Report From the CoalFace: Lessons Learnt Building A General-Purpose Always-On Provenance System

Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Andy Hopper
University of Cambridge, Computer Lab
{*firstname.lastname*}@cl.cam.ac.uk

Abstract

Over the past year we have implemented OPUS, an always-on system for observed provenance capture in user-space. In this paper we present some important lessons for anyone hoping to implement a general purpose provenance system operating at user-level. In particular, we highlight the problems and solutions associated with the explosion of interposition requirements attributable to function variants, challenges in maintaining semantic equivalence with POSIX and the importance of deactivating function interception in response to runtime errors. We also provide some insights on choosing the right database to manage provenance data.

1 Introduction

For provenance to be useful to a broad range of potential users it must exist as a core component of modern systems. For the implementation of OPUS [1] our always-on general purpose provenance system that runs in user-space we chose to integrate with the runtime environment, collecting provenance from each process as it runs. There are a variety of other approaches to provenance and specific capture methods. With OPUS we focus mainly on data provenance and capturing provenance for POSIX function calls.

In this paper we outline some of the lessons we have learnt in building OPUS. While these are specific to our implementation we advocate that the general principles we outline are useful to others building runtime interposition systems. Specifically we cover function stems and their growing numbers of variants, maintaining semantic equivalence with POSIX, deactivating interposition in case of errors and finally a discussion of storage systems.

2 OPUS

The rate of adoption of provenance systems is currently very low amongst the user community. This is attributable to the following reasons:

I. Users are reluctant to deploy systems that require installation of a new kernel, kernel modules, standalone servers etc., as they consider the upfront installation and configuration cost as overhead.

II. Users think of provenance as insurance [4] since there may not be an immediate necessity to use the provenance data but there is an ongoing cost incurred in the form of an increase in time and space requirements for the execution of applications.

III. Finally, users expect provenance systems to not be application specific but be general enough to cater to a wide variety of application types.

Based on these points our goal in the design of OPUS has been to build a system that is fast, seamless, lightweight, general purpose, always-on, implemented entirely in user space and, implemented using easy to install and configure components. Our vision is to enable provenance as a first class construct of future computation systems.

2.1 General design of OPUS

The current implementation of OPUS targets applications running on GNU/Linux systems. A majority of applications running on Linux dynamically link with the GNU C library and invoke functions in the library to carry out process and I/O operations. OPUS leverages the `LD_PRELOAD` feature provided by the runtime linker to override the application's symbol table and interpose these C library function calls. We prefer this approach to other user space interposition techniques such as ptrace (used by CDE [2]) and FUSE (used by StoryBook [5]) since `LD_PRELOAD` has a lower overhead, requires no extra components or setup and allows function call level capture to be extended to arbitrary third party libraries.

OPUS has a modular design and consists of two major components, a frontend and a backend. The frontend is implemented as a library that captures function level provenance metadata and immediately sends it to the backend. The backend is divided into sub-components that handle incoming connections, process provenance metadata into objects and relations and a storage layer that talks to an embedded graph database, Neo4j [1].

3 Lessons Learnt

3.1 Function Capture: Stems and Variants

We will be starting with the premise that we want to interpose some of the functionality of the system C library. Let us assume that we are interested in the files that a program produces, we are likely then to want to capture the `open`, `close` and `write` functions.

Continuing our example if we are interested in `write` then we would also be interested in `printf` as it is just a more specialised version of `write`. Thus you will need to construct a function to interpose `printf`, however then you will also need another to interpose `fprintf`. Then there are `vprintf` and `vfprintf` that take a slightly different argument form. Also `wprintf`, `wfprintf`, `wvprintf` and `wvfprintf` that take wide char arguments. Figure 1 gives an example of how such a tree of functions can be connected together. We choose to refer to the function at the base of the tree that relies on no other function a 'stem', the rest of the functions in the tree are 'variants'. As you can see the number of functions requiring interposition can quickly grow when there are a lot of variant functions based off the same stem function.

