Relational access to Unix kernel data structures

Marios Fragkoulis
Athens University of Economics and Business
mfg@aueb.gr

Diomidis Spinellis
Athens University of Economics and Business
dds@aueb.gr

Panos Louridas
Athens University of Economics and Business
louridas@aueb.gr

Angelos Bilas
University of Crete and FORTH-ICS
bilas@ics.forth.gr

Abstract

State of the art kernel diagnostic tools like DTrace and Systemtap provide a procedural interface for expressing analysis tasks. We argue that a relational interface to kernel data structures can offer complementary benefits for kernel diagnostics.

This work contributes a method and an implementation for mapping a kernel's data structures to a relational interface. The Pico COllections Query Library (PiCO QL) Linux kernel module uses a domain specific language to define a relational representation of accessible Linux kernel data structures, a parser to analyze the definitions, and a compiler to implement an SQL interface to the data structures. It then evaluates queries written in SQL against the kernel's data structures. PiCO QL queries are interactive and type safe. Unlike SystemTap and DTrace, PiCO QL is less intrusive because it does not require kernel instrumentation; instead it hooks to existing kernel data structures through the module's source code. PiCO QL imposes no overhead when idle and needs only access to the kernel data structures that contain relevant information for answering the input queries.

We demonstrate PiCO QL's usefulness by presenting Linux kernel queries that provide meaningful custom views of system resources and pinpoint issues, such as security vulnerabilities and performance problems.

Categories and Subject Descriptors D2.5 [*Testing and Debugging*]: Diagnostics

Keywords Unix, kernel, diagnostics, SQL

1. Introduction

Kernel diagnostic tools like DTrace [7] and SystemTap [27] provide a procedural interface for diagnosing system issues. We argue that a relational interface can offer complementary advantages; this is our motivation behind PiCO QL. Various imperative programming languages offer relational interfaces; some have become very popular [21, 25]. Previous work proposes relational interfaces in place of imperative ones for performing validation tasks at the operating system level [14, 29]. Windows-based operating systems provide the Windows Management Instrumentation (WMI) infrastructure [33] and the WMI Query Language (WQL), which adopts a relational syntax, to provide access to management data and operations.

Extending operating system kernels with high level tools [4, 7] is a growing trend. PiCO QL is such a tool used for performing kernel diagnostic actions. Our work contributes:

- a method for creating an extensible relational model of Unix kernel data structures (Sections 2.1 and 2.2),
- a method to provide relational query evaluation on Linux kernel data structures (Section 2.3), and
- a diagnostic tool to extract relational views of the kernel's state at runtime (Section 3). Each view is a custom image of the kernel's state defined in the form of an SQL query.

A Unix kernel running PiCO QL provides dynamic analysis of accessible data structures with the following features.

SQL SELECT queries: Queries conform to the SQL92 standard [8].

Type safety: Queries perform checks to ensure data structure types are used in a safe manner. PiCO QL does not affect the execution of kernel code, because it does not require its modification.

Low performance impact: PiCO QL's presence in the Unix kernel does not affect system performance. In addition, query execution consumes only the necessary kernel resources, that is, data structures whose relational representations appear in the query. Thus, PiCO QL minimizes its impact on kernel operation. Section 4.2 presents a quantitative evaluation of this impact.

Relational views: To reuse queries efficiently and store important queries, standard relational non-materialized

EuroSys '14, April 13–16, 2014, Amsterdam, The Netherlands.

http://dx.doi.org/10.1145/2592798.2592802

views can be defined in the PiCO QL Domain Specific Language (DSL).

Unix kernel data structures include C structs, linked lists, arrays, and unions to name but a few [2, 6]. In the context of this paper, and unless otherwise noted, the term data structure refers in general to Unix kernel data structures. Although PiCO QL's implementation targets the Linux kernel, the relational interface is equally applicable to other versions of Unix featuring loadable kernel modules and the `/proc` file system.

We evaluate our tool by presenting SQL queries, which diagnose interesting effects in two domains that concern a system's operation, namely security and performance. We find that our tool can identify security vulnerabilities and performance problems with less effort than other approaches. We present several use cases to describe our tool's contribution to the kernel diagnostics tool stack.

2. A relational interface to the Unix kernel data structures

Our method for mapping the kernel's data structures to a relational interface addresses two challenges: first, how to provide a relational representation for the kernel's data structures; second, how to evaluate SQL queries to these data structures. The key points of the design that address these challenges include:

- rules for providing a relational representation of the kernel's data structures,
- a domain specific language (DSL) for specifying relational representations and required information about data structures, and
- the leveraging of SQLite's virtual table hooks to evaluate SQL queries.

2.1 Relational representation of data structures

The problem of mapping a program's in-memory data structures to a relational representation, and vice versa, has been studied thoroughly in the literature [28]. We address this problem from a different angle: we provide a relational interface to data structures without storing them in a relational database management system; the issue at hand is not the transformation of data from procedural code data structures to relational structures, but the representation of the same data in different models. In other words, we do not transform the data model used for procedural programming; instead we want to provide a relational view on top of it. Issues that emerge in the Object-Relational mapping problem are not relevant to our work.

To provide a relational representation we must solve the representation mismatch between relations and procedural programming data structures. Relations consist of a set of columns that host scalar values, while data structures form graphs of arbitrary structure.

Our method defines three entities: data structures, data structure associations, and virtual relational tables. Specifically, it provides a relational representation of data structures and data structure associations in the form of virtual relational tables. Data structures can be unary instances or containers grouping multiple instances.

Our method is expressive; it can represent *has-a* associations, *many-to-many* associations, and object-oriented features like inheritance and polymorphism [10]. In this paper we exemplify the representation of *has-a* associations, since this is the widespread type of associations in the kernel.

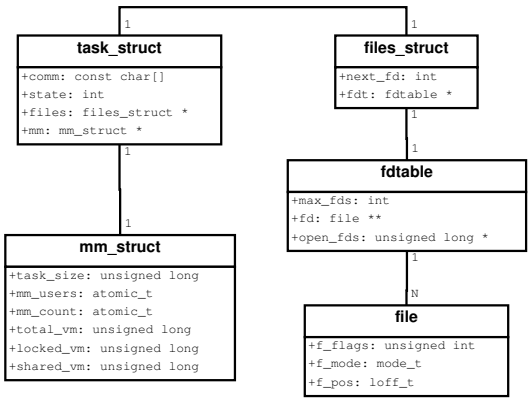
2.1.1 *Has-a* associations

Has-a associations are of two types: *has-one* and *has-many*. To represent data structures as first normal form relations, we normalize *has-a* associations between data structures, that is we define a separate relational representation for them. For instance Figure 1(a) shows a simplified kernel data structure model for the Linux kernel's files, processes, and virtual memory. Figure 1(b) shows the respective virtual table schema. There, a process's associated virtual memory structure has been represented in an associated table. The same applies for a process's open files, a *has-many* association. Notably, this normalization process is only required for *has-many* associations. In the same schema, the associated *files_struct* has been included within the process's representation. In addition, the structure *fdtable* has been included in its associated *files_struct* and it is also part of the process representation; each member of *fdtable* and *files_struct* occupies a column in Process_VT. By allowing to represent a *has-one* association as a separate table or inside the containing instance's table, the relational representation flexibly expands or folds to meet representation objectives.

