Advanced Linux tracing technologies for Online Surveillance of Software Systems

M. Couture
DRDC – Valcartier Research Centre

Defence Research and Development Canada

Scientific Report
DRDC-RDDC-2015-R211
October 2015

© Her Majesty the Queen in Right of Canada, as represented by the Minister of National Defence, 2015

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2015

Abstract

This scientific report provides a description of the concepts and technologies that were developed in the Poly-Tracing Project. The main product that was made available at the end of this four-year project is a new software tracer called Linux Trace Toolkit next generation (LTTng). LTTng actually represents the centre of gravity around which much more applied research and development took place in the project.

As shown in this document, the technologies that were produced allow the exploitation of the LTTng tracer and its output for two main purposes: a- solving today's increasingly complex software bugs and b- improving the detection of anomalies within live information systems. This new technology will enable the development of a set of new tools to help detect the presence of malware and malicious activity within information systems during operations.

Significance to defence and security

All the technologies described in this document result from collaborative R&D efforts (the Poly-Tracing project), which involved the participation of NSERC, Ericsson Canada, DRDC – Valcartier Research Centre, and the following Canadian universities: Montreal Polytechnique, Laval University, Concordia University, and the University of Ottawa. This scientific report should be considered as one of the Platform-to-Assembly Secured Systems (PASS) deliverables. The sponsors of the project are VCDS/CFD/DG Cyber and ADM(IM)/COS(IM).

These technologies open the door for the development of new advanced leading-edge anomaly detection algorithms. These would allow the detection of malicious activities within critical DND computing systems much earlier in the cyber-attack process.

Résumé

Ce rapport scientifique donne une description des concepts et technologies qui ont été développés dans le cadre du projet Poly-Tracing. Le principal produit qui a été rendu disponible à la fin du projet de quatre ans est un nouveau traceur logiciel appelé « Linux Trace Toolkit next generation (LTTng) ». LTTng représente en fait le centre de gravité autour duquel plusieurs autres travaux de recherche et développement ont pris place dans le projet.

Tel que montré dans ce document, les technologies qui ont été produites permettent l'exploitation du traceur LTTng et de son extrant pour deux principales raisons : a- le débogage logiciel qui est de plus en plus complexe de nos jours et b- l'amélioration de la détection en ligne d'anomalies dans les systèmes d'information. Cette nouvelle technologie va permettre de développer un ensemble d'outils pour aider la détection de la présence de logiciels et d'activités malicieux dans les systèmes d'information pendant les opérations.

Importance pour la défense et la sécurité

Toutes les technologies décrites dans ce document sont le résultat d'efforts collaboratifs de R & D (le projet Poly-Tracing), qui ont impliqué la participation de CRSNG, Ericsson Canada, RDDC – Centre de recherches de Valcartier et les universités canadiennes suivantes : la Polytechnique de Montréal, l'université Laval, l'université Concordia et l'université d'Ottawa. Ce rapport scientifique devrait être considéré comme un des livrables du projet « Platform-to-Assembly Secured Systems » (PASS). Les commanditaires du projet sont VCEMD/CDF/DG Cyber et SMA(GI)/CEM(GI).

Ces technologies ouvrent la porte au développement de nouveaux algorithmes très poussés de détection d'anomalies en ligne. Ces derniers seront en mesure de détecter, beaucoup plus tôt dans le processus de la cyber-attaque, les activités malicieuses dans les systèmes informatiques critiques des Forces.

Table of contents

Abstract . . . i
Significance to defence and security . . . i
Résumé . . . ii
Importance pour la défense et la sécurité . . . ii
Table of contents . . . iii
List of figures . . . iv
List of tables . . . v
1 Introduction . . . 1
    1.1 Purposes of this work . . . 1
    1.2 Goals of this document . . . 2
    1.3 Methodology . . . 3
    1.4 How to read this document . . . 4
2 Software tracing on Linux . . . 5
    2.1 The need for a very effective/efficient software tracer on Linux . . . 5
    2.2 The Linux Trace Toolkit next generation . . . 7
    2.3 The instrumentation of software applications . . . 10
    2.4 Software instrumentation with Umple . . . 12
    2.5 The output of LTTng – Execution traces . . . 13
    2.6 The synchronization of execution traces . . . 14
    2.7 The storing of system resource states . . . 15
3 Trace abstraction and analysis . . . 17
    3.1 The abstraction of execution traces . . . 17
    3.2 The detection of scenarios within execution traces . . . 19
    3.3 Other complementary studies . . . 20
4 Concluding remarks and recommendations . . . 23
References/Bibliography . . . 26
List of symbols/abbreviations/acronyms/initialisms . . . 31

List of figures

Figure 1: The methodology used in the Poly-Tracing Project.. 3
Figure 2: Main components of the LTTng software tracer.. 8
Figure 3: High-level functional view of an instance of the LTTng tracer and tools. 9
Figure 4: Static and dynamic instrumentation (the arrows indicate the execution flow). 11
Figure 5: The Umple Framework [11]. 12
Figure 6: How trace events can be organized in trace files. 14
Figure 7: Trace abstraction technologies [22]. 18
Figure 8: The creation of a scenario library and trace scanning.. 20

List of tables

Table 1: List of required/actual LTTng characteristics. 6

This page intentionally left blank.

1 Introduction

Networked software systems supporting DND platforms have reached unprecedented degrees of technological complexity that make their complete debugging and full certification impossible. Such systems will always contain software flaws that will manifest themselves as errors and service failures at runtime. Such errors and failures can lead to exploitable vulnerabilities. From an operational perspective, the failure of a software system to deliver trusted services in a timely manner may take the form of a sudden service interruption at a very critical moment. Other vulnerabilities could lead to the theft of classified operational data over extended periods of time.

Clearly, additional efforts should be deployed to provide software engineers with leading-edge tools that can address increasingly complex debugging problems. Moreover, new tools are also needed to improve the surveillance of these software systems during their execution to achieve early detection of anomalies. Herein this domain is called *Online Surveillance of Software Systems* (OS3). An overview of this vision is given in [1].

Software anomalies can be defined as: *software behaviours and states that diverge from the design specifications*. They can be caused by the manifestation of design flaws, the bad utilization of the system or cyber-attacks/malware. They may take place both at the level of software applications in the user space as well as deep within the operating system itself, in the kernel space. The numerous possible sources of software anomalies combined with the high degree of complexity of current software systems and cyber-attacks/malware make their detection very difficult at runtime.

1.1 Purposes of this work

This document describes the work done in a four-year applied R&D project—the *Poly-Tracing Project* [2]—to develop advanced technologies for software tracing and analysis. This project involved the *Natural Sciences and Engineering Research Council* of Canada (*NSERC*), Ericsson Canada and Defence Research and Development Canada (DRDC) as the main sponsors. Main work was performed by Canadian universities: Montreal Polytechnique, Laval University, Concordia University, and the University of Ottawa. This multidisciplinary team of experts worked collaboratively to develop the technologies for advanced debugging of complex information systems, and for the OS3, the two main motivations driving the work in the Poly-Tracing Project. The following lines briefly describe both aspects. It is important to note at this point that these advanced technologies were developed in such a way that they can be used to address both aspects of the project. They can be used *online* during operations for OS3, and *offline* in the laboratory for debugging purposes.

Motivation 1: Advanced debugging of software systems

Debugging problems with ever-more complex software and hardware technologies are much harder to solve with current development tools and frameworks. For example, these are often timing-related bugs that only manifest under real loads, when the hardware and software are interacting in real-time. Also, current viewing tools such as specialized graphic user interfaces for

debugging cannot handle the huge amounts of information that can easily be generated with newer high performance multi-core processors.

The lack of adequate debugging tools results in longer development cycles, operations and maintenance support problems and, importantly, poorly optimized and buggy software systems. More specifically, advanced software tracing tools are needed to capture precise data from the system while minimizing the overhead on the system. For example, the tools must handle the millions of significant software events per second that must be expected on a multi-core system running at several gigahertz. The lack of adequate tracing and debugging tools is a critical challenge for the deployment of today's software systems in multi-core hardware environments.

Motivation 2: Online surveillance of software systems (OS3)

The use of sophisticated debugging tools is not sufficient to completely remove all defects in software systems. The probability that a fielded system is completely bug-free is extremely low. As mentioned earlier, undetected flaws and bugs may become critical vulnerabilities if they are discovered by hackers. Another strategy must thus be used at runtime to further protect these systems from undesired software behaviours and states. This strategy consists in using cyber-security monitoring systems that aim at detecting malware and malicious activities at runtime.

Cyber-security monitoring systems can use different mechanisms. Two important ones are *signature-based detection* and *anomaly-based detection*. Signature-based detection systems (e.g. antivirus, host-based intrusion detection systems, network-based intrusion detection systems) are widely used. They search for known signatures of malware, such as predefined static sequences of bytes in the system's persistent storage, network transmission packets, or active memory. Their detection efficiency is limited because the number of new signatures that can be generated every day compared to the massive daily production of malware is very small. A characteristic of these is that they can only detect malware for which signatures are defined.

Anomaly-based detection can detect unknown or unforeseen malware. This involves specialized models that express the normalcy of software behaviours. Advanced detection mechanisms use these models to detect software behaviours that deviate from the ones contained in the models. Building comprehensive normalcy models expressing *all* normal behaviours and states of the software system is hard to achieve. These models are often incomplete and many false alarms are often generated for this reason.

1.2 Goals of this document

It was expected in the Poly-Tracing project that the exploitation of a new, very efficient software tracer would contribute to significantly improve the detection of undesired software behaviours and states at runtime. This document provides an integrated overview of the concepts and technologies that were conceived and developed in this project. Preliminary studies were also initiated to identify how the new software tracer, the Linux Trace Toolkit next generation (LTTng) [3] could be exploited to improve OS3. This document also shows that this technology represents a huge potential for current and future cyber-detection systems.

1.3 Methodology

DRDC addresses DND's technological problems in two different contexts; efforts that are a) *external to DND* and b) efforts that are *internal to DND* (blue and red rectangles in Figure 1). External applied research and development involves Canadian partners that originate from academic, industrial and governmental organizations. DRDC itself is also involved in external works.