It would seem then that, even with this proliferation of variants, we would still only need to interpose the stem functions as eventually all of the variants will result in calls to the stem functions. This is however not possible due to the symbol attribute PROTECTED. PROTECTED is used by library creators to stop their symbols accidentally being masked by other libraries early in the load order. When symbols are designated PROTECTED and they are referenced locally in the library, these references are computed at compile time, as opposed to the normal procedure of resolving symbols at run-time. Due to this compile time resolution we are not capable of interposing intra-library calls meaning that even if a call to `printf` eventually results in a call to `write` our interposition function for `write` will not be triggered. Due to this inability to interpose calls within a library we have to interpose all of the variants for the stems that we are interested in. Otherwise we will fail to intercept some of the functionality we are interested in.

Lesson Learnt: Even trying to interpose only a small amount of the functionality of the system libraries can easily produce a rapidly growing number of functions requiring interposition. The PROTECTED attribute requires that all of the variants of a stem function must be interposed.

3.2 The Utility of Templating Interposition

As we learned in the previous section, the number of functions requiring interposition in a `LD_PRELOAD` based system can quickly grow.

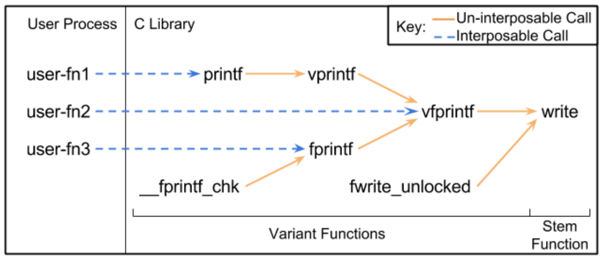

Figure 1: A stem function and a group of its variants. Also shows which calls to a function can and cannot be interposed due to the PROTECTED attribute.

This is a problem as writing code for such a quantity of functions by hand will quickly produce large problems of maintainability. For instance the OPUS system interposes 150 functions from the GNU C library, resulting in approximately 12,000 lines of interposition code. Even in a best case situation an interposition function will require at least 10 lines of code per interposed function as a minimum.

Clearly an automated system is required. In the case of OPUS we chose to use code generation by means of templating. This allows us to maintain a single template and a structured document storing all of the relevant information about a given function such as its name, arguments and return value. A script then reads the structured document and fills out the template for each function substituting values where appropriate and including or omitting code, depending on configuration.

This approach reduces the amount of code for each interposition function from approximately 80 lines of C to approximately 10 lines of structured YAML [2]. This also has the advantage that all of the interposition functions are produced from the template meaning that we can keep the code of all the interposition functions consistent easily. Another advantage this approach provides is that it allows for the system to be extended to new libraries quickly and efficiently, requiring, in the worst case that a supplementary template for the new library be created.

Lesson Learnt: Templating reduces the amount of development effort required per interposition function, reducing maintenance effort and allowing for easy extension of the set of interposed functions.

However there are still functions that do not template well, this is usually due to the function having some side effects that must be handled. For example `signal` requires maintenance of some additional data structures as described in section 3.3.2 or some of the variants on `exec` are modified to persist the environment variables that load OPUS. Even though these functions resist templating we still make use of techniques such as macros to reduce the amount of code duplication.

3.3 Intricacies of POSIX execution

Interposing at the C standard library level imposes a number of challenges. The primary challenge is in maintaining semantic equivalence where, from an application's perspective, there should be no difference in the behaviour expected from making a C library call with or without interposition. In our implementation we attempt to maintain semantic equivalence with POSIX [3] for all standard C library functions being interposed. For most standard library functions adding an interposition layer does not alter the semantics of the function call since the interposition function merely captures the passed arguments, calls the underlying function in the library using its address then records the functions return value and errno value. However, for two cases `vfork` and `signal`, adding an interposition layer breaks the semantics expected by the application for different reasons. These are discussed below along with our solution.