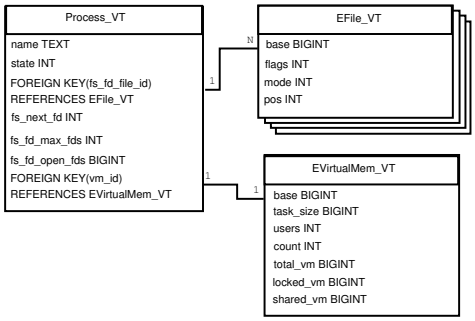
Combining virtual tables in queries is achieved through join operations. In table Process_VT foreign key column *fs_fd_file_id* identifies the set of files that a process retains open. A process's open file information can be retrieved by specifying in a query a join to the file table (EFile_VT). By specifying a join to the file table, an instantiation happens. The instantiation of the file table is process specific; it contains the open files of a specific process only. For another process another instantiation would be created. Thus, multiple potential instances of *EFile_VT* exist implicitly, as Figure 1(b) shows.

2.2 Domain Specific Language design

Our method's DSL includes *struct view* definitions that describe a virtual table's columns, *virtual table* definitions that link a *struct view* definition to a data structure type, *lock directive* definitions to leverage existing locking facilities when querying data structures, and standard *relational view* definitions to provide easy query reuse.



(a) Linux model



(b) Virtual table schema

Figure 1. *Has-one* and *has-many* associations between data structures are normalized or denormalized in the virtual relational schema.

Listing 1. Struct view definition includes the mapping of *has-a* relationships (Figure 1) in the virtual table schema.

```
CREATE STRUCT VIEW Process_SV (
 name TEXT FROM comm,
 state  INT FROM state,
 FOREIGN KEY(fs_fd_file_id) FROM files_fdtable (
    tuple_iter −>files) REFERENCES EFile_VT POINTER,
 fs_next_fd  INT FROM files−>next_fd,
 fs_fd_max_fds  INT FROM files_fdtable (
    tuple_iter −>files)−>max_fds,
 fs_fd_open_fds  BIGINT FROM files_fdtable (
    tuple_iter −>files)−>open_fds,
 FOREIGN KEY(vm_id) FROM mm
   REFERENCES EVirtualMem_VT POINTER)
```

2.2.1 Struct view definition

Struct view definitions (Listings 1, 2, and 3) describe the columns of a virtual table. They resemble relational table definitions. Struct view definitions include the struct view's name and its attributes, where the attribute information includes the name, data type, and access path, that is, how the column value will be retrieved from the data structure.

The foreign key column definition (Listing 1) supports relationships between virtual tables, which in turn represent a *has-a* association between the underlying data structures. A foreign key specification resembles its relational counterpart except that no matching column of the referenced table is specified. This is because the foreign key column matches against the *base* column of the referenced virtual table (see Section 2.3 for the use of the *base* column).

Listing 2. Struct view definition includes the mapping of *has-a* inclusion (Figure 1(b)) in the virtual table schema.

```
CREATE STRUCT VIEW FilesStruct_SV (
 next_fd  INT FROM next_fd,
 INCLUDES STRUCT VIEW Fdtable_SV FROM
   files_fdtable ( tuple_iter ))
```

The INCLUDES STRUCT VIEW directive allows including a relational representation into another. Section 2.1.1 explains how structures *files_struct* and *fdtable* have been represented inline in a process's relational representation (Process_SV). Listing 1 shows the respective DSL definition and Figure 1(b) shows the respective virtual table schema. Instead of mapping structures' *fdtable* and *files_struct* fields manually to Process_SV columns, one can use the INCLUDES STRUCT VIEW directive to reuse struct view definitions. Listing 2 shows how the relational representation of *fdtable* (Fdtable_SV) defined elsewhere can be included to structure's *files_struct* relational representation (FilesStruct_SV). Mapping an associated *fdtable* to another data structure's relational representation besides *files_struct*'s requires only a similar INCLUDES STRUCT VIEW directive to the one presented in Listing 2.

Listing 1 demonstrates some important characteristics of the DSL. First, the special identifier POINTER is used to denote that the column maps to a pointer type. It is required to generate code that meets mapped data structures' types. Second, the extensive use of path expressions [11] in access paths is a core aspect of the mapping method and of the query evaluation technique. Path expressions identify a data structure by specifying how to navigate to it in a graph of data structures. Third, accessing kernel information in a safe manner often requires additional constructs, such as calling kernel functions and macros. For instance, the file descriptor table should be accessed through kernel function `files_fdtable()` in order to secure the *files_struct* pointer dereference. The DSL accepts calls to kernel functions and macros and defines a reserved keyword, *tuple_iter* to allow references to a virtual table's tuple iterator. *tuple_iter* provides a uniform interface to each of a virtual table's tuples regardless of the virtual table's tuple set size. For instance, since a process has one *files_struct*, the *files_struct* virtual table has tuple set size of one and *tuple_iter* refers to this one tuple (Listing 2).

From our experience in querying data structures, it is also very useful to be able to write a block of code relevant to an access path in order to hold and release locks or define conditions. For this purpose, a DSL file starts with boilerplate C code. This comprises include directives, macro definitions and function definitions. Functions and macros can then be called from an access path context as shown in Listing 3.

Listing 3. Struct view definition can contain calls to custom functions in column access paths.

```
long check_kvm(struct file *f) {
  if ((!strcmp(f->f_path.dentry->d_name.name,
               "kvm-vm")) &&
      (f->f_owner.uid == 0) &&
      (f->f_owner.euid == 0))
    return (long)f->private_data;
  return 0;
}
$
 ...
CREATE STRUCT VIEW File_SV (
  ...
  FOREIGN KEY(kvm_id) FROM check_kvm(tuple_iter)
    REFERENCES EKVM_VT POINTER) )
```

Listing 4. Virtual table definition links to a *struct view* one.

```
CREATE VIRTUAL TABLE Process_VT
USING STRUCT VIEW Process_SV
WITH REGISTERED C NAME processes
WITH REGISTERED C TYPE struct task_struct *
USING LOOP list_for_each_entry_rcu( tuple_iter ,
                          &base->tasks, tasks)
USING LOCK RCU
```

The defined function `check_kvm()` accepts an open file and tests whether the file belongs to a Kernel-based Virtual Machine (KVM) [16] virtual machine (VM) instance. KVM is manageable through file descriptors and a set of *ioctl* calls that operate on open file handles. Through this interface authorized users can manipulate VM instances and allocated virtual CPUs. Behind the scenes, a data structure modelling an open file – the usual *struct file* – maps back to a data structure modelling a KVM VM or virtual CPU instance depending on the *ioctl* call. We leverage this mapping to hook to KVM related data structures. With `check_kvm()` we ensure that the file is indeed an interface to a KVM VM or virtual CPU by checking the file's name and file ownership against the root user's account id.

2.2.2 Virtual table definition

Virtual table definitions (Listing 4) link a data structure to its relational representation. They carry the virtual table's name and information about the kernel data structure it represents. Data structure information includes an identifier (C NAME) and a type (C TYPE); the identifier introduces a convention required for mapping a Linux kernel data structure to its virtual table representation, and the type must agree with the data structure's programming language type. The identifier is omitted if the data structure is nested in another data structure. A virtual table definition always links to a struct view definition through the USING STRUCT VIEW syntax.