Internal works involve people from DND, DRDC and also industrial organizations under contract. The Defence Scientists from DRDC act as *interfaces* between both contexts, segregating the data, information and knowledge to be exchanged between the partners.

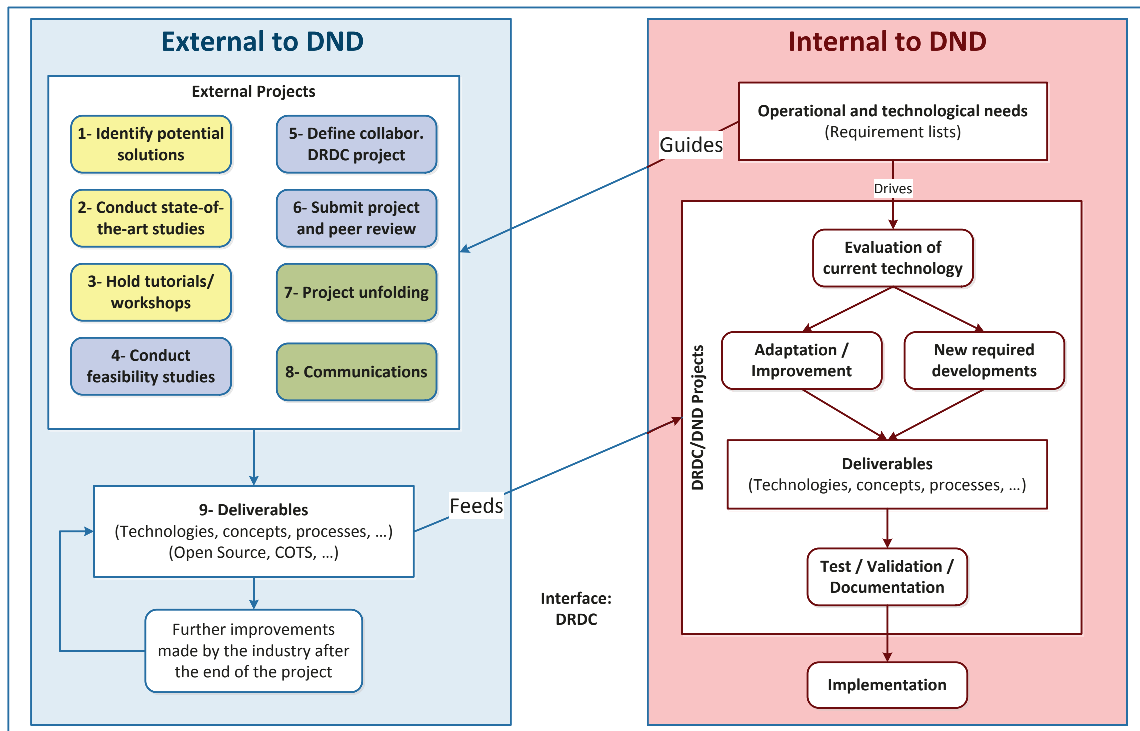

Figure 1: The methodology used in the Poly-Tracing Project.

External DRDC projects are defined according to DND's priority requirements. Preliminary studies are conducted to identify potential solution options (box 1 in Figure 1), conduct state-of-the-art studies on identified solution options (box 2), inform DND and capture feedback from the clients—specific branches or components of DND—regarding options that should be pushed further (box 3), and conduct feasibility studies on these selected options (box 4). Based on results obtained in these preliminary studies, a project is defined and submitted to DRDC for approval and financial support (boxes 5 and 6).

Once started, the Defence Scientist enforces the bridging between the two contexts, and regularly organize subsequent *client meetings* (box 3) to keep DND informed of the latest external developments and capture the latest information regarding internal requirements, priorities, and constraints. This information will be used to adapt the work that is done in the external context to

the internal one. Once the project is finished, all relevant technologies are transferred to DND. Further adaptations, improvements, and developments are then made. Subsequent internal DRDC projects can be defined and initiated for these purposes.

This methodology enables the definition of major projects in which government, academia and industry collaborate to develop solutions that are well aligned with the latest government guidelines. The presence of an industrial partner complements the presence of academic researchers. The latter tend to work at conceptual and theoretical levels while experts from the industry tend to be more at the applied level, aiming at delivering implementation-ready products at the end of a project. Collaborative projects involving government, academic researchers and experts from the industry tend to produce high-quality technologies that have higher Technology Readiness Level (TRL) values, focussed on real operational technological problems.

1.4 How to read this document

This document should not be considered as the official documentation of these technologies. It rather presents brief overviews of all the concepts and technologies that were developed in the project. References can be visited to get complete documents, videos, demonstrations and the software components themselves.

Chapter 2 describes software tracing on the Linux operating system. It provides a description of the LTTng software tracer as well as the tools allowing the instrumentation of software applications, the synchronization of trace events in execution traces originating from different CPUs, and the capture and storage of these traces.

Chapter 3 presents some preliminary studies to start identifying how the LTTng software tracer and its output execution traces can be exploited for OS3. A mechanism for trace abstraction is described, as well as a language and algorithms allowing to search for scenarios and to detect anomalies in traces.

Chapter 4 concludes this document.

2 Software tracing on Linux

A software tracer requires that *trace probes* be inserted at specific locations within the source code or the binary executable of a program. Once a processor executes a trace probe during the execution of this program, a *trace event* is generated by the tracer. This event and other selected contextual parameters are then captured, transformed and recorded into an *execution trace* for further analysis. Execution traces potentially contain a huge number of trace events that reflect the software behaviours and states of the traced program. Activating/deactivating trace probes at will during the execution of the program provides control over the focus and granularity of the content of execution traces.

Desfossez [4] and Couture et al. [5] provide a review of literature as well as an overview of software tracers on Linux and MS Windows. Software tracing tools existed for Linux before the Poly-Tracing Project but they could not be used for advanced debugging/surveillance because none simultaneously provided all the needed capabilities (such as the ones listed in Table 1). Nevertheless, one of these tracers seemed to contain the basic technological assets that could be pushed further to produce the needed technology; the Open Source Linux Trace Toolkit (LTT) software tracer [6].

The LTT software tracer, which was already supported by Autodesk, IBM and Google, was selected as the best candidate and pushed further in the Poly-Tracing Project. The end-result of this work is the Linux Trace Toolkit next generation (LTTng) [3]. LTTng software probes are now included by default in the latest versions of the Linux kernel (from version 2.6.36), the tracer is also available in major Linux distributions such as Ubuntu and Red Hat Enterprise Linux 7.

This chapter describes LTTng and other tools supporting software tracing on the Linux operating system. Section 2.1 presents the list of requirements that were considered at the beginning of the project, and the list of characteristics LTTng offers at the end of the project. Section 2.2 provides a relatively detailed overview of LTTng and related tools; LTTng being the centre of gravity around which all other work revolved.

Section 2.3 describes the process of software instrumentation and presents a new technology allowing the instrumentation of huge legacy software systems. Section 2.4 provides a description of how LTTng execution traces can be structured. An overview of the algorithm used to synchronize trace events that originate from different processors is presented in Section 2.5. Finally, Section 2.6 presents a new technology allowing the storage of the evolution of system states in a specialized database. All these new technologies were developed in the context of the Poly-Tracing Project.

2.1 The need for a very effective/efficient software tracer on Linux

Table 1 lists the requirements that were considered at the beginning of the Poly-Tracing Project for the development of a software tracer for the Linux operating system and the characteristics of the LTTng software tracer as of the end of the Project [3].

Not mentioned in Table 1 is the fact that LTTng provides the users with a simplified integrated control interface to manage tracing processes. Simultaneous capture and recording of multiple traces is supported through multiple *tracing sessions*. These are created and can be concurrently active, each with its own instrumentation set. Non-root users that are registered to a predefined *tracing group* are given the permission to perform both kernel and user space tracing. The LTTng interface API library allows any software application to embed LTTng and have full control over the tracing process. The Tracing and Monitoring Framework (TMF) [7] is an example of an environment in which LTTng can be embedded.

Table 1: List of required/actual LTTng characteristics.

Required characteristics	**LTTng software tracer characteristics at the end of the Poly-Tracing Project**
Online tracing of software systems that are running on computing systems ranging from small-scale imbedded systems to complex multi-core systems.	Supported architectures (as of Dec. 2013) are: x86, x64, PowerPC 32/64, ARM (ARMv7 OMAP3 in particular), MIPS, sh, sparc64, s390. Other supported architectures are added every year ([3]).
Local and remote control of both the focus and granularity of the generated execution traces at all times during the tracing process.	Control of the monitoring process can be local or remote. Live controlled monitoring supports the activation or deactivation of any static or dynamic probe at runtime, making possible the online selection of both the focus of the tracing process (what is traced within the system), and the granularity of the content of the execution trace (the number of probes and their specific locations).
Ability to send execution traces to both local and remote computers.	The choice of execution trace destination (local or remote disk) can be made.
Optimization of the tracing mechanisms so that they do not impose significant overhead performance penalties on the traced system; the system and traced processes should be perturbed as little as possible by the tracing process itself.	The tracepoints (LTTng main static probes) involve no trap mechanisms neither any kernel system calls, yielding very low performance impacts on the system. Probes that are not activated at runtime have almost zero performance impact. LTTng also imposes a very low overhead because it uses the Read-Copy-Update synchronization mechanism (a wait-free alternative to the reader-writer lock mechanism), it also uses efficient per-CPU buffers to capture the trace events and uses non-blocking atomic operations.
Use of the same tracer for both complex debugging and online system monitoring.	LTTng provides full support of both complex debugging and OS3. For example, it can be used on systems to solve very hard to find bugs such as race conditions in multi-core hardware environments, where current debuggers are too intrusive.
Ability to run on most Linux distributions.	LTTng can be implemented and run on many types of computing platforms and most Linux distributions. An experimental version of LTTng is also available for CYGWIN under MS Windows.
Ability to scale to a high number of CPU cores and to support real-time computing.	LTTng scales to a high number of CPU cores and supports strong real-time.

