transparent Computer Common Data Model (CDM) to carry *Provenance Assertions* derived from userspace-supplied MetaIO information (see CDM Augmentation section).

3.1.2.2. Tools for Information Flow Analysis

We developed several tools for analyzing information flows within userspace software during the course of this program. We began by adding pure LLVM operand flow tracking within LLVM-Prov. This tracking used LLVM's existing API for exposing value/instruction operands to build up use–def chains from information flow sources (e.g., read(2)) to information flow sinks (e.g., write(2)) within a single procedure. We began by hard-coding the set of I/O source and sink system calls within LLVM-Prov, then developed software that would allow C-language annotations to be applied to the header files containing I/O source and sink declarations. We improved our approach to information-flow analysis by adding memory-based data dependencies using LLVM's MemorySSA framework. This API provides the ability to augment the use–def graph with mod–ref edges based on an analysis of instructions and functions that are available in the LLVM intermediate representation (IR). For example, a function that can be proved to only read from memory can only act as a sink, whereas functions with unknown memory access patterns could potentially write to arbitrary memory. Analysis precision is improved through the use of alias analysis in conjunction with MemorySSA, but this analysis is still entirely confined to a single procedure.

We took several approaches to the discovery of interprocedural information flows during the program. First, we build software to export use–def chains into a form that could be read by external graph computation tools. We built a tool called py-cdg (Python Call and Data Graph) that could ingest this information and expose it to standard graph analyses provided by the Python NetworkX library. After adding mod–ref information, however, naming the various memory states in a manner that is portable across the LLVM passes and the external Python tool became impractical. We then turned to evaluating the suitable of the Static Value–Flow (SVF) interprocedural analysis in our environment.

SVF analysis is a technique and set of software tools for building interprocedural information-flow graphs in LLVM IR bitcode. We explored the approach as potentially providing additional information to enhance LLVM-Prov–based instrumentation, but its license, GNU General Public License version 3 (GPLv3) was problematic for the CADETS project. Towards the end of the program we were starting to explore mechanisms for running external SVF tools on uninstrumented binaries, capturing information-flow information from their output and bringing that information back into our LLVM-Prov library, under the University of Illinois (UIUC)/National Center for Supercomputing Applications (NCSA) Open Source License). This work remains exploratory at the conclusion of the program.

3.1.2.3. MetaIO

The MetaIO mechanism allows user- and kernel-space tracing mechanisms to reliably name objects consistently without races. Existing naming mechanisms such as file descriptors are inherently racy: it is possible for one userspace thread to begin a system call with respect to a particular file descriptor number while another thread replaces that descriptor with another descriptor — leading to confusion about which file is actually being accessed — or for a userspace trace record to use a file descriptor number to refer to a different file than the corresponding kernel trace

record if a file has been closed and a new file opened. The MetaIO mechanism prevents such confusion by adding a new set of system calls that either emit or accept more explicit naming information, e.g., UUIDs for files or lightweight message IDs for inter-process communication. Information about which file is being read from, for example, comes not from the system-call interface layer but from the deeper kernel layer that is actually performing the I/O. This information can then be copied into userspace in the case of I/O sources; information can then be passed through user space and back into the kernel when an I/O sink system call is invoked. This explicit passing of I/O metadata allows tracing information from both user- and kernel-space to synchronize their naming of files and other relevant kernel objects.

3.1.2.4. Loom-based MetaIO transformation

Given the availability of information-flow analysis, LLVM-Prov is able to detect information flows from I/O sources to I/O sinks. In order to address the *n* × *m problem*, these sources and sinks must be adapted to use their MetaIO counterparts and the program must be modified to pass I/O metadata along the same paths as the data flowing through the program. LLVM-Prov uses the Loom instrumentation framework (see Loom section) to perform this modification, using Loom's external API to augment the relevant system calls to take a pointer to a MetaIO structure, and ensuring that memory for the MetaIO structure is allocated in a place that will allow the information to be preserved from source to sink. Figure 5 shows an example of how code is instrumented for MetaIO.

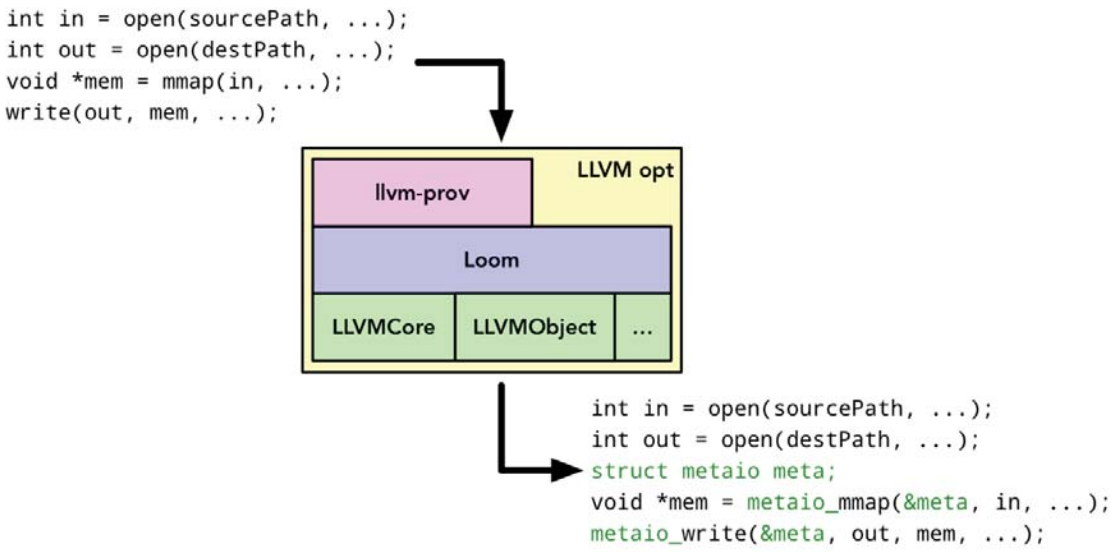

Figure 5 MetaIO Instrumentation Example

3.1.2.5. CDM Augmentation

In order to propagate MetaIO information to TA2 groups, we proposed additions to the CDM. These additions added a new type, ProvenanceAssertion, which asserts that the data it is attached to was derived from a specific source (identified by UUID). The CDM Value type was modified to accept an optional ProvenanceAssertion field, allowing

any Value received by a system call to be annotated with provenance information supplied by instrumented userspace software. Since this assertion comes from potentially untrusted sources (userspace programs), a ProvenanceAssertion can itself include a ProvenanceAssertion about its own provenance, e.g., “this host says that the kernel says that this program says that the data came from this specific file”.

3.2 Tracing Kernel Space

Kernel tracing serves a number of key functions:

- Many security-essential events are implemented within the kernel, including file, network, and IPC events.
- The kernel is key part of the Trusted Computing Base (TCB) for the operating system.
- The kernel can provide strong security and reliability guarantees for trace capture and storage.

However, the existing kernel tracing mechanisms, Audit, which implements Common Criteria security-event auditing, and DTrace, which provides highly configurable implementation-oriented tracing, fail to provide the detailed events required by CADETS, nor a suitable framework in which to capture and distribute them. To this end, we have implemented a number of new components, in both userspace and kernel, to address these gaps:

- FreeBSD Audit has been extended to capture Transparent Computing-relevant events, arguments, and return values beyond those specified by the Common Criteria.
- The kernel has been extended to associate UUIDs and Message IDs with key kernel subjects, objects, and IPC messages.
- A new DTrace Audit Provider has been implemented, providing configurable access to audit events from within the DTrace implementation.
- New guest-from-host/hypervisor tracing support has been added to DTrace to allow the hypervisor (host) operating system to instrument guest virtual machines.
- The new DDTrace implementation allows logs to be reliably stored to disk by the kernel, and then be shipped securely and reliably over the network to a distributed Kafka message queue.
- Significant improvements have been made to the reliability and completeness of Audit and DTrace to allow capture of early boot, steady state, late shutdown events without loss.

3.2.1. Integrating FreeBSD Audit and DTrace

One of our key hypotheses was that we could integrate previously entirely independent approaches to CAPP security event auditing and tracing designed for debugging. Security event auditing, as embodied by OpenBSM, has historically focused on secure, accurate, and reliable capture of security-critical events such as

file-access system calls—at the cost of significantly impacting performance, excluding support for other operation types, and accepting a fail-stop approach to reliability. Debugging and tracing mechanisms have instead been focused on performance analysis and optimization, accepting unreliable behavior in return for continued system optimization, and likewise potentially inaccurate access to data (e.g., due to race conditions) rather than interfering with kernel implementation. They have also provided much deeper opportunity to access and trace the implementation rather than well-defined specified interfaces. We believe that combining these approaches—in particular, utilizing audit to capture security-critical events, but DTrace's mechanisms for data representation and capture, as well as greater access to implementation where required, will better meet the goals of the Transparent Computing.

3.2.2. A Transparent Operating-System Design

Another key hypothesis was the notion that an operating system can be (re-)designed to be a transparent operating system: i.e., to inherently integrate a number of features that make detailed security tracing more natural to its structure. We have focused in particular on making key operating-system objects reliably identifiable in traces by providing UUIDs on subjects and objects, and Message IDs on ephemeral messages, allowing tools consuming traces to identify system elements and communications between them more explicitly. For example, Message IDs will allow datagrams transmitted over the loopback network interface, or over the distributed systems, to be tagged and directly correlated, rather than having to infer them from their side effects, timestamps, and so on.

3.2.3. Distributed DTrace

We have engaged with distributed tracing in two phases: first, an investigation into a more limited form of distributed tracing across a set of virtual machines on the same physical host; and second, true distributed tracing across a disjoint set of hosts. A number of key distributed-system challenges necessarily apply:

- The system must tolerate node and edge reboot and failure, requiring careful management of persistency and, as needed, retransmission.
- The consistency of replicated data must be explicitly managed.
- Where ordering is depended on, it must be explicitly tracked.
- Where timestamps are to be compared, time must be explicitly synchronized.
- Nodes may be significantly heterogeneous: different computer architectures, operating-system versions, different applications, and eventually different operating systems.
- Latency between nodes eliminates the opportunity to use synchronous communication, instead requiring asynchronous communication, mitigation of latency through batching, and at times, distributed computation models.

As part of this work, we have also developed a formal model of the DTrace Interface Format (DIF) and its container formats, which describe executable DTrace scripts. This has not only allowed us to identify and fix several bugs in the DTrace

implementation, but also explore the semantics of distributed DTrace execution, evaluating a broad corpus of current scripts for potential bugs when executed in a distributed environment. In the future, we hope to use these formal semantics to automatically distribute work using a partial compute model, permitting more distributed script execution.

