Final Report:
Institute for Scalable Application Development Software

November 15 2006 to November 2011
DE-FC02-ER25802
(UW account 144-PT48)

Barton P. Miller

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
bart@cs.wisc.edu

1 Technical Accomplishments

Work by the University of Wisconsin as part of the DOE SciDAC CScADS includes the following accomplishments:

- Research on tool componentization, with concentration on the:
  - InstructionAPI and InstructionSemanticsAPI
  - ParseAPI
  - DataflowAPI
- Co-organized a series of high successful workshops with Prof. John Mellor-Crummey, Rice University, on *Performance Tools for Petascale Computing*, held in Snowbird, Utah and Lake Tahoe, California in July or August of 2007 through 2012.
- Investigated the use of multicore in numerical libraries
- Dyninst porting to 32- and 64bit Power/PowerPC (including BlueGene) and 32- and 64-bit Pentium platforms.
- Applying our toolkits to advanced problems in binary code parsing associated with dealing with legacy and malicious code.

1.1 InstructionAPI and InstructionSemanticsAPI

A key project goal has been the "deconstruction of Dyninst" – that is, the identification of key abstract components, design of interfaces for these components, producing libraries that isolate the new components, and then restructuring Dyninst to be based on these new libraries. The result of such an effort is to allow tool builders more flexibility in reusing and sharing tool technology and reducing re-implementations.

Under this funding, we completed the InstructionAPI, the part of binary code parsing that disassembles machine code and provides an abstract representation of each instruction. This representation must be sufficient for serving as the foundation for control and data flow analyses, symbolic disassembly, and code generation and modification. A key part of this design (as with other components in the Dyninst deconstruction) is to create portable and multi-platform abstractions for the interfaces.

The x86/Linux (32 and 64 bit) versions of the InstructionAPI were completed and publicly distributed as InstructionAPI version 1.0. Dyninst was modified to incorporate the InstructionAPI as its foundational instruction parsing mechanism.

The InstructionAPI [14] underwent significant optimization after its initial release in June 2009. We performed extensive profiling and achieved an order of magnitude decrease in the CPU time taken per instruction decode compared to its initial release. This brought decoding performance in line with previous Dyninst instruction decoding, even though it resulted in a much more comprehensive representation of the instructions involved. Key elements of this optimization work included caching, lazy decoding of operands, and memory pools with optimized small object allocators. These optimizations were released as part of Dyninst 6.1.

The InstructionAPI has also added support for the Power/PowerPC instruction sets, including the Double Hummer instructions used on BG/P, and including both 32-bit and 64-bit dialects of PPC machine language. We now fully support the x86 and PPC families of processors. This port was released starting with Dyninst 7.0.

The bind/eval mechanism for evaluating operand abstract syntax trees (AST's) in InstructionAPI was augmented with a visitor interface. Users can extend the visitor class provided in InstructionAPI to perform any algorithm based on a postfix-order traversal of an operand AST. This extension did not require any modification of the AST classes by users to support their algorithms. This allows more general operations on operand AST's than are possible through bind/eval; for example, it is possible to use this interface to translate InstructionAPI AST's to their analogues in another set of AST classes. This functionality was released in Dyninst 7.0.

We also developed a new API that captures instruction semantics. While the InstructionAPI breaks an instruction down into components, providing a machine-independent representation of the operation, dataflow information, and operand calculation, it does not capture the semantics of the instruction. Such semantics are useful for symbolic and partial evaluation, simulation, and testing uses. Within Dyninst, such facilities can give improved resolution of control transfers (call and branch) through pointers, and improved jump and virtual function table parsing. The semantics effort is substantial and involves participants from multiple grants.

The InstructionSemanticsAPI incorporates instruction specification modules from the Rose system at LLNL (directed by Dan Quinlin). This use of Rose's module is a nice demonstration of our increased ability to both share our technology and incorporate that developed by others.

1.2 ParseAPI

Dyninst relies on an interprocedural control flow graph (CFG) abstraction of program binaries to support its analysis and instrumentation facilities. Generation of this CFG from program binaries ("parsing" the binary code) has historically been tightly coupled to Dyninst's internal algorithms and data structures. We separated the parsing algorithms and data structures from the Dyninst library internals. The ParseAPI component presents an interface to programmatically parse binary code, presenting a graph-structured abstraction of the program's control flow. This component is intended to be used directly to retrieve and traverse the structure of binary programs, and to be extended in tools (like Dyninst) that build analysis or other functionality on CFG structures.

