Security R&D Projects using LLVM

Quarkslab

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Carnegie
Mellon
University

Penn

NC STATE UNIVERSITY

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Verimag

UNSW SYDNEY Engineering
Computer Science and Engineering

POLITECNICO
MILANO 1863

UNIVERSITY OF
CAMBRIDGE

PRINCETON
UNIVERSITY

Grenoble INP

NASA

Microsoft

THE
UNIVERSITY
OF UTAH

deep
spec

université
PARIS-SACLAY

UNIVERSITY
OF TWENTE.

galois

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

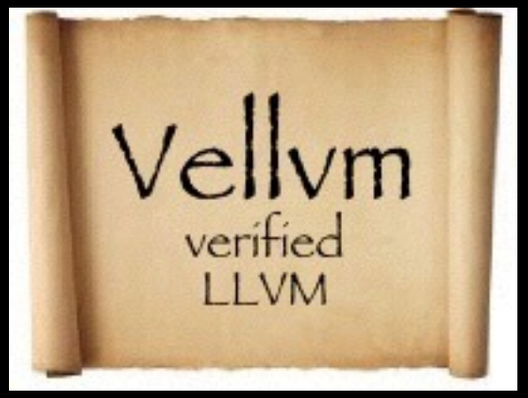
informatics mathematics
Inria

institute
IMDEA

VUSec

TRAIL OF BITS

UNIVERSITÉ
Grenoble
Alpes



Vellvm: Verified LLVM

- Model syntax and semantics of LLVM IR so as to…
- Reason about code expressed in IR in order to…
- Prove properties about LLVM passes in Coq

- OCaml extracted, ran unit tests validating model
- Prior work: reasoning on Instruction Combiner pass
- Recent work: reasoning on FastTrack (TSan)
  - Toward proof of implementation: 18900 LoC in proofs

Hybrid static/dynamic “sanitizers”

AddressSanitizer	• RT lib replaces malloc/free/etc • UAF, UAR	• “asan-module” ModulePass • “asan” FunctionPass • Taint/poison memory • Analyze those points on use • Instruction Visitor collect return loc’s (UAR)
MemorySanitizer	• Detect reads of uninit memory • Subset of valgrind	• “msan” FunctionPass • Instrument each function to taint a few bits of app memory • Instruction Visitor does much of the work to add propagation elements
ThreadSanitizer	• Detect race conditions & deadlocks	• “tsan” FunctionPass • Instrument load/store insts to track read and writes to mem
DataFlowSanitizer	• Allows for create your own dynamic taint flow analysis	“DFSan”
DangSan	• Detect UAF (at scale)	• instruments programs written in C or C++ to invalidate pointers whenever a block of memory is freed
TypeSan	• Check casts in C++ code	

Dynamic analysis: fuzzing

Project	Action	LLVM Usage
LibFuzzer	• Dynamic analysis • Evolutionary fuzzer engine • In-process, coverage guided • Wrap code to make target functions	• Target code instrumented with DFSan and instruction tracing • Taint applied to every byte of input
American Fuzzy Lop (AFL)	• Dynamic analysis • Genetic Algorithm based Fuzzer • Instruments code for guided coverage	• LLVM mode option allows for instrumentation via IR • Implemented as a ModulePass • Uses IRBuilder
Iafindel's compare splitter	• Improve getting pass multi-byte compares by splitting them • See "Circumventing Fuzzing …" in projects appendix listing	• Split compares module pass • Split switch statements module pass • Work toward strcmp, memcmp's
TokenCap	• Similar to above • Find tokens (magic values) etc • Use that to feed your fuzzing	• Implemented as a ModulePass

Static Program Analysis

Project	Action	LLVM Usage
Static Value Flow (SVF)	• Static program analysis • Andersen's Alias Analysis • Interprocedural value flow • Build your own program analysis	• Uses LLVM as means for all analysis • Lifting to simplifying auto-analysis • Decomposing some GetElementPtr uses
IKOS	• Static Program analysis • find buffer overflows, etc) • Abstract Interpretation	• Function pass that lifts to ARBOS IR (a text based rep) • Module pass that determines read-only globals and *lowers* them to constants • Lower select to 3 BB and a phi node

Infer	• Static Program analysis
	  • Memory leaks
	  • Null deref
	  • Resource leak
	• Build-time analysis
		• Clang static analyzer plugin