3.2.3.1. HyperTrace: Guest-from-Host Tracing

Guest-from-host, or hypervisor tracing, allows a single DTrace script installed in a host VM to be applied across a set of guest VMs. HyperTrace executes the DTrace script in the host DTrace interpreter, but accepts probes and argument data provided by the guest VM's DTrace implementation via a new DTrace hypercall. New front-end and back-end VirtIO drivers allow the host DTrace instance to control instrumentation in the guest. The host and guests can be configured to tag network packets distributed via VirtIO networking with host and message information so that events can be correlated across multiple nodes.

This approach offers significant ease of use, and takes advantage of a number of non-distributed behaviors to improve performance and semantics: there is a single time domain (no clock drift between nodes), affordable synchronous traps, high-performance VirtIO communication channels, reliable detection of guest failure, and a single executing instance of the DTrace interpreter avoids the need for work distribution and consistency management for the script itself. This environment offers the potential to explore issues such as heterogeneity and distributed semantics without taking on the full challenge of a networked distributed system. However, a number of challenges arise—not least, as the host executes the script, it must access guest memory, which requires the HyperTrace implementation to implement a manual nested page-table walk, and lacks the stronger host-kernel reliability guarantees for memory access usually present with DTrace.

We are also interested in using HyperTrace to improve TCB security for guest tracing: while a compromised guest may choose not to trace events it wishes to omit, or to provide falsified information, strong non-repudiation can be achieved by storing log data in the host rather than the guest. Deploying this model, an attacker must compromise the host operating system to make retrospective changes to captured traces.

3.2.3.2. Distributed DTrace

Full Distributed DTrace operates over a set of nodes connected only by a network, rather than being able to assume access to a hypercall. Unlike HyperTrace, due to high network latency, it cannot utilize a single executing instance of the DTrace interpreter, as kernel and userspace execution cannot be halted awaiting synchronous replies from a central instance. Distributed DTrace must also tolerate a variety of host and network failures, as well as address heterogeneity.

To this end, we have utilized the Kafka distributed message protocol and server implementation to reliably stream trace records from distributed nodes back to a central tracing system (tracing source), as shown in Figure 6. DTrace scripts execute on each individual traced node, capturing data and computing based on local state.

The command-line DDTrace client accepts a stream of DTrace trace records aggregated from multiple tracing nodes and delivered by Kafka, converting them to human- or machine-readable output as specified by the script. CADETS end hosts can be configured to carry host and message information via IP options, in a manner similar to VirtIO packet tagging in the HyperTrace DTrace, allowing scripts on different nodes to describe common events.

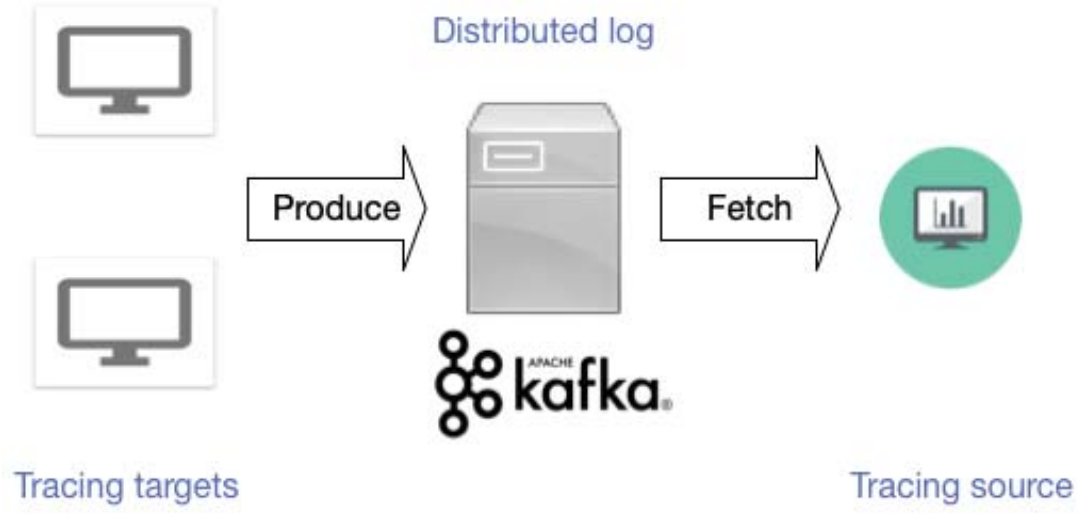

Figure 6 High-level View of DDTrace

Each traced node implements the Kafka protocol via dlog, which persists traces to disk reliably for eventual network transmission via Kafka, tolerating node reboots and network failures. A new kernel module, ddtrace, manages DTrace trace buffers to prevent record loss from full buffers. The new trace handling has required significant changes to the trace output path for DTrace: a new dlogd userspace daemon configures distributed scripts, and directs the kernel DTrace implementation to send output into dlog, a kernel subsystem that reliably writes records to on-disk logs, which can then be streamed over the network by dlogd. dlog provides an upper bound on loss in the event of a node failure. The mechanism is robust in the presence of system reboot, and dlogd implements the Kafka protocol allowing reliable persistent distribution to the DDTrace client. Figure 7 shows the new components of DDTrace (in green) and their relationships with the original DTrace components (blue).

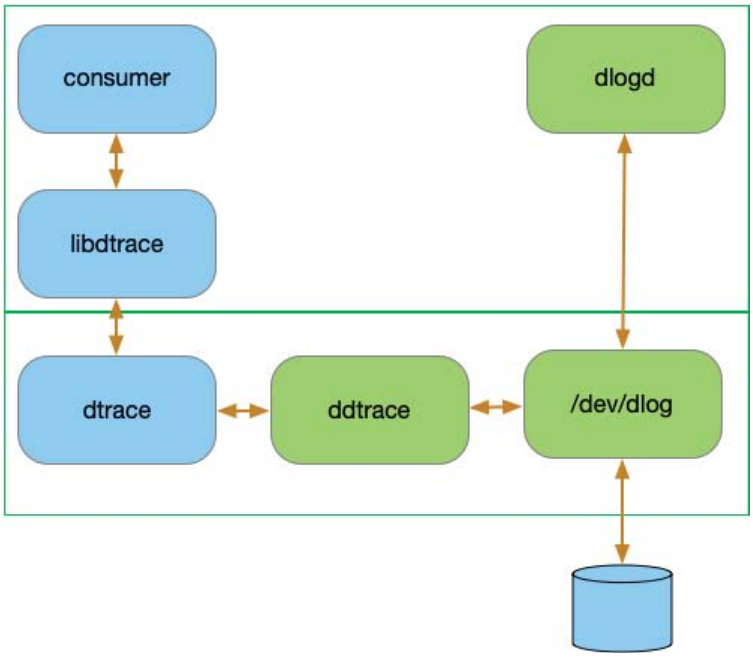

Figure 7 DDTrace Overview

Distributed-system heterogeneity is addressed at a basic level by having individual traced nodes submit type information required to interpret trace entries to Kafka as well; this allows the client to present trace records using type information from the traced node that generated it.

It remains a challenge to the DTrace script programmer, and hence future work, to address other forms of heterogeneity, and there remain unanswered questions about network event ordering and distributed compute. We anticipate future Distributed DTrace features to allow compute requested by the script to have its execution split between distributed nodes (with access to local information) or the central node (with a global view).

3.3 CDM Translator

The goal of the CDM Translator is to quickly and faithfully convert the CADETS JSON traces to the CDM used by the TC program. It can read data from a file or Kafka, and output to a file or Kafka. This flexibility makes testing and debugging simpler, as most tests don't require a full Kafka setup. For the first few engagements, CADETS traces were saved to disk, and the translator read from those files. For the last two engagements, the translator both read and wrote to Kafka. In order to support translation of MetaIO data to CDM, a new event type, ProvenanceAssertion was added (see CDM Augmentation section).

Other than producing CDM for the other teams, the priorities in implementation of the translator were that it should be able to keep up with the traces as they were generated, and that it should be able to run for weeks at a time without issue. For this reason, the translator keeps minimal state. The only state the translator keeps is

a list the UUIDs of objects that have been sent in CDM. This prevents it from doing things like repeatedly sending file objects every time a file is mentioned.

This choice of priorities had an effect on the CDM traces generated, as it prevented compiling information from multiple events to generate a more complete record when defining files or other objects.

Along with the CDM translator, there is a separate, but related Python program. This is the correlator. The correlator takes the CADETS trace and looks for events that can be correlated between CADETS machines. Currently, this is limited to identifying network connections. Information about these correlations is added to the CADETS trace for the CDM Translator to use.

3.4 LibPVM

The hypothesis behind libPVM is that TA2 algorithms should not reconstruct or infer the semantics of TA1-produced graphs by just applying learning on the data. Such a model-less approach would be prone to miss the subtle semantic guarantees that can be made about the trace data, which can differ from one TA1 to another, depending on the types of instrumentation employed, the location where it is captured, resistance to adversaries, etc.

Instead, libPVM proposes an underlying semantic model on the data, where a clear mapping exists between the data captured for every entity in the system (e.g. system call) and the changes that are made to the provenance graph as a consequence of having observed the corresponding data.

The second hypotheses of libPVM is that relying on the model, transformations can be applied to the graph that preserve the correctness of the result (even if they, for example, change the abstraction level of the representation). Such transformations should help machine learning algorithms in ignoring noise, possibly intentionally introduced in the system by adversaries in order to distract from attack identification.

As part of CADETS, libPVM was used as a data quality control tool, trying to identify in-house whether there are sufficient details in the tracing data for producing meaningful provenance graphs.

On the implementation side, libPVM makes the assumption that provenance graphs represent an index over the persistent, distributed log containing raw capture data. As a side effect of this, it allows for the exploration of analysis and pattern matching over dynamic graphs: graphs that can be re-created and expanded to contain more detail in regions where that is needed, or contracted to get a higher-level view of system activity. This means that local subgraphs may be re-created starting from the raw data, following rules that are user-driven (i.e for process with PID X, create a graph with detailed versioning on every read and write, in order to make an analysis of cross-process data visibility).

This has also informed a more nuanced position on the model generating the graph: it is acceptable for it not to make all data fully accessible for analysis in the default view, and it gives the model the freedom to make some information harder to reach

(motivated, for example, by the fact that the information cannot be always correctly collected by the OS-level tools). This translates to some graph queries that are simple and fast (common queries about processes and files), while others are allowed to be slower and multi-step (path-dependent queries, fine-grained data visibility queries).

