Towards Automated Collection of Application-Level Data Provenance

Dawood Tariq Maisem Ali* Ashish Gehani
SRI International

Abstract

Gathering data provenance at the operating system level is useful for capturing system-wide activity. However, many modern programs are complex and can perform numerous tasks concurrently. Capturing their provenance at this level, where processes are treated as single entities, may lead to the loss of useful intra-process detail. This can, in turn, produce false dependencies in the provenance graph. Using the LLVM compiler framework and SPADE provenance infrastructure, we investigate adding provenance instrumentation to allow intra-process provenance to be captured automatically. This results in a more accurate representation of the provenance relationships and eliminates some false dependencies. Since the capture of fine-grained provenance incurs increased overhead for storage and querying, we minimize the records retained by allowing users to declare aspects of interest and then automatically infer which provenance records are unnecessary and can be discarded.

1 Introduction

Provenance refers to the history of ownership and usage of an object. In the context of computation, provenance refers to the source of data, as well as the details of how the data has been used, modified, and transformed over time. Tracking the computational provenance of data has many useful applications, including ensuring the reproducibility of experiments, determining code and data dependencies when sharing research, and estimating the quality of data.

The granularity at which provenance metadata is captured can have significant implications for its usability. Capturing provenance at too coarse a granularity can result in the omission of important details and relationships. Though provenance captured at the operating system level allows visibility into system-wide behavior and ensures that users are not restricted to the views of activity from a single application, the approach suffers from two shortfalls – details within an application are lost, and false dependencies are introduced in the provenance graph.

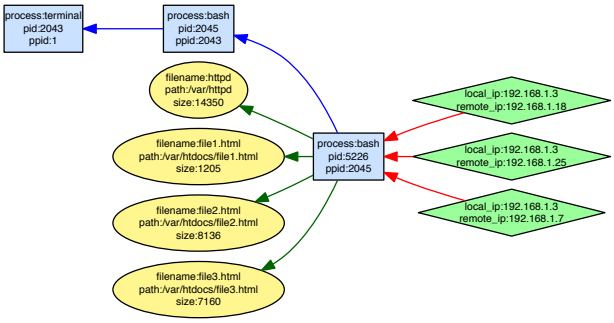

Figure 1: Provenance showing a Web server reading multiple files and transmitting data to multiple clients, as captured at the operating system level by SPADE.

To illustrate the issue, consider a simple Web server that runs as a single process and serves Web pages to clients through incoming network connections. As shown in Figure 1, the data provenance (at the operating system level) of such an application shows a process reading multiple files, and transferring data to multiple connections across the network. While this representation is correct, it fails to allow a user to determine precisely which files were transferred to each connection. As a result of this ambiguity, the provenance relationships indicate false dependencies – they show that each network connection is dependent on all the Web page files that have been read by the server up to the point that a request is served. For long-running processes, this can have cascading effects in provenance queries as the number of false dependencies increases.

Our experience with scientists who manually instru-

*Done while visiting SRI.

mented their scripts to generate application-level provenance confirmed that this requires a substantial effort, is error-prone, requires significant coordination between collaborators, and is hard to maintain. This motivated us to investigate automated application-provenance collection. However, operating system processes can be complex, with multiple threads interacting through filesystems, databases, and networks. While tracking intra-process data provenance allows finer-grained behavior to be recorded, providing a more accurate representation of dataflow, the approach results in increased overhead for collecting, storing, and querying provenance metadata. For example, a single operating system level process vertex representing a Web server would require a complex graph of dependencies between functions and variables to represent intra-process relationships. Further, much of this metadata is not of interest to the user. To address the issue, we allow the user to specify intra-process elements of interest, and then use the information to decide what to retain and which records can be discarded.

2 Approach

Our approach for adding provenance instrumentation to a target application is to insert it during compilation. To achieve this, we use the LLVM [4] framework, which was developed to investigate dynamic compilation techniques. The compiler infrastructure allows both compile-time and link-time optimizations of programs written in a range of source languages, including C, C++, Objective-C, and Java. Using the LLVM toolchain, programs written in these languages can be converted into an intermediate representation, also known as *bitcode*. The LLVM optimizer can then be used to perform built-in and user-defined transformations of the bitcode. Finally, the bitcode can be converted into native objects and linked into an executable for the target hardware architecture.