The primary control flow abstractions presented by the ParseAPI are Functions, Basic Block and Edges. Functions are analogous to source language procedure constructs. Basic Blocks are defined

as linear sequences of contiguous instructions such that the first instruction dominates the last and the last postdominates the first; that is, the instructions always are executed in sequence as a unit. Basic Blocks are connected by typed control flow Edges, where these Edges are labeled as direct branches, indirect branches, calls, or returns. Functions can be thought of as collections of Basic Blocks, although it is important to note that the ParseAPI supports a "shared code" abstraction in which a given Basic Block can be contained in multiple Functions (this situation occurs in both compiled and library code and typically not handle by other tools).

The underlying parsing algorithms use a recursive traversal strategy to parse from known entry points into the binary code (e.g., the program entry point or function locations retrieved from debug symbols). The parser uses best-effort recovery of function return status to schedule its exploration of the binary, to avoid pitfalls such as non-returning call instructions. An optional speculative heuristic parsing mode will also scan the program binary for recognizable function entry preambles in the remaining "gap" areas after recursive traversal parsing has completed.

The ParseAPI provides various interfaces for retrieving the CFG components of a binary. Functions and Basic Blocks can be looked up by specifying the starting address or by providing an arbitrary address to retrieve all CFG objects that span that point (importantly, ParseAPI supports the notion of overlapping Basic Blocks, which can arise due to overlapping instruction streams on variable length instruction set architectures such as IA-32 and x86-64). Functions can also be looked up by name, provided the underlying program file format supports symbol or debug information. Several interfaces are provided to traverse the control flow graph, either intraprocedurally (such as listing the Basic Blocks in a Function) or interprocedurally by following call or return edges through the graph.

Extensibility is a key feature of the ParseAPI. By extending the Function, Basic Block and Edge C++ classes, users can directly incorporate the ParseAPI's binary code parser into a larger system that uses these CFG abstractions. A callback interface gives fine-grain notification for events during the actual parsing of binary code; users can notified at events such as discovery of call edges, unresolvable control flow (such as computed indirect branch targets), or even with the details of every instruction. The ParseAPI ships with support for ELF, COFF, XCOFF, and PE program file formats, but is designed for easy extension to additional formats though a small "code source" interface. Effectively, any source of binary code that can be opened with mmap can be parsed by the ParseAPI with minimal effort.

1.3 DataFlowAPI

The purpose of the DataflowAPI is to provide the building blocks for more complex dataflow analyses; we seek to provide implementations of concepts (such as abstract locations or slices) that are used in complex code analyses. The end goal is to enable interesting analysis without a lot of reinventions of dataflow wheels.

The DataflowAPI can be divided into two parts: *building blocks* and complete analyses. There are many interactions between the two, as a complete analysis can be used to further refine the data produced by either a building block or a complete analysis. We currently support three building blocks (*abstract locations, slicing, and symbolic expansion*), and one complete analysis:

Abstract locations: the concept of an abstract location (or absloc for short) is to allow an analysis to treat register or memory contents equivalently. We provide a simple absloc implementation that can represent a register, slot on the stack (for a particular function), known address in memory, or

the widened version of each of these. The remainder of the DataflowAPI is based on our absloc representation.

Slicing: many analyses uses the data dependence or control dependence graph rather than the CFG. In our experience, these analyses are more frequently interested in a subgraph that starts (or ends) at a particular known point rather than the entire graph. With that in mind, we have an algorithm that uses search techniques to build such subgraphs, rather than the fix-point algorithm normally used to build the entire graph. While our search strategy is more expensive than the global approach if used to derive the entire DDG, it is substantially less so if users are interested in small graphs. The "slicer" is highly parameterized for such characteristics on when to end or widen the slice and whether to be inter- or intra-procedural. As previously mentioned, it is currently based only on data dependence; this is strictly a result of the uses it has been supporting and not a fundamental problem.