The end result, as shown in Figure 8, is an ingestion pipeline which implements the semantic model (PVM) and pushed the resulting graph into a Neo4j database. Optimizations on this pipeline were performed in order to be able to cope with high event rates (with the goal being to keep up with system generated events).

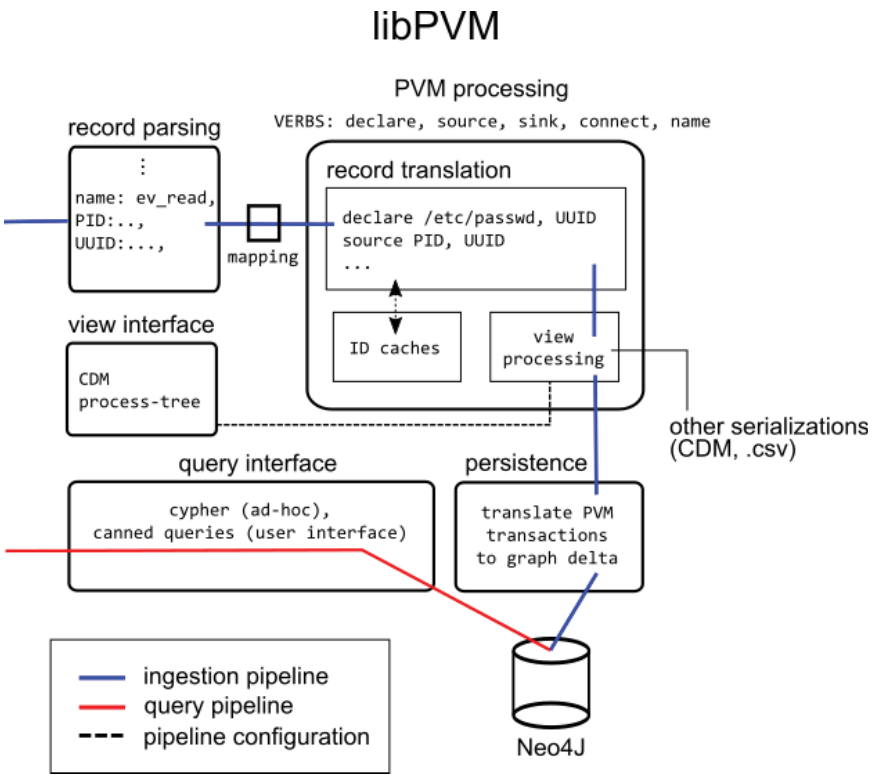

Figure 8 libPVM Overview

3.4.1. The PVM Model

The model itself makes the assumption that any system trace (irrespective of its granularity) can be de-composed into datatypes that are subtypes of four main abstract entities: Actors, Objects, Stores and ChannelEP (Channel End Points). Versioning rules are defined for those entities, in order to represent changes over time and to track their provenance. The abstract entities form a shallow hierarchy, with Stores and ChannelEP being Objects.

The model assumes that the high-level properties of the abstract entities are sufficient for declaring dependencies between concrete subtypes and maintaining consistent semantics in a provenance graph, across multiple types of data sources:

- Actors are entities that perform actions; generally they represent processing units such as UNIX processes, even though they can also represent more abstract notions of processing (central processing units, accelerators, program functions). They may act upon objects or other actors.
- Objects are data-carrying entities, used and acted-upon by actors. Currently, two abstract object subtypes exist:
    - Stores are an object type storing data internally. Under the model, such objects version on first write and on last concurrent write. The libPVM implementation uses a concrete Store subclass called EditSession to represent intervals where a Store is concurrently modified by multiple users, without asserting precise data visibility rules. If thread or process interleaving of data within an EditSession is important, one needs to explore the original events trace and its guarantees about such matters on captured data.
    - ChannelEPs are an object type through which data flows, without internal storage (or for which any ephemeral internal storage is side-effect free in terms of the provenance graph). Those objects do not version.

For each type of trace that needs to be processed, one needs to define a mapping from events in that trace to a series of verbs existent in the PVM model. As part of this mapping, developers of the tracing system would also define concrete entity subtypes, by specifying a schema, in the form of key-value pairs that should be expected as properties of each concrete subtype.

Figure 9 shows as an example, under the CADETS mapping of raw data, pseudo teletypes (PTTYs) are defined as subtypes of Conduits:

```
PPTY : Conduit
 name: “ptty”
 properties: [owner_uid, owner_gid, mode]
```

Figure 9 PTTY Mapping to PVM Example

Similarly, the exec system call is mapped to the following series of PVM verbs, as shown in Figure 10:

```
exec(process, bin_file_uuid, bin_path, cmdline){
 bin = declare(FILE, bin_file_uuid);  # declare the existence of the binary file
 name(bin, bin_path)                  # specify that it was referred to by this
                                      # name
 meta(process, “cmdline”, cmdline);   # add process metadata (command line)
 source(process, bin);                # declare that the process reads data from
                                      # its binary file
}
```

Figure 10 exec Syscall Mapping to PVM Example

Together, the verb mapping and the concrete subtypes allow implementations of the PVM model to take events in a trace and an existing output graph (possibly empty) and have a well-defined specification for what transformations should be applied to the output graph as a response. Those transformations are well-known side effects of "applying" each PVM verb to an existent graph.

The following verbs exist:

- declare(concrete_type, uuid) -> entity_handle e This declares the existence of an entity of the given concrete type, with the given unique identifier. As a side effect, this forces the creation of entity e with the given UUID, if it doesn't already exist. Either the newly created graph entity or the one existing with the same UUID is returned.
- sink(src:Actor, dst:Entity) This declares that the src actor has transferred information into the dst entity, as an atomic operation (verbs which are not atomic, for example sink_start and sink_end also exist). Depending on the destination Entity, this can trigger versioning (for example, if it is the first sink for an entity that is a subtype of Store)
- source(src:Entity, dst:Actor) The reverse of sink, this declares that the corresponding data/information was obtained by the dst actor from the src entity.
- connect(ep1:ChannelEP, ep2:ChannelEP, direction) This declares that two channel nodes have connected to each other, and data can flow between them. The direction argument indicates the direction of flow, either mono (ep1-> ep2) or bi-directional (ep1 <-> ep2).
- mention(e:Entity, n:Name) This declares that at this point, entity e has been referred to using the name n. If it doesn't already exist, this will generate a new node for the name, and then link e to it using a "NAMED" relation. In PVM, non-UUID names are considered non-reliable and temporary.
- unlink(e:Entity, n:Name) This declares that at this point, the entity e has been disassociated from the name n. A historic record about this fact is recorded in the graph, without removing previously existing links (data is just added to a provenance graph, never removed).
- property(e:Entity, key:str, value:str) Set the given key to the corresponding value for entity e. The key/value pair should have been defined as part of the type of e.

The assumption is that those limited verbs take the role of a DSL for specifying how provenance graphs can be created in response to events from system-level traces.

3.4.2. The libPVM Implementation

The implementation assumes that it is possible to create a data ingestion/ transformation pipeline that takes system trace events as they are produced and transforms them into the corresponding provenance graph, at a rate which keeps up with the frequency of input. For ingesting CADETS data, this meant significant optimizations and parallelization for JSON parsing and for the storage backend (Neo4j).

The assumption is also that libPVM can be linked with a wide variety of applications in order to allow them to either ingest provenance data, transform it (through plugin filters and views) and query it.

3.5 Neo4j Database Optimization

Motivated by the identification of Neo4j as one of the major bottlenecks in our LibPVM data ingestion pipeline, we have proposed that additional work was necessary to improve Neo4j data ingestion characteristics for write-heavy workloads.

In particular, the workloads generated by the fine-grained provenance-capturing mechanisms of TA1s are quite different from traditional workloads for which graph databases are currently optimized for. At the same time, storing the collected data and performing provenance analysis is most naturally done on backends supporting a graph data model.

We have characterized the workload generated by system level-provenance capture to graph database backends as producing a high number of concurrent writes, and triggering updates to a limited number of elements (nodes) compared to the size of the entire graph (write-heavy, contention-heavy). This is a direct consequence of new nodes and edges in the graph attaching predominantly to an existing “boundary” of nodes as determined by the set of active applications and services running on the underlying system. More concretely, when compared to other graph structures, such as the ones generated by social-network use cases, the probability of attaching new edges to existing nodes in the graph is not uniform, with most nodes in large provenance graphs being effectively “frozen”. For such nodes, our optimization target moved from improving the latency and throughput of concurrent writes to one of not penalizing reads (a balance which is difficult to achieve for arbitrary workloads).

We have tested our assumptions using the CADETS TA1 and libPVM as an entity generating graphs from distributed provenance logs, but we believe that improvements are generalizable to other TA1 performers. This means optimizations were done for the type of data produced by provenance loggers rather than being libPVM-specific.

Furthermore, our optimization work does not assume that all captured data will end up in graph databases (with some information being more naturally stored in key-value stores or relational databases), which allows for even more flexibility in terms of design choices for optimization (we focus optimization on common graph cases and access patterns, without trying to improve things across the board). However, not all our optimizations are specific to provenance data and could also improve other write-heavy workloads which share some of the same underlying characteristics.

Two primary targets of optimization were identified: the transactional engine of Neo4j (which effectively serializes updates for the type of workload considered) and the storage layer (space inefficient and tightly coupled to the transactional engine). A third major bottleneck, the Neo4j query planner, was also identified but we have

decided not to focus optimizations efforts on it as in the absence of a clear algebraic model for graph queries it is significantly less tractable in terms of possible improvements. Furthermore, we have assessed that it can be easily bypassed by writing the queries directly in a lower-level Neo4j API with more direct access to the underlying storage.