3.3.1 Case 1: vfork

The POSIX standard presents us with computation abstractions such as thread, process etc. The GNU C library provides us with functions that can be used to manipulate these computation abstractions. For example, with the `fork` function call the specification states that a child process can be created with a copy of its parent's resources (fds, memory, environment etc.). From an interposition standpoint it is critical that we preserve this semantic of program execution. OPUS implements this by interposing the call to `fork` and within the interposition function calls the actual C library `fork`. OPUS then collects all required provenance metadata and allows both the parent and child process to continue execution. However, there are other functions such as `vfork` that make it impossible to do this easily. `vfork` was intended as an optimization of `fork` since it does not duplicate the parent's address space, instead both the child and parent share the same address space. The execution of the parent process is suspended until the child either `execs` another binary or terminates by calling `exit`. On modern Linux systems applications no longer have to use the `vfork` optimization since `fork` has been reimplemented using copy-on-write pages [3]. However, we have observed that a number of programs such as `bash`, `gcc`, `make` etc. still use `vfork`, hence OPUS has no option but to interpose the function call.

Since the entire memory address space (including state of the stack) is shared, the POSIX specification does not allow the child process to return from the current function calling `vfork` as it will destroy the shared stack frame. This makes interposition problematic since the interposition function is basically a wrapper on top on the C library `vfork` function. In fact, the standard C library has the problem of preserving the stack frame as well since the `vfork` library function is itself a wrapper for making the `vfork` system call. To solve this problem the C library implementers have to break the function call abstraction provided by their high level language (in this case C) and implement the `vfork` function using assembly code. Within the assembly routine the return address from the function is saved in a register that is preserved across system calls. Thus when the child process returns, even though the stack frame is destroyed the parent process can still read the saved return address from the register.

With the implementation of our interposition function we have to take a similar approach and resort to assembly code. Our initial implementation of `vfork` interposition used the C library approach of stashing the return address in a register, however we found that it is architecture dependant and requires details about system call calling conventions specific to the OS. Our current implementation uses a small assembly routine to read the return address and store it in heap memory. This approach has a higher overhead but makes the architecture specific part of the code much smaller. One special case that requires addressing is to not overwrite the return address value in case of nested `vfork` calls. This is solved by storing the return addresses in a user stack.

Lesson Learnt: When dealing with functions such as `vfork` that are implemented as a close coupling between the architecture and high level standard, maintaining semantic equivalence becomes difficult and can result in having to provide architecture specific implementations of the interposition function.

3.3.2 Case 2: Signal handling

Capturing signalling events received by an application can be valid provenance information especially in the context where the signal causes the application to terminate, either automatically because the signal was not handled or manually where the application calls `exit` within its handler. The OPUS frontend intercepts such signalling events using a default signal handler and records the occurrence of the event. OPUS then calls the application's signal handler if present. OPUS is aware of the application's signal handler as it interposes the signal registration functions (`signal` and `sigaction`).

This approach, although viable, modifies the expected behaviour of the signal registration functions. We found that many programs in coreutils use signal registration functions purely to determine the previous signal disposition (signal handle, masks and flags). The kernel maintains the current signal disposition for each signal and returns it on query. Thus, by registering our (OPUS) signal handler, we were causing the kernel to return an invalid signal disposition from the application's perspective. To maintain semantic equivalence OPUS must be transparent to the application and provide the illusion that it does not exist. OPUS achieves this by maintaining extra state in memory which consists of a lookup table that stores a mapping between a signal number and the current disposition. Hence, when the application queries the previous signal disposition, OPUS can intercept the request and return the correct data directly from the lookup table.

Although some of the functionality in the kernel has been duplicated in our interposition library, the data (per signal disposition) stored within OPUS is different from what the kernel observes. This presents us with yet another side-effect. If interposition is turned off due to a runtime error or manually by the user, deactivating signal interposition becomes complicated. We essentially need to re-register all relevant signals and restore the state seen by the kernel.

Lesson Learnt: Certain functions such as signal registration functions can store state internally. A naive approach to interposing such functions may result in unintended side effects since subtle semantics may be overlooked. In order to maintain semantic equivalence the interposition layer must be carefully implemented to cover all cases.

