Towards Automatically Extracting the Relational Model of an Operating System

Brian Choi
Yale University
USA

George Neville-Neil
Yale University
USA

Alex Yuan
Yale University
USA

Avi Silberschatz
Yale University
USA

Peter Alvaro
UCSC
USA

Robert Soulé
Yale University
USA

Abstract

Several recent projects have argued that users would benefit from having a relational interface to operating systems state. In this paper, we report on the progress of a tool to automate this data modeling process. While automating schema inference in the general case appears to be intractable, we argue that it is feasible in the limited domain of operating system. Our approach leverages static analysis on the tightly-controlled coding styles common in OS kernels and hybridized techniques from the programming languages and databases literature for inferring and propagating constraints. We use a simple static analysis to extract relation names, types, and arity from constrained operating system source code patterns. We then leverage LLMs and iterative refinement to identify constraints. Our prototype implementation automates the extraction of 10% of the operating system structure into the relational model.

CCS Concepts

• **Software and its engineering** → **Operating systems**; • **Information systems** → *Entity relationship models*; **Data access methods**.

Keywords

Operating Systems, Databases, Structured Query Language (SQL)

ACM Reference Format:
Brian Choi, George Neville-Neil, Alex Yuan, Avi Silberschatz, Peter Alvaro, and Robert Soulé. 2025. Towards Automatically Extracting the Relational Model of an Operating System. In *The 19th International Symposium on Database Programming Languages (DBPL '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3735106.3736534

1 Introduction

The relational model is always finding ways to escape the walled garden of databases. Over the last 20 years, adopting a uniform representation of system state that is amenable to composition via *join* has shown benefits in the domains of networks [9], distributed systems [6], program analysis [8], and many others. One research area, however, abounds with proposals for relational interfaces [17–19] but has yet to yield a full-featured prototype: operating systems (OS). This is a pity, as operating systems present uniquely difficult data management challenges.

The idea of a relational OS, while old [18], has received recent attention. DBOS [17] is a clean-slate proposal to implement all OS state as relations and OS policy as queries atop a minimalist kernel, while OSDB [19] embeds a query processor in existing OSs and presents existing data, as tables, which provide both read and udpate semantics. Something that both proposals have in common is that they are a long way from their goal of a realistic modern operating system. To the best of our understanding, DBOS cannot even be prototyped until its micro-kernel is developed, and the OSDB prototype models only a handful of OS abstractions (e.g., processes and sockets). That progress should be slow is unsurprising given the magnitude of the task of developing a data model for production operating systems, which are enormous code bases that have been developed over decades. As an example, consider FreeBSD, which includes code that goes back to the 1980s, and is currently 9,533,488 lines of code spread across 16,376 files.

Given that there is such broad agreement that OS should more closely resemble databases, why don't we have such an OS—and how long will we need to wait? In this paper, we present a vision alongside early evidence that we can reach the goal of a complete relational interface to a full-featured, modern OS very soon. This vision favors pragmatism over purity, and relies on two principles: *incrementality* and *automation*. The incremental approach, exemplified by OSDB, makes it possible to convert OS state into relations *one relation at a time*, all the while measuring the benefits of queries over that newly-represented state. We believe this to be the only realistic path forward: the clean-slate approach is unlikely to ever catch up with a state-of-the-art that keeps moving forward more quickly than any research team can.

However, incrementality is not enough, and OSDB is a long way from a full-featured relational OS. We also need some mechanism to accelerate the data modeling of OS kernel state. We believe that this seemingly intractable problem of *automating* schema inference may be attacked by exploiting the tightly-controlled coding styles common in OS kernels and by hybridizing techniques from the programming languages and databases literature for inferring and propagating constraints. We use a simple static analysis to extract relation names, types, and arity from constrained operating system source code patterns. We then leverage LLMs and iterative refinement to identify constraints.

In the database community, there has been a wealth of prior work on sound inference of functional dependencies [1, 2, 4, 7, 10,

ACM ISBN 979-8-4007-1919-6/2025/06
https://doi.org/10.1145/3735106.3736534

13, 15, 16, 21–23]. Our approach differs from this prior work in three key respects. First, we are explicitly not trying to solve this for the general case, but rather, we focus exclusively on the domain of OS kernels. Second, rather than taking an incomplete schema (i.e., one missing dependencies) as input, we attempt to extract both the structure and semantics from the kernel code. Finally, and most significantly, we do not strive for soundness. Rather, we prioritize expediency and practicality, relying on the richness of OS workload traces to provide iterative refinement of our guesses.