Ability to trace both the user and kernel spaces of the operating system.	Both the user space and the kernel space domains of the Linux operating system can be traced by LTTng. Probes can be inserted anywhere in both spaces, including in interruption contexts.
Use of many types of tracing probes that could be installed within the system.	LTTng allows the concurrent utilization of both static probes (tracepoints) and dynamic probes (kprobes).
Precise common timing of trace events, the event that is generated when a trace probe is executed by a CPU.	The timestamp of a trace event accurately tracks individual processor cycles, and the event specifies on which processor it occurred in multi-core systems. Timestamps allow event ordering across cores/CPUs (except if hardware makes it impossible). The timestamp ordering of trace events originating from the user and kernel spaces is also made using *the same base*, allowing correlations of trace events originating from both domains.
Standardization of the output of the software tracer.	Execution traces are formatted according to the Common Trace Format (CTF) [8]. CTF is a versatile self-described binary formatting standard that is highly optimized for compactness.

2.2 The Linux Trace Toolkit next generation (LTTng) software tracer

This section presents a relatively detailed overview of the LTTng software tracer. More technical details are available on the Web site [3] and the official DRDC documentation [9]. Figure 2 and 3 are used to illustrate how LTTng is made and how its components can be used. Figure 2 illustrates the main components of LTTng and related tools, and some dependency links between them. In Figure 2, the *traced system* was separated from the *observer system* to ease the understanding of the different components of LTTng. Both computers are connected through SSH. Figure 3 provides a functional view of the same instantiation of LTTng.

Figure 2 shows that *specialized probes* such as tracepoints and kprobes can be used to instrument and trace any C/C++, Java, or Erlang software applications, as well as any kernel services (including modules) of the traced system. As shown later in this chapter, static instrumentation inserts tracepoint probes in the source code of a program, which is then recompiled and linked with specific LTTng libraries; liburcu, liblttng-ust (and the LTTng VM adaptors if tracing takes place in a virtual machine).

This instrumented program is then run on the computer. Every time an active probe is executed, a *trace event* is generated and captured in local buffers. The LTTng Consumer Daemon (lttng-consumerd in Figure 2) then transfers the buffer contents to their destination (local or remote computers). When *streaming* a trace to a remote destination, the Linux daemon lttng-relayd listens to the network on the receiving end and stores the trace on the remote system's local storage.

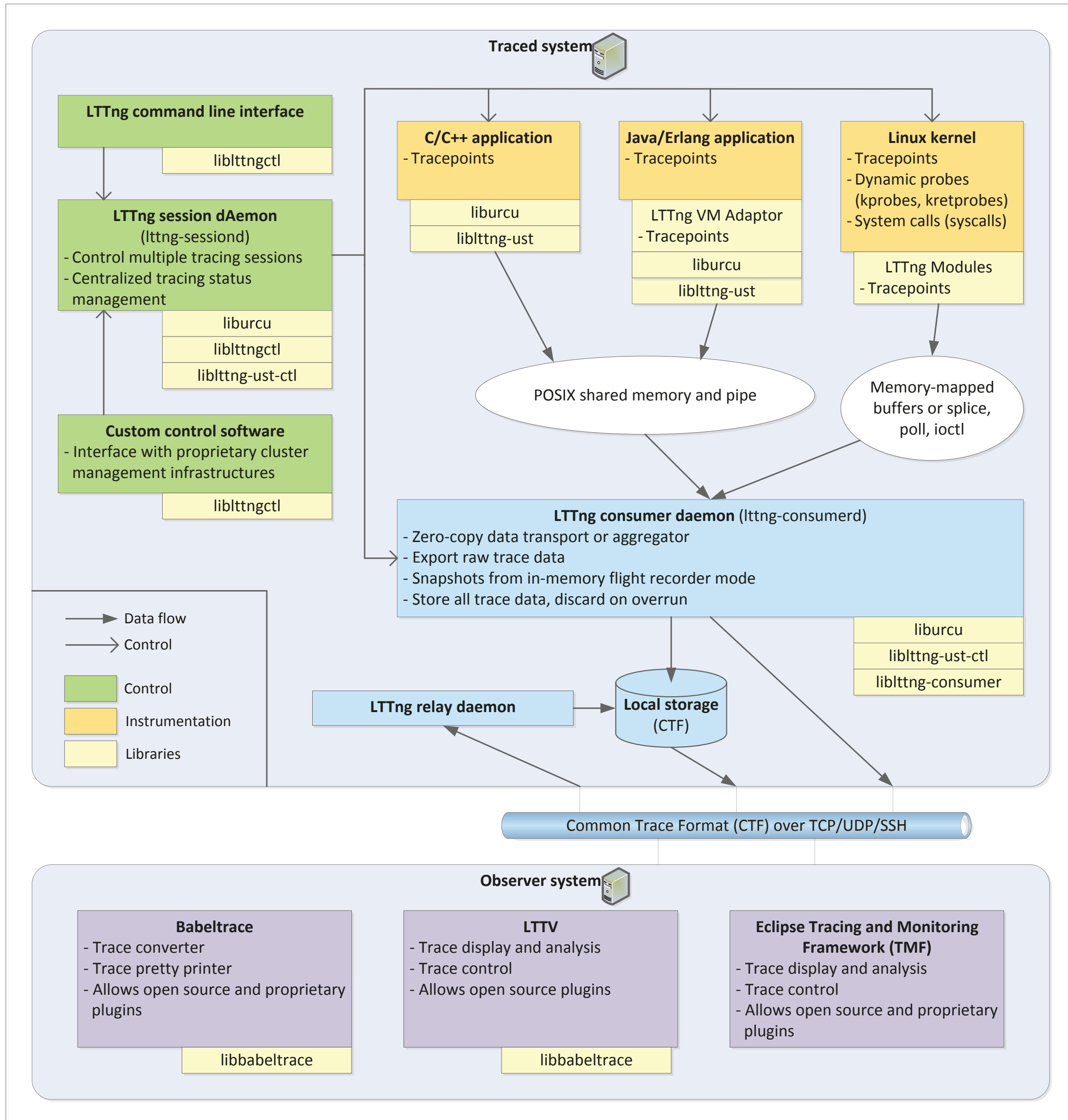

Figure 2: Main components of the LTTng software tracer.

A further option is *live tracing*, which allows a trace viewer to access trace events as soon as they arrive. The viewer works with both streamed and local traces—in the latter case a local relay daemon is used. At the other extreme of consumer daemon activity, LTTng can leave trace events in the local buffer, newer events overwriting older ones, until a *snapshot* command is issued, at which point the consumer daemon does its duty. This is sometimes referred to as *flight recorder mode* because it is similar to how aviation's flight recorders (popularly known as "black boxes") work, saving a record of the last two hours of the flight's events.

Figure 3 shows that a number of tools can be used on the observer system to manage generated trace events. The babeltrace library can be used to convert execution traces and make their content available to other software applications such as specialized trace analysis, trace visualization and storage. The Common Trace Format (CTF) [8] is used to represent and format execution traces. The Linux Trace Toolkit Viewer (LTTV) [3] and the Tracing and Monitoring Framework (TMF) [7] are currently used to visualize the trace events, control the tracing process, and analyse execution traces. TMF allows the addition of new algorithms for trace management and analysis through plug-ins.

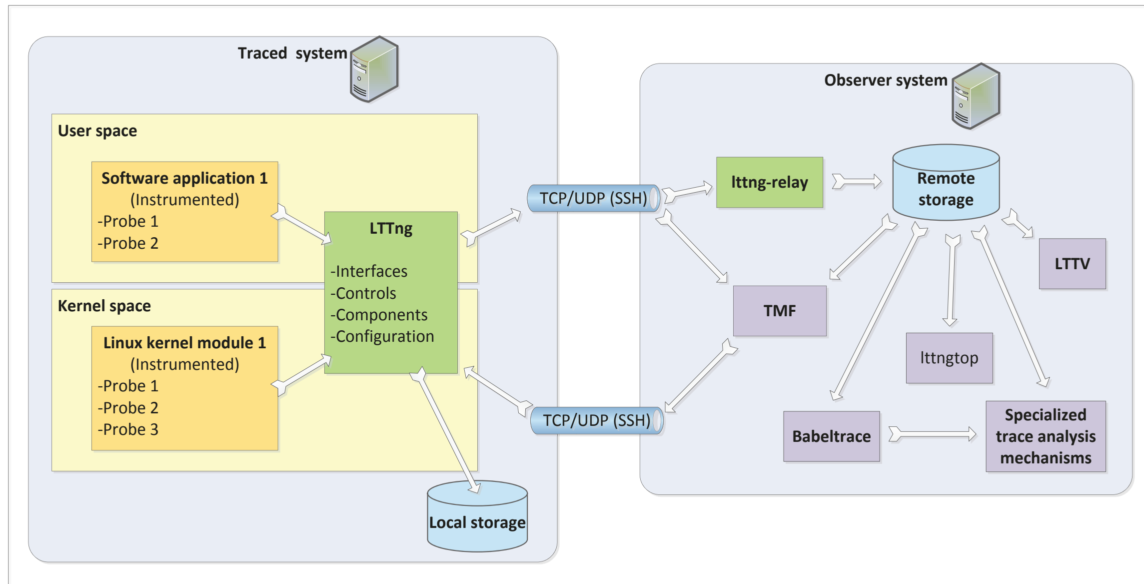

Figure 3: High-level functional view of an instance of the LTTng tracer and tools.

The control of the tracing process is achieved through the LTTng session daemon (lttng-sessiond; the central green rectangle in Figure 2). This centralized management allows the control of one or many simultaneously tracing sessions on the traced system. The command line interface or custom control software (such as TMF) can be used to achieve this control (locally or remotely).

Some of the most important commands are listed in the following lines.

- **lttng**: allows interaction with the tracer and tracing sessions to control both kernel and user-space tracing. Available functionalities are:
  - add context to one or many channels;
  - evaluate overhead and calibrate;
  - create, destroy, start, or stop (suspend) tracing sessions;
  - view the trace of a tracing session;
  - list the information related to a tracing session;
  - switch between tracing sessions;

- enable or disable one or many channels;
- enable or disable one or many events; and
- grab the contents of the tracing buffers in order to export them to a destination specified by the user.