Figure 11 shows changes to Neo4j in red:

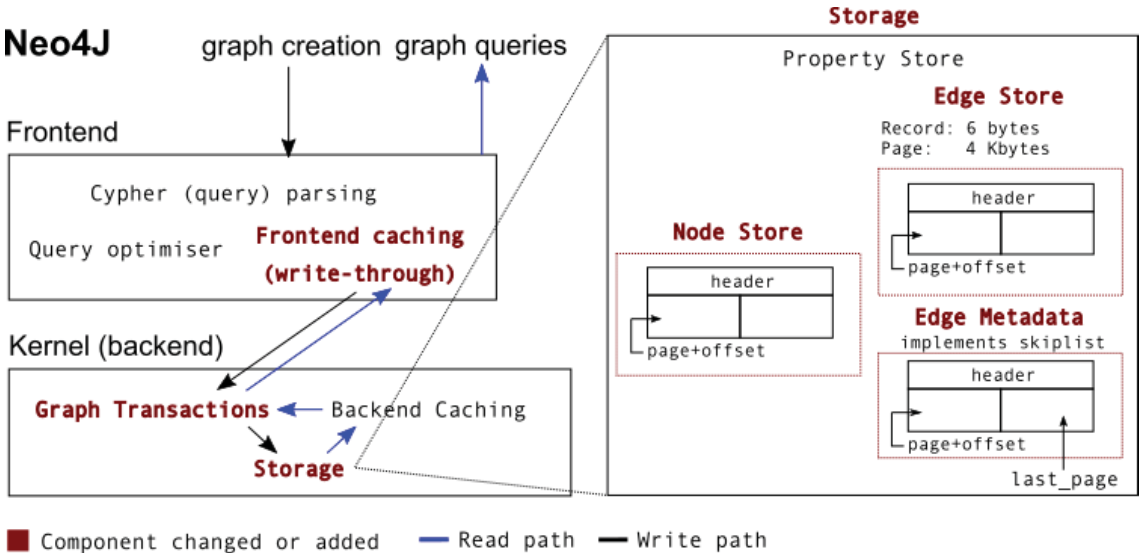

Figure 11 Neo4j Optimized Design

A new storage backend was built, offering both space-savings and allowing for faster lookups and concurrent updates without penalizing reads. It is optimized for a transaction layer implementing a form of SSI (Serializable Snapshot Isolation), a transaction isolation level that guarantees that the result of executing multiple transactions in parallel is identical to at least one serial execution of the same transactions. For optimizing information lookups during writes (a read-amplification factor exists due to the need for finding existent nodes and edges), the edge store is physically split into core edge data and metadata used for lookup (used as an efficient index for lookups and finding insert positions).

The existent transaction components have been modified in order to work with the new storage backend, allowing more concurrency during updates but maintaining Atomicity, Consistency, Isolation, Durability (ACID) properties.

A new write-through caching layer was overlaid on top of the existing transaction engine, to reduce read-write conflicts and allow higher concurrency in writes at the storage-engine level.

3.6 User Interface

The CADETS user interface was initially developed to support the demonstration we performed in July 2017. Although the CADETS project does not interface directly

with end users, we realized that an adequate demonstration of our data collection and correlation required visualization. The visualization capabilities built into Neo4j are helpful for viewing sets of nodes, but not for the iterative sense making process that a security analysis might use when developing a hypothesis about an attack using CADETS. As a result, we developed our UI to simulate activities that an analyst might need to do while investigating an attack: inspect individual nodes (files, pipes, processes, sockets, etc.) and build up a graph that represents a working hypothesis of how an attack proceeded. Figure 12 shows an example view of the UI.

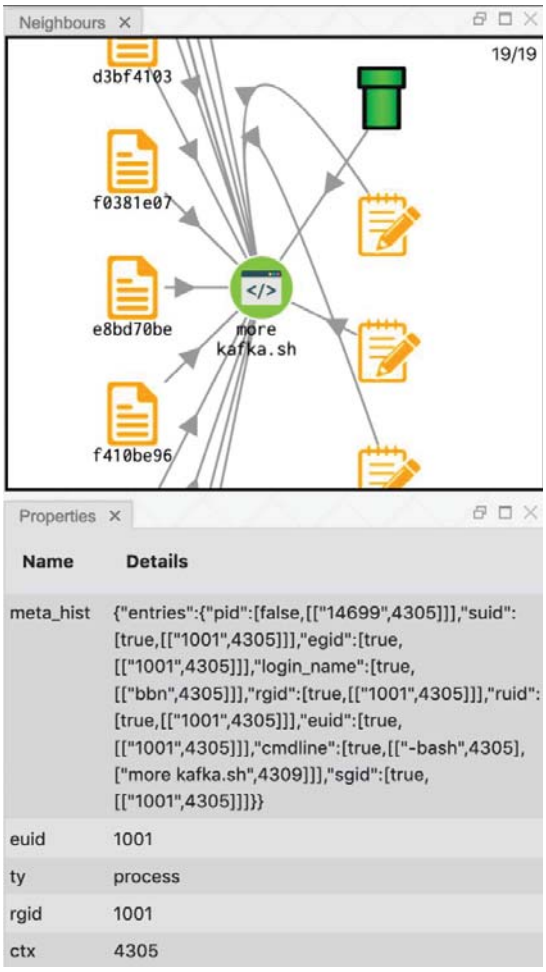

Figure 12 CADETS UI Focused on a Single Node

The initial UI was developed as a research prototype without the level of rigor and software engineering that has been applied to other aspects of the CADETS project. As our team had little experience with JavaScript-based user interfaces, we developed a tool that worked for the demonstration but was difficult to maintain over the longer term. After the initial demonstration, we brought in undergraduate research assistants to help restructure code and package dependencies to increase reproducibility and maintainability. We then switched much of the core UI infrastructure away from *ad hoc* JavaScript and towards mature.

Throughout the design and implementation process for the user interface, we used our understanding of analysts' needs and workflow — based on limited engagement with a Global Security Operations Center's (GSOC) employees — to drive design decisions. We also employed the UI to a limited extent internally, to view data from engagements.

4 RESULTS AND DISCUSSION

4.1 Engagement 1 (E1)

4.1.1. New Features

The first engagement took place September 6-9, 2016. We started with DTrace and FreeBSD 11.0-RC2 and added some features. FreeBSD comes with support for security event auditing. We modified this system, which provides detailed information about syscalls, to make the information accessible from DTrace. A feature available on Solaris, but not FreeBSD was the ability to print some file path information when tracing read and write events. We ported this feature to FreeBSD. While file paths are very human friendly to look at, they are not actually reliable identifiers for files. To that end, we modified FreeBSD to have UUIDs associated with many kernel objects. Files, sockets, and even processes have intrinsic UUIDs. These are meant to be more reliable than the identifiers typically used on a system. While other identifiers, such as file paths, inodes, or pids, may be reused over time on a running system, the UUIDs will not. For E1, we produced CDM13 for the other teams to consume.

4.1.2. Results

CADETS had no stability issues during E1. CADETS generated approximately 25-30 CADETS records per second during E1. For the Bovia and Pandex attack scenarios combined, CADETS generated approximately 8 million events. In CDM format, this was represented by about 10 million nodes and 15 million edges.

Our data was used by all TA2s. Feedback indicated that all TA2s were able to make good use of the data and had good accuracy finding attacks. The tracing also did not interfere with system performance.

4.2 Engagement 2 (E2)

4.2.1. New Features

The second engagement took place May 5-23, 2017. Along with bug fixes, we added information about memory map events allowing slightly more information to be provided for in-memory attacks. MetaIO provenance information was added, but was not seen in the engagement. CADETS produced CDMv14 for the other teams to consume.

4.2.2. Results

CADETS had no stability issues during E2. As different attacks were used on different TA1 systems, it is hard to directly compare performance between TA1 systems. Nevertheless, CADETS did well, and the majority of attacks were identified. Attacks using the network or file system were generally identified, while attacks fully in memory or making use of custom syscalls installed via a new kernel module left fewer tracks and were mostly missed.

4.3 Engagement 3 (E3)

4.3.1. New Features

The third engagement took place April 6-13, 2018. CADETS was run for the length of the engagement. For E3, we began adding features that would be more important once Distributed DTrace was implemented. The traces included more details about the machine being traced, and additional networking information was added to help reconcile network events across hosts. Lastly, MetaIO integration was completed, and we generated provenance information for some executables.

4.3.2. Results

The CADETS system had minor stability issues during E3 (see gaps in E3 data streams, as shown in Figure 13), but not caused by instability in CADETS itself. Any APTs interacting with the kernel risk causing instability in the underlying system. There were numerous attacks against CADETS. Overall, the accuracy for the three TA2s was 8%, 46%, and 75%. Overall, the traces had the same weaknesses as during E2, and while MetaIO events were found in the trace, the events were not key to identifying attacks. Despite these observations, CADETS was the only TA1 system named as one of the best TA1/TA2 combinations for each of the TA2 performers.

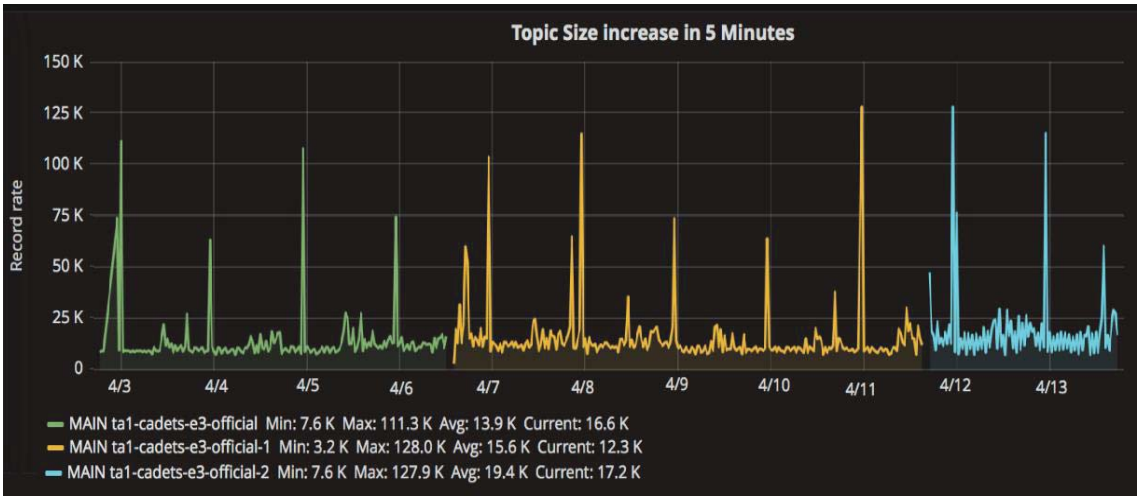

Figure 13 Activity on E3 Kafka Topics

4.4 Engagement 4 (E4)

4.4.1. New Features

The fourth engagement took place November 8-21, 2018. CADETS was run on November 14th and 21st, for a total of about 12 hours.

For E4, we moved from a basis of FreeBSD 11 to FreeBSD 12. We also moved from using the slightly modified version of DTrace that we had used in previous engagements to Distributed DTrace. Distributed DTrace still makes use of much of the underlying DTrace code, but is run in multiple parts, on multiple machines. This allowed us to move some trace processing off of the machine being traced.