An interface is required for traversing data structures like arrays and linked lists to execute queries. The USING LOOP directive serves this purpose. In our approach, a container/iterator-based uniform abstraction is utilized that wraps diverse types of data structures. Listing 4 makes use

Listing 5. Virtual table definition includes a customized loop variant for traversing file array.

```
#define EFile_VT_decl(X) struct file *X; \
        int bit = 0
#define EFile_VT_begin(X, Y, Z) \
        (X) = (Y)[(Z)]
#define EFile_VT_advance(X, Y, Z) \
        EFile_VT_begin(X,Y,Z)

CREATE VIRTUAL TABLE EFile_VT
USING STRUCT VIEW File_SV
WITH REGISTERED C TYPE struct fdtable:struct file*
USING LOOP for (
        EFile_VT_begin( tuple_iter , base->fd,
                (bit = find_first_bit (
                (unsigned long *)base->open_fds,
                base->max_fds)));
        bit < base->max_fds;
        EFile_VT_advance( tuple_iter , base->fd,
                (bit = find_next_bit (
                (unsigned long *)base->open_fds,
                base->max_fds, bit + 1))))
```

Listing 6. Lock/unlock directive definition includes lock-/unlock calls to a synchronization primitive's interface.

```
CREATE LOCK RCU
HOLD WITH rcu_read_lock()
RELEASE WITH rcu_read_unlock()
```

of a Linux kernel built-in macro for traversing linked lists. If such a mechanism does not exist, user defined macros in the first part of a DSL description may customize the loop variant by means of iterator methods (`declare, begin, end, advance`) as is the case in Listing 5. The DSL parser will substitute references to `base`, which abstracts the data structure instantiation, with an appropriate variable instance.

To synchronize access with other execution paths, the USING LOCK directive selects a locking method from those defined in the DSL. Section 2.2.3 details lock directive definitions.

2.2.3 Lock directive definition

In-place querying requires locking data structures to ensure synchronization for concurrent accesses. Since our method requires only read access to kernel data structures, it primarily relies on synchronization primitives optimized for read access, where these are utilized. For the Linux kernel, the DSL's lock/unlock directives can be seen in Listing 6.

2.2.4 Standard relational views

To support recurring queries and minimize time-to-query in these cases relational non-materialized views can be defined in the DSL using the standard CREATE VIEW notation, as exemplified in Listing 7. The particular view defines an alias for queries that hook to KVM virtual machine instances in the system. A similar view definition wraps the representation of virtual CPU instances.

Listing 7. Relational view definition identifies KVM VM instances.

```
CREATE VIEW KVM_View AS
  SELECT P.name AS kvm_process_name, users AS kvm_users,
    P.inode_name AS kvm_inode_name, online_vcpus AS kvm_online_vcpus,
     stats_id  AS kvm_stats_id ,  online_vcpus_id  AS kvm_online_vcpus_id,
     tlbs_dirty  AS kvm_tlbs_dirty ,  pit_state_id  AS kvm_pit_state_id
  FROM Process_VT as P
  JOIN EFile_VT as F
  ON F.base = P.  fs_fd_file_id
  JOIN EKVM_VT AS KVM
  ON KVM.base = F.kvm_id;
```

Listing 8. Join query combines processes and associated virtual memory.

```
SELECT * FROM Process_VT
JOIN EVirtualMem_VT
ON EVirtualMem_VT.base = Process_VT.vm_id;
```

2.3 Query evaluation

In our method, the relational representation of a data structure comprises one or more virtual tables. Each virtual table in the representation enables access to some part of data structure using path expressions. For instance, Process_VT represents some fields of *task_struct*. Since *task_struct*'s *has-a* association with *mm_struct* has been modeled as a separate virtual table (EVirtualMem_VT), the latter provides access to the associated *mm_struct* fields. Field access is provided by path expressions according to the DSL specification.

Virtual tables may be combined in SQL queries by means of join operations (Listing 8). Data structures may span arbitrary levels of nesting. Although the nested data structure may be represented as one or more virtual table(s) in the relational interface, access to it is available through a parent data structure only. The virtual table representing the nested data structure (VT_n) can only be used in SQL queries combined with the virtual table representing a parent data structure (VT_p). For instance, one cannot select a process' associated virtual memory representation without first selecting the process. If such a query is input, it terminates with an error.

To allow querying of VT_ns a join is required to instantiate the VT_n. The join uses the column of the VT_p that refers to the nested data structure (similar to a foreign key) and the VT_n's *base* column, which acts as an internal identifier. When a join operation references the VT_n's *base* column it instantiates the VT_n by setting the foreign key column's value to the *base* column. This drives the new instantiation thereby performing the equivalent of a join operation: *for each value of the join attribute, that is the foreign key column, the operation finds the collection of tuples in each table participating in the join that contain that value*. In our case the join is essentially a precomputed one and, therefore, it has the cost of a pointer traversal. The base column acts as the activation interface of a VT_n and we guarantee type-safety by checking that the VT_n's specification is appropriate for representing the nested data structure.

Listing 9. Relational join query shows which processes have same files open.

```
SELECT P1.name, F1.inode_name, P2.name, F2.inode_name
FROM Process_VT AS P1
JOIN EFile_VT AS F1
ON F1.base = P1.  fs_fd_file_id  ,
Process_VT AS P2
JOIN EFile_VT AS F2
ON F2.base = P2.  fs_fd_file_id
WHERE P1.pid <> P2.pid
AND F1.path_mount = F2.path_mount
AND F1.path_dentry = F2. path_dentry
AND F1.inode_name NOT IN ('null','');
```

In addition to combining relational representations of associated data structures in an SQL query, joins may also be used to combine relational representations of unassociated data structures; this is implemented through a nested loop join [30, p. 542]. For example, consider a query to find out which processes have the same files open (Listing 9).

3. PICO QL implementation specifics

PiCO QL has been implemented as a loadable kernel module (LKM) for Linux.

The cornerstone of the PiCO QL implementation is a meta-programming technique devised:

- to create a relational representation of arbitrary data structures using a relational specification of these data structures and
- to generate C code for querying the data structures through their relational representation.

PiCO QL relies heavily on SQLite [1], an embeddable SQL database, and leverages SQLite's virtual table module [32] to support SQL queries to in-memory kernel data structures. We provide details on SQL support and describe briefly the integration of PiCO QL with Linux. Our kernel module uses the `/proc` file system for receiving queries and communicating query result sets. A high level interface that runs in user space is also available. Security is handled by implementing `/proc` callback functions that provide access control and synchronization is achieved by leveraging the kernel's locking utilities; we discuss consistency concerns in the context of our work. Finally, we present PiCO QL's maintainability in the course of the kernel's evolution.

3.1 Generative programming

Our solution to the problem of representing any data structure as a virtual table is based on a user-driven relational specification and a meta-programming technique, generative programming. Specifically, a parser analyzes specifications written in the PiCO QL DSL and kernel data structure information. Then a compiler generates virtual table definitions and C callback functions for SQL queries. These functions implement constraint evaluation and value retrieval for each

of a virtual table's columns. The generative programming component of PiCO QL was implemented in Ruby.