- **lttng-gen-tp**: is a python script that simplifies the generation of user space tracepoint source code files.
- **lttngtop**: is an ncurses-based interface for reading and browsing execution traces and find software bugs. It is similar to the top programs on Linux systems. It provides a number of statistics that are related to tracing sessions.
- **babeltrace-log**: is a tool to convert a text log (read from standard input) to the Common Trace Format (CTF) and to write it out as a trace file.

LTTng controls the flow of events into a trace at several levels. At the highest level, an entire trace session can be suspended, temporarily stopping any events from being traced. Within the session, events are organized in channels, and each of these channels can be activated and deactivated separately. Within a channel, each individual event can also be activated and deactivated separately. Lastly, a filter can be applied to each event (see 2.5, below).

2.3 The instrumentation of software applications

As mentioned, trace probes need to be inserted at specific locations within programs to tell the software tracer when to generate trace events. These events are then transformed and captured in execution traces for online or post-mortem analyses. The process of inserting probes within a program is called *program instrumentation*.

Online tracing with LTTng can be done through the utilization of static and/or dynamic tracing probes. *Static instrumentation* (Figure 4A) requires the addition of probes (called tracepoints) directly in the source code of the software, which is then recompiled and run. *Dynamic instrumentation* (Figure 4B) requires the dynamic addition of probes at specific locations within the binary code, while the software is running.

The following describes the different probing mechanisms that can be used with LTTng.

The tracepoint mechanism

The tracepoint mechanism (Figure 4A) was developed to allow static tracing of software running in either the user or kernel space. A tracepoint is inserted at a specific location within the source code of a software component, which is then recompiled. When this tracepoint is executed, it will call a function, the probe. This function can be provided at runtime using standard dynamic linking.

Tracepoint activation and de-activation is achieved by changing a single byte in the tracepoint's prologue. Performance penalties are very low if a tracepoint is not activated, as normal execution resumes in very few steps. When activated, the user-provided function will be executed every time the tracepoint is executed. The ability to activate or deactivate tracepoints at runtime in the

system allows the selection of both the focus and granularity of the tracing process, which define the content of execution traces.

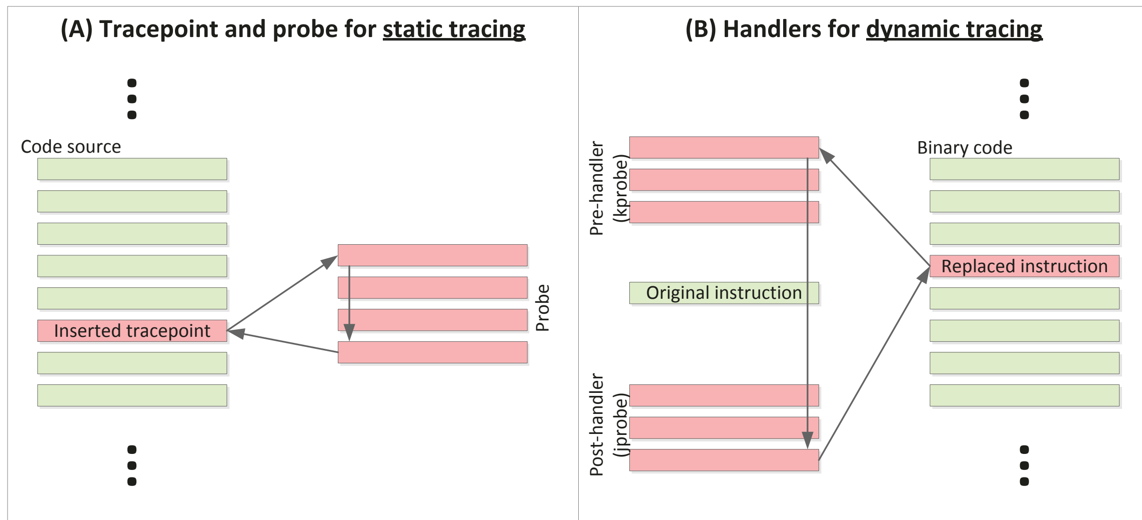

Figure 4: Static and dynamic instrumentation (the arrows indicate the execution flow).

The kprobe mechanism

The kprobe mechanism (Figure 4B) was developed by IBM to allow the dynamic tracing of software that is running in the kernel space of the Linux operating system. It is built as a kernel module that can be loaded or unloaded, no recompiling (and rebooting) of the kernel is necessary. A specific instruction in the kernel that needs to be traced is associated with a set of handlers: a pre-handler, the *kprobe,* and a post-handler, the *jprobe*.

More precisely, this instruction is saved in a buffer and is replaced by a *breakpoint instruction*. When the processor encounters this breakpoint, the trap mechanism is put in play: the pre-handler is invoked, the instruction in the buffer is then executed, followed by the execution of the post-handler. The handlers contain the instructions generating the tracing events. This binary instrumentation imposes a greater penalty on the performance of the system than the tracepoint mecanism.

This mechanism can be used by LTTng to dynamically monitor the execution of any instruction in the kernel. It supports refined debugging of the kernel and online monitoring of events that take place deep within the operating system of a production system.

Other supported probing mechanisms

Perf [10] is a multi-core, multi-user-aware performance analysis tool for Linux. It allows statistical profiling of the entire system in both the kernel and user spaces. It supports hardware performance counters, tracepoints, software performance counters (e.g. hrtimer), and dynamic probes (e.g. kprobes). LTTng allows the addition of performance counter data to execution traces *on a per event basis*. For example, it would be possible to add the number of CPU cycles at each generated LTTng event. The lttng command *help* (add-context) provides all possible the performance counter values.

2.4 Software instrumentation with Umple

The instrumentation of legacy programs (code source) may represent a problem when its documentation has not been updated along with the evolution and major changes of its source code. This is particularly true for very large software applications that have often seen their architecture changed by different people over time. The instrumentation of these programs would require a programmer to spend a lot of time trying to understand the logic of the program to identify the numerous critical locations where trace probes should be inserted.

The Umple Framework

A new technology allowing both source code improvement and instrumentation has been developed in the Poly-Tracing Project [11]. As shown in Figure 5, the technique uses the Umple Framework [12, 13]. Umple can be used in the *forward engineering mode* to generate entire software systems. It can also be used in the *reverse engineering mode* to recover models, documentation and design from existing legacy software systems. Figure 5 illustrates the process for both modes.

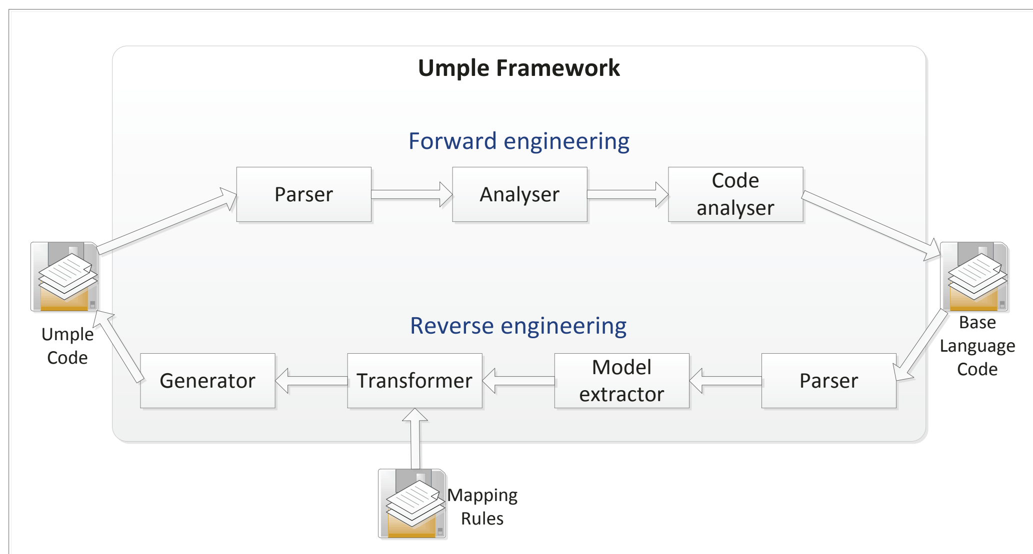

Figure 5: The Umple Framework [11].

Forward engineering

Starting from the Umple code, the parser tokenizes the model and transfers it to the analyser. The analyser processes the tokens and converts them into an internal representation. Then, the code generator translates the internal representation into other additional models (e.g. Papyrus XMI, EMF, Yum, Xuml), various diagrams, or source code (e.g. Java, C++, PHP, Ruby or SQL).

Reverse engineering

Starting from Java or C/C++ source code, the parser creates an Abstract Syntax Tree (AST) of the code. The AST is then taken by the model extractor to build a base-language model according to

the base-language's meta-model. This base-language model is then transformed into an Umple model using a pre-defined set of mapping rules. The generator validates this model and generates the Umple code from it. The Umple code is a friendly human-readable modelling notation seamlessly integrated with algorithmic code.

Instrumenting legacy software applications with Umple

Most techniques for software instrumentation consist in adding trace probes directly in the source code, not at the level of the model. Umple proposes a new method for adding trace probes at the *model level*. Umple uses the concept of *trace directives* as a means of specifying how the source code will be instrumented. Trace directives define a new tracing language called Model Oriented Tracing Language (MOTL). These can be placed anywhere within the *Umple code* (Figure 5)—which describes a model—allowing the tracing of attributes and state machines at the model level.

Tracing can be limited based on pre-defined constraints such as a condition being true or a maximum number of trace event occurrences being reached. Developers can also specify that they want a certain state transition or event to be traced, have the ability to limit the scope of tracing to a certain level of sub-states, or trigger tracing when certain trace-based conditions become true.

The instrumentation of the source code of a legacy software application is done by first reverse engineering it using Umple (Figure 5). Once generated, the Umple code representing the program is instrumented using trace directives. Then, the source code of the instrumented program is generated using the forward engineering mode. The instrumented program is then ready to be compiled and run.

This technology and associated documentation can be found at this web site [11]. A web-based version of Umple (UmpleOnline [14]) is also available, it allows experimenting with Umple on the Web.