One of the advantages of Distributed DTrace is that we were able to expand our tracing to include part of the tracing system itself. While this added a large number of events that were not directly useful for TA2s in this engagement, these events could provide additional events if an attacker was trying to interfere with tracing. Also, due to the fact that intermediate traces are stored in Kafka rather than on the local file system, the attacker would need to compromise multiple systems to remove their actions from the log.

4.4.2. Results

CADETS had no stability issues during the run of E4, but did have issues keeping up with real-time. Figure 14 indicates that overall, we generated approximately 1700 records per second over two machines. For E4, there were 4 attacks against CADETS. Due to performance issues, the 4th attack of the day did not make it into the traces available to the TA2s. Given the 3 remaining attacks, the accuracy for the three TA2s was 9%, 25%, and 65%, with 2 attacks almost entirely identified by one TA2.

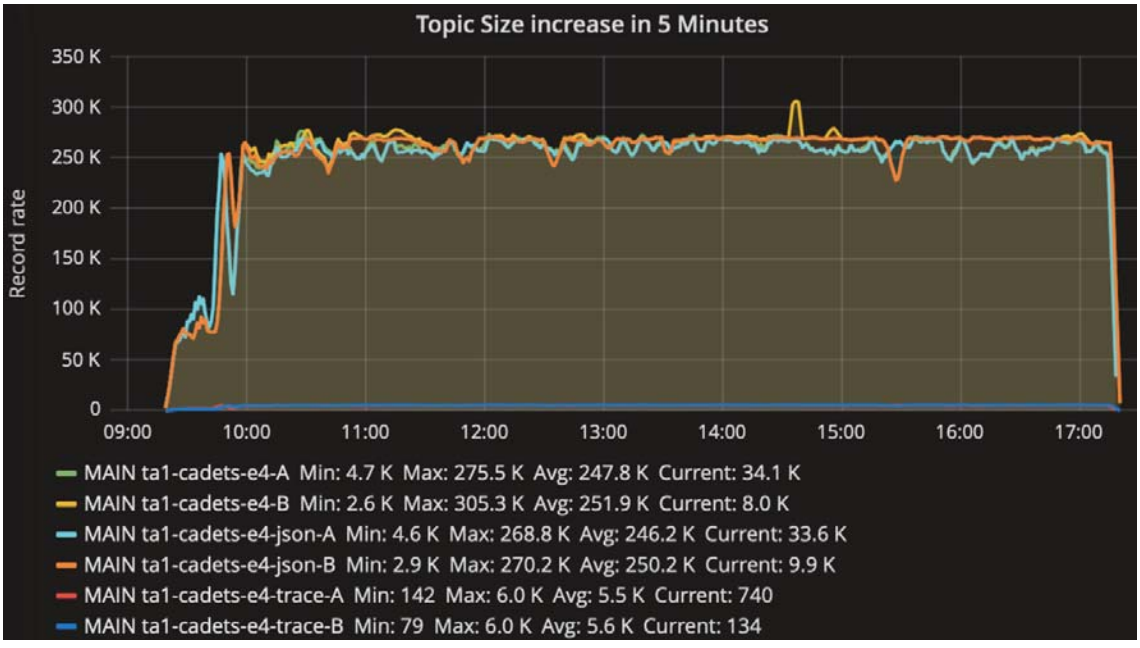

Figure 14 Activity on E4 Kafka Topics

4.5 Engagement 5 (E5)

4.5.1.1. New Features

The fifth engagement took place May 6-17, 2019. Like other TA1s, CADETS was run from the 7th through the 17th. For E5, our code was based off of FreeBSD 13.0. We removed the file path information from some events in the traces. Events with more reliable file information continue to report it, they were excluded from events where the file paths may be relative or incomplete was it removed. More information (e.g. initial sequence numbers) was added to the network events to improve tracing cross-host interactions.

4.5.1.2. Results

There were two hurdles to CADETS stability in E5. While unfortunate, the bugs were located such that there was minimal, if any, data lost. One of our components, dlogd, had a memory leak. As it took time to start back up, this was handled by restarting it intentionally outside of TA5.1's attack hours to reduce impact. Later in the engagement, a fix was pushed to speed up restart time. The amount of data being generated during this engagement revealed a bug in the CDM Translator where the CDM Translator enqueued data to send to Kafka faster than it could be sent. This caused the program to crash. Once this bug was identified, a fix was implemented and provided to deploy in case of another crash.

The other components, the ddtrace_producer and ddtrace_consumer, remained stable throughout the engagement.

After approximately 11 days running, CADETS trace generation was still keeping up with real-time. While there were lags when a component needed to be restarted, the system caught up each time.

As in E4, CADETS generated much more data than it did in the first few engagements. While the data rate did not change significantly for E5, due to the fact that E5 was set up to run data generation 24 hours a day for the entire engagement and on an increased number of hosts, a much higher quantity of data was generated. Overall, we generated approximately 1250 records per second over three machines, as shown in Figure 15. At the time of this writing, attack detection results have not yet been evaluated.

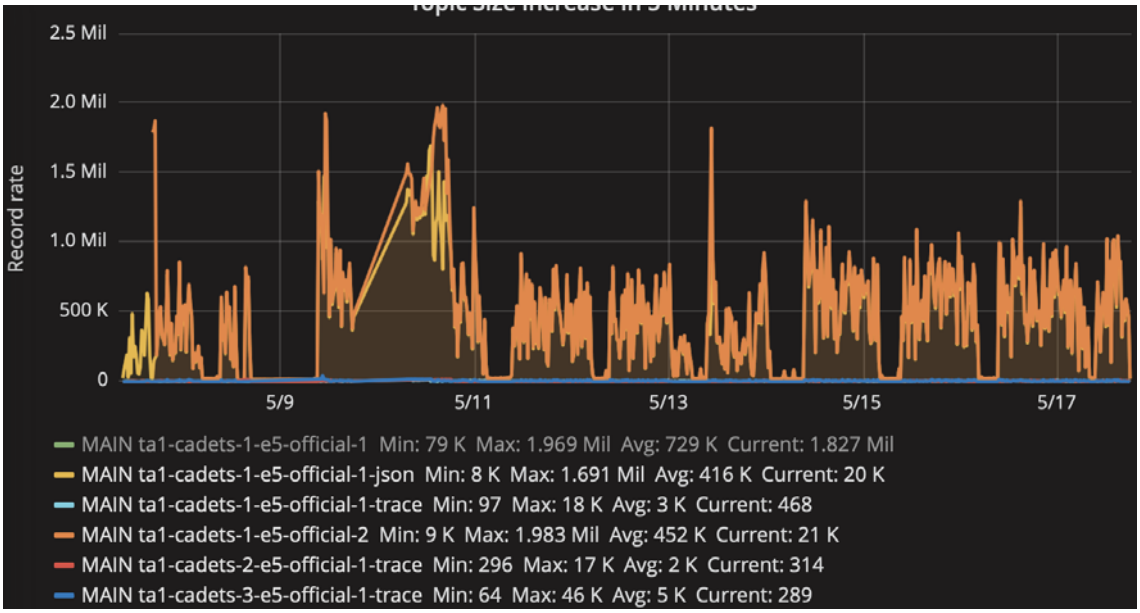

Figure 15: Activity on E5 Kafka Topics

4.6 Userspace Tracing

4.6.1. Loom

Loom was initially envisioned to help provide additional trace data and corresponding semantics to user space applications on FreeBSD. Although DTrace could instrument applications, its components to do so were coarse and did not provide much context to the data. Loom was implemented as a framework to provide additional instrumentation in user space application and libraries to better understand their data flows.

The initial functionality provided by Loom allowed for instrumentation of function calls and data structure accesses and modifications. This was enough functionality to support the development of LLVM-Prov, which used to provide data provenance in FreeBSD applications. The result for LLVM-Prov within the project are discussion in section LLVM-Prov.

Loom development continued independently of LLVM-Prov to provide a general purpose instrumentation tool. Loom functionality was expanded to allow it to provide instrumentation of both FreeBSD applications and libraries. Modifications were also made to the FreeBSD build system to allow for instrumentation during the build process for the operating system.

Although the final Loom improvements did not align with the E5 schedule (fixed after the feature freeze), internal testing showed promising results. With a small of amount of instrumentation added to the Pluggable Authentication Module (PAM) user space library, we were able to capture authentication events and associated data across all applications using PAM.

To get to the point where Loom was able to instrument both programs and libraries many avenues were explored, creating enhancement ideas for the framework. Future plans for expansion of the framework includes the addition of a runtime component, allowing for additional logic to be applied to instrumentation data before incurring the expensive cost of submitting it to kernel space frameworks. Additionally, there is ongoing research to expand the language used to define instrumentation points, allowing for more complex logic (include temporal logic) to be applied to the decision of what to trace.

4.6.2. LLVM-Prov

LLVM-prov was applied to 368 binaries that are a part of the FreeBSD base system. By the final engagement, LLVM-Prov was able to add MetaIO instrumentation to 93 binaries in the CADETS FreeBSD base system. MetaIO would not be applicable to a further 95 binaries, as they act as either sources or sinks from an information-flow perspective but not both. A further 180 binaries could not be automatically instrumented for MetaIO. Further development or integration of interprocedural information-flow analysis would allow additional binaries to be automatically instrumented.

4.6.2.1. Fully-instrumented Binaries

Table 1 lists binaries that were fully instrumented for MetaIO, i.e., all instances of source and sink system calls were converted into their MetaIO equivalents.

Table 1 List of Fully-instrumented Binaries

Binaries	
/bin/cp	/bin/mv
/sbin/fsirand	/sbin/ggated
/sbin/ggatel	/sbin/newfs_msdos
/sbin/nos-tun	/sbin/recoverdisk-
/usr/bin/brandelf	/usr/bin/bspatch
/usr/bin/ctfdump	/usr/bin/dpv
/usr/bin/elf2aout	/usr/bin/indent
/usr/bin/reset	/usr/bin/tee
/usr/bin/tset	/usr/sbin/dtrace
/usr/sbin/nfsd	/usr/sbin/tzsetup
/usr/sbin/watch	

4.6.2.2. Partially-instrumented Binaries

Table 2 lists binaries that were partially instrumented for MetaIO, i.e., LLVM-Prov was able to find some information flows and instrument them but other flows were opaque to LLVM-Prov.