3.2 Virtual table implementation

The virtual table module of SQLite, which consists of a set of callback functions, allows querying of user-defined data sources. Once a module implementation is registered with SQLite, queries to virtual tables that belong to the module are resolved by calling the module's callback functions.

PiCO QL implements an SQLite virtual table module, that is those callback functions that specify a PiCO QL virtual table's behavior. These are: create, destroy, connect, disconnect, open, close, filter, column, plan, advance_cursor, and eof. The SQLite query engine calls these functions when performing a query on a PiCO QL virtual table. Hence queries are resolved by executing the implemented callback functions, which operate on the kernel data structures.

In query processing, PiCO QL and SQLite share responsibilities. PiCO QL controls query planning and carries out constraint and data management for each virtual table. The hook in the query planner ensures that the constraint referencing the base column has the highest priority in the constraint set for the VT_n and, therefore, the instantiation will happen prior to evaluating any real constraints. This is important for ensuring integrity in query processing. SQLite performs high level query evaluation and optimization [1, p. 360]. For the most part, query efficiency mirrors SQLite's query processing algorithms enhanced by simply following pointers in memory in the case of path expressions and join operations that provide a new virtual table instantiation.

3.3 SQL support and query optimization

PiCO QL supports all relational algebra operators as implemented by SQLite in [1], that is, the SELECT part of SQL92 excluding right outer joins and full outer joins. Queries expressed using the latter, however, may be rewritten with supported operators [1, p. 76]. For a right outer join, rearranging the order of tables in the join produces a left outer join. A full outer join may be transformed using compound queries. The query engine is a standard relational query engine.

Concerning query optimizations, PiCO QL benefits from SQLite's query rewrite optimizations, e.g. subquery flattening. The SQLite query optimizer [31] offers OR, BETWEEN, and LIKE optimizations in the WHERE clause, join optimizations, and order by optimizations associated with indices. However, these are not currently supported by PiCO QL because the PiCO QL index implementation is a future work plan. SQLite provides a virtual table API for indices and a command that scans all indices of a database where there might be a choice between two or more indices and gathers statistics on the appropriateness of those indices.

Providing relational views of data structures imposes one requirement to SQL queries. VT_ps have to be specified before VT_ns in the FROM clause. This stems from the implementation of SQLite's syntactic join evaluation and does not impose a limitation to query expressiveness.

3.4 Loadable module implementation for the Linux kernel

The process for implementing PiCO QL in the Linux kernel involved four steps. First we supplied a relational representation of the target data structures and data structure information through the PiCO QL DSL. Second, we executed the DSL parser to generate code based on the DSL description for querying the data structures. Third, we implemented the kernel module's initialization, permission, ioctl, open, input, output, close and exit routines in order to provide an interface to PiCO QL. The query library's involvement in this step is to register virtual tables representing kernel data structures and start the query library at the module's initialization routine. It then receives a query from the module's input buffer, places a result set into the module's output buffer and terminates the query library through the module's exit routine. Finally, we compiled PiCO QL LKM with the Linux kernel.

The main challenge in producing a loadable module was fitting SQLite to the Linux kernel. This included omitting floating point data types and operations, omitting support for threads, eliminating SQLite's file I/O activity, and substituting user-space library function calls with the corresponding ones available to kernel code.

Floating point support and thread support for SQLite can be controlled through compile flags. File I/O is unnecessary since PiCO QL stores nothing. SQLite's in-memory mode alleviates this complexity except for the journaling mechanism, which can be turned off with a pragma query following the establishment of a database connection. SQLite's flexible configuration removed significant roadblocks with minimal effort.

The C library facilities available within the Linux kernel are spartan. Some user space library function calls were matched through a macro definition to implemented kernel functions. This was the case for the memory allocation routines. A few other library function calls were substituted with dummy macro definitions and a couple were implemented in a straightforward manner by combining kernel functions.

3.5 Query interfaces

A `/proc` file attached to the PiCO QL module is the main interface for inputting queries and collecting result sets. PiCO QL's `/proc` output is produced in a number of ways including the standard Unix header-less column format. In addition to the `/proc` file, an HTTP-based query interface was produced through SWILL [17], a library that adds a web interface to C programs. Each web page served by SWILL is implemented by a C function. For a query interface three such functions are essential, one to input queries, one to output query results, and one to display errors.

3.6 Security

The kernel provides facilities for securing a module's use and access. First, loading and unloading a kernel module requires elevated privileges. Second, the `/proc` file system, PiCO QL's interface, provides an API for specifying file access permissions.

Our access control security policy covers interface access control and kernel access control, that is access from other kernel modules. The mechanism we provide to restrict access to the kernel is based on the ownership of the `/proc` file. This should be configured to belong to a user or group with elevated privileges, who will be allowed to execute PiCO QL queries. When creating the PiCO QL `/proc` entry with `create_proc_entry()` we specify the file's access permissions for granting access to the owner and the owner's group. Additionally, the `/proc` interface provides a number of callback functions for manipulating an entry's behavior. By implementing the `.permission` callback function we control `inode` operations to the `/proc` file by allowing access only to the owner and the owner's group. Kernel modules communicate with each other and with the kernel leveraging exported symbols, that is functions and variables. PiCO QL exports none, thus no other module can exploit PiCO QL's symbols.

3.7 Synchronized access to kernel data structures

Ensuring synchronized access to kernel data structures is an important requirement for our work. For the purpose of our work we define consistency and describe how we treat synchronization in PiCO QL. Finally, we discuss requirements for safe use of PiCO QL, the impact of misusing our tool to the kernel, and the impact of a corrupted kernel to PiCO QL.

3.7.1 Definition of consistency

A PiCO QL query extracts a view of a subset of the kernel's state as reflected by the data structures referenced in the query. Synchronized access to the referenced data structures is required to obtain this view and the view's quality depends on it.

Consistency requires that a kernel state view is extracted while the referenced data structures are in the same consistent state as if by acquiring exclusive read access to them atomically, for the whole query evaluation period. This definition involves two important characteristics. First, it refers to a part of the kernel's state that regards the state of particular data structures; not its global state. Second, it refers to consistency in the strong sense, that is, all data structures referenced in a query do not change state during query evaluation.

In many cases non-protected kernel data pose a limitation to the consistency we can achieve. The individual fields of many kernel data structures are not necessarily protected by a lock. In fact, this also holds for data structures protected through non-blocking synchronization such as Read-Copy-Update [19] (RCU). RCU guarantees that the protected pointers will be alive within a critical section, but the data they point to need not be consistent. The values of non-protected fields can change without notice.

Consider the list of processes as an example. The list is protected but not its individual elements. The *pid* of each process has the same value for its whole lifecycle, but this is not true for, say, its *virtual memory resident size*. The latter can indeed change in the course of a PiCO QL query so that SUM(RSS) provides a different result in two consecutive traversals of the process list while the list itself is locked.

3.7.2 Synchronization in PiCO QL

The kernel features non-blocking and blocking synchronization disciplines; PiCO QL supports both.

Non-blocking synchronization Non-blocking synchronization that favours read accesses is useful to our work in terms of performance and usability, but does not provide any consistency guarantees. The Linux kernel implements RCU locking, which provides wait-free low overhead read access.

