Loom: Weaving Instrumentation for Program Analysis

MEMORIAL UNIVERSITY

Brian Kidney and Jonathan Anderson

What is Loom?

Loom is an instrumentation framework built upon the LLVM compiler tool chain. It is designed to allow for instrumentation of existing software without the need for modification of the original source code. This is achieved by modification of software, with an LLVM optimization (opt) pass in intermediate representation (IR) format, before the final machine code is produced. Loom provides both a stand-alone opt pass (using user defined instrumentation policies) and a API upon which custom tools can be written to perform instrumentation or transformation. Loom was originally developed as part of the CADETS Project, a part of the DARPA Transparent Computing program, used for instrumenting the FreeBSD operating system for defensive security applications.

How Does Loom Work?

Existing instrumentation frameworks often rely on custom build processes or specific binary formats. X-Ray required customizations to the compiler to insert instrumentation points that can be enabled through a runtime [1]. CSI requires the end user to use link-time optimization (LTO) in order to remove unused instrumentation hooks that would otherwise have an effect on application performance [2]. And Intel's Pin only work with binaries from specific architectures [3]. Loom is implemented as an LLVM optimization pass, allowing it to be used as part of an LLVM build process without frontend or backend modifications.

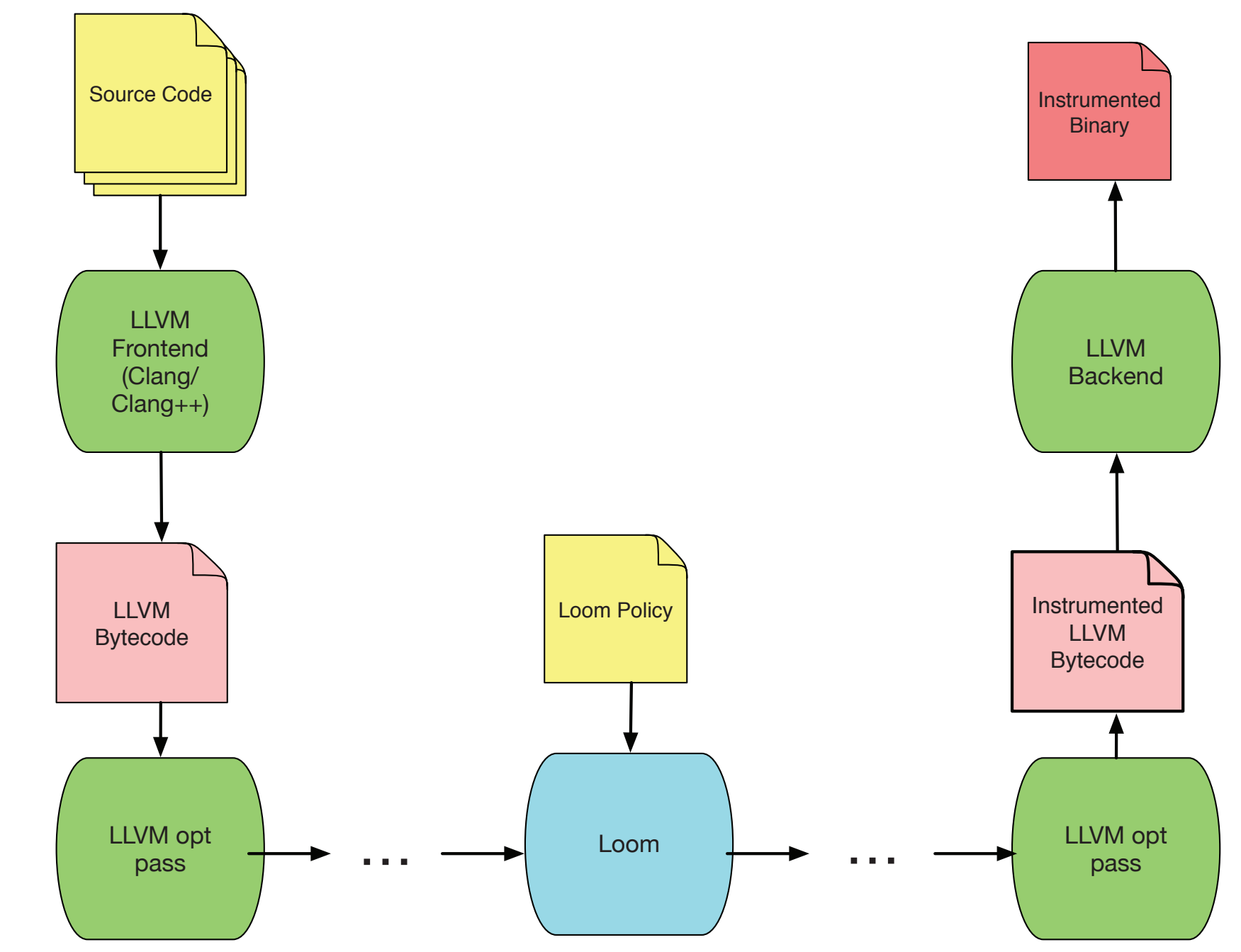