2.5 The output of LTTng – Execution traces

As mentioned, a trace event is a record in an execution trace. It is associated with the execution of an LTTng probe by a CPU. A trace event that is recorded in an execution trace may contain many parameters like the event's timestamp, some identification numbers (CPU ID, user ID, process ID), the event's type and the payload. The payload is tracepoint-defined. It may contain context data that would be particularly helpful for advanced debugging or OS3 purposes. Couture et al. [15] provide a description of LTTng execution traces.

Figure 6 illustrates the different options for determining how trace events will be organized within execution trace files. It is important to mention that *LTTng segregates generated trace events into a number of streams that correspond to the number of CPU cores*. For example, Figure 6 shows that tracepoint number 2 generates four streams of trace events, indicating a quad-core system. Trace events that are generated by any active probe will be inserted in one of these four streams (typically, a given thread of an instrumented application is executed on a single core at a time, but the operating system's task scheduler may migrate it between cores).

Figure 6 shows that *filters* can also be used to exclude trace events during the tracing process. A filter can be defined for any *type of trace event at the channel level*. Each filter is a simple expression that is evaluated against the event's fields and which must be true for the event to be included in the trace.

The concept of *channel* offers even more flexibility regarding the configuration of execution traces. This mechanism may contribute to ease or accelerate subsequent trace pre-treatments and analyses. The user can specify which trace events (trace probes) will be handled by which channel(s). Note that a trace event can be assigned to one or more channels. It is possible to create and reconfigure channels during the tracing process, and channels can be activated or deactivated at will during tracing.

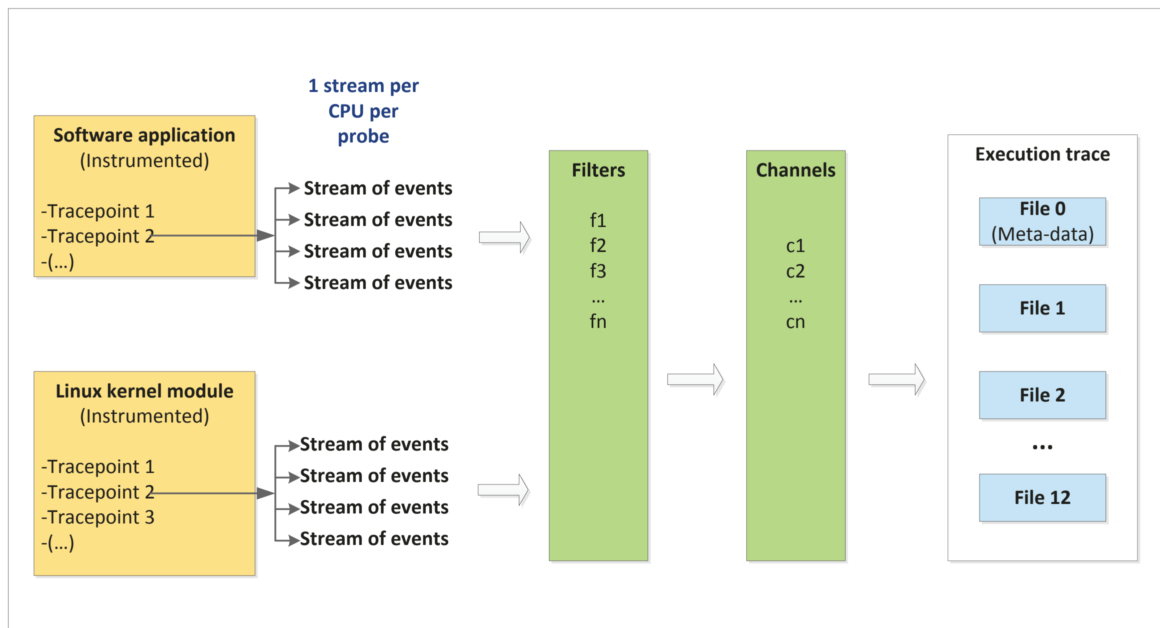

Figure 6: How trace events can be organized in trace files.

Finally, the number of files associated with an execution trace is always equal to the number of channels times the number of CPU cores, plus one additional file containing the meta-data describing the contents of the execution trace. Following the example in Figure 6, the number of channels is 3; there are 4 cores plus the meta-data file so 3 x 4 + 1 = 13 files.

Execution traces are transferred over the network and saved on disk according to a new standardized format which was developed by the Multicore Association and the Linux community: the Common Trace Format (CTF) [8]. The babeltrace software tool [3] can be used to translate execution traces from the CTF format to any other user-defined format.

2.6 The synchronization of execution traces

LTTng can control many tracing sessions that are taking place on the computing systems of a LAN, and the transfer of the generated execution traces to a single computer for further analysis.

This capability represents new opportunities for remote OS3 and cyber-surveillance. It may, for example, contribute to help detect the propagation of a malicious software anomaly from one computer to another on the LAN by identifying the manifestation of its presence from one execution trace to another over time.

Before implementing such a *LAN-wide cyber-analysis* of execution traces, a problem remains to be resolved: the synchronization of trace event timestamps *originating from different remote CPUs* (remote computers). A trace event that is generated on a system receives a timestamp that is not synchronized (not in the same time-frame) with those of trace events generated on other (remote) CPUs. Whereas the CPU clocks on a single machine are closely synchronized, clocks belonging to different machines may have relatively large systematic differences. The problem is further compounded by the fact that different CPU clocks are drifting at different rates.

Efforts were made in the Poly-Tracing Project to develop new mechanisms to synchronize trace events that originate on distributed computers. The *convex hull algorithm* from Duda et al. [16] was studied and used to estimate a conversion function between a pair of distributed computers [17]. Trace events were generated by LTTng on each computer, using tracepoints that were located in the Linux kernel, just before NIC driver calls, to reduce time-stamping delays. Execution traces were analysed offline. The main conclusions of this work are reproduced here from [18]: The implemented convex hull algorithm "guarantees no message inversion and gives strict bounds on accuracy at any point in the traces. Its run time is quadratic in the worst case but it scales almost linearly on practical traces." Factors that affect offline synchronization were also studied. Accuracy could be improved "by using a network with lower latency and by using a higher message rate."

The effects of the linear clock assumption were also shown experimentally. "With a constant message rate, lengthening the trace duration reduces precision and gives a false impression of improving accuracy. (…) During our experiment, we have achieved a synchronization accuracy of +/– 15 µs and an estimated precision of 9 ns on a network with an estimated minimum latency of 39 µs." The developed algorithm could be extended to long-running and streaming traces by considering sub-intervals. "It could also be extended to systems of more than two nodes" by using algorithms that would "propagate the factors (of the linear conversion function between pairs of nodes) efficiently while adjusting the accuracy bounds (…) a separate factor-propagation step is thus needed when synchronizing more than two nodes" [17].

Algorithms were successfully developed to compute a common reference timeframe from timestamp values, allowing for the fusion of traces for their global display and analysis. New algorithms for the online streaming mode were then successfully developed [19, 20]. They were first implemented in an experimental viewer but have recently been implemented into the Tracing and Monitoring Framework (TMF) [7].

This technology and its associated documentation can be found at this web site [3].

2.7 The storing of system resource states

Execution traces can be used for complex debugging, OS3, and in certain cases, post-mortem forensic analyses. Most of the time, they must be read, visualized, replayed many times, and

submitted to different specialized algorithms for analysis. The volume of data that is generated by LTTng can be quite large, especially if many components of the kernel and software applications running on a multicore system are traced. This represents a problem if long tracing sessions (many hours and many tens of terabytes of data) must be saved to disk of limited capacity.

Solutions were proposed in the Poly-Tracing Project to address this problem. The first option may help capture relevant information for forensic analysis. It consists in creating *fixed-size circular buffers* on the hard disk in which trace events are recorded in a round robin fashion. In this mode, the latest events that were generated—over a timespan that is determined by the size of the circular buffer and the production rate of trace events—will be included in the on-disk circular buffer. After a system crash, a forensic analyst would then have the very last captured data. The appropriateness of the focus and granularity of the content of the trace will determine the type of forensic analyses that can be conducted.

Another solution was conceived and prototyped in the Poly-Tracing Project. Instead of storing trace events, the new system stores *system resource states* such as running processes, open files, scheduled CPUs, the number of bytes read/written on the network card, etc. [21] The developed storage technologies and corresponding data structure allow one to model, store, manage and retrieve the state values for any number of system resources. The objects that are saved in the database are *state intervals*. They are actually changes of a resource. The reconstitution of the evolution of the states of a resource at any time in the past is made by using these state intervals.

The database is a tree-based state history that is disk-based instead of RAM-based for scalability purposes. Currently, the reconstitution of the evolution of the state of a resource over time is made with the help of a viewer. The modelled state at any point in time can be reloaded very rapidly in the viewer, which is suitable for interactive navigation through huge execution traces [21].

Extracted state values can be used online and/or offline to track problems with the resources of the system. In case of cyber-attacks or performance degradation, the ability to rebuild the state of the involved resources of the system will contribute to help understand the causes of the problem, and possibly find its roots.

This technology and associated documentation can be found at this web site [3].

3 Trace abstraction and analysis

As mentioned, the technology that is currently used for the OS3 is clearly not sufficient to detect all undesired software behaviours/states during operations. Huge improvements are still needed in this domain. The new software tracer LTTng offers tracing capabilities that were not available until now. It opens the door for solutions that will help fulfill current technological gaps in this domain.

This chapter provides high-level overviews of preliminary studies that were conducted to start identifying how the new LTTng software tracer and its output can be exploited for OS3. Section 3.1 describes a mechanism that can be used to abstract low-level execution traces into more meaningful human-readable traces. Abstraction objects form a library that can be used in different types of analysis. A descriptive language and its associated detection technologies are described in Section 3.2. Using this language, scenarios describing suites of software behaviours can be defined, and the detection engine can be used to check for their presence in execution traces. Section 3.3 presents other preliminary studies that were conducted within the Poly-Tracing Project.

3.1 The abstraction of execution traces

The stream of trace events that is generated per unit of time on one computer can be huge if many components are simultaneously traced and if the computer is multicore. The large volume of data makes the content of the execution trace hard to understand. This content is even more complex when the traced components are low-level, such as systems calls or kernel modules. A first effort was made in the Poly-Tracing Project to *abstract trace events in execution traces*. The main goal was to ease the understanding of complex execution traces, to reduce their sizes, to possibly ease trace analysis, and to build a library of abstract objects (patterns) that can be used in different types of analysis. The correlation analysis of two abstract traces is an example application.

