Analyzing Program Behavior with Cross Layer Information

Avi Saven[1]

[1]University of Rochester

Introduction

- Programs are highly measurable objects
  - The code itself can be measured
  - Various performance measurements can be taken
  - How the program is executed can be recorded (program traces, memory traces)
- These measurements are highly connected
  - Performance measurements happen during the execution of the binary code
  - The binary that is being executed is derived from source code
  - These points identify the two ways we analyze software: runtime and static behavior
- These measurable aspects of programs, we call **program behavior**
  - Practically any measurable aspect of a program
  - Dynamic Information such as cache performance, or, memory/register traces
  - Static information such as source code, abstract syntax trees, or disassembly
- There are existing tools that measure certain program behaviors
  - *cachegrind*, *drcachesim* can measure cache behavior during runtime
  - *valgrind*, *drmemory* can detect memory safety errors during runtime
  - *TEMU*, *BitBlaze* can generate instruction and memory traces by emulating the program

Motivation

- Current tooling is inconsistent, and requires custom code to interpret and merge results
- Tooling becomes specialized to specific analyses and becomes difficult to reuse

Methodology

- We propose a new framework for the representation and querying of program behavior
- This is designed with a few important goals in mind
  - Bring together tooling into one unified representation that can be reused throughout various analyses
  - Allow for all queries to be written in a single query language
- Connect static information and dynamic behaviors of a program into one labeled graph
- Allow queries on the graph to facilitate behavioral evaluation using a query language
- Use static information to augment dynamic behaviors for analysis

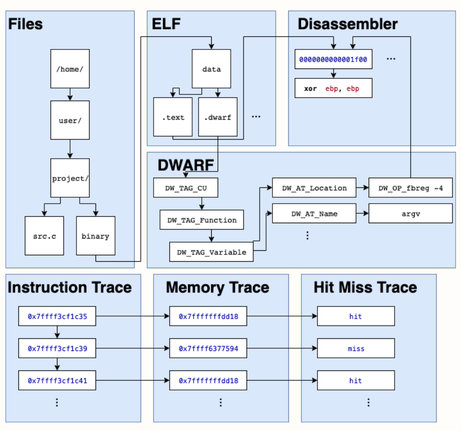


Figure 1: Visualization of a PBG

Deriving Graphs

- Graph nodes are derived from existing tools
- Graph labels are picked based on the semantic relationship between information
- DynamoRIO used to dynamically instrument binaries to collect runtime information
  - Instruction Traces
  - Memory Traces
  - Allocation Traces (malloc/free/realloc/calloc)
- Traces require a notion of time in the graph:
  - Program counters aren't sufficient representation of time: they are reused in loops and functions called more than once
  - Rather, temporal nodes are added to the graph: a monotonically increasing node with connections to trace information to give a time reference
- Uses Capstone [5] to generate the disassembly of a binary
- Imports DWARF information into the graph for type/function/source and assembly conversion information
- **All items become interconnected in the labeled graph**:
  - Cache misses and memory allocations, are connected to a temporal counter in the graph
  - Temporal counters are connected with the instruction and memory trace in the graph
  - Program counters are associated with source and type information
  - Cross-layer information becomes connected by their semantic relationships.
- Analyses which required custom tooling and instrumentation now are queries on a graph.

Querying Graphs

- With this framework we can answer questions about the behavior of our programs using queries
- Several query languages exist for exploring relational data
  - *Gizmo* [6] is an ES5 JavaScript-based query language, provided natively by the utilized graph database Cayley [6]
  - *Soufflé* [7] is an implementation of Datalog, which is a subset of Prolog used for databases and static analysis, it requires an export of the graph to Datalog facts

Case Study #1

- TCC is a small, self-hosting, C compiler by Fabrice Bellard
- Using a PBG, can we answer the question: what line of TCC has the worst cache performance?
- Traditionally, this is done by programs such as *cachegrind* or *drcachesim*
  - However, these are specialized utilities, and it's difficult to reuse the data produced by these tools
  - Additionally, they may require more tooling to answer more specific questions
- We can answer this question using a PBG:
  - Cache trace, instruction trace, disassembly, and, source code are in the graph
  - Finding which line has the worst performance becomes a query on the graph
  - Can be trivially extended to ask more questions about the cache performance
    - Such as: which function has the worst cache performance? which type has the worst cache performance?