UNSW Engineering
Computer Science and Engineering

Lifting MC to IR

Project	Action	LLVM Usage
Mc Sema, remill	• Binary patching + instrumentation • Reverse engineering • symbolic exec • Bug finding • Retargeting • Worth reading the CGC blog post **&&**	• Control flow reproduction in IR • IR code generation • MC to IR translation • compilable
rev.ng	• Reverse engineering • Symbolic|concolic execution • Worth reading llvm devmtg slides **&&**	• Translation from QEMU to IR • IR code generation
fdc	• Reverse engineering • Output C-like pseudocode with IR	• IR code generation • MC to IR translation
s2e	• Reverse engineering • Bug finding • Symbolic or relaxed execution • path analysis	• Uses Klee for symbolic exec

FM Verification

Project	Action	LLVM Usage
Software Analysis Workbench (SAW)	• Verification: mode/code equivalence • Translate code to formal model • Use SMT/SAT *solvers* to prove equiv • (C/C++) bitcode <-> java <-> cryptol	• LLVM symbolic simulator: • haskell based xlate to symbolic IR
Divine	• Verification: Explicit state model checker • Explores attempting to hit asserts • Implements DIVINE VM	• “divine cc prog.c” will generate prog.bc • DiVM symexec’s IR
Seahorn	• Verification: Model checking integral assertions • intra-procedural	• Inject calls in C for assume and assert style checking • See end possible projects slide • Lifts to CRAB (abstract rep) for reasoning on values
Ilstar	• Verification Model checking • Pre|Post condition checking • Translates to coreStar for sym.exec	• Ingests Bitcode • Translates it to coreStar

Translate IR to Formal Language

Project	Action	LLVM Usage
llmc	• Verification: Transform to Labeled Transition System (LTS) • Map IR to PINS interface of LTS_min	• llmc git is empty • Thesis available with code discussion **&&**
pmGen	• Verification: Translation IR to Promela • Process or Protocol Meta Language Verification Modeling Language • Promela is checked in SPIN	Uses IR reader, but does not use pass API.
Smack	• Verification: Translation to Boogie (IVL) • Checks user-supplied assestions	• Translates LLVM to Boogie • Can inject into bc __VERIFIER_assert()

Application Hardening

Project	Action	LLVM Usage
Security-Oriented Analysis of Application Programs (SOAAP)	• Help with (fine grained) app compartmentalization • Semi-automated static/dynamic analysis • Act as an aide to development	• Pass manager • Passes analyzing code and compare with injected annotations (policy)
Temporally Enhanced Security Logic Assertions (TESLA)	• Developer specify temporal properties • Auto-gen runtime checks for the properties • “previously”, “eventually”… • Borrowed from model checking	• Injection of tesla_ routines • Injection of runtime checks
Causal, Adaptive, Distributed, & Efficient Tracing System (CADETS)	• Improve tracing and audit implementations based on lessons learned from *TrustedBSD Audit* and *DTrace*	TBD
SAFECode	• Code hardening • Static analysis for runtime safety • Goal to reduce runtime checks	• Virtual instruction set is LLVM IR + more • Compile to that and code is analyzed / verified • Then translated to proper MC
ASAP	• Optimizing “hardened” code • Analyzes runtime checks to determine which may be safely removed	• Implements cost calculators as module passes • Show use of SmallPtrSet container • Metadata node use

General Code Hardening & SW Resilience

Project	Action	LLVM Usage
KULFI	• Software resiliency analysis • Instruction level fault injection	• Module pass that injects static corruption and dynamic corruption
Return-less-code	• Code hardening • Investigate eliminating ROP via no returns • Generated return-less FreeBSD kernel	• Iterate through Instructions finding call/ret and replacing • Implement backend (MC)
PRESAGE	• Software resiliency • Protect structured address computations against soft errors (e.g. bit flips from alpha particles)	• Translate GetElementPtr array accesses to have dependencies that broken are a signal of soft error • Read paper for gory details on GEP

Obfuscator-LLVM	• Code obfuscation
	• Tamper-proofing
		• Operator substitution is FunctionPass
		• Bogus CFG injection is FunctionPass
		  • splits basic blocks
		  • introduce bogus loops

Many small, one-off passes or tools as well…

Project	Action	LLVM Usage
kryptonite	• Code obfuscation	Implemented function pass to transform IR

Passes from QuarksLab	• Example passes:
	  • ObfuscateZero
	  • Others
		• OZ is BasicBlockPass

whole program llvm	• Help merge multiple bitcode files into one	

- https://github.com/roachspray/opcde2017/blob/master/projects.md
- Please send me additions