2.1 Provenance Instrumentation

Intra-process data provenance can be tracked at a range of granularities. We opted to target function call boundaries since they correspond to interfaces most likely to be meaningful to the end user. The LLVM framework includes a *FunctionPass* class that can be used to modify each function in the targeted application. We extended it to create an *LLVM Tracer* class that inserts provenance instrumentation at each function entry and exit. When the LLVM optimizer *opt* is run with the LLVM Tracer pass on a target application's bitcode, it produces a version with instructions inserted to emit the name of the function, its argument types, and argument values on function entry, and the return value on function exit. This results in function-call-level provenance being emitted when the target application executes.

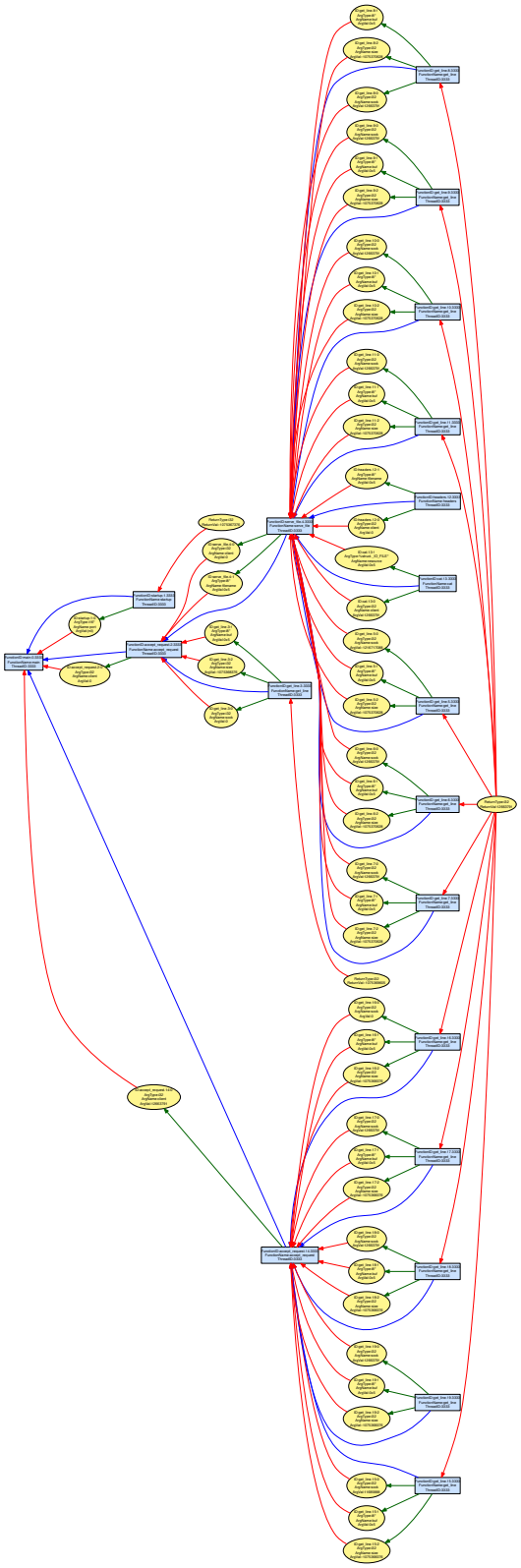

Figure 2: Unfiltered provenance metadata captured using the LLVM Reporter. The graph shows all the function calls and arguments in the instrumented program.