In our OSDB [19] paper, our prototype included 5 hand-written tables. In a few short months, using the algorithms and tools described in this paper, we have increased the number of tables that we can query in the operating system by two orders of magnitude. Moreover, the number of fields in each table has increased significantly as well. For example, the process table in the OSDB [19] paper had 7 fields and now includes 110.

Below, we describe the design of a tool to automatically extract the relational model from the operating system kernel source code (§2). We then present initial results from using the tool on the FreeBSD operating system (§3). We then discuss limitations and future work (§4). Finally, we briefly discuss related work (§5) and conclude (§6).

2 System Design

Data modeling involves identifying and organizing the data entities and their relationships to create a structured representation. In the relational model [3], the structures are tables (relations) with rows (tuples) and columns (attributes). The structural model is further refined via *constraints*, which restrict which tuples constitute a valid instance of the schema.

To automate data modeling, a tool must perform the following tasks: (i) determine the name of each table, (ii) choose the arity of the tables and assign each column a unique name, (iii) decide the domain of the elements of the columns, and (iv) decide the constraints on the permitted values. We perform steps i-iii via a direct structural translation. For step (iv), we use a combination of heuristics, learning, and refinement to indentify primary keys, and static code analysis to identify foreign key constraints.

2.1 Structural Extraction

The modeling task is aided by a few key observations about operating system code. First, the majority of modern operating systems are developed in low-level, systems programming languages (i.e., C) with few, or no, built-in complex data types. Second, the overall structure of operating systems are relatively simple, as developers tend to avoid complex abstractions in favor of low overhead. Indeed, the majority of the data structures in an operating system kernel are lists of C structures. In FreeBSD nearly every complex data structure depends on macros from a single header file (`queue.h`) which provides singly and doubly linked lists, and tail queues. Finally, as with any large code base, operating system developers have already naturally modularized their code into the relevant entities.

These observations lead us to a design that allows us to automate the structural extraction of a significant portion of the operating system. Lists of flat structures map directly onto the relational model, with every such list represented as a table, every element (struct) in the list as a tuple (row), and every data member of a structure as an attribute (column). We can re-use the names of structs and fields as well as the declared types directly in our tables.

The implementation of this design is complicated by two factors. First, the `queue.h` header file in both FreeBSD and Linux provide a set of macros that define and operate on the list structures. As a consequence, the static analysis to identify list and queue uses must be able to analyze the C preprocessor (CPP) directives. Second, the operating system code makes frequent use of `typedef` statements to alias existing primitive types (e.g., `pid_t` is a 32-bit, signed `int`). Thus, the static analyzer must be able to perform the type resolution. To address these complications, the structural extractor is implemented as a plugin to the Clang compiler, which provides an API to access CPP directives and the symbol table after the compiler's semantic analysis pass. We explored an alternative implementation that extracted the symbols from the gdb debugger, but it was not able to meet these requirements.

2.2 Semantic Extraction

The structural extraction to relations is relatively straightforward, in large part due to the regularity of operating system data organization. However, identifying an appropriate set of constraints would appear to be impossible. After all, just as one cannot infer what a box is meant to contain just by looking at it, we cannot in general determine a set of constraints merely by studying a schema or instances of a schema. A human expert—the person doing the modeling—is typically required to supply these semantics. And yet, as we argued above, doing the data modeling for the OS by hand would be a tremendous effort.

In this section, we discuss how we combine a variety of techniques for constraint inference, ranging from simple syntactic analysis to heuristics to approaches that leverage machine learning (and hence from sound to unsound). We discuss how by combining covering workloads with constraint checking in the query processor can provide the necessary guardrails to quickly repair incorrect guesses. We focus for now on primary and foreign key constraints.

2.2.1 *Workloads and Traces.*

The viability of our approach, which will involve a lot of guesswork (and hence a lot of mistakes!) hinges on two features of data management systems and operating systems, respectively. First, although constraints are difficult to soundly infer, they are extremely cheap to check. A hypothesis about a candidate key can always be tested by creating a checked constraint in the database and watching for violations. Second, unlike application software for which representative workloads may not exist or may be shallow, OS benchmarks (e.g. UNIXBench, Spec CPU, FIO) are abundant. We exploit these and other workload generators not to measure the performance of the OS but to drive the system under test into a large number of different states. Each such state is a distinct schema instance, allowing us to check all of our hypotheses and refine them as needed.