Figure 1: Loom Build Process

Loom can be invoked from the command line either as part of a LLVM build process or built into a custom tool using our API. When invoking Loom at the command line, the user must provide a policy file defining what to instrument and how to output the results. This file uses YAML syntax with declarations provided by Loom.

An example of a custom instrumentation tool, llvm-prov, with Loom. The goal of LLVM-prov was to add provenance information to data, allowing it to be traced through system calls. The tool rewrites system calls in the augmented program allow the addition of the provenance information to systems calls called MetaIO.

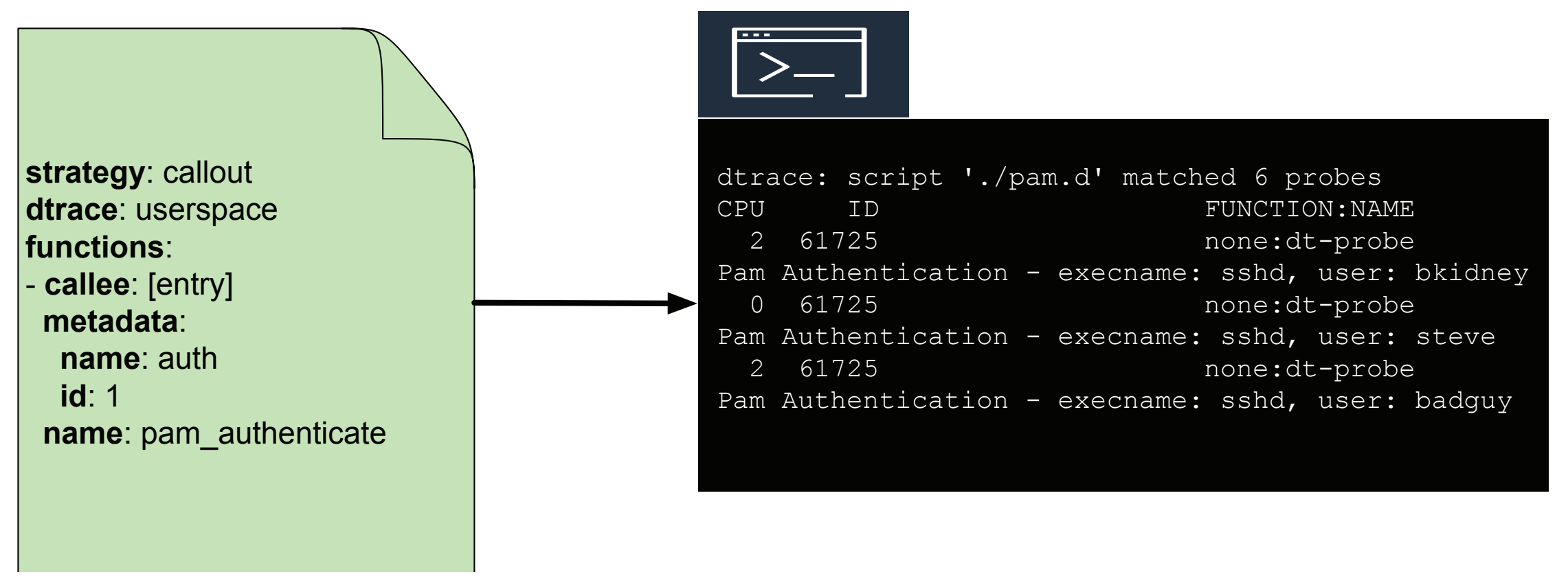
Once the software has been instrumented it undergoes the final stage of compilation. The final application adds instrumentation at the points of interest in the software are reached and the added code is executed. Output of the instrumentation depends on the logger defined by the user.

```
$ make gzip.full.bc
cc -emit-llvm  -pipe -g -std=gnu99 -fstack-protector-strong -Wsystem-headers
-Werror -Wall -Wno-format-y2k -W -Wno-unused-parameter -Wstrict-prototypes
-Wmissing-prototypes
-Wpointer-arith -Wreturn-type -Wcast-qual -Wwrite-strings -Wswitch -Wshadow
-Wunused-parameter -Wcast-align -Wchar-subscripts -Winline -Wnested-externs
-Wredundant-decls
-Wold-style-definition -Wno-pointer-sign -Wmissing-variable-declarations
-Wthread-safety -Wno-empty-body -Wno-string-plus-int -Wno-unused-const-variable
-Qunused-arguments
-c gzip.c -o gzip.bco
llvm-link -o gzip.full.bc gzip.bco
```