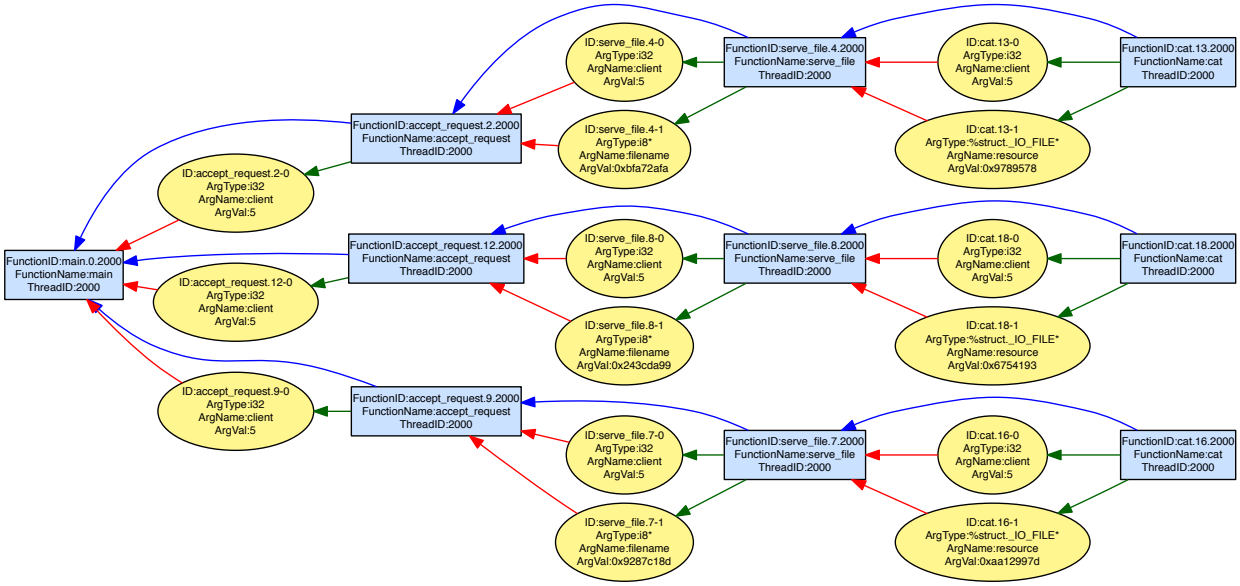

Figure 3: Provenance captured using the LLVM Reporter shows *tinyhttpd* reading files and transmitting data to clients. Since the provenance is recorded at the function call level, network connection artifacts (not shown) can be correlated with the specific files served to the remote clients.

2.2 Collecting LLVM Emittances

SPADE [6] is a modular provenance management infrastructure. Its provenance kernel is agnostic to the domain from which provenance is collected. Domain-specific activity is transformed into a provenance record by a SPADE *Reporter*. The *LLVM Reporter* is a Java class that parses the output of the LLVM Tracer described above, and sends appropriate provenance events to the SPADE kernel.

A SPADE Reporter is an extension of the provenance kernel and runs in the same address space. Since the LLVM Tracer is compiled into the target application, it runs in a different address space. To send provenance metadata from the target application to the LLVM Reporter, our initial design considered printing the metadata to the standard output stream of the application. The approach has significant limitations. First, the provenance metadata is interleaved with the application's output, making interactive applications unusable. Second, applications may fork child processes and then exit. In this case, monitoring will terminate when the application exits and will not extend to any child processes since the original output stream is closed. Third, monitoring will not be possible for servers that close their output streams when launched (as many daemons do).

A more robust approach that addresses the above shortcomings uses TCP sockets to send the provenance metadata. When the LLVM Reporter is loaded by the SPADE kernel, it launches a multi-threaded server that accepts incoming connections. The LLVM Tracer compiled into each target application establishes and maintains a separate connection to the Reporter, sending provenance metadata when function calls and exits occur. Application calls to *close()* the socket are ignored.

2.3 Reporting Application Provenance

To imbue the output with Open Provenance Model (OPM) [5] semantics, we represent a function as an OPM *Process* and the arguments and return values as OPM *Artifacts*. The calling function is responsible for generating the argument artifacts that the callee function uses. The caller is connected to the artifacts with OPM *wasGeneratedBy* edges, whereas the callee is connected to the same artifacts with OPM *used* edges, signifying the dataflow direction. Conversely, the callee function uses these argument artifacts and generates a return value artifact that can be used by the calling function. The return value is connected to the callee with an OPM *wasGeneratedBy* edge and the caller with an OPM *used* edge.