In the OSDB paper [19], we discussed the snapshot mechanism by which OSDB creates a transactionally-consistent, time-series view of the structurally-transformed kernel state. We have modified that mechanism such that the snapshot data is not just stored in memory, but written to persistent disk. The structural extraction process significantly increases the number of tables that OSDB snapshots, allowing us to collect a fairly complete trace of kernel state after running the combined workloads.

2.2.2 Primary Key Constraints. As a first step, we start with trying to identify a primary key, i.e., a column or a set of columns that uniquely identifies each row in a table, for each table that we extracted. In general, the primary key would have to be given from the domain expert or derived from a set of functional dependencies which, again, were provided from the domain expert. Following our principle of aggressive automation we seek to remove this requirement in the common case.

One straw-man approach for finding a primary key could be the following. We know that although we can't look at the data to determine if an attribute is a primary key, we can look at the data to determine that a candidate key is *not* a primary key. So, if we had a sufficient amount of data, we might try an iterative approach: we first guess, "is this a key?" and then we check if it is not one against a sufficiently size trace. If it is not a key, then we make another guess, and repeat.

Of course, this straw-man approach is an exponential-time algorithm in the size of the problem. For a set of n attributes, there are $2^n - 1$ non-empty subsets, and trying them all would be prohibitively expensive. So, to reduce this search space, we might try to use heuristics for choosing candidate keys. There are certain "patterns" in the kernel source code that might suggest that an attribute or set of attributes is a primary key. As a simple example, if an attribute name includes "id" in it, then the attribute is probably an identifier, and possibly unique. This is the case, for example, with `pid` in the `process` table and `tid` in the `threads` table. However, this hunch about variables that end in "id" is not always correct. For example, the `files` table includes an attribute `p_uid` and the `vnodes` table includes the attribute `f_fsid`, neither of which are primary keys.

While an experienced operating system developer might be able to articulate a good set of heuristics in the form of patterns in the code to look for, Large Language Models (LLMs) are *really* good at looking for patterns. So, while we hate to follow the trend of "just throw AI at the problem", this seemed like a viable use. The strength of our approach is that we can treat the LLM as a black box for "guessing keys". It doesn't need to be correct, since we have guardrails in the form of traces to check if a guess is wrong.

2.2.3 Foreign Key Constraints. Having identified primary keys, we next try to identify foreign keys, i.e., a set of attributes in one table that refers to the primary key of another table. Foreign keys help enforce referential integrity. For foreign keys, we again examine structural patterns in operating systems implementations from which we can soundly infer foreign key relationships. There are two cases we consider.

Nested structures. A common pattern in OS data structures besides lists of flat structures is lists of structures with nested struct elements. When one structure is nested inside another, either directly or via a pointer, we can use the primary key of the enclosing structure as a primary key in the inner structure. In the FreeBSD kernel, for example, a `proc` structure contains an array of pointers to `thread` structures. We can then model `thread` as a relation, with a foreign key constraint on `proc.pid`, even though this attribute does not occur in the thread struct, because we know that every thread belongs to a process. As was the case with primary keys, this nesting pattern may not actually be a foreign key relationship, as the enclosing structure may not contain the *only* pointers to the nested structure. Again, we can revert to examining data traces for violations.

We enact this design with a three step algorithm. First, we perform an initial structural extraction to record the inner-outer relationship. We then perform a semantic extraction pass, as described above, to infer the primary key of the outer struct. Finally, we make a second structural extraction pass, incorporating the primary key of the outer struct into the schema of the inner struct.

Invariant pointers. There are times when we may not know the primary key of the outer structure. This can occur in C code, for example, when the relationship between in-memory structures are established through pointers. In such situations, we can build on an idea of Erik Meijer, who argues that memory pointers are foreign keys that go the wrong way [12]. In these cases, we include pointers as a field in many table representations of OS structures. We note that the pointers in an operating system kernel are unique because the kernel program executes in a single address space, making them a suitable choice for foreign keys. Once a kernel data structure is allocated, it is never moved and its address never changes.

3 Initial Results

Prototype Implementation. We have implemented a prototype for automatically extracting the relational model from the operating system. The structural extraction was implemented as a Clang 14.0.5 plugin in approximately 600 lines of C code. We integrated this plugin with the kernel build process. The semantic extraction was implemented in 223 lines of Python code, using the ChatGPT API.