Figure 2: Example of FreeBSD IR Build

Logging PAM Authentication Calls

As part of the CADETS project we wanted to be able to instrument the Pluggable Authentication Module (PAM) libraries to capture details of login attempts to FreeBSD. This was achieved by instrumenting the PAM shared library to log all calls to its authentication methods and their parameters.



```
strategy: callout
dtrace: userspace
functions:
- callee: [entry]
  metadata:
    name: auth
    id: 1
  name: pam_authenticate
```

```
dtrace: script './pam.d' matched 6 probes
CPU     ID                    FUNCTION:NAME
  2  61725                    none:dt-probe
Pam Authentication - execname: sshd, user: bkidney
  0  61725                    none:dt-probe
Pam Authentication - execname: sshd, user: steve
  2  61725                    none:dt-probe
Pam Authentication - execname: sshd, user: badguy
```

Figure 3: Policy File and Output of PAM Instrumentation

FreeBSD has been using LLVM as its default compiler since version 10. As part of this work the build system was expanded to allow developers to build IR versions of both applications and libraries. Leverging the ability to build shared libraries to IR we were able to use Loom and a policy file like the simplified version below instrument the calls of interest. With the binary instrumented, all logins using PAM were logged to our tracing system through the instrumentation injected by Loom.

Transforming System Call APIs

An additional goal of the CADETS work was to allow for tracking of the provenance of userspace data as it flowed through system calls. To do this MetaIO versions of the standard POSIX system calls were created by extending the existing functions to include metadata. Loom was then used to transform existing software, replacing the original system calls with there MetaIO versions.

Transforming code that makes systems calls against one API, in order to make calls against a slightly different API, is a more sophisticated use of instrumentation, one not achievable with the Loom policy file interface or with other instrumentation tools. In the current implementation of Loom this work required the use of the Loom API to build a custom tool to transform the code. Domain specific code was written to identify the calls that needed to be transformed and generate the provenance data. This code then emits the transformed the calls using Loom's Instrumenter::Extend method to perform the instrumentation. Work is currently being done to add an instrumentation and transformation language to Loom enabling this type of instrumentation to be done without custom development.