Table 2 List of Partially-instrumented Binaries

Binaries	
/bin/dd	/bin/pax
/sbin/decryptcore	/sbin/fsck_msdosfs
/sbin/gbde	/sbin/gcache
/sbin/gconcat	/sbin/geli
/sbin/geom	/sbin/gjournal
/sbin/glabel	/sbin/gmirror
/sbin/gmountver	/sbin/gmultipath
/sbin/gnop	/sbin/gpart
/sbin/graid3	/sbin/graid
/sbin/gsched	/sbin/gshsec
/sbin/gstripe	/sbin/gvirstor
/sbin/hastd	/sbin/natd
/sbin/ping6	/sbin/ping
/sbin/restore	/sbin/rrestore
/sbin/savecore	/usr/bin/calendar
/usr/bin/cu	/usr/bin/diff
/usr/bin/gunzip	/usr/bin/gzcat
/usr/bin/gzip	/usr/bin/ibv_rc_pingpong
/usr/bin/ibv_uc_pingpong	/usr/bin/ibv_ud_pingpong
/usr/bin/install	/usr/bin/lpr
/usr/bin/patch	/usr/bin/script
/usr/bin/sdiff	/usr/bin/split
/usr/bin/tcopy	/usr/bin/tip
/usr/bin/zcat	/usr/sbin/acpidb
/usr/sbin/bcmfw	/usr/sbin/bhyve
/usr/sbin/ctladm	/usr/sbin/fwcontrol
/usr/sbin/hostapd	/usr/sbin/hv_kvp_daemon
/usr/sbin/inetd	/usr/sbin/ipfwpcap
/usr/sbin/lpd	/usr/sbin/nghook
/usr/sbin/ntpd	/usr/sbin/ppp
/usr/sbin/pppctl	/usr/sbin/pw
/usr/sbin/rarpd	/usr/sbin/rmt
/usr/sbin/rpc.statd	/usr/sbin/rwhod
/usr/sbin/sshd	/usr/sbin/timed
/usr/sbin/timedc	/usr/sbin/uathload
/usr/sbin/wpa_cli	/usr/sbin/wpa_supplicant

4.7 Kernel Tracing

4.7.1. FreeBSD Audit and the DTrace Audit Provider

The key hypothesis of this work was that conventional security event auditing and debugging mechanisms such as DTrace could be converged to offer the benefits of both in the Transparent Computing environment. An additional hypothesis was that modest kernel extensions to introduce unique identifiers for key kernel data structures would better support the Transparent Computing use case. We explored this hypothesis through substantive engineering and experimental activities, including:

- Significant extensions to the baseline FreeBSD audit implementation to capture non-CAPP events and additional context relevant to Transparent Computing.
- Introducing new UUIDs and Message ID to allow explicit event linkage rather than relying on statistical correlation techniques.
- Integrated audit and DTrace via the implementation of an Audit DTrace Provider that allows DTrace scripts to control audit event capture and processing.

These changes were present across multiple adversarial engagements, although additional refinements were made as the program progressed (e.g., to add additional events and context, as well as to fix bugs). UUIDs were used by TA2 teams to reliably identify a variety of subject and object types in the system, including processes, IPC sockets, and files. While we provided unique Message IDs for datagram sockets (e.g., loopback UDP, UNIX domain sockets), we are not aware that TA2s made use of this data. We did not provide access to Message IDs on stream sockets as part of the adversarial engagement, and this would be a natural next direction. We found that the DTrace Audit provider provided substantially higher-quality trace data and more maintainable tracing scripts than using DTrace's existing providers.

We have upstreamed support for audit improvements, excluding UUIDs and Message IDs, as well as the DTrace Audit Provider, to FreeBSD, accomplishing successful open-source transition. To further transition UUID and Message ID support, additional performance characterization and use-case exploration would be required—e.g., by seeing similar successful transition and adoption of TA2 technologies able to consume this additional OS metadata.

4.7.2. Distributed DTrace

The key hypotheses in this work were that inherent distributed-system problems relating to security, reliability, performance, timing, event correlation, and heterogeneity could be overcome in adapting DTrace for both multi-VM and wider distributed-system environments. We explored these hypotheses through substantive engineering and experimental activities, including:

- Introduced DTrace support for reliably tracing early boot and through shutdown.
- Introduced the dlog reliable persistent logging facility to improve DTrace reliability, and to distribute trace data across multiple hosts, even in the event of network or server outages. dlog is encrypted using TLS on the wire.
- Introduced dlog compression support and other efficiency improvements to allow DTrace to be used efficiently in a persistent network environment.
- Adapted DTrace to ship event and type metadata, not just trace data, over dlog, to account for host heterogeneity.

- Introduced IP-option-based packet tagging to carry message and host IDs between supporting IPv4 nodes, allowing distributed events to be correlated.

Distributed DTrace, as with all distributed systems, is a complex engineering artifact. We stepped towards full DDTrace support over several engagements, initially using off-the-shelf Kafka submission components to pipe local DTrace tracing on each individual nodes into Kafka. In this approach, conversion to the final CDM format occurs on individual tracing nodes, with DTrace controlling only local processing.

In the fourth adversarial engagement, we introduced dlog support, allowing separation of DTrace data capture, on tracing nodes, and presentation and conversion, on a central node, with the Kafka broker in between. dlog provides much stronger local and end-to-end reliability guarantees, including placing a strict bound on potential record loss when a tracing node crashes. dlog is integrated into the kernel, avoiding unbounded userspace buffering (as is present if the DTrace command-line tool is used), and ensures that records are synchronized to the filesystem in a controlled manner. The dlogd daemon is then responsible for reliably shipping data over the network, protected by TLS, recovering from a variety of failure modes including its own node crash, Kafka server unreachability, and Kafka server crash.

Deploying in the fourth engagement was bumpy: with network and workload conditions substantially different from our testing environment, and a late-binding request to add TLS support despite the risks involved, we had to debug a number of in-field issues relating to performance and connection reliability. In addition, while dlogd performed well respect to its specified and documented configuration, a number of in-field configuration changes were made during the engagement that reduced is reliability by taking it outside of its specified use—for example, by making configuration changes without fully restarting the system during deployment. Our experience gained was primarily operational: improved debug logging and tools, as well as configuration simplification, where necessary. We also discovered that DTrace made extremely inefficient use of storage with our audit.d script, causing performance issues in the field. After pre-engagement debugging and working through configuration issues, the system was stable, albeit slow during E4.

In the fifth engagement, we deployed a substantially improved prototype, which included improved monitoring tools, better crash recovery and misconfiguration detection, and compression support to reduce log size.

At the end of the program, our Distributed DTrace prototype is able to reliably instrument multiple hosts, dealing with moderate host heterogeneity (e.g., differing OS revisions), and addressing a variety of host and network failure modes. The prototype is not yet in a production-ready state for transition, requiring substantial further research and engineering. In particular, we have identified that modest DTrace language extensions would substantially improve usability, as well as better integrating support for event correlation. We hope also to pursue a more mature partial-compute model, in which portions of scripts run on tracing nodes (as is the case today), while other portions run centrally offering post-reconciliation processing of data originating from multiple nodes.

While Distributed DTrace has served the Transparent Computing use case well, it is easy to see it also addressing widespread problems with distributed performance analysis—e.g., with applications running on multiple nodes using distributed file systems. We have begun to engage with a number of vendors of such systems, including NetApp, Apple, and others, to understand their related use cases and potential transition challenges better.

4.7.2.1. HyperTrace

To implement HyperTrace, we:

- Added support for guest-from-host tracing via new front-end and back-end VirtIO drivers allowing the host to configure guest DTrace, and for the guest to issue hypercalls to the host DTrace engine to process firing probes.
- Add VirtIO message tagging to allow messages passing between guest VMs and the host, and between multiple guest VMs, can be linked.

We were not able to deploy HyperTrace in the final engagement, despite a strong desire to do so.

4.7.2.2. Distributed DTrace

On the whole, our implementation has validated our hypotheses, providing increasing levels of event tracking, reliability, and distributed correlation with escalating adversarial engagements. Further work will be required to address known limitations of the prototype:

- Introduce automatic distribution of Distributed DTrace configuration, including scripts.
- Support the dynamic updating of in-execution scripts, allowing them to be adapted to system-wide changes, or to increase or decrease data gathering.
- Explicitly address time synchronization requirements of scripts utilizing time—e.g., to measure latency.
- Introduce better language-level support for managing and analyzing distributed events.

4.8 Performance Overhead

An important concern of the TC program is minimizing performance overhead. Instrumentation overhead is dependent on the application that is being instrumented and the type of information that is being traced (i.e. system calls, function entries/exits) as well as its granularity. While CPU, network, disk IO performance are reasonable metrics, a baseline needed to be established.Due to the unavailability of an engagement baseline, we have provided time overheads on limited scenarios.

Post E4, we performed benchmarking of time overhead. Our internal testing showed a 17.7% time overhead when tracing is turned on using the same tracing *script* (

audit.d) that was used in E4 (although without DDTrace) when *building* the FreeBSD kernel. This was on a bhyve VM with 4 CPUs and 16G RAM.

We also measured the time to compile the FreeBSD kernel under 3 conditions:

- no tracing turned on
- tracing with HyperTrace enabled, and
- tracing from the guest only

Figure 16 below provides examples of different levels of granularity possible when tracing. The overhead of compiling FreeBSD on a guest VM with these respective levels of tracing is shown in Figure 17. The results in Figure 17 show an average time overhead of:

- 28% when *compiling* the FreeBSD kernel on a guest and tracing a wide variety of info with different granularities. This is the average additional overhead when “Traced inside the guest” compared to the “No tracing”.
- 110% when *compiling* the FreeBSD kernel on a guest and tracing, **from the host**, a wide variety of info with different granularities. This is the average additional overhead when “Traced using DTrace-virt” compared to the “No tracing”. Please note that DTrace-virt was the former name of HyperTrace.