3.4 Deactivating Interposition

While an interposition system is running it may encounter a situation that requires it to be deactivated; for instance, an unreasonable amount of overhead is being imposed on the users workflow and they would prefer to continue without it. Alternatively, the system has encountered an error sufficiently severe that is cannot continue interposing but not severe enough to prevent the interposed process from continuing. For example, in OPUS we consider losing the ability to communicate with the backend process to be in this category of error.

In these situations we need to find a way to deactivate the interposition functions, however they are very tightly integrated with the function resolution order once they have been called at least once. Due to this property it is very difficult to remove them as you would have to rewrite the symbol addresses in the process procedure linkage table, which, if done incorrectly, will render the process incapable of calling those symbols.

Lesson Learnt: As `LD_PRELOAD` based interposition cannot just be "turned-off" a system based on it should include mechanisms to short-cut interposition functions to their underlying calls as a core design factor.

3.5 Choice of storage system

We have described some of the major challenges faced in implementing the frontend module of OPUS that deals with provenance capture. In this section we will look at the storage system in the backend and how its selection can affect querying capabilities.

The OPUS backend receives raw provenance metadata that is analysed and converted into a directed graph which is then persisted in a database on disk. In the initial implementation of OPUS the database used was LevelDB[4], a NoSQL key-value store that is schema-less, embedded and allows us to model graphs fairly easily. We found that these properties although useful for storage of provenance data did not satisfy our query requirements as the inherent limitation of LevelDB is the lack of support for indexes or query languages.

A user query to obtain the provenance of a given file or process entity typically translates into locating a starting node in the provenance graph and traversing the graph to retrieve the required data. In order to achieve this with LevelDB we have to manually create and maintain the indexes (during storage) and write code in the application layer to express traversals. This can become tedious and inefficient. Therefore there is a necessity for a storage system that provides the ability to index data and a query language to express complex traversals.

Given this requirement, it may be tempting to use an on disk relational database such as SQLite. However the SQL language features supported by SQLite do not provide support for recursive queries and therefore have to be implemented in the application layer. This has led us to choose Neo4j, an embedded graph database that allows us to model graphs naturally, create indexes automatically on node properties and provides a query language (Cypher).

Lesson Learnt: Among existing data store models, graph databases provide the best match for the functional and data representation requirements of general purpose provenance systems.

4 Conclusions

Interposition using LD PRELOAD is a viable method for provenance capture. However the use of this method has highlighted a number of interesting challenges. To build a complete system these challenges have to be addressed and we have shown that they can be solved by careful thinking, planning and effort to implement correctly, if not there can be functional repercussions.

We have also discussed the importance of choosing the right storage system and how certain properties of the storage system influence query capabilities.

Acknowledgements

We acknowledge Sherif Akoush, Lucian Carata, Oliver Chick, George Coulouris and James Snee for their advice on shaping this paper.

References

[1] Balakrishnan, N., Bytheway, T., Sohan, R., and Hopper, A. Opus: A lightweight system for observational provenance in user space. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance* (Berkeley, CA, USA, 2013), TaPP '13, USENIX Association, pp. 8:1–8:4.

[2] Guo, P. J., and Engler, D. Cde: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association.

[3] Love, R. *Linux Kernel Development, Third Edition*. Addison-Wesley Professional, 2010, ch. Process Management.

[4] Seltzer, M. World domination through provenance. Presented at the 5th USENIX Workshop on the Theory and Practice of Provenance.

[5] Spillane, R., Sears, R., Yalamanchili, C., Gaikwad, S., Chinni, M., and Zadok, E. Story book: an efficient extensible provenance framework. In *First workshop on on Theory and practice of provenance* (Berkeley, CA, USA, 2009), TAPP'09, USENIX Association.

Notes

[1]`http://www.neo4j.org/`
[2]`http://yaml.org/spec/`
[3]`http://pubs.opengroup.org/onlinepubs/9699919799/`
[4]`http://code.google.com/p/leveldb/`