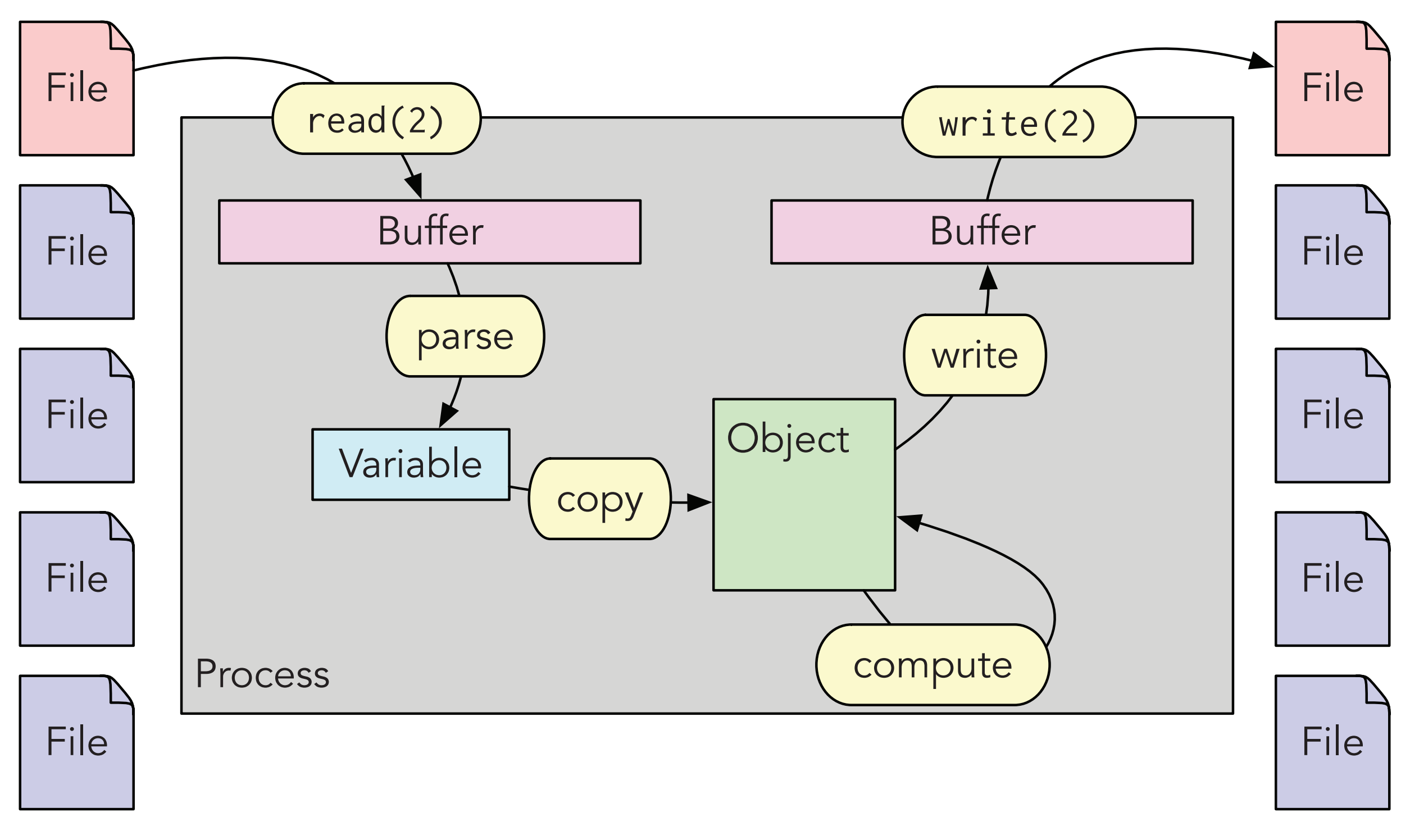


Figure 4: Tracing Data Through an Application

Loom Policy File

In order to define the instrumentation points in an opt pass, Loom uses a policy file. This file is plain text and uses a standard YAML format. The following parameters can be defined for instrumentation.

General Configuration		
strategy	Defines an instrumentation strategy	Options: callout, inline
hook_prefix	Prefix added to callout function names	default: __loom
Logging Configuration		
logging	Use a simple logger	Options: none, printf, xo (json)
ktrace	Log to KTrace	Option: utrace
dtrace	Log to DTrace	Option: userspace
serialization	Serialization to use with logger	Option: nv
Instrumentation Configuration		
function	Specify functions to instrument	Options: name, caller, callee
structures	Specify structures to implement	Options: name, fields (name, read/write)
globals	Specify globals to instrument	Options: name, fields (name, read/write)
pointerInsts	Instrument all pointer instructions	
Additional Instrumentation Configuration (currently functions only)		
metadata	Add additional metadata to instrumentation data	Options: id, name
Transforms	Transform a value before logging	Options: arg, fn

Loom API

The Loom library can also be integrated directly into other software and invoked via API calls. This is the approach employed by the LLVM-Prov tool. This mode of operation allows for customized instrumentation to be injected into target bitcode without re-inventing Loom's generic instrumentation strategies.

When using Loom as a library, a client constructs an InstrStrategy object that describes how instrumentation should be injected, then uses that to construct a Loom Instrumenter. The Instrumenter can then be used to apply instrumentation to LLVM abstractions including function calls, function definitions, structure field accesses and even all low-level instructions. The Instrumenter can also be used to extend function calls, translating the usage of one API into an extended API that requires additional arguments and uses different function names. Methods of an Instrumenter object can, therefore, be used to describe what should be instrumented; InstrStrategy, Instrumentation and Logger objects are used to describe how they should be instrumented.