A call stack is maintained in order to determine the calling function for any callee function. Since a program may be running multiple threads, each with its own call stack, it is necessary to monitor the thread identifier for each function entry and exit. Moreover, since a function may be invoked at arbitrary points during the execution of a program as well as by different threads, it is important to distinguish the calling context of the invocations. We define a function identifier based on the

function name, thread identifier, and a counter of function calls in the call stack. This function identifier is used to disambiguate invocations of the same function.

2.4 Minimizing Provenance

One implication of tracking provenance at the function call level is that it may result in a deluge of information, as shown in Figure 2. Users may be only interested in specific functions. For example, consider the case of the Web server described earlier. When the server sends a file to a client, it executes functions that perform the following tasks: accept a request from a client, parse the request, open a local file, read lines from the file into a buffer, generate and send HTTP headers, and send the content of the buffer to the client. Not all of these functions need to be monitored in order to generate useful provenance. Calls to read lines from a file may be invoked numerous times. However, knowledge of these calls is not needed to determine the dependency between a remote client and the files it received.

To minimize the provenance record, we allow users to specify the functions that in which are interested. The call graph of the target application is generated by the LLVM optimizer and used to perform a reverse reachability analysis from the list of desired functions. This is then passed to a SPADE filter, which intercepts all provenance elements being reported to the kernel before they are committed to persistent storage. The filter discards all provenance metadata that is not needed, as shown in Figure 3.

3 Evaluation

To evaluate our approach, we used *tinyhttpd* [7] as a target application. We compared the provenance metadata collected at the operating system level, shown in Figure 1, with provenance gathered at the intra-process level. The steps below were used to add provenance instrumentation to *tinyhttpd*. The same workflow, illustrated in Figure 4, can be used for instrumenting any application.

1. The target application *tinyhttpd* is converted to bitcode with the LLVM C compiler *clang*.

2. The LLVM Tracer module is compiled as a shared library.

3. The LLVM optimizer *opt* is run with the Tracer pass to add provenance instrumentation to the *tinyhttpd* bitcode.

4. The LLVM bitcode compiler *llc* converts the instrumented bitcode into assembly.

5. The socket code for bridging the LLVM Tracer and Reporter is compiled into assembly.

6. The native compiler *gcc* (and implicitly the linker *ld*) is used to compile and link the instrumented application and bridge assembly code into an executable binary that sends provenance metadata to the SPADE kernel.

This workflow is automated for simple applications on Linux and Mac OS X. A user can download SPADE [6], edit the `LLVM_PATH` variable in the `Makefile`, and invoke `make llvm TARGET=<program>` to compile and instrument `<program>`. When the resulting binary is run, it emits provenance to a socket. If the SPADE kernel is running and the LLVM Reporter has been loaded, the provenance events will be read from the socket and recorded.

Figure 3 shows the intra-process provenance collected when running *tinyhttpd* to serve Web pages to multiple remote clients. The storage overhead of the provenance instrumentation was just under 10%, with the binary increasing in size by 2008 bytes from 20136 to 22144 bytes. The execution time overhead was of a similar order of magnitude, and is detailed for four functions in Table 1. We anticipate that this overhead will drop significantly in the case of larger applications where most functions will not be reported if the user remains interested in a small number of functions.

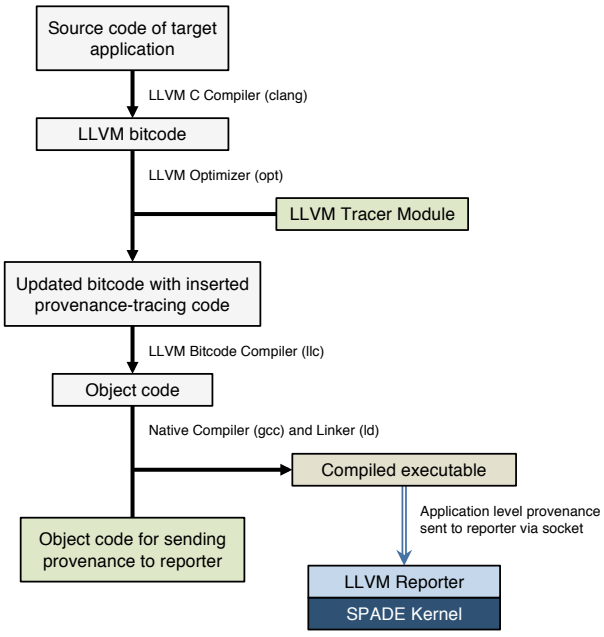

Figure 4: Workflow for collecting application-level provenance of a target program using LLVM and SPADE.