Blocking synchronization PiCO QL supports blocking synchronization, but combining the different protocols requires caution. It is the responsibility of the person defining the virtual table specification to map the kernel data structure specifics correctly. Specifically, the writer of a data structure specification is responsible for three things: a) to specify the proper locking scheme/protocol required by a kernel data structure, b) to ensure that the specified data structure's synchronization protocol is compatible with the synchronization protocols of the existing data structure specifications, and c) queries across related data structures entail a valid locking order. For queries across unrelated data structures protected through blocking synchronization, the ordering of the locking primitives is the responsibility of the query writer. This technique imitates how kernel code would access the data structures collectively. This is a generic solution, but it is not always easy to implement, since it requires verifying that it is safe to use a specific combination of data structures in a given query.

Lock acquisition in PiCO QL works in two respects: a) all locks for globally accessible data structures are acquired prior to query execution and released at the end, and b) locks regarding data structures nested to the first are acquired during the query at the time the virtual table is instantiated (recall sections 2.2.3 and 2.3) and released once the query's evaluation has progressed to the next instantiation. Currently we use locks to provide for correct operation and consistency.

The Linux kernel does not have a unique rule for acquiring locks in order. PiCO QL has a deterministic lock ordering rule: it acquires locks according to the syntactic position of virtual tables in a query. Notably, from our experience, the majority of join operations in queries concern nested data

Listing 10. Blocking lock/unlock directive definition includes argument specifying the lock type.

```
unsigned long flags ;
 ...

CREATE LOCK SPINLOCK−IRQ(x)
HOLD WITH spin_lock_save(x, flags)
RELEASE WITH spin_unlock_restore(x, flags)

CREATE VIRTUAL TABLE ESockRcvQueue_VT
USING STRUCT VIEW SkBuff_SV
WITH REGISTERED C TYPE struct sock:struct sk_buff *
USING LOOP skb_queue_walk(&base−>sk_receive_queue,
                          tuple_iter )
USING LOCK SPINLOCK−IRQ(&base−>sk_receive_queue.lock)
```

Listing 11. Retrieve socket and socket buffer data for all open sockets.

```
SELECT name, inode_name, socket_state,
  socket_type , drops, errors , errors_soft , skbuff_len
FROM Process_VT AS P
JOIN EFile_VT AS F
ON F.base = P. fs_fd_file_id
JOIN ESocket_VT AS SKT
ON SKT.base = F.socket_id
JOIN ESock_VT AS SK
ON SK.base = SKT.sock_id
JOIN ESockRcvQueue_VT Rcv
ON Rcv.base=receive_queue_id;
```

structures; hence, in these cases syntactic means outer to inner and our traversal method corresponds to the appropriate locking order.

The illustrated query in Listing 11 combines the list of processes in the system protected through RCU, the list of a process's open files protected through RCU, and the list of socket buffers, that is a socket's receive buffer queue, protected through a spin lock. The specification for the latter data structure can be seen in Listing 10 where we reference a kernel function, which, apart from acquiring the spinlock, disables interrupts until unlock time and saves/restores the active interrupt flags to resume appropriately after a query. The `flags` variable can be defined at the boilerplate part of the DSL file.

Except for unprotected data, PiCO QL guarantees consistency for queries across unrelated data structures protected though blocking synchronization if the correct locking order is followed; queries across related ones do not provide consistency due to incremental lock acquisition.

Our implementation currently does not offer the same transaction isolation levels as a database system due to incremental lock acquisition and non-protected kernel data including data referenced by RCU-protected pointers. Notably, the treatment of synchronization in this work is implementation dependent up to an extent. It is possible to alter our implementation, or provide different configurations, so that all necessary locks for a given query are acquired in consecutive instructions prior to query evaluation and interrupts are disabled for the duration of the query.

Listing 12. C-like macro condition configures the relational interface according to the underlying kernel version.

```
#if KERNEL_VERSION > 2.6.32
        pinned_vm BIGINT FROM pinned_vm,
#endif
```

3.7.3 When things can go wrong

The virtual tables we defined for accessing Linux kernel's data structures are correct, use the right synchronization protocols, and can be safely combined together in queries. However, inappropriate virtual table definitions and ordering of virtual tables in queries might result in a kernel crash.

Incorrect locking in virtual tables and kernel corruption can result in erroneous values in PiCO QL queries. To provide some protection against kernel bugs and incorrect locking, PiCO QL checks pointers with *virt_addr_valid()* kernel function before they are dereferenced to ensure they fall within some mapped range of page areas. Caught invalid pointers show up in the result set as INVALID_P. However, the kernel can still corrupt PiCO QL via e.g. mapped but incorrect pointers.

3.8 Deployment and maintenance

PiCO QL, similar to any loadable kernel module that is not built into the kernel, requires recompilation and reloading with each freshly installed version of the Linux kernel. The process is simple:

```
make && sudo insmod[1] picoQL.ko
```

Maintenance action, if any, regards only PiCO QL's virtual relational schema, not its source code. A number of cases where the kernel violates the assumptions encoded in a struct view will be caught by the C compiler, e.g. data structure field renaming or removal. PiCO QL's debug mode, will point to the line of the DSL description where the field's representation was described.

Maintenance costs comprise adding C-like macro conditions in parts of the DSL data structure specifications that differ across kernel versions (Listing 12). The macros reference the kernel's version where the data structure's definition differs. Then they are interpreted by the DSL compiler, which generates code according to the underlying kernel version and the kernel version tag it sees on the macro. Thus, the cost of evolving PiCO QL as the kernel changes over time is minimized.

4. Evaluation

The contribution of PiCO QL is a relational interface for simplifying the writing of kernel diagnostics tasks. PiCO QL's evaluation involves three axes of analysis, use cases, quantitative, and consistency. Use cases examine how PiCO QL can aid system diagnostics. Quantitative evaluation presents the

[1] `insmod` is the Linux kernel's facility for dynamic loading of kernel modules.

Listing 13. Identify normal users who execute processes with `root` privileges and do not belong to the `admin` or `sudo` groups.

```
SELECT PG.name, PG.cred_uid, PG.ecred_euid,
  PG.ecred_egid , G.gid
FROM (
  SELECT name, cred_uid, ecred_euid,
         ecred_egid , group_set_id
  FROM Process_VT AS P
  WHERE NOT EXISTS (
        SELECT gid
        FROM EGroup_VT
        WHERE EGroup_VT.base = P.group_set_id
        AND gid IN (4,27))
  ) PG
JOIN EGroup_VT AS G
ON G.base=PG.group_set_id
WHERE PG.cred_uid > 0
AND PG.ecred_euid = 0;
```

execution cost for a variety of queries with diverse computation requirements and consistency evaluation discusses the level of consistency that the presented use cases achieve.

4.1 Use cases

In the context of the presented use cases we show how PiCO QL can be used to aid system diagnostics activities, such as performance evaluation and security audits.

4.1.1 Security

PiCO QL can improve security by expressing queries that spot suspicious behavior. For example, normal users, who do not belong to the `admin` or `sudo` groups should rarely be able to execute processes with `root` privileges; The query in Listing 13 displays processes that do not obey this rule.

Another security requirement is verifying that processes have rightful access to the files they open. The query in Listing 14 reports processes that have a file open for reading without the necessary read access permissions. This query returns forty four records for our Linux kernel (Table 1). It can identify files whose access has leaked (unintentionally or by design) to processes that have dropped elevated privileges.