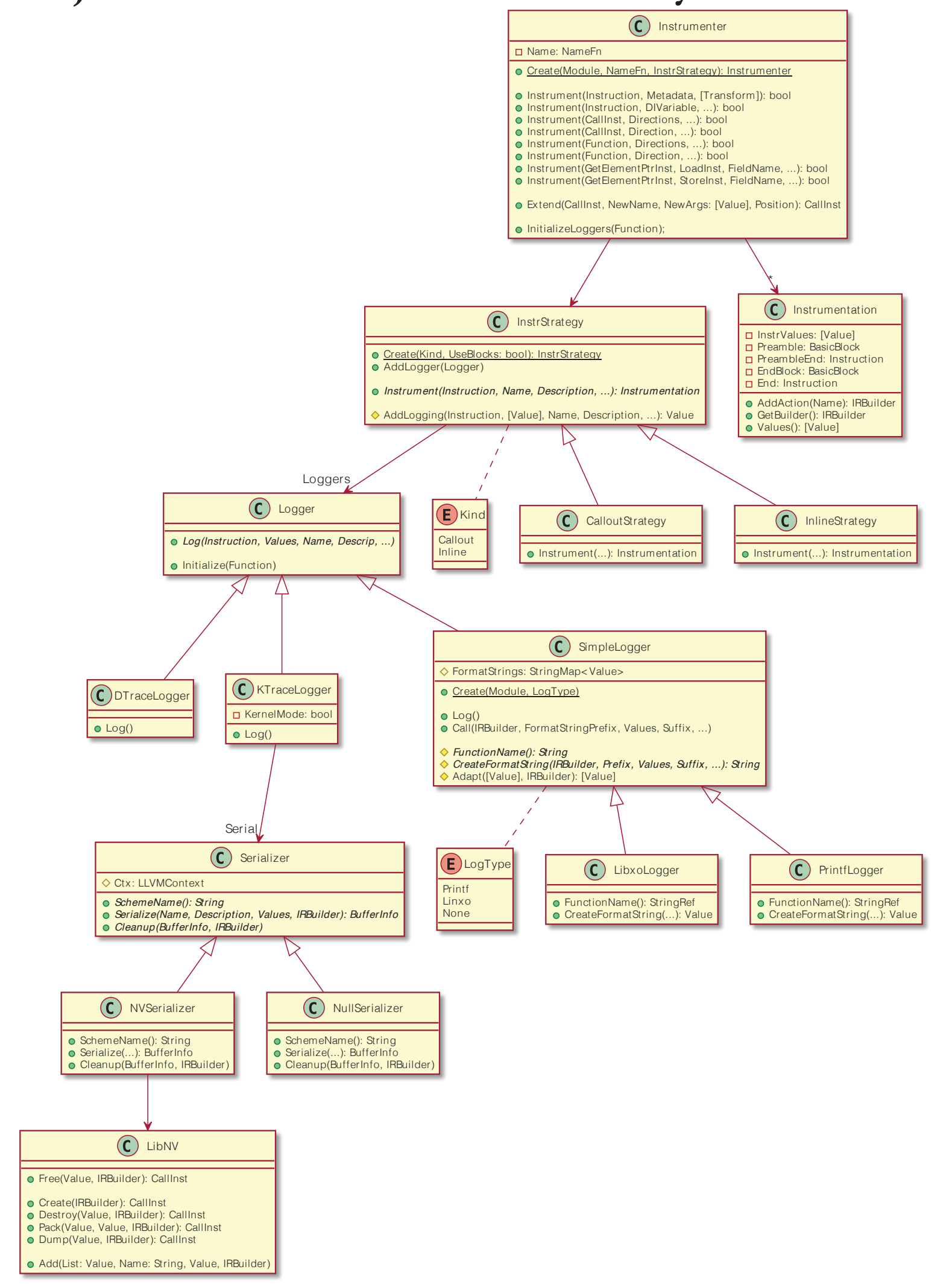

Instrumenter
Name: NameFn
Create(Module, NameFn, InstrStrategy): Instrumenter
Instrument(Instruction, Metadata, [Transform]): bool
Instrument(Instruction, DIVariable, ...): bool
Instrument(CallInst, Directions, ...): bool
Instrument(CallInst, Direction, ...): bool
Instrument(Function, Directions, ...): bool
Instrument(Function, Direction, ...): bool
Instrument(GetElementPtrInst, LoadInst, FieldName, ...): bool
Instrument(GetElementPtrInst, StoreInst, FieldName, ...): bool
Extend(CallInst, NewName, NewArgs: [Value], Position): CallInst
InitializeLoggers(Function);
InstrStrategy
Create(Kind, UseBlocks: bool): InstrStrategy
AddLogger(Logger)
Instrument(Instruction, Name, Description, ...): Instrumentation
AddLogging(Instruction, [Value], Name, Description, ...): Value
Instrumentation
InstrValues: [Value]
Preamble: BasicBlock
PreambleEnd: Instruction
EndBlock: BasicBlock
End: Instruction
AddAction(Name): IRBuilder
GetBuilder(): IRBuilder
Values(): [Value]
Loggers
Logger
Log(Instruction, Values, Name, Descrip, ...)
Initialize(Function)
Kind
Callout
Inline
CalloutStrategy
Instrument(...): Instrumentation
InlineStrategy
Instrument(...): Instrumentation
DTraceLogger
Log()
KTraceLogger
KernelMode: bool
Log()
SimpleLogger
FormatStrings: StringMap<Value>
Create(Module, LogType)
Log()
Call(IRBuilder, FormatStringPrefix, Values, Suffix, ...)
FunctionName(): String