A new trace abstraction algorithm [22] and a library of 120 patterns modelling the normal utilization of system calls were developed. In this work, patterns are represented by state machines. They are composed of trace events (*transitions*) and system modes (*states*). The system uses the pattern library to transform low-level *system call trace events* into high-level *abstract objects* without losing information.

Figure 7 shows that the trace abstraction algorithm takes as input an execution trace. It scans the content of the trace, searching for patterns in the library. Once the process is completed, a new higher-level trace is generated. Trace events corresponding to the patterns that were found in the LTTng trace are replaced by their corresponding abstract objects. Trace events that could not be abstracted are kept and tagged as *noise* by a utility program.

Examples of low-level patterns manually implemented in the library are: file management (open, read, write, close, access, stat), socket management for both TCP and UDP (create, bind, connect, listen, accept, send, receive, close), process management (clone, execute, exit) and memory management and page fault patterns. The technique has been used to abstract many execution traces originating from different Linux processes. A 65% compression ratio could be obtained in

many cases [22] and the replacement of low-level events by their abstract objects in the abstract trace made its content much easier to understand.

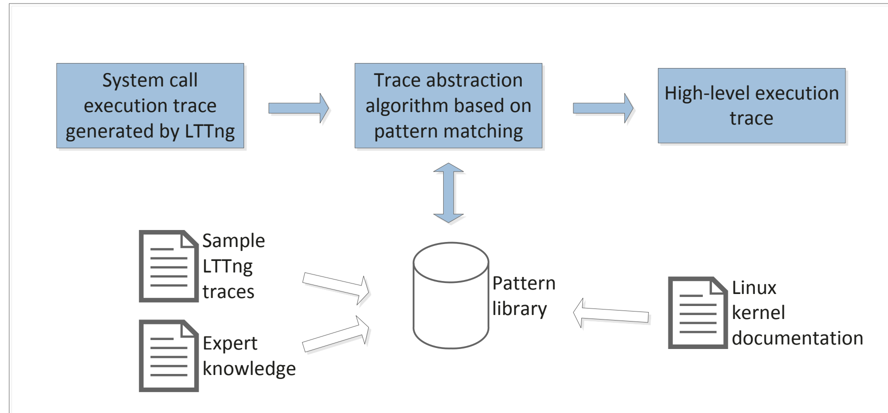

Figure 7: Trace abstraction technologies [22].

This technology and associated documentation can be found at this web site [23].

Another preliminary study was conducted in the Poly-Tracing Project to help understand large execution traces [24, 25]. It consisted in developing a novel trace analysis framework that divides the content of the trace into smaller, meaningful trace segments corresponding to the program's main *execution phases*. An execution phase is a portion of the program that exhibits common behaviours. Each phase represents an essential step of the general program, and phase transitions may represent the logic of the program from one step to another.

The generation of phases from execution traces is made using an approach inspired by the *Gestalt laws of perception*; an overview of the approach can be found in [24, 25]. These laws are based on the concepts of similarity, proximity and good continuation. They are used in psychology to describe how the brain visually recognizes objects and shapes as wholes and not just as points and lines, or how the human perceptual system segments local elements against their context and integrates them as structured collections of meaningful objects.

The framework is currently an early prototype that mimics how the human brain and its perception system automatically (not voluntarily) deal with huge volume of visual data considering limited short-term memory and required short response times. It involves three components; the Trace segmenter, the Gist builder, and the Content prioritizer. The Trace segmenter uses the Gestalt laws of perception to divide the large execution trace into meaningful segments representing execution phases. Then, the execution phases are scanned (random selection of trace elements from each execution phase) by the Gist builder to produce a global perception. Finally, the execution phases are analysed in more detail by the Content prioritizer.

This type of analysis usually produces pop-out effects, which consist in rapid detection of elements that differ greatly from their surrounding elements. These results are preliminary. More studies and work are required to calibrate the three components of the framework.

3.2 The detection of scenarios within execution traces

The scanning of large execution traces for the detection of complex series of trace events forming complex scenarios was also studied [26]. The framework that was developed allows the definition of *scenarios of software behaviours*, which are built from the set of easy-to-use pre-defined instructions of a new programming language. A specialized search engine can then be used to scan a trace, searching for trace events corresponding to the elements of a user-defined scenario. A scenario can thus be built and used to search for any type of software behaviour in execution traces.

The framework allows recursive scenarios and the grouping and naming of many related scenarios. For example, a *security group* would contain all the scenarios defining cyber-attacks, and a *management group* would contain all the scenarios that are related to management tasks that are conducted on the system. The detection of scenarios pertaining to the first group will of course trigger actions that are very different from the ones pertaining to the second group.

The instructions that are used to define a scenario are based on a declarative language that was conceived and tested during the Poly-Tracing Project. As shown in Figure 8, defined scenarios are verified and optimized before being included in the scenario library. The Advanced Fault Identification (AFI) system [26] takes as input the execution trace to be scanned. It searches the trace for the presence of any scenario that is included in the library of scenarios. Once a scenario has been detected, specific actions can be taken. Examples of such actions are: displaying graphically the detected scenarios, logging a record in a file, creating a message to be sent to a remote computer, executing a given Linux command, shutting the computer down, etc.

Many types of tests were conducted on this technology. The results show that "the execution time of AFI is linear with respect to: the number of events in the trace, the number of concerned scenarios, the complexity of the scenarios, and the number of coexisting scenarios" [26]. Two problems remain to be solved with the AFI technology: the fact that trace events do not always appear in the same order in the execution trace, and the imprecision of the definition of software behaviours in a scenario.

The AFI technology was thus pushed further to address these problems and provide system administrators with precise diagnostics of the state of their systems. The goal was to develop a system that is able to reproduce, at least partially in this project, the reasoning of an expert during the analysis of a cyber-attack. A new expert system was thus developed. As shown in Figure 8, it captures AFI's results, logically analyses the trace and reports the evolution of the system's states. This approach is called "Logical approach for the analysis of execution traces" [27].

The proposed approach uses a set of properties that describe the state of the system, and consists in searching for suspect software behaviours by analysing the effects any software actions may have on the resources of the system. Compared to signature-based detection engines, this approach may accelerate the detection of cyber-attacks because it is based on the *effects* software *actions impose on the system*, instead of searching for precise sequences of software actions.

Proofs of concept were successfully made using software actions on the Linux file system. The language used for the specification of knowledge and the representation of the states of the system is sufficiently generic to be used to represent any type of software actions on any resource

of the system. Results of this work show that this technology is able to detect known, known but slightly modified, and unknown cyber-attacks [27].

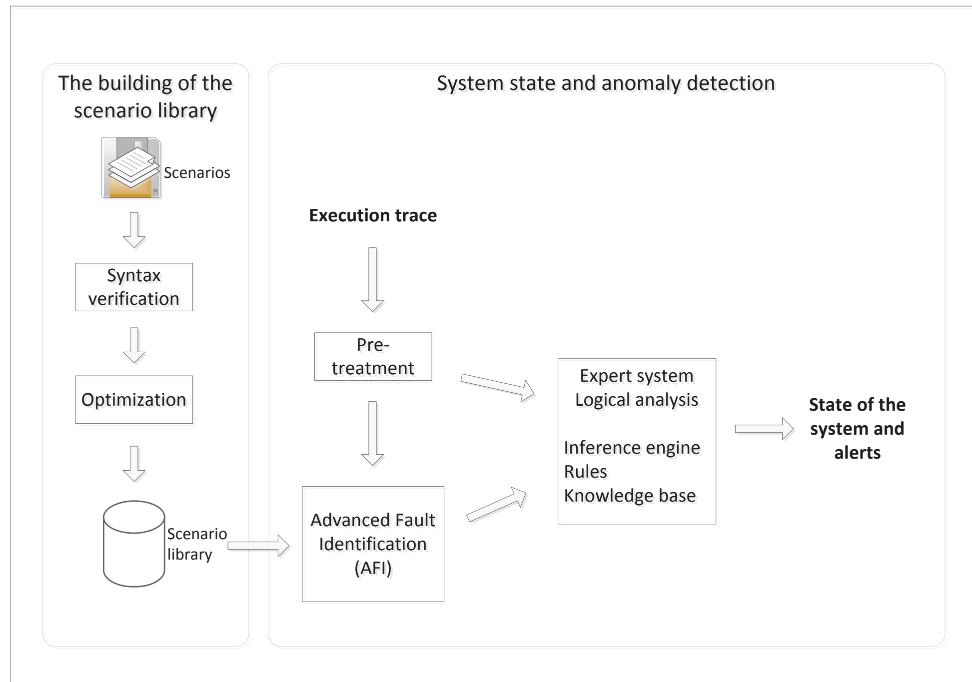

Figure 8: The creation of a scenario library and trace scanning.

This technology is still a prototype that remains to be completed and tested using other Linux resources such as network, memory management, etc. The knowledge base needs to be fed with additional rules that capture some of the expertise of an expert in cyber-security. Other sources of data could be added as inputs to the expert system (such as Snort's output [36]). The use of ontologies and abstracted low-level software behaviours (previous section) could also contribute to improve the efficiency of the system.

This technology and associated documentation can be found at this web site [2].

3.3 Other complementary studies

Other techniques were studied in the Poly-Tracing Project to identify how LTTng execution traces could be exploited to improve the OS3. A brief description of each study is presented in this section.

Tracing and analysing the system calls

Signature-based detection systems have limited capabilities for detecting undesired software behaviours and states during operations. Anomaly-based detection mechanisms would be a good complementary solution if they did not have the tendency to generate many false alarms due to the difficulty of building normalcy models.

One of the most popular anomaly-based detection techniques involves the utilization of a Hidden Markov Model (HMM) [37, 38] as a means of modelling normal software behaviours at the level of system calls—the interface between the user and kernel spaces of the Linux operating system. These techniques require the HMM to be *trained* using system call trace events that reflect the *normal* utilization of the system. The training is however very long for this type of model, often requiring huge execution traces. This makes this type of model less attractive despite its very good accuracy in detecting cyber-attacks and faults.