Queries like the above could be used to detect the potential presence of a rootkit by retrieving details of processes with elevated privileges and unauthorized access to files. Advanced rootkits use dynamic kernel object manipulation attacks. to mutate non-control data, that is kernel data structures. Baliga *et al.* [3] present a number of such possible attacks, which tamper kernel data structures. An attack involves adding a malicious binary format in the list of binary formats used by the kernel to load binary images of processes and shared libraries. PiCO QL queries the list of formats exposing a registered binary handler's memory addresses of load functions as shown in Listing 15.

Hardware virtualization environments suffer from vulnerabilities as well [26]. CVE-2009-3290 [22] describes how guest VMs, operating at Ring 3 level, abuse hyper calls, normally issued by Ring 0 processes, to cause denial of service or to control a KVM host. The SQL query in Listing 16 retrieves the current privilege level of each online virtual CPU and its eligibility to execute hypercalls, and displays VMs violating hypercalls. This query could be used to detect the corresponding attacks.

Perez-Botero *et al.* [26] report vulnerabilities in the KVM hypervisor due to lack of data structure state validation (CVE-2010-0309 [23]). Lack of data validation in the programmable interval timer (PIT) data structures allowed a malicious VM to drive a full host operating system crash by attempting to read from `/dev/port`, which should only be used for writing. The query in Listing 17 provides a view of the PIT channel state array, which mirrors the permitted access modes as array indexes; read access is masked to an index that falls out of bounds and triggers the crash when later dereferenced. Accessing this information in the form of a simple SQL query can help with automatic validation of data structure state during testing and prevent vulnerabilities of this kind.

Listing 14. Identify files open for reading by processes that do not currently have corresponding read access permissions.

```
SELECT DISTINCT P.name, F.inode_name, F.inode_mode&400,
  F.inode_mode&40, F.inode_mode&4
FROM Process_VT AS P
JOIN EFile_VT AS F
ON F.base=P. fs_fd_file_id
WHERE F.fmode&1
AND (F.fowner_euid != P. ecred_fsuid
  OR NOT F.inode_mode&400)
AND (F.fcred_egid NOT IN (
    SELECT gid
    FROM EGRoup_VT AS G
    WHERE G.base = P.group_set_id)
  OR NOT F.inode_mode&40)
AND NOT F.inode_mode&4;
```

Listing 15. Retrieve binary formats not used by any processes.

```
SELECT load_bin_addr, load_shlib_addr , core_dump_addr
FROM BinaryFormat_VT;
```

Listing 16. Return the current privilege level of each KVM virtual online CPU and whether it is allowed to execute hypercalls.

```
SELECT cpu, vcpu_id, vcpu_mode, vcpu_requests,
    current_privilege_level , hypercalls_allowed
FROM KVM_VCPU_View;
```

Listing 17. Return the contents of the PIT channel state array.

```
SELECT kvm_users, APCS.count, latched_count, count_latched ,
       status_latched , status , read_state , write_state , rw_mode,
       mode, bcd, gate , count_load_time
FROM KVM_View AS KVM
JOIN EKVMArchPitChannelState_VT AS APCS
ON APCS.base=KVM.kvm_pit_state_id;
```

Listing 18. Present fine-grained page cache information per file for KVM related processes.

```
SELECT name, inode_name, file_offset , page_offset , inode_size_bytes ,
  pages_in_cache , inode_size_pages , pages_in_cache_contig_start ,
   pages_in_cache_contig_current_offset , pages_in_cache_tag_dirty ,
  pages_in_cache_tag_writeback , pages_in_cache_tag_towrite ,
FROM Process_VT AS P
JOIN EFile_VT AS F
ON F.base=P. fs_fd_file_id
WHERE pages_in_cache_tag_dirty
AND name LIKE '%kvm%';
```

Listing 19. Present a view of socket files' state.

```
SELECT name, pid, gid, utime, stime, total_vm , nr_ptes , inode_name,
  inode_no, rem_ip, rem_port, local_ip , local_port , tx_queue, rx_queue
FROM Process_VT AS P
JOIN EVirtualMem_VT AS VM
ON VM.base = P.vm_id
JOIN EFile_VT AS F
ON F.base = P. fs_fd_file_id
JOIN ESocket_VT AS SKT
ON SKT.base = F.socket_id
JOIN ESock_VT AS SK
ON SK.base = SKT.sock_id
WHERE proto_name LIKE 'tcp';
```

4.1.2 Performance

Regulating system resources is a requirement for system stability. PiCO QL can provide a custom view of a system's resources and help discover conditions that hinder its performance. The query in Listing 18 extracts a view of the page cache detailing per file information for KVM related processes. It shows how effectively virtual machine I/O requests are serviced by the host page cache assuming that the guest operating system is executing direct I/O calls.

One of PiCO QL's advantages is its extensible ability to offer a unified view of information across the kernel. The combined use of diagnostic tools can point to the solution in most situations, but for some of those situations, PiCO QL alone provides the answer. For instance, consider Listing 19, which shows how to combine data structures to extract detailed performance views of and across important kernel subsystems, namely process, CPU, virtual memory, file, and network. Adding to this list means only expanding the representation of data structures. The rationale behind PiCO QL is to leverage the knowledge of a widely-used data management language for diagnosing issues evident in the system's data structures.

Performance debugging is a complex task, which is usually undertaken by observing the symptoms, forming a theory, and checking it using appropriate tools. PiCO QL significantly simplifies the final step. Let us consider an example drawn from reference [13], which regards retrieving per process memory mappings including virtual memory, RSS, private anonymous memory, permissions, and mapped file. In this case, Gregg [13, p. 307] proposes the use of `pmap` to see this information. We present the corresponding PiCO QL query in Listing 20.

Listing 20. Present virtual memory mappings per process.

```
SELECT vm_start, anon_vmas, vm_page_prot, vm_file
FROM Process_VT AS P
JOIN EVirtualMem_VT AS VT
ON VT.base = P.vm_id;
```

4.2 Quantitative evaluation

PiCO QL's quantitative evaluation presents the execution efficiency of SQL queries in the Linux kernel (Table 1). We compute the query execution time as the difference with cycle-accurate precision between two timestamps, acquired at the end and at the start of each query respectively. Tests were carried out in an otherwise idle machine (`1GB RAM, 500GB disk, 2 cores`) running the Linux kernel (v3.6.10). The mean of at least three runs is reported for each query.

The rows in Table 1 contain queries with diverse characteristics and columns contain query properties and measurement metrics. Specifically, columns include references to queries presented in the paper, a label characterizing each query's execution plan, the lines of SQL code for expressing each query, the number of records returned, the total set size evaluated, the execution space and time, and the average time spent for each record.

Table 1 indicates the programming effort for writing some representative queries in terms of LOC. As there is no standard way to count SQL lines of code, we count logical lines of code, that is each line that begins with an SQL keyword excluding AS, which can be omitted, and the various WHERE clause binary comparison operators. At a minimum, SQL requires two lines of code to retrieve some information (SELECT…FROM…;). The PiCO QL evaluation queries listed in Table 1 require between six and thirteen LOC. This is not a small figure, but it is a price we pay for sophisticated queries like most of those used in our evaluation. Moreover, a large part of this programming cost can be abstracted away since queries can be composed from other queries leveraging standard relational views (Listing 7). This is the case with Listings 16 and 17 whose LOC drop to less than half of the original. Finally, PICO QL provides an advantage of qualitative nature: it allows users to focus on the information retrieval aspects of an analysis task. This can not be measured by LOC.