CreateFormatString(IRBuilder, Prefix, Values, Suffix, ...): String
Adapt([Value], IRBuilder): [Value]
Serial
Serializer
Ctx: LLVMContext
SchemeName(): String
Serialize(Name, Description, Values, IRBuilder): BufferInfo
Cleanup(BufferInfo, IRBuilder)
LogType
Printf
Linxo
None
LibxoLogger
FunctionName(): StringRef
CreateFormatString(...): Value
PrintfLogger
FunctionName(): StringRef
CreateFormatString(...): Value
NVSerializer
SchemeName(): String
Serialize(...): BufferInfo
Cleanup(BufferInfo, IRBuilder)
NullSerializer
SchemeName(): String
Serialize(...): BufferInfo
Cleanup(BufferInfo, IRBuilder)
LibNV
Free(Value, IRBuilder): CallInst
Create(IRBuilder): CallInst
Destroy(Value, IRBuilder): CallInst
Pack(Value, Value, IRBuilder): CallInst
Dump(Value, IRBuilder): CallInst
Add(List: Value, Name: String, Value, IRBuilder)

Figure 5: Loom Class Structure

Benchmarking

To measure the overhead of Loom as compared to other instrumentation frameworks we wrote a micro-benchmark around instrumenting function calls. The benchmark is a set of nested calls that are called in a loop. Measurement of the total program time is made using clock_gettime(3) within the program. The program is compiled and instrumented using CSI, DTrace USDT, XRay and Loom. The versions are then run on a Xeon E5-2407 service with 16GiB ram, running FreeBSD with no additional applications. Test are run 10 times and the results are compiled below.