Symbolic expansion: The symbolic expansion engine takes as input an instruction or graph of instructions and produces a symbolic representation of the semantics of this input in terms of an AST tree. This tree can then be analyzed by the user of the DataflowAPI to see if the input instruction(s) have a particular behavior of interest. The advantage of performing such an expansion, rather than operating directly on the instructions themselves, is that the analysis can be written in terms of a machine-independent, limited set of fundamental operations (e.g., add, subtract, or shift) rather than a specific instruction set. Currently Dyninst uses the expansion capabilities for many reasons, including:

1. Detecting modifications of a function's return address;
2. Detecting if the return address is used for any other reason;
3. Statically determining the input parameters to a system call;
4. Concolic (combined symbolic/concrete) execution of a program (Todd);
5. An experimental prototype that determines the targets of an indirect branch or call.

We also provide one complete analysis, a stack analysis that determines the value of the stack pointer and whether the frame pointer exists for each point in a function. In addition to being generally useful, this analysis is a key part of our abstract location derivation.

1.4 Dealing with Legacy and Malicious Code

We developed a novel application of the machine learning technique call *structured classification* to an important problem in binary code analysis: identifying function entry points (FEPs, the starting byte of each function). Such identification is the crucial first step in analyzing many malicious, commercial and legacy software, which lack full symbol information that specifies FEPs. Existing pattern-matching FEP detection techniques are insufficient due to variable instruction sequences introduced by compiler and link-time optimizations. We formulated the FEP identification problem as structured classification using Conditional Random Fields. Our Conditional Random Fields incorporate both idiom features to represent the sequence of instructions surrounding FEPs, and control flow structure features to represent the interaction among FEPs. These features allowed us to jointly label all FEPs in the binary. We performed feature selection and present an approximate inference method for massive program binaries. We evaluated our models on a large set of real-world test binaries, and results showed that our models dramatically outperform two best existing, standard disassemblers.

2 Publications

6. G.L. Lee, D.H. Ahn, D.C. Arnold, B.R. de Supinski, B.P. Miller, and M. Schulz, "Benchmarking the Stack Trace Analysis Tool for BlueGene/L", *Parallel Computing 2007 (Parco), Minisymposium on Scalability and Usability of HPC Programming Tools*, Aachen/Jülich, Germany, September 2007.
7. D.C. Arnold, D.H. Ahn, B.R. de Supinski, G. Lee, B.P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging", *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, California, March 2007.
8. J. Mellor-Crummey, P. Beckman, J. Dongarra, B.P. Miller, and K. Yelick, "Software Technology for Leadership-Class Computing", *SciDAC Review* **5**, Fall 2007, Department of Energy, pp. 36-45.
9. D.H. Ahn, D.C. Arnold, B.R. de Supinski, G.L. Lee, B.P. Miller, and M. Schulz, "Overcoming Scalability Challenges for Tool Daemon Launching", *37th International Conference on Parallel Processing (ICPP-08)*, Portland, OR, September 2008.
10. G.L. Lee, D.H. Ahn, D.C. Arnold, B.R. de Supinski, B.P. Miller, M. Schulz, and B. Liblit, "Lessons learned at 208K: Towards Debugging Millions of Cores", *Supercomputing 2008 (SC2008)*, Austin, TX, November 2008.
11. A. Nataraj, A.D. Malony, A. Morris D.C. Arnold, and B.P. Miller, "In Search of Sweet-Spots in Parallel Performance Monitoring", *2008 IEEE International Conference on Cluster Computing (Cluster 2008),* Tsukuba, Japan, September 2008.
12. Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu, "Extracting Compiler Provenance from Program Binaries", *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Toronto, Canada, June 2010.
13. Paradyn Project. **StackwalkerAPI Programmer's Guide**, University of Wisconsin, Version 1.1.
14. Paradyn Project. **InstructionAPI Programmer's Guide**, University of Wisconsin, Version 1.1.

3 Students Supported and Student Progress

Graduate Students Supported:

Sunlung Suen **Nathan Rosenblum**

No post doctoral nor undergraduate student were supported.