Query measurements allow a number of observations. First, query evaluation appears to scale well as total set size increases. The average amount of time spent for each record in the relational join query (Listing 9) is the smallest beating even the average record evaluation time of the query performing arithmetic operations (Listing 19). The cartesian product evaluated for the former approximates 700,000 records. Second, nested subqueries (Listing 13) perform well as opposed to DISTINCT evaluation (Listing 14), which seems to be the source of large average execution time per record. Third, multiple page cache accesses (Listing 18) dur-

Table 1. Present SQL query execution cost for 10 diverse queries.

PICO QL query	Query label	LOC	Records returned	Total set size (records)	Execution space (KB)	Execution time (ms)	Record evaluation time (μs)
Listing 9	Relational join	10	80	683929	1667.10	231.90	0.34
Listing 16	Join – virtual table context switch ($\times$2)	3(9)	1	827	33.27	1.60	1.94
Listing 17	Join – virtual table context switch ($\times$3)	4(10)	1	827	32.61	1.66	2.01
Listing 13	Nested subquery (FROM, WHERE)	13	0	132	27.37	0.25	1.89
Listing 14	Nested subquery (WHERE), OR evaluation, bitwise logical operations, DISTINCT records	13	44	827	3445.89	10.69	12.93
Listing 18	Page cache access, string constraint evaluation	6	16	827	26.33	0.57	0.69
Listing 19	Arithmetic operations, string constraint evaluation	11	0	827	76.11	0.59	0.71
SELECT 1;	Query overhead	1	1	1	18.65	0.05	50.00

ing a query evaluation are affordable incurring the second best record evaluation time and topping even arithmetic operations (Listing 19), which, as expected, are very efficient.

According to the memory space measurements during query execution, PiCO QL has modest memory space requirements for most query plans. We distinguish two queries due to their significantly larger memory space footprint. The relational join query (Listing 9) requires 1.5MB but this can be justified by the evaluated total set size, which approximates 700K records. The second query with large memory space requirements, almost 3.5MB, is the one involving DISTINCT evaluation (Listing 14). In this case, we would expect a more modest footprint, since the query returns 40 records; the DISTINCT algorithm is evaluated on this set.

4.3 Consistency evaluation

Most of our use cases depend on non-blocking synchronization, perform incremental lock acquisitions and, involve unprotected kernel data structure fields, thus the extracted views do not reflect a consistent state. The distance between the obtained view and a consistent view depends on the changes performed on the referenced data structures from other CPUs a) between incremental data structure lock acquisitions and b) to non-protected data structure fields including those referenced by RCU-protected pointers. However, consistency does not matter too much for a number of scenarios we tackle, such as performance profiling. The queries in Listings 9, 11, 18, 19, and 20 might provide inconsistent views, but the views will still be meaningful in that they communicate real values of system resources.

On the other hand, for data structures that use proper locking PiCO QL provides a consistent view of their contents. For instance, the query in Listing 15 provides a consistent view of a kernel's list of binary formats. The list is protected by a read-write lock and PiCO QL will never build a view of an inconsistent list. Each element in the list may change independently, however, this is no different from the existing kernel code, where there is no mechanism to guarantee that the contents of the elements are consistent with the list structure itself.

5. Related work

Our work is related to relational interfaces within operating systems and kernel diagnostic tools.

5.1 Operating system relational interfaces

Relations are part of the structure of some operating systems. The Barrelfish operating system features a general purpose system knowledge base (SKB) used for many core purposes, such as declarative device configuration [29] and cache-aware placement of relational database operators [12] among others. The knowledge base runs as a user-level process, collects information from the kernel about the various system services, and provides an interface to a constraint logic programming engine (CLP) written in Prolog. The collected information forms the input for the CLP algorithm and the produced solution is used to derive kernel data structures for the appropriate system service. Barrelfish's logic engine is of equivalent expressivity to relational calculus.

The Pick operating system [5] is built around a multivalued database representation of hash files, which form nested representations. The data manipulation language includes access queries with two commands LIST, which can express selection using a syntax similar to a WHERE clause, and SORT. Pick features no data typing and data integrity is left to the application programmer.

In contrast to the Barrelfish SKB and Pick, which use a declarative specification to derive kernel data structures, PiCO QL represents existing kernel data structures in relational terms and provides SQL queries on them through their relational representation.

Relational-like interfaces exist at the operating system level for carrying out validation tasks. The work by Gunawi *et al.* [14] contributes a declarative file system checker that improves over `e2fsck` [20], mainly because of its high-level abstraction. It imports the file system's metadata in a MySQL database, carries out file system integrity checks by means of SQL queries, and performs repairs accordingly. PiCO QL would alleviate the need for storing the file system's metadata in a database, but would require loading the metadata in memory. For a disk-based file system this would create a major overhead, but not for a main-memory one. In a potential near future where non-volatile RAM substitutes DRAM PiCO QL could be a useful tool for verifying main memory file systems [24].

5.2 Kernel diagnostic tools

Kernel diagnostic tools implement a variety of approaches. We discuss tools for dynamic analysis of production systems in terms of their instrumentation techniques and analysis methods.

Program instrumentation techniques augment a system's code with special-purpose code, typically in an automated manner, to create system execution metadata.

Dynamic or binary code instrumentation that replaces or rewrites binary code is used more widely than static or source code instrumentation. Tools like DTrace [7] and SystemTap [27] perform dynamic or binary code instrumentation. In contrast, LTTng [9] performs automated source code instrumentation. LTTng is a tracer; it records statically determined events in trace buffers written to files and offers post-mortem analysis. Source code instrumentation requires a fresh kernel build for each tracing attempt.

Binary code instrumentation tools are of varying nature. DTrace employs a virtual machine and a set of kernel modules that manage a probe infrastructure in the Solaris kernel. SystemTap is a meta-instrumentation tool, which compiles scripts written in the SystemTap procedural language into instrumenting loadable kernel modules. System execution metadata typically capture events, but KLASY [34] provides data structure selection and filtering in C. KLASY is a dynamic aspect-oriented system.

PiCO QL consumes directly lower level operating system data, that is kernel data structures. PiCO QL's hook technique is probe-based but very different than typical probe-based instrumentation techniques, like DTrace's, in three ways. First, as mentioned, PiCO QL hooks to data structures; it does not instrument events as is the typical case with most tools like SystemTap and DTrace. Second, PiCO QL requires a single hook for each data structure in order to track it. Event-based tools generate even thousands of probes to achieve instrumentation coverage [7]. Third, PiCO QL's hooking is performed manually at the module's source code. Its footprint is negligible compared to automatic binary instrumentation of event-based tools. Consequently, PiCO QL incurs zero performance overhead in idle state, because PiCO QL's "probes" are actually part of the loadable module and not part of the kernel. The other mentioned tools also achieve zero probe effect. When PiCO QL is active, the performance overhead is that related to evaluating the input queries and using the required data structures.