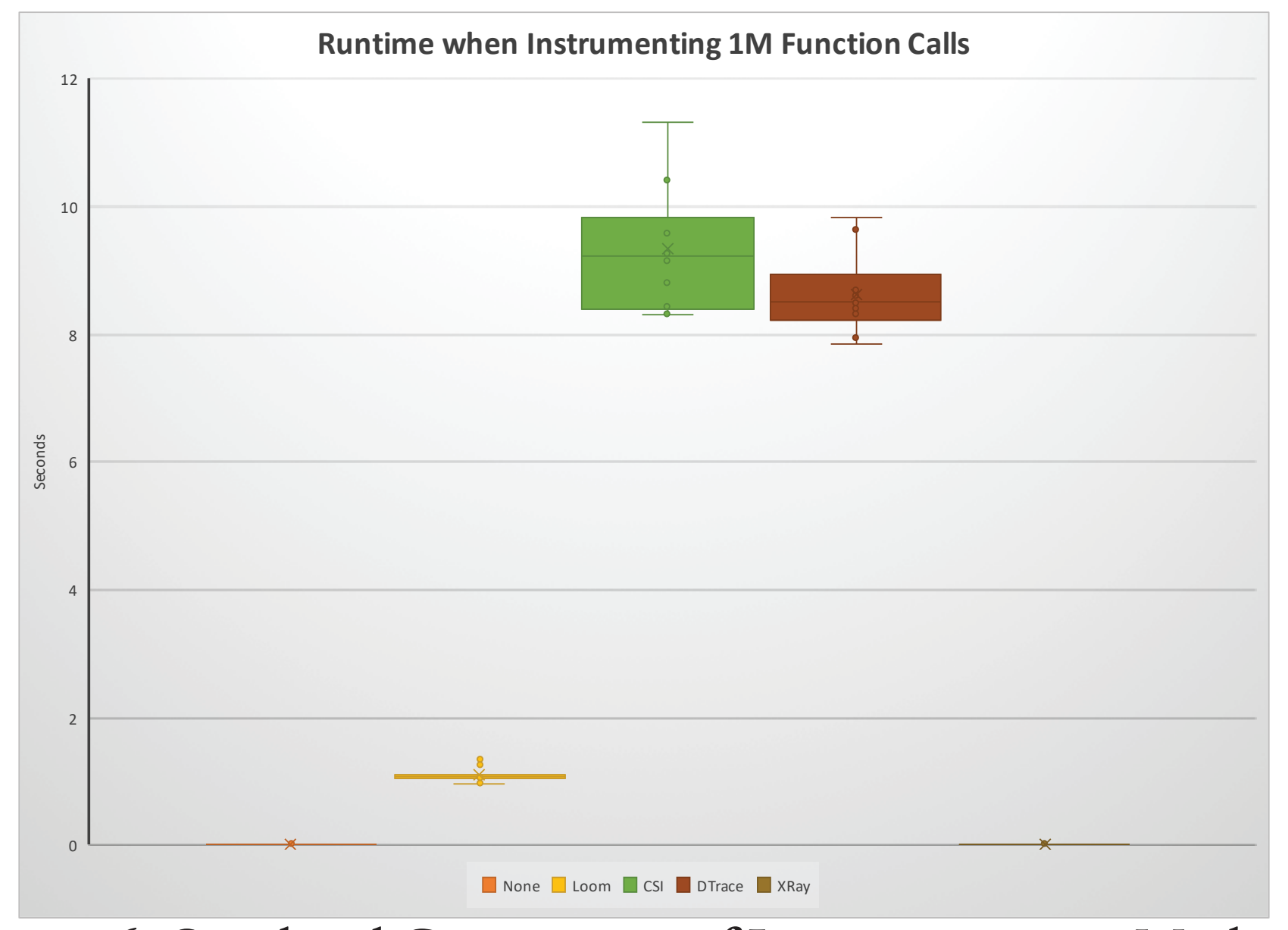


Figure 6: Overhead Comparison of Instrumentation Methods

The overhead for XRay is relatively non-existent compared to the other methods. This is due to the fact that is generates staticstics during the running of the program without reporting them. Running an additional tool is required to report the results.

Conclusions

The Loom instrumentation framework expands the functionality of traditional software instrumentation. Loom allows the user to probe inside of a running program to gather information for an expanding set of use cases, including debugging and security tracing. By leveraging LLVM all of this is achieved at the IR level without having to modify the source code, and with no custom front-end or backend for the compiler. As we have show, the additional overhead compares favorable to that added by other instrumentation tools.

References

[1] Dean M. Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. 2016. XRay: A Function Call Tracing System. Technical Report. A white paper on XRay, a function call tracing system developed at Google.

[2] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2017. The CSI Framework for Compiler-Inserted Program Instrumentation. Proc. ACM Meas. Anal. Comput. Syst. 1, 2, Article 43 (December 2017), 25 pages. DOI: https://doi.org/10.1145/3154502

[3] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05). ACM, New York, NY, USA, 190-200. DOI: https://doi.org/10.1145/1065010.1065034

Acknowledgements

The authors gratefully acknowledge the support of the NSERC Discovery program (contract RGPIN-2015-06048).

This work was also funded under DARPA/AFRL contract FA8650-15-C-7558 ("CADETS"), as part of the DARPA Transparent Computing (TC) research program. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.