Provider	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5	Trace 6	Trace 7	Trace 8	Trace 9
fbt	UFS	UFS (open, close, create)	UFS (open, close, create)	UFS	UFS (open, close, create)	UFS (open, close, create)	UFS, a*	UFS, a*, b*, v*	none
syscall	all	all (entry)	all	all	all (entry)	all	all	all	none
vfs	all	write, read, open, close	write, read, open, close	all	write, read, open, close	write, read, open, close	none	none	all
sched	none	none	none	all	all	all	none	none	none

Figure 16 Varying Levels of Granularity for Different Traces

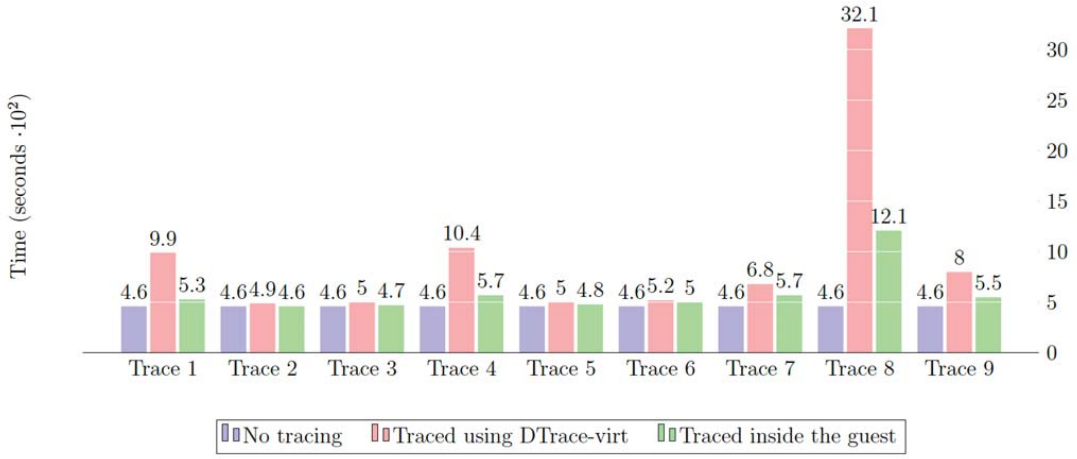

Figure 17 Comparison of Time Required to Compile FreeBSD While Tracing

4.9 LibPVM

LibPVM and its previous prototypes have been used on data resulting from engagements since E3 in order to:

- tune the ingestion pipeline
- characterize system provenance workloads
- debug missing data issues, test new types of provided data

Since E4, libPVM has also been capable of outputting a CDM serialization of graphs that have PVM semantics. libPVM has run on-line during E5, on a VM not part of the engagement.

Below we show that the resulting pipeline is capable of easily sustaining data production rates such as the ones produced during engagements; This also suggests that raw trace data coming from multiple hosts (10-100) could be processed by a single PVM instance, depending on the per-host data rates. In production scenarios beyond 100 hosts or that have high levels of per-host activity, libPVM's current design would also support working in a fully distributed environment with multiple libPVM producers sending data to a distributed store. However, truly distributed graph databases are yet in their infancy, with few products mature enough and able to sustain the type of write throughputs required.

In terms of scaling performance, libPVM shows almost perfect linear characteristics for the range of trace sizes analysed, as evidenced in Figure 18. The data points are taken by ingesting events produced during E3 and E4 (covering the Bovia and Pandex attack scenarios) as quickly as possible.

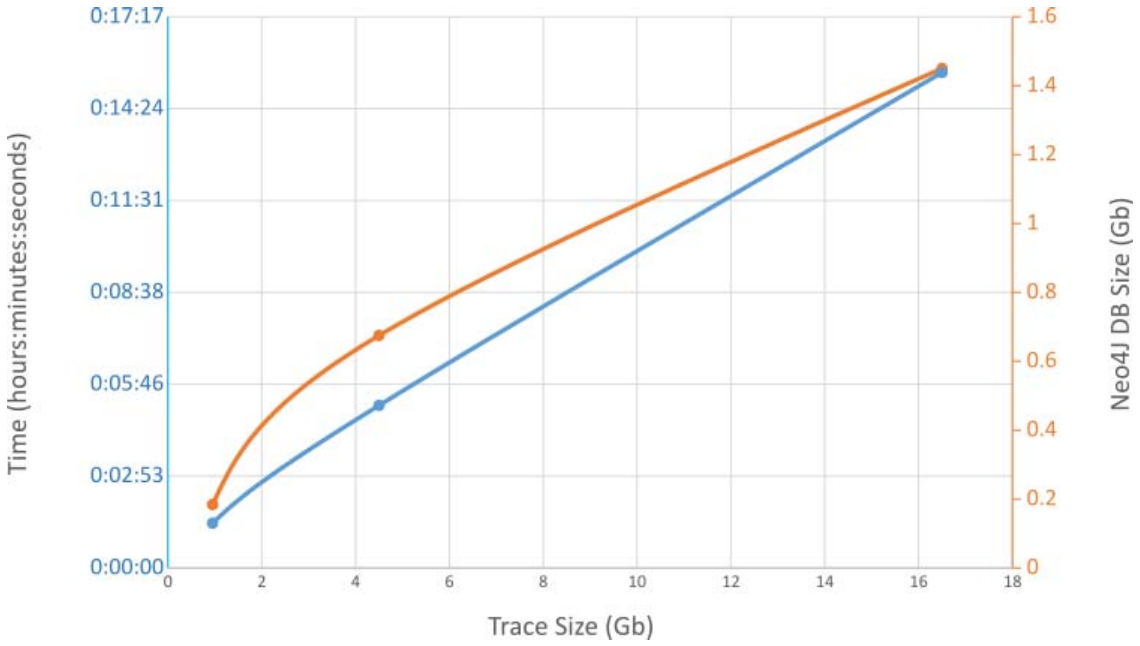

Figure 18 libPVM performance by trace size

4.10 Neo4J Database Optimization

Early on in the work, when profiling existing Neo4J bottlenecks, we have determined that for typical libPVM ingestion workloads, 80% of the time was spent in the transaction manager, with little parallelism being used because of coarse-grained locking. In particular, adding multiple edges to a single “source” node was serialized despite no real underlying data conflicts between transactions. We have determined that in such cases, a node and its corresponding edge-list can be treated as separate entities, with the possibility of implementing the edge list as a structure allowing high-concurrency on additions (for example, using lock-free data structures and optimizing for particular hardware architectures such as x86).

A second element of concern beyond the granularity of the locking was the way the locks themselves were managed, with all active locks stored in a HashMap with poor scalability (look-ups in this HashMap represented 14% of the total time).

This was despite significant improvements to the write throughput made in upstream Neo4j since 2.2 (by replacing two-phase commit with a unified transaction log). The identified cause was the fact that the provenance ingestion workload represents a pathological case for the existent Neo4J transaction/storage architecture.

In our aim to increase parallelism while maintaining transactional ACID properties, we have proposed to improve the on-disk format of the data. In particular, we have set to create an on-disk format that allows us to experiment with different transaction engines, and in particular ones implementing SSI, In terms of the consistency model, serializability means that the effect of executing multiple transactions concurrently is equivalent to the effect of executing the transactions in *some* serial order. This choice was made based on the assessment that such engines would behave significantly better for the given TC workloads.

Indeed, when creating and obtaining measurements from a non-Neo4j based prototype, we have observed the on-disk layout as able to scale better than even the batch-importing ingestion of Neo4j (which was previously the fastest available method for data ingestion, albeit not transactional or appropriate for on-line data ingestion). This is shown in the Figure 19 below, where our non-Neo4j proof-of-concept prototype (graph-ssi) is compared against standard Neo4j 3.2.2, as well as its batch-ingestion mode (graph-batch). Llama, a different, in-memory graph engine that is non-transactional is also shown for comparison.

This validates our assumptions about the suitability of our graph on-disk format and SSI transaction engine for TC workloads. Following from this, the prototype was integrated in the Neo4j ingestion pipeline.

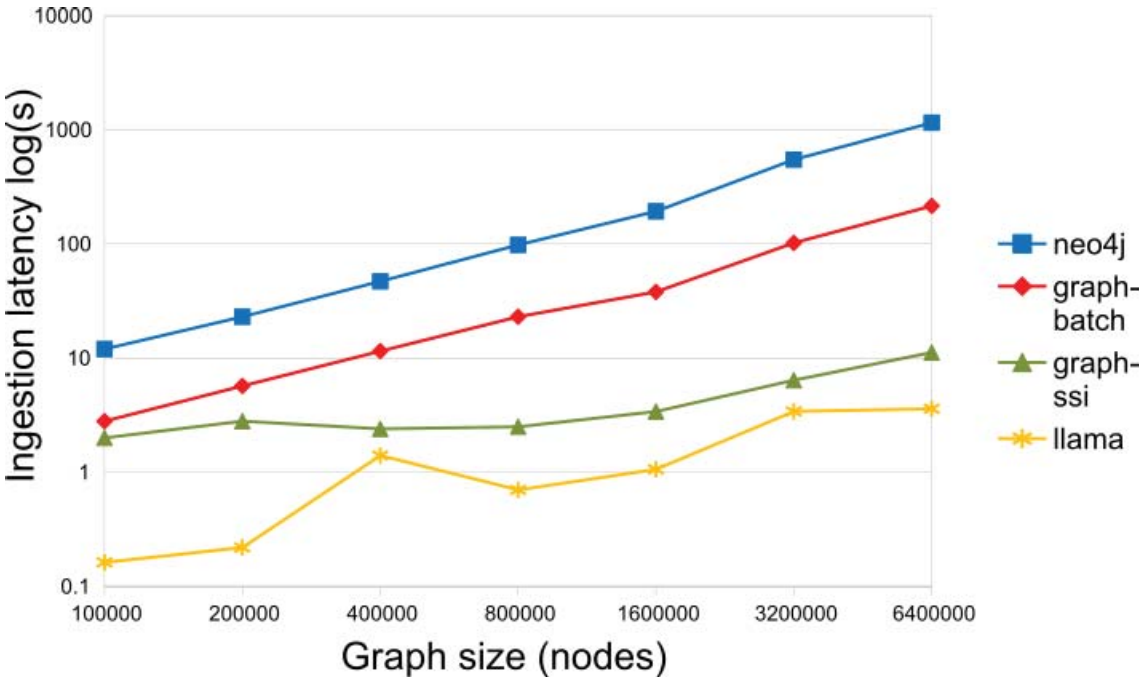

Figure 19 Performance of Different Transaction Engines for Neo4j

Notably, the performance of the integrated prototype considering all improvements (Neo4j-Modified-Cached) is worse than the batch insert from a CSV file (Neo4j-csv), as shown in Figure 20. This is because of all the overheads imposed by layers above the transaction layer (Frontend protocols, Query processing and optimization). Those were not present in the non-integrated prototype and represent separate optimization targets. However, it is worth noting that Neo4j-CSV is a non-transactional, batch-import mode not suitable for real-time ingestion. This shows that the bottleneck has shifted from the Neo4j backend to the frontend. Even so, we do observe an average improvement of 10% when compared to stock Neo4j.