4 Outreach and Transitions

We continue to work with several key groups in government labs, research organizations, academia, and industry. A few recent interactions include:

- BitBlazer project at UC Berkeley, ROSE project at LLNL, and PIN project at Intel: We are worked with several groups, including Dawn Song's group at Berkeley and Dan Quinlan's group at LLNL to integrate full instruction semantics as a layer on top of the InstructionAPI. We are also in discussions with Rob-

ert Cohn at Intel in an effort to unify the Dyninst and PIN instruction decoding.

- TAU group at the University of Oregon: This project used Dyninst to allow instrumentation of binary programs on a variety of HPC systems.
- Barcelona Supercomputer Center: The group used Dyninst for both dynamic and static instrumentation of binary programs for their Paraver performance tools on IBM HPC systems.
- Red Hat now uses Dyninst tool kits to support their SystemTap monitoring primitives, and we are part of the standard Red Hat Enterprise Linux distribution.
- Open|Speedshop: This project, originally started by SGI and now at Krell Institute, funded by NNSA, developed open source performing monitoring tools. We provided the Dyninst tool kits are a foundation for these tools and MRNet was integrated into Open|Speedshop to provide scalable control and monitoring on large-scale parallel systems.
- We held monthly discussions with the IBM BlueGene team, as we extended our various tools to run in the BG/P environment.

5 External Presentations

In addition to conference and workshop papers listed in Section 2 above, the following presentations were given:.

- "Tool Futures: A Look Back and a Look Forward", *DOE Workshop on Software Development Tools for Petascale Computing*, Washington D.C., August 2007.
- **Distinguished Lecturer**, "A Framework for Binary Code Analysis and Static and Dynamic Patching", *Georgia Institute of Technology*, Atlanta, April 2007.
- Invited Talk, "A Framework for Binary Code Analysis and Static and Dynamic Patching", *Los Alamos National Lab*, January 2007.
- Invited Talk, "A Framework for Binary Code Analysis and Static and Dynamic Patching", *University of New Mexico,* Albuquerque, January 2007.
- **Distinguished Lecturer,** "A Framework for Binary Code Analysis and Static and Dynamic Patching", *Ben-Gurion University*, Israel, January 2008.
- Invited Lecturer, "Scalable Middleware for Large Scale Systems", *The Technion*, Israel, January 2008.
- Invited lecture: "Tree-based Overlay Networks for Scalable Applications and Analysis", *University of Colorado, Computer Science Dept.*, March 2008.
- W. Williams, "Deconstructing Dyninst: The Instruction API", *CScADS Workshop on Performance Tools for Petascale Computing*, Snowbird, Utah, July 2008.
- Invited lecture: "Scalable Middleware for Large Scale Systems", *International Advanced Research Workshop on High Performance Computing and Grids*, Cetraro, Italy, June/July 2008.
- Invited Talk, Universitat Autònoma de Barcelona, "Hybrid Static and Dynamic Analysis of Analysis-Resistant Program Binaries", October 2009
- Invited Talk, Epic System, Madison, Wisconsin, "Random Testing with 'Fuzz': Almost 20 Years of Finding Bugs", March 2009.
- Panel Speaker, IGT2008: World Summit of Cloud Computing, Ramat Gan, Israel, "Cloud Security", December 2008.
- Invited talk, Hebrew University of Jerusalem, "Tree-based Overlay Networks for Scalable Middleware and Systems", December 2008.
- Invited Talk, Forschungszentrum Jülich, Germany, "Random Testing with 'Fuzz': 18 Years of Finding Bugs", October 2008.
- Distinguished Lecture, Google, Seattle, "Scaling Up to Large (Really Large) Systems", August 2010
- Keynote Speaker, Workshop on Large-scale Systems and Applications Performance (LSAP2010), in

conjunction with HPDC, Chicago, June 2010.
`http://www.lsap2010.org/`

- Keynote Speaker, Open Grid Forum 28, Munich, Germany, March 2010.
- Invited Talk, Computer Science Department, University of California, San Diego, "Scalable Middleware for Large (Really Large) Scale Systems", February 2010.
- Invited Talk, Computer Architecture and Operating Systems Department, Autonomous University of Barcelona, "Hybrid Static and Dynamic Analysis of Analysis-Resistant Program Binaries", October 2009.