Structural Extraction. Overall, there are 3,286 non-static C structs in the FreeBSD kernel, including the driver code. When we published the OSDB paper [19], our prototype included 5 tables: processes, threads, files, TCP connections, and UDP connections. We wrote these tables by hand and, for expediency, included only the subset of attributes that we needed to conduct our experiments. For example, the processes table included 7 fields, while the actual process struct from FreeBSD includes 110 members. Our TCP connections table had 10 fields, while the `tcpcb` struct has 153 members. Using our Clang plugin to automatically extract table definitions, OSDB now supports 328 tables, or 10% of the structs in the kernel. These structures were extracted because they exist in natural tables, lists of C structures, which are amenable to being exposed as relations. Moreover, every table definition includes all fields in every struct.

Semantic Extraction. Table 1 lists a subset of relations we extracted and their predicted primary keys. For processes and threads, the predicted keys match our expectations, i.e., our framework accurately identifies `pid` and `tid` as one attribute keys. We also see that for TCP and UDP connections, our framework correctly identifies the classic 5-tuple as the primary key, demonstrating that our approach can identify even multi-attribute keys.

For `file` and `vnode`, our framework does not predict a primary key. Perhaps counter-intuitively, this result is encouraging. Indeed,

Relation	Primary Key
proc	pid
thread	tid
tcp	laddr, lport, faddr, fport, inp_ip_p
udp	laddr, lport, faddr, fport, inp_ip_p
file	No primary key predicted.
vnode	No primary key predicted.

Table 1: Predicted Primary Keys

```
struct file {
    void     *f_data;        /* File-specific data */
    struct   fileops *f_ops; /* File operations */
    struct   ucred *f_cred;  /* Associated credentials */
    struct   vnode *f_vnode; /* Vnode  */
    off_t    f_offset;       /* File offset */
    int      f_flag;         /* Open flags */
    int      f_count;        /* Reference count */
    struct   mtx f_mtx;      /* Mutex to protect struct */
    struct   knlist *f_knlist; /* Kernel event list */
    short    f_type;         /* Type of the file */
    volatile u_int f_count_atomic; /* Reference count */
};
```

Figure 1: FreeBSD file structure

if we examine the fields of the `file` struct, shown in Figure 1, we can confirm that their is no field that would serve as a suitable primary key. This example serves to remind us that kernel developers did not organize their data structures with the relational model in mind. If they had, perhaps the file struct would have included a unique file id field. As we continue developing our prototype, we seek to find the limits of an incremental and automatic approach. We continue to discuss this in the next section.

4 Limitations and Future Work

In this paper, we have demonstrated first steps towards accelerating the data modeling of OS state. Motivated by practical concerns, we threw the kitchen sink at the problem, combining sound and unsound approaches and relying on constraint enforcement and covering workloads to catch mistakes. In our race to build a data model, we skipped directly to the problem of inferring primary and foreign keys on relations, without first understanding the functional dependencies (FDs) among attributes that would have guided a more principled data modeling effort.

With our proof-of-concept in place, the next phase of our research will focus on modeling FDs. Our experience combining heuristics, structural rules, and machine learning to make informed guesses, and using rich workloads and dynamic assertions to provide guard rails, generalizes to the problem of identifying likely FDs. Once we have seeded our inference system with ground truth by combining these techniques, we may apply techniques such as the chase [11] to soundly infer additional FDs that are implied by those that we have identified.

5 Related Work

Inferring Functional Dependencies. The idea of automating the determination of functional dependencies dates back at least 40 years to the work of Dina Bitton [2]. Some more recent efforts include Tane [7], Fun [13], FD_Mine [22, 23], DFD [1], Dep-Miner [10], FastFDs [21], Fdep [4], and Papenbrock et al. [15, 16]. In general, the problem cannot be solved without additional information. Most efforts, very broadly speaking, attempt to generate candidate keys, prune the search space, and validate the results. We differ in that we are not only trying to infer the functional dependencies, but are also trying to extract the structure of the relations. Our approach also differs in that we are not trying to solve the problem in general, but concentrate on the limited domain of operating systems.

Relational Interfaces for Operating Systems. Operating systems and databases have a complex relationship. Early on, databases offered an alternative to general-purpose operating systems for resource and data management [5]. Today, databases are often thought of as applications that make use of the operating system's facilities for managing hardware resources. However, databases and operating systems are often at odds, e.g., with respect to cache replacement, scheduling, and file management [20].