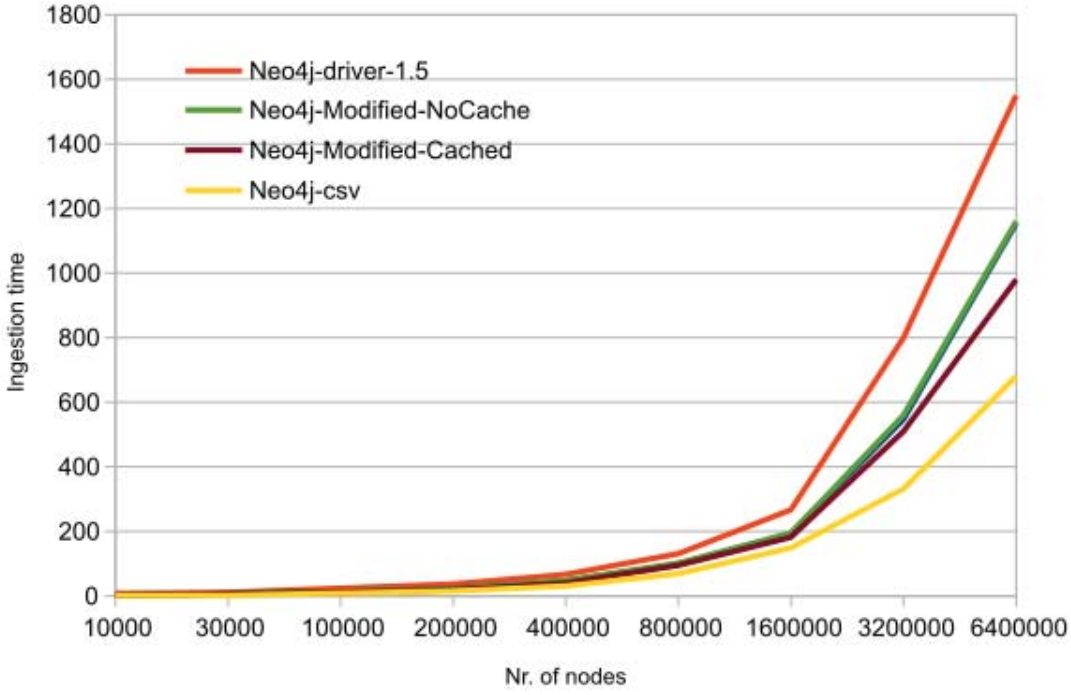

Figure 20 Comparison of Ingestion Speed

In terms of the new storage engine, Figure 21 shows how we have also managed to reduce edge storage requirements, with reductions between 25% (worst case) and 40% (best). Despite fixed records for edges, the information stored for a given edge can vary depending on the amount of information stored in indexes for finding that edge.

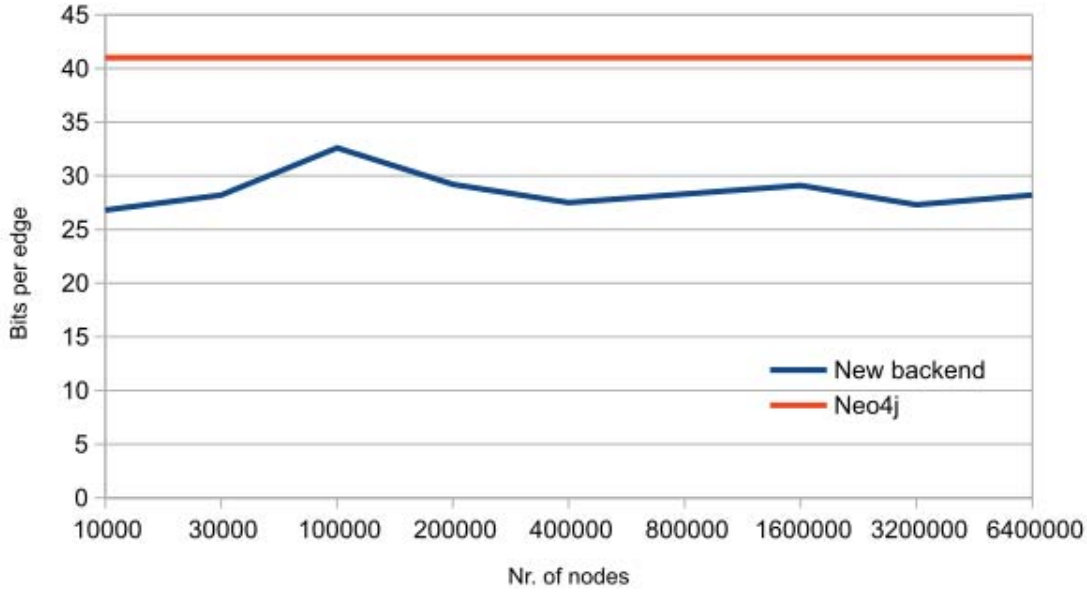

Figure 21 Storage Needed Per Edge in Graph

4.11 UI

We accomplished the primary objective for the CADETS user interface: supporting the demonstration at the July 2017 PI meeting. We also used it internally to view PVM data that flowed out of engagements, supporting our own team in a limited way.

5 CONCLUSIONS

CADETS unifies kernel and userspace tracing to provide provenance information that interconnects events across multiple nodes in a distributed system. Our system combines and improves on the security-event auditing provided by FreeBSD Audit and the dynamic and configurable debugging capability provided by DTrace tracing. The FreeBSD Audit system has been expanded to assign UUIDs to subjects and objects and Message IDs to ephemeral messages, to allow for reliable tracking of their flow. The types of events as well as the details of those events have been augmented to enhance contextual awareness. A new DTrace Audit Provider allows the triggering of such capture in a way that avoids potential race conditions and simplifies configuration. This integration of FreeBSD Audit and DTrace brings together two historically distinct subsystems to provide reliable information flow data.

To advance our instrumentation to a distributed environment, CADETS has extended the original DTrace design with Distributed DTrace (DDTrace) to support distributed tracing across multiple hosts. DDTrace offers fully distributed tracing with built-in functionality that takes advantage of the Kafka message broker to address complex issues in consistency, persistence, time synchronicity, and latency. In addition, HyperTrace is a separate and “simplified” variation of distributed tracing, whereby instrumentation is coordinated among a single host VM and multiple guest VMs. DDTrace offers potential for an enterprise solution while HyperTrace offers an approach that is suitable for a more scoped exploration and perhaps a more secured model to perform the exploration under.

While DTrace can be used for userspace tracing, it requires modification to the source code, a requirement that does not scale well. We have implemented Loom technology which leverages the LLVM compiler for software instrumentation that is free from this intrusive caveat. Loom’s instrumentation is specified by either a policy file or via use of its API, allowing for flexibility in determining what is instrumented and how to output the results. Once the Loom framework was built, information flow was further enhanced and integrated with kernel tracing with the development of LLVM-prov. LLVM-prov builds on Loom’s transformation to map normal syscalls to MetaIO calls that facilitate the passing of provenance information. Various static analysis tools and techniques have been explored to improve LLVM-prov’s precision, allowing it to address the *n x m problem* of a combinatorial explosion in provenance propagation.

Our instrumentation technology have been deployed to the TC engagement environments with the exception of HyperTrace, which is currently still undergoing testing. Despite the complexity in the deployment of DDTrace, we were able to achieve stability during the live portion of its first engagement (E4). By the last engagement (E5) of the program, our system was proven capable of handling generation of 1250 records per second with data incoming from 3 hosts. Although we were not able to maintain continuous uptime for the full 11 days of data generation, the engagement helped define the improvements that are needed to further operational use. Overall, CADETS data was able to be used to identify the majority of attacks during the engagements.

Downstream, other components contribute to the practical use of CADETS' instrumentation output. The CDM Translator converts CADETS trace data to TC CDM semantics for TA2 consumption. The libPVM library provides a model that avoids misinterpretation of the CADETS data. It enforces a clear semantic model while also offering layers of data abstraction that strives to improve on the signal-to-noise ratio as well as flexible context viewing. Our Neo4j optimization work considers the write-heavy characteristics of TC workloads and addresses the shortcomings. It introduces a new storage engine optimized for serializable snapshot isolation support and the addition of a caching layer to the transactional engine to realize efficiency. Finally, a UI provides end users with an interactive tool to investigate and build provenance graphs of system activities using CADETS data.

CADETS enables whole system inspection and reliable provenance tracking across networked hosts. While the advancement of userspace and kernel space instrumentation were the cornerstone of our research, we've also built out our system to include tools that transform trace data to a pragmatic model, optimize data storage, and enhance user experience.

6 REFERENCES

[1] Stolfa, Domagoj, “Tracing Virtual Machines in Real-Time”, Final Year, University of Rijeka, Faculty of Engineering, 2017. https://urn.nsk.hr/urn:nbn:hr:190:120631

List of Symbols, Abbreviations, and Acronyms

Acronym	Description
ACID	Atomicity, Consistency, Isolation, Durability
API	application programming interface
BSD	Berkeley Software Distribution
BSM	Sun Basic Security Module
CADETS	Casual, Adaptive, Distributed, and Efficient Tracing System
CAPP	Common Access Protection Profile
CC	Common Criteria
CDM	Common Data Model
CI	continuous integration
CRASH	Clean-Slate Resilient Adaptive Secure Hosts
CSV	comma-separated values
CTSRD	Clean Slate Trustworthy Secure Research and Development
DARPA	Defense Advanced Research Projects Agency
DDTrace	Distributed DTrace
DEQUE	Distributed Event Queries for EQ
DIF	DTrace Interface Format
DTrace	Sun's Dynamic Tracing system
E1, E2, E3, E4, E5	Engagement 1, 2, 3, 4, 5
ELF	Executable and Linkable Format
EQ	Event Query
GPLv3	GNU General Public License version 3
GSOC	Global Security Operations Center
IPC	inter-process communication
IR	intermediate representation
JSON	JavaScript Object Notation
NCSA	National Center for Supercomputing Applications
OpenBSM	Open-source Basic Security Module
OPUS	Observed Provenance in User Space
OS	operating system
POSIX	Portable Operating System Interface
PTTY	pseudo teletype
PVM	Provenance and Versioning Model
PVMv2	PVM version 2
py-cdg	Python Call and Data Graph
SDT	Statically defined tracing
SSDD	System and subsystem design document
SSI	Serializable Snapshot Isolation
SVF	Static Value–Flow

Acronym	Description
TC	Transparent Computing
TCB	Trusted Computing Base
TESLA	Temporally-Enhanced Security Logic Assertions
TRACE	Tracking and Analysis of Causality at Enterprise level
UFS	Unix file system
UI	user interface
UIUC	University of Illinois at Urbana-Champaign
USDT	Userland Statically Defined Tracing (USDT)
UUID	universally unique identifier
VM	virtual machine
XML	Extensible markup language
YAML	Yet-Another-Markup-Language