A first preliminary study was conducted on the utilization of HMM in the context of OS3. An improved HMM was proposed, which reduces the training time of HMMs [28]. The developed technique uses the concept of frequent common patterns. The models are built based on extracting the largest n-grams (patterns) in the execution traces instead of taking every trace event as-is. The case study shows that the training time could be reduced by 32% to 48%, compared to the original algorithm. As expected, the use of n-grams resulted in less accuracy than the original HMM algorithm. However, preliminary studies also showed that this loss of accuracy could potentially be reduced by improving the trace coverage during the construction of the models. Further studies must be conducted to evaluate the real gain of using the n-gram approach.

Another preliminary study was conducted to identify how false positive rates could be reduced when using anomaly-based detection algorithms [29]. The fact that high false positive rates are generated despite the type of detection algorithm could suggest that the problem is not only caused by the application of algorithms but also by the characteristics of the dataset. In this study, the authors have investigated whether the removal of contiguous system calls (a form of trace generalization) impacts the accuracy (false positive/true positive rates) and the performance of an anomaly detection algorithm. They have used a sliding window algorithm and the HMM on a) execution traces that do not contain contiguous repetitions of system calls, and b) on actual traces with repetitions of system calls.

The results show that training the sliding window on traces without contiguous repetitions of system calls lower the average false positive rate by 31%. The true positive rate however remains the same. The removal of contiguous repetitions reduced the size of the traces by approximately 40%, and reduced the time to preprocess the traces by 10% to 50% [29]. Further improvements of the accuracy could be made if the arguments of the system calls were considered in the detection analysis.

Multi-level tracing for complex debugging and OS3

A study was conducted to address the problem of the utilization of the LTTng software tracer by software developers and system administrators for the identification of software faults in information systems (virtualized or not) [4]. It produced two main contributions and many improvements to the LTTng tracer.

The first contribution is a mechanism that allows the synchronization of execution traces that originate from different software levels on one information system (e.g. kernel space, user space, hypervisor). The goal was to timestamp all trace events originating from different software levels of the system using the *same timeframe*. A review of all the different options that are available for trace event synchronization and timestamping on one system allowed the identification of the most efficient mechanism, which also minimizes the impacts on the system's performance. As all

trace events are using the same timeframe, including the events that are generated within a virtualized environment, they can be studied altogether. Moreover, this development contributed to significantly improve the performance of the LTTng tracer (in user space).

The second contribution simplified the utilization of LTTng by software developers and system administrators. There are cases where the use of traditional tools, such as “top” [30], “ftrace” [31], “perf” [10], or “systemtap” [32] for example, cannot help solve complex bugs in the system. Components of the LTTng tracer were improved and a new tool called “lttngtop” was developed. This tool works like the “top” tool except that it reads execution traces and provides the administrator with the high-level information that is needed to solve complex low-level software problems. The impact of using the tool on the performances of the system are similar to those of “top”, which is very low.

System health monitoring and proactive response activation

The results of this preliminary study [33] will be summarized in another report.

The use of redundancy and diversity in software architectures

The results of this preliminary study [34, 35] will be summarized in another report.

4 Concluding remarks and recommendations

This report summarizes a number of studies and developments that were made in the context of the Poly-Tracing Project. The main goal of the project was to develop advanced tools that can be used to analyse any software component of the Linux operating system and detect the presence of undesired behaviours and states while the system is used during operations.

The LTTng software tracer represents the central development from which all other research and development was made in the project. LTTng allows the online tracing of any software components that run in either the user or kernel spaces of the Linux operating system (or hypervisors). The tracer gives the users full control over both the focus and granularity of the tracing process, which define the content of the execution traces. These can be recorded locally on the hard drive of the traced computer, or remotely on another computer through standard network links. The generation of synchronized execution traces from distributed systems as well as from virtualized systems was made possible. LTTng produces data having advanced characteristics that could not be generated with traditional similar tools.

Other types of improvements were made in the Poly-Tracing Project. The utilization of LTTng to trace legacy software applications was made possible. Using the Umple framework, it is now possible to instrument these programs through a complex process, which involves the insertion of LTTng probes at the model level instead of at the source code level.

A new descriptive programming language was also developed for the definition of scenarios, which represent user-defined software behaviours. A specialized search engine can be used to scan LTTng execution traces, searching for the presence of these pre-defined scenarios. An expert system was then developed for the logical analysis of execution traces, providing the state of the system at any time.

Anomaly-based detection techniques using the content of LTTng execution traces were also studied. Some improvements were made regarding the training of Hidden Markov Models (HMMs), which are known to give good results in this type of classification problem. As anomaly-based detection algorithms often generate a lot of false positives, new ways to reduce these using HMM models were also studied. These preliminary studies gave promising results, showing that they should be pushed further.

The ease of utilization of LTTng by software developers and system administrators was also addressed. A new Linux-like tool “lttngtop” was developed to allow the capture and dynamic tabular representation of the low-level information contained in execution traces. This tool complements other traditional Linux tools such as “top”, “ftrace”, “systemtap” and other specialized debugging tools. The new information that is generated by LTTng allows the discovery of complex bugs, which was not possible using traditional tools.

The capture and storage on disk of huge execution traces represented a problem for long LTTng tracing sessions. A new storage mechanism was developed to solve this problem. Instead of saving trace events, the mechanism saves the changes of states of the system’s resources over time. These are saved in a new specialized state history database. Using the content of this

database, it is possible to reconstitute the evolution of the state of a specific resource over time, which could help forensic investigations.

The Eclipse-based Tracing and Monitoring Framework (TMF), which embeds LTTng, has shown new interesting possibilities that could be successfully used at runtime. TMF exploits the LTTng API to provide the users with full control over tracing sessions. It includes a visualizer that can zoom in/out of huge execution traces as they are generated. The selection of one or many trace events in the viewer automatically brings up all the relevant information regarding selected events such as process IDs, start/end times, states, etc.

TMF also offers the possibility to simultaneously execute many different algorithms in its plug-in environment. Examples are algorithms for trace analysis, anomaly detection, scenario detection, trace event synchronization, specialized visualization, etc. This capability opens the door for the development of a future cyber-dashboard that could provide system administrators with all the graphical utilities and controls they need for online decision making when the system is attacked. For example, it could allow/ease:

- the online identification of the specific software components that should be traced, and the activation of related LTTng probes according to the granularity that is needed in the generated execution traces;
- the online selection of specific detection algorithms for specific analysis purposes;
- the online decision to transfer execution traces from the traced system to one or many remote systems for a) more in-depth detection analysis and b) to avoid performance impacts of these algorithms on the traced systems;
- the online selection of the system's resources for which the states should be captured and saved on disk for further post-mortem analysis; and
- the selection and organization of results from one or many trace analyses, using specialized graphical representations that ease their understanding.

Technologies that were developed in the Poly-Tracing Project are mature enough to support all these functionalities. One of the great successes of the Poly-Tracing Project is the fact that LTTng is now used by many major enterprises from all around the world, making its further development and debugging rates very fast. The tracer is also part of Red Hat Enterprise Linux 7, which is used on many governmental and industrial critical information systems.

One of the future efforts should be to investigate the possibility of harmonizing the utilization of LTTng with other selected cyber-security monitoring systems, such as Intrusion Detection Systems (IDS), antivirus, log analysis systems, etc. The following example illustrates the advantage of doing so. The sudden detection of an anomaly by a network-based IDS could automatically trigger the activation of specific LTTng probes within appropriate software components. These probes would then produce data having the best focus and granularity for the subsequent analysis of the system for the assessment of the anomaly. Providing the best data to detection analysis algorithms will certainly contribute to generating more precise results regarding the state of the system. This cause-and-effect control of LTTng is called *feedback-directed tracing*.

References/Bibliography

[1] M. Couture, A. Hamou-Lhadj, M. Dagenais and A. Goel. “Online surveillance of computerized systems—Analysis of current and future needs”. In Proceedings of the NATO Symposium on Information Assurance and Cyber Defence, NATO unclassified publications, RTO SET-183/IST-112, Quebec city, Qc, paper 25, 2012.

[2] The Poly-Tracing Project. Web site (last accessed July, 2015).
http://dmct.dorsal.polymtl.ca

[3] LTTng. Web sites (last accessed July, 2015).
http://lttng.org
http://wiki.eclipse.org/index.php/Linux_Tools_Project/LTTng/User_Guide
http://dmct.dorsal.polymtl.ca/node/11

[4] J. Desfossez. “Résolution de problème par suivi de métriques dans les systèmes virtualisés”. Master thesis, École Polytechnique de Montréal, Qc, 2011.

[5] M. Couture, M. Dagenais, D. Toupin, R. Charpentier, G. Matni, M. Desnoyers, P.-M. Fournier. “Monitoring and tracing of critical software systems - State of the work and project definition”. DRDC – Valcartier Research Centre, Technical Memorendum, TM 2008-144, 2008.

[6] K. Yaghmour. “Analyse de performance et caractérisation de comportement à l’aide d’enregistrement d’événements noyau”. Master thesis, École Polytechnique de Montréal, Qc, 2001.

[7] The Tracing Monitoring Framework (TMF). Web sites (last accessed July, 2015).
http://dmct.dorsal.polymtl.ca/node/27
http://projects.eclipse.org/proposals/trace-compass

[8] The Common Trace Format (CTF). Web sites (last accessed July, 2015).
http://www.efficios.com/ctf
http://git.efficios.com/?p=ctf.git

[9] D. Thibault. “LTTng: The Linux Trace Toolkit Next Generation—A Comprehensive User’s Guide.” DRDC – Valcartier Research Centre. Submitted to DRDC Editorial Office. Another source of information can be found at the following Web site (last accessed July, 2015).
http://lttng.org/docs/#doc-getting-started

[10] Perf. Web site (last accessed July, 2015).
https://perf.wiki.kernel.org/index.php/Main_Page

[11] UMPLE. Web sites (last accessed July, 2015).
http://cruise.eecs.uottawa.ca/umple/GettingStarted.html
https://code.google.com/p/umple/wiki/Tutorials

[12] H. Aljamaan, T. Lethbridge, O. Badreddin, G. Guest, and A. Forward. “Specifying Trace Directives for UML Attributes and State Machine.” In the 2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Lisbon, Portugal, Jan. 2014.