ROSI [18] is the first work that we are aware of to propose a relational interface to the operating system to improve usability. osquery [14] mirrors important OS state in a user-space database, but offers only weak semantics including stale reads and non-atomic updates to kernel state. At the other end of the design spectrum, DBOS [17] advocates for a "clean slate" approach that entirely replaces the OS with a query processor. Our own OSDB [19] shares similar goals but offers an incremental approach.

6 Conclusion

The idea of applying the relational model to operating system data is not a new one. Our previous results, using an incremental approach to exposing existing data structures with an in-kernel query processor has proven that such is a system is not only possible but that there is a clear path to a fully relational operating system.

Our present work automates the tasks of extracting structural and semantic knowledge from the operating system's source as well as execution state to accelerate work that would normally require a set of human experts and a great deal of tedious effort. The structural extraction is straightforward because of the narrow way in which operating systems implement and utilize their internal data structures. The extraction of semantic knowledge is the more challenging task and we have brought to bear several techniques, including syntactic analysis, inference using LLMs, and refinement, to take the first steps in achieving fully automatic relational modeling of the operating system's state. We have shown that our system can correctly identify primary keys within the operating system state, a significant first step along the path to full automation.

Acknowledgments

We thank Brent Pappas and Paul Gazzillo for help in developing the Clang plugin.

References

[1] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. DFD: Efficient Functional Dependency Discovery. In *ACM International Conference on Information and Knowledge Management*, November 2014.

[2] Dina Bitton, Jeffrey Millman, and Solveig Torgersen. A Feasibility and Performance Study of Dependency Inference (Database Design). In *IEEE International Conference on Data Engineering*, February 1989.

[3] Edgar. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), June 1970.

[4] Peter A. Flach and Iztok Savnik. Database Dependency Discovery: A Machine Learning Approach. *AI Communications*, 12(3):139–160, August 1999.

[5] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, January 1978.

[6] Joseph M. Hellerstein and Peter Alvaro. Keeping CALM: When Distributed Consistency Is Easy. *Communications of the ACM*, 63(9), September 2020.

[7] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2), March 1999.

[8] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-Sensitive Program Analysis as Database Queries. In *ACM Symposium on Principles of Database Systems*, June 2005.

[9] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. *Communications of the ACM*, 52(11), November 2009.

[10] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient Discovery of Functional Dependencies and Armstrong Relations. In *International Conference on Extending Database Technology*, January 2000.

[11] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems*, 4(4), December 1979.

[12] Erik Meijer and Gavin Bierman. A Co-Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 54(4):49–58, April 2011.

[13] Noel Novelli and Rosine Cicchetti. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies. In *Proceedings of the 8th International Conference on Database Theory*, January 2001.

[14] osquery. Online. https://osquery.io [Accessed July 2024].

[15] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Very Large Databases Endowment*, 8(10), June 2015.

[16] Thorsten Papenbrock and Felix Naumann. A Hybrid Approach to Functional Dependency Discovery. In *ACM SIGMOD International Conference on Management of Data*, June 2016.

[17] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. DBOS: A DBMS-Oriented Operating System. *Very Large Databases Endowment*, 15(1):21–30, September 2021.

[18] Robert Soulé, Peter Alvaro, Henry F. Korth, and Abraham Silberschatz. Research pearl: The ROSI operating system interface. *CoRR*, abs/2409.14241, July 2024.

[19] Robert Soulé, George Neville-Neil, Stelios Kasouridis, Alex Yuan, Avi Silberschatz, and Alvaro Peter. OSDB: Exposing the Operating System's Inner Database. In *The Biennial Conference on Innovative Data Systems Research*, January 2025.

[20] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7), July 1981.

[21] Catharine Wyss, Chris Giannella, and Edward Robertson. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances Extended Abstract. In *ACM Data Warehousing and Knowledge Discovery*, January 2001.

[22] Hong Yao, H.J. Hamilton, and C.J. Butz. FD_Mine: Discovering Functional Dependencies In A Database Using Equivalences. In *IEEE International Conference on Data Mining*, December 2002.

[23] Hong Yao and Howard J. Hamilton. Mining Functional Dependencies From Data. *Data Mining and Knowledge Discovery*, 16(2), April 2008.