Function	Normal Execution	Traced Execution	Overhead (in μs)	Overhead (in %)
accept_request()	386.3 μs	414.6 μs	28.3 μs	7.33%
serve_file()	253.2 μs	273.9 μs	20.7 μs	8.18%
headers()	23.2 μs	24.6 μs	3.2 μs	13.79%
cat()	70.4 μs	73.9 μs	3.5 μs	4.97%

Table 1: Average execution time and overhead of four *tinyhttpd* functions (calculated with 10 invocations of each).

4 Limitations

While the infrastructure described can be used to obtain data provenance within applications, it has notable limitations. First, the approach requires the user to have access to the source code of the target application. Second, monitoring is limited to function entry and exit points and cannot discriminate between different intra-function behaviors. Monitoring at the level of variable assignments is not employed due to the potential for high processing and storage overhead (though LLVM bitcode's static single assignment could be exploited to reduce this). Third, we obtain visibility only into arguments and return values that are primitive types. For example, we obtain the value of an integer but not of an integer array. Currently, we report pointer values only for complex data types since the source language type information is not available in the LLVM bitcode.

5 Related Work

Related research has been performed in a wide variety of domains under the rubrics of data flow analysis, dynamic slicing, data provenance, database lineage, filesystem lineage, and taint analysis. Some have focused on specific languages or paradigms, such as Python [3], network operations [2], or graphical interfaces [1], whereas our system aims to work with any system and source language that is supported by LLVM (and in turn by *gcc* with the *dragonegg* plugin). Others have tracked dependencies at the X86 binary level, for database lineage [8] and malware tainting [9], for example. However, this is too fine-grained for scientific or other end users to specify their interest (which is typically no more detailed than a variable or function in the source language). The increased overhead of such monitoring precludes its use for interactive applications.

6 Conclusion

We described a method to let applications generate intra-process provenance using the LLVM framework. The framework modifies the bitcode generated from source code, making our approach convenient for users since it does not require them to understand or modify the source code of their applications. While this approach captures provenance only at function entry and exit and does not provide complete intra-application provenance, it can be used as a basis for further work. Specifically, improvements can be made to capture intra-function provenance including variable assignments. In addition to LLVM primitive types as function arguments and return values, support for values in complex data structures and arrays can be added.

Acknowledgments. We thank Ian Mason for his help on separating the provenance and application output streams.

This material is based upon work supported by the National Science Foundation under Grant IIS-1116414. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Steven Callahan, Juliana Freire, Carlos Scheidegger, Cludio Silva, and Huy Vo, Towards provenance-enabling ParaView, *2nd International Provenance and Annotation Workshop*, 2008.

[2] Rodrigo Fonseca, George Porter, Randy Katz, Scott Shenker, and Ion Stoica, X-Trace: A pervasive network tracing framework, *4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[3] Philip Guo and Dawson Engler, Towards practical incremental recomputation for scientists: An implementation for the Python language, *2nd Workshop on the Theory and Practice of Provenance*, 2010.

[4] LLVM, `http://llvm.org/`

[5] Open Provenance Model, `http://www.openprovenance.org/`

[6] Support for Provenance Auditing in Distributed Environments, `http://spade.csl.sri.com/`

[7] tinyhttpd, `http://sourceforge.net/projects/tinyhttpd/`

[8] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, Sunil Prabhakar, Tracing lineage beyond relational operators, *33rd International Conference on Very Large Databases*, 2007.

[9] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall, TaintEraser: Protecting sensitive data leaks using application-level taint tracking, *ACM Operating Systems Review*, 2011.