[13] M. A. Garzón Torre. “Reverse Engineering Object-Oriented Systems into Umple: An Incremental and Rule-Based Approach.” Ph.D. thesis, University of Ottawa, 2015.

[14] UMPLE online. Web sites (last accessed July, 2015).
http://cruise.eecs.uottawa.ca/umpleonline
http://dmct.dorsal.polymtl.ca/node/16

[15] M. Couture, F. Lajeunesse-Robert, F. Prenoveau, M. Dagenais, B. Ktari. “The Tracing, monitoring and analysis of distributed multi-core systems - Selected feasibility studies”. DRDC – Valcartier Research Centre, Technical Report, TR 2008-300, 2009.

[16 A. Duda, G. Harrus, Y. Haddad, and G. Bernard. “Estimating global time in distributed systems.” Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, vol. 18, 1987.

[17] B. Poirier. “Synchronisation de traces distribuées à l'aide d'événements de bas niveau.” Master thesis, École Polytechnique de Montréal, Apr. 2010. Another source of information can be found at the following Web site (last accessed July, 2015).
http://dmct.dorsal.polymtl.ca/node/12

[18] B. Poirier, R. Roy, and M. Dagenais. “Accurate offline synchronization of distributed traces using kernel-level events.” *Operating Systems Review*, 2010.

[19] M. Jabbarifar, A. S. Sendi, A. Sadighian, N. E. Jivan, and M. Dagenais. “A reliable and efficient time synchronization protocol for heterogeneous wireless sensor network.” *Wireless Sensor Network*, Aug. 2011.

[20] M. Jabbarifar, M. Dagenais. “On Line Trace Synchronization for Large Scale Distributed Systems." Ph.D. Thesis, École Polytechnique de Montréal, 2013. Another source of information can be found at the following Web site (last accessed July, 2015).
http://dmct.dorsal.polymtl.ca/node/12

[21] A. Montplaisir. “Stockage sur disque pour accès rapide d'attributs avec intervalles de temps.” Master thesis, École Polytechnique de Montréal, Dec. 2011. Another source of information can be found at the following Web site (last accessed July, 2015).
http://dmct.dorsal.polymtl.ca/node/25

[22] W. Fadel. “Techniques for the Abstraction of System Call Traces to Facilitate the Understanding of the Behavioural Aspects of the Linux Kernel.” Master thesis, Concordia University, 2012.

[23] Trace abstraction, analysis and correlation. Web site (last accessed July, 2015).
http://dmct.dorsal.polymtl.ca/node/13

[24] H. Pirzadeh and A. Hamou-Lhadj. “A Software Behaviour Analysis Framework Based on the Human Perception System.” In Proceedings of the 33rd International Conference on Software Engineering (ICSE'12 NIER Track), pp. 948–951, 2011.

[25] H. Pirzadeh, A. Hamou-Lhadj. “A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension.” In Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '11), 2011.

[26] H. Waly. “Automated fault identification.” Master thesis, Laval University, 2011. Another source of information can be found at the following Web sites (last accessed July, 2015).
https://svn.fsg.ulaval.ca/svn-pub/afi-20
http://dmct.dorsal.polymtl.ca/node/28

[27] R. Zribi. “Approche logique pour l’analyse de traces d’exécution.” Master thesis, Laval University, 2013.

[28] A. Sultana, A. Hamou-Lhadj, S. Murtaza, M. Couture. “An Improved Hidden Markov Model for Anomaly Detection Using Frequent Common Patterns.” In Proceedings of the IEEE International Conference on Communications, The Communication and Information Systems Security Symposium, pp. 1113 - 1117, 2012.

[29] S. S. Murtaza, A. Sultana, A. Hamou-Lhadj, M. Couture. “On the Comparison of User Space and Kernel Space Traces in Identification of Software Anomalies.” Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12), pp. 127–136, 2012.

[30] top. Web site (last accessed July, 2015).
https://en.wikipedia.org/wiki/Top_(software)

[31] ftrace. Web sites (last accessed July, 2015).
http://elinux.org/Ftrace
https://en.wikipedia.org/wiki/Ftrace

[32] systemtap. Web sites (last accessed July, 2015).
https://en.wikipedia.org/wiki/SystemTap
https://sourceware.org/systemtap/documentation.html

[33] A. Shameli-Sendi, M. Dagenais, “System Health Monitoring and Proactive Response Activation.” Ph.D. Thesis, École Polytechnique de Montréal, 2013. Another source of information can be found at the following Web site (last accessed July, 2015).
http://dmct.dorsal.polymtl.ca/node/15

[34] R. Khoury, A. Hamou-Lhadj and M. Couture. “Towards a Formal Framework for Evaluating the Effectiveness of Diversity when Applied to Security.” In Proceedings of the IEEE Symposium on Computational Intelligence for Security and Defence Applications (CISDA'12), IEEE Computational Intelligence Society, pp. 1–7, 2012.

[35] R. Khoury, A. Hamou-Lhadj, M. Couture, and R. Charpentier, “Diversity through N-Version Programming: Current State, Challenges and Recommendations.” International Journal of Information Technology and Computer Science (IJITCS) 4(2), pp.56–64, 2012.

[36] snort. Web sites (last accessed July, 2015).
https://www.snort.org

[37] C. Warrender, S. Forrest, B. Pearlmutter, “Detecting intrusion using system calls: alternative data models” Proceedings of the 1999 IEEE Symposium on Security and Privacy, IEEE Computer Society, pp. 133–145, 1999.

[38] B. Gao, H.Y. Ma, Y.H. Yang, “HMMS based on anomaly intrusion detection method” Proceedings of the First International Conference on Machine Learning and Cybernetics, Beijing, 2002.

List of symbols/abbreviations/acronyms/initialisms

AFI	Advanced Fault Identification
ARM	*originally* Advanced RISC Machine
AST	Abstract Syntax Tree
AV	Antivirus
CPU	Central Processing Unit
CTF	Common Trace Format
DMRS	Director Maritime Requirements Sea. The title "Director Maritime Requirements Sea" (DMRS) was replaced by "Director Naval Requirements" (DNR) on March 28, 2013.
DND	Department of National Defence
DRDC	Defence Research and Development Canada
EMF	Eclipse Modeling Framework
HIDS	Host-based Intrusion Detection System
HMM	Hidden Markov Model
ID	Identifier
IDS	Intrusion Detection System
LAN	Local Area Network
LTT	Linux Trace Toolkit
LTTng	Linux Trace Toolkit next generation (lttng.org)
LTTV	Linux Trace Toolkit Viewer
MIPS	*originally* Microprocessor without Interlocked Pipeline Stages
MOTL	Model Oriented Tracing Language
MS	Microsoft
NIC	Network Interface Controller
NIDS	Network-based Intrusion Detection System
NSERC	Natural Sciences and Engineering Research Council (of Canada)
OMAP	Open Multimedia Applications Platform
OS3	Online Surveillance of Software Systems
PASS-RTM	Platform-to-Assembly Secured Systems – Real-Time Monitoring
PHP	PHP: Hypertext Preprocessor (*originally* Personal Home Page)
R&D	Research & Development
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computing
SQL	Structured Query Language
TCP	Transmission Control Protocol
TMF	Tracing and Monitoring Framework
TRL	Technology Readiness Level
UDP	User Datagram Protocol
VM	Virtual Machine

DOCUMENT CONTROL DATA

(Security markings for the title, abstract and indexing annotation must be entered when the document is Classified or Designated)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g., Centre sponsoring a contractor's report, or tasking agency, are entered in Section 8.)

   DRDC – Valcartier Research Centre
   Defence Research and Development Canada
   2459 Route de la Bravoure
   Quebec (Quebec) G3J 1X5
   Canada

2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.)

   UNCLASSIFIED

2b. CONTROLLED GOODS

   (NON-CONTROLLED GOODS)
   DMC A
   REVIEW: GCEC DEC 2013

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

   Advanced Linux tracing technologies for Online Surveillance of Software Systems

4. AUTHORS (last name, followed by initials – ranks, titles, etc., not to be used)

   Couture, M.

5. DATE OF PUBLICATION (Month and year of publication of document.)

   October 2015

6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)

   31

6b. NO. OF REFS (Total cited in document.)

   38

7. DESCRIPTIVE NOTES (The category of the document, e.g., technical report, technical note or memorandum. If appropriate, enter the type of report, e.g., interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

   Scientific Report

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

   DRDC – Valcartier Research Centre
   Defence Research and Development Canada
   2459 route de la Bravoure
   Quebec (Quebec) G3J 1X5
   Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)

9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)

   DRDC-RDDC-2015-R211

10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

   Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

   Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

This scientific report provides a description of the concepts and technologies that were developed in the Poly-Tracing Project. The main product that was made available at the end of this four-year project is a new software tracer called Linux Trace Toolkit next generation (LTTng). LTTng actually represents the centre of gravity around which much more applied research and development took place in the project.

As shown in this document, the technologies that were produced allow the exploitation of the LTTng tracer and its output for two main purposes: a- solving today’s increasingly complex software bugs and b- improving the detection of anomalies within live information systems. This new technology will enable the development of a set of new tools to help detect the presence of malware and malicious activity within information systems during operations.

Ce rapport scientifique donne une description des concepts et technologies qui ont été développés dans le cadre du projet Poly-Tracing. Le principal produit qui a été rendu disponible à la fin du projet de quatre ans est un nouveau traceur logiciel appelé « Linux Trace Toolkit next generation (LTTng) ». LTTng représente en fait le centre de gravité autour duquel plusieurs autres travaux de recherche et développement ont pris place dans le projet.

Tel que montré dans ce document, les technologies qui ont été produites permettent l’exploitation du traceur LTTng et de son extrant pour deux principales raisons : a- le débogage logiciel qui est de plus en plus complexe de nos jours et b- l’amélioration de la détection en ligne d’anomalies dans les systèmes d’information. Cette nouvelle technologie va permettre de développer un ensemble d’outils pour aider la détection de la présence de logiciels et d’activités malicieux dans les systèmes d’information pendant les opérations.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g., Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Software tracer, LTTng, online surveillance, online system analysis, cyber, malware