Case Study #2

- Valgrind [4] is a runtime analysis utility for finding memory safety errors
- What information does Valgrind use?
  - Instruction trace: What instructions are executed, and when
  - Memory trace: What memory addresses are accessed, and when
  - Allocation trace: What memory is allocated, and when
- These are all pieces of information already in the PBG
  - We can check for memory errors and leaked memory through a query on the PBG
  - Note: not at runtime, rather, is a post-mortem check for memory errors
- Additionally, because the analysis only a query, we can extend it to include new information:
  - *Useless Reallocations*
    - Memory is often grown/extended in anticipation of new data being added
    - However, if it is grown and not written to, then the growth was a waste
    - By a small extension to our query, we can catch instances of this

Results

- Tests were run a server with the following configuration:
  - **CPU:** AMD Ryzen Threadripper 2950 @ 2.2GHz
  - **RAM:** 32GB
  - **HDD:** TOSHIBA MG03ACA4 4TB 7200RPM Disk
- Tests were executed on the following selection of software
  - *basic:* demo program for source/binary validation (6 LOC)
  - *basic_malloc:* demo program for memory allocation issues (8 LOC)
  - *structs:* demo program for DWARF validation (12 LOC)
  - *tcc_forth:* a TCC compilation of a small Forth interpreter by Leif Bruder. (1124 LOC)
  - *sqlite3:* a TCC compilation of SQLite 3 (228449 LOC)
- The testing was executed in three stages on each program
  - First, the PBGs were formed (stage *c*)
  - Secondly, a datalog export was made (stage *d*)
  - Thirdly, a query is run on the exported datalog to detect memory issues (stage *q*)
  - Time spent in userspace, average/maximum memory usage is collected for each stage

Figure 2: Time of PBG

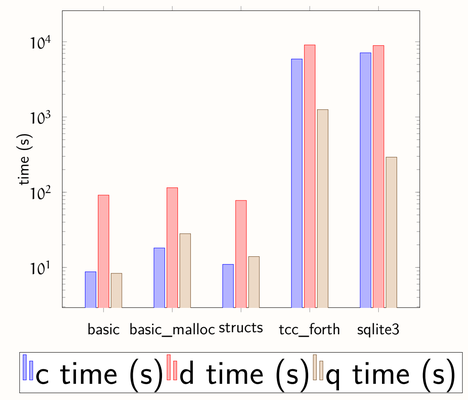
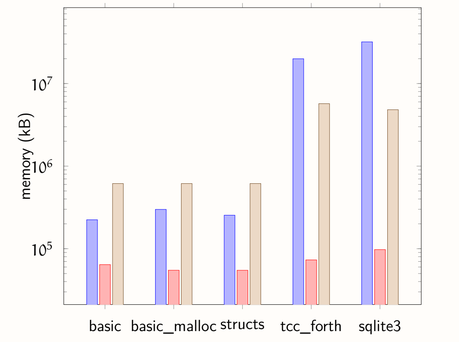


Figure 3: Memory usage of PBG



Conclusion

- We're able to correlate cross-layer information into a single unified representation
- Using the unified representation, it is possible to execute analyses through queries on the graph, in order to answer questions without new tools
- Issues come from scale:
  - Large program execution generates lots of information, which leads to performance issues in generating and querying PBGs
- In the future, it would be beneficial to time/space if we can take information out of the graph and derive it at query time.

References

[1] https://bellard.org/tcc/

[2] http://dynamorio.org/dynamorio_docs/page_drcachesim.html

[3] Song, Dawn, et al. "BitBlaze: A new approach to computer security via binary analysis." *International Conference on Information Systems Security*. Springer, Berlin, Heidelberg, 2008.

[4] Nethercote, Nicholas, and Julian Deward. "Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42.6 (2007): 89-100

[5] http://www.capstone-engine.org

[6] https://cayley.io

[7] Jordan, Herbert, Bernhard Scholz, and Pavle Subotić. "Soufflé: On synthesis of program analyzers." *International Conference on Computer Aided Verification*. Springer, Cham, 2016.

Undergraduate Research Competition, April 17th 2020, Rochester, NY