Analysis methods The tools we cited employ various analysis methods for dynamic analysis. We could not find art that allows relational queries to kernel data structures in the manner provided by PiCO QL. DTrace and SystemTap define an event-based procedural language for expressing analysis tasks. Dynamic aspect-oriented tools like KLASY define pointcuts, that is predicates, which select matching function calls and/or data structure fields.

Windows-based operating systems provide the Windows Management Instrumentation (WMI) [33] infrastructure, which provides access to management data and operations. Management data are divided in namespaces, which contain classes. Object associations are modeled in association classes instead of embedded object references. WMI supports a query language, WQL, which implements a subset of SQL. A WQL query targets a specific WMI class hierarchy; joins are not allowed, nor cross-namespace queries or associations. WQL is in fact syntactic sugar over the object-oriented WMI data model. In contrast, PiCO QL exposes a relational model view of the underlying hierarchical data model. This is what makes it queryable through standard SQL.

6. Discussion

The layered design of PiCO QL's representation and user interface, provide flexibility in the types of applications that it can support. The amount of effort for writing struct views in the PiCO QL DSL varies mostly in the level of knowledge of the kernel needed and much less in terms of writing the DSL description. In fact, for each line of code of a kernel data structure definition, the DSL specification requires one line of code for the *struct view* definition, that is to represent fields in columns. The *virtual table* definition adds six lines of code on average to the total cost. To eliminate this effort completely, we examine deriving data structure specifications automatically from data structure definitions or appropriate annotations. This is an avenue for future work.

PiCO QL provides a separation of concerns between an operating system's operation and diagnostics. C is appropriate for programming the kernel, whereas SQL is appropriate for modelling a data schema and performing analysis tasks. With PiCO QL we can invest in a relational representation for understanding kernel data structures and extracting knowledge from them.

Queries in PiCO QL can execute on demand. However, users cannot specify execution points where queries should automatically be evaluated. A partial solution would be to combine PiCO QL with a facility like `cron` to provide a form of periodic execution. Conditional query execution would require kernel instrumentation.

We distinguish two avenues for further research related to this work. First, we can improve PiCO QL's usability, safety, and consistency by adapting our treatment of synchronization. In particular, to provide queries that acquire locks in the correct order, our plan is to leverage the rules of the kernel's lock validator [15] to establish a correct query plan at our module's respective callback function at runtime. The lock validator finds and reports many synchronization problems. To enhance consistency of kernel state views, our plan is to provide lockless queries to snapshots of kernel data structures. Across data structures protected through blocking synchronization, this approach would provide consistent kernel state views; for the rest, it would minimize the gap to consistency.

Second, we plan to investigate locking intricacies during SQL queries to kernel data structures. Locking dependencies have important repercussions to safety and performance; examining these for a new kernel data access model might offer insight with respect to the way we use locks in the kernel or in relational queries.

7. Conclusions

We present a relational interface to accessible Unix kernel data structures supported by an implementation for the Linux kernel. Our work uses SQL queries to provide a custom high-level view of low-level system information, which is hidden in complex data structures. The implementation is type-safe and secure. We evaluate the potential benefits of our work and exemplify PiCO QL's contribution in diagnosing security vulnerabilities and in obtaining non-trivial views of system resources. Our evaluation demonstrates that this approach is efficient and scalable by measuring query execution cost.

In this paper we present a small number of queries on top of PiCO QL's relational interface, which currently amounts to 40 virtual tables. However, the kernel's relational representation with PiCO QL is only limited by the kernel's data structures; more than 500 are directly available. In fact, it is easy for everyone to roll their own probes by following the tutorial available online [18].

8. Availability

PiCO QL is open source software distributed under the Apache license. Code and other project data are available through Github at https://github.com/mfragkoulis/PiCO_QL.

Acknowledgments

This research has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thalis — Athens University of Economics and Business — Software Engineering Research Platform. We would like to thank Georgios Gousios and the reviewers for their comments on the paper and in particular our shepherd Timothy Roscoe, whose comments have helped us clarify important aspects of our work.

References

[1] G. Allen and M. Owens. *The Definitive Guide to SQLite, Second Edition*. Apress, Berkeley, CA, USA, 2010.

[2] M. J. Bach. *The design of the UNIX operating system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[3] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[4] M. Balmer. Lua in the NetBSD kernel. In *Research Room @ FOSDEM 2013*, Brussels, Belgium, 02 2013.

[5] R. J. Bourdon. *The PICK Operating System: A Practical Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In A. Arpaci-Dusseau and R. Arpaci-Dusseau, editors, *Proceedings of the USENIX 2004 Annual Technical Conference*, ATEC '04, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.

[8] C. J. Date and H. Darwen. *A guide to the SQL standard (4th ed.): a user's guide to the standard database language SQL*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] M. Desnoyers and M. R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the 2006 Ottawa Linux Symposium (OLS)*, 2006.

[10] M. Fragkoulis, D. Spinellis, and P. Louridas. An interactive relational interface to main-memory data structures. Technical report, Department of Management Science and Technology, Athens University of Economics and Business, February 2014. Available online February 2014.

[11] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 273–284, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[12] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. Cod: Database / operating system co-design. In *CIDR, Sixth Biennial Conference on Innovative Data Systems*

Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings, 2013.

[13] B. Gregg. *Systems performance: entreprise and the cloud*. Prentice Hall, Upper Saddle River, NJ, 2013.

[14] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: a declarative file system checker. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 131–146, Berkeley, CA, USA, 2008. USENIX Association.

[15] I. Molnar and A. van de Ven. Runtime locking correctness validator, 2014. Available online Current March 2014.

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[17] S. Lampoudi and D. M. Beazley. SWILL: A simple embedded web server library. In C. G. Demetriou, editor, *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 19–27. USENIX, 2002.

[18] M. Fragkoulis. The PiCO QL Linux kernel module tutorial, 2013. Available online January 2013.

[19] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[20] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program, April 1986.

[21] E. Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, Oct. 2011.

[22] National Institute of Standards and Technology. CVE-2009-3290, National Vulnerability Database, 2009. September 22, 2009.

[23] National Institute of Standards and Technology. CVE-2010-0309, National Vulnerability Database, 2010. January 12, 2010.

[24] S. Oikawa. Integrating memory management with a file system on a non-volatile main memory system. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1589–1594, New York, NY, USA, 2013. ACM.

[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[26] D. Perez-Botero, J. Szefer, and R. B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[27] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium (OLS)*, 2005.

[28] R. E. Sanders. *ODBC 3.5 Developer's Guide*. McGraw-Hill Professional, 1998.

[29] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter. A declarative language approach to device configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 119–132, New York, NY, USA, 2011. ACM.

[30] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.

[31] The SQLite team. The SQLite Query Planner Release 3.8.0., 2014. Available online Current March 2014.

[32] The SQLite team. The virtual table mechanism of SQLite, 2014. Available online Current March 2014.

[33] C. Tunstall and G. Cole. *Developing WMI solutions: a guide to Windows management instrumentation*. The Addison-Wesley Microsoft Technology Series. Addison-Wesley, 2003.

[34] Y. Yanagisawa, K. Kourai, and S. Chiba. A dynamic aspect-oriented system for OS kernels. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 69–78, New York, NY, USA, 2006. ACM.