Assured Information Security, Inc.
&
Dartmouth College

Crema

A LangSec-Inspired Programming Language

Jacob Torrey (@JacobTorrey) & Sergey Bratus (@sergeybratus)
`torreyj@ainfosec.com` & `sergey@cs.dartmouth.edu`

August 4, 2015

Outline

Thesis

In a Nutshell

Programming languages provide more computational power than most programmers need, LangSec has shown that in this excess expressiveness lurks weird machines, difficulties of verification and state-space explosion.

By providing a language that forces programmers to more accurately express their intent, security wins are possible!

Who we are

@JacobTorrey

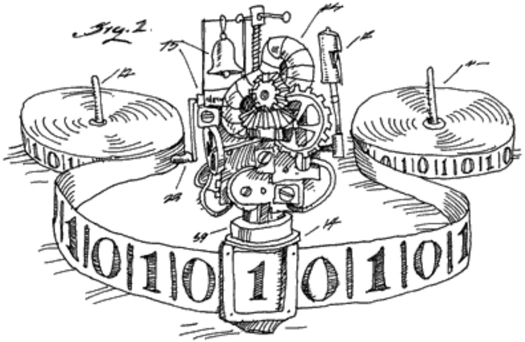
- Advising Engineer as Assured Information Security
- Leads Computer Architectures group
- Plays in x86 rings ≤ 0
- Ultra-runner/cyclist, traveler and foodie

@SergeyBratus

- Professor at Dartmouth College
- Co-founder of LangSec field
- Chair of IEEE S&P LangSec Workshop
- Had to miss at last minute :(

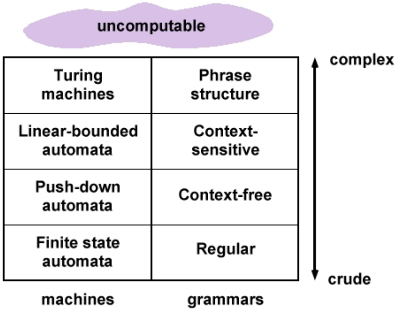
Some Definitions

- Turing Complete (TC) — If a system can simulate the widely-known Turing machine; computers today are finite, physical Turing machines
- Halting Problem — A classic problem in computer science that it is provably *undecidable* in general to determine if a program will halt on a given input



Some Definitions (cont.)

- “Undecidablity Cliff” — The more complex an execution environment is, the more difficult to analyze; eventually complexity reaches a “cliff” that is impossible to recover from
- Chomsky Hierarchy — A hierarchy of formal language classes of complexity and the corresponding automatons which will accept/recognize them



machines	grammars
Turing machines	Phrase structure
Linear-bounded automata	Context-sensitive
Push-down automata	Context-free
Finite state automata	Regular

What is LangSec

In a nutshell:

Language-Theoretical Security

Internet insecurity is a consequence of *ad hoc programming of input handling* at all layers of network stacks, and in other kinds of software stacks. LangSec posits that the only path to trustworthy software that takes untrusted inputs is treating *all expected inputs as a formal language*, and the respective input-handling routines as a *recognizer* for that language.

Ok... What...

We've built the internet on faith that developers will properly sanitize inputs, and trusted them the full power of Turing completeness. Like letting an unproven 16 year old drive a Ferrari.

LangSec Perspective

- The story of InfoSec: code meets input, code goes crazy, computation "elopes" beyond wildest expectations
- Static analysis of binary code for effects of (complex enough) inputs is *typically* intractable
  - Undecidable for Turing-complete cases, intended or accidental
  - State explosion even when technically decidable
- (Complex enough) inputs are Turing-complete on the code/processor/execution model that handles them
  - *Weird machines:* examples of unexpected emergent computation/programming models driven by input that seems to be purely "data" or "tables"

Data Drives Execution

"The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program."

"Any input is a program"

A complex data format is indistinguishable from bytecode, its handling code from a VM [Java-like, not VMWare-like] for that bytecode.

(apologies to A. Clarke)

Your input elements are an ISA; your code is a processor for that ISA. Pray it's exactly as powerful as you intended, and not more.

- "How will this input byte change the state of my system?"

–almost no one, ever

The Gap

There is a huge gap between the programmer's model and the execution model. It's likely big enough for Turing-completeness.

It's almost like the State of California, which knows that almost every place or thing is dangerous to your health.

Bad for your computer's trustworthiness:

- features in your CPU
- features in your programming language's standard library (``%n'', anyone?)
- features in your compiler's optimizer
- etc...

The Angle

LangSec-perfect

You should co-design your data and code/execution logic to yield predictable computation & tractable analysis on every input.

But what if you can't? Then the Turing Beast will likely free—and devour your chances of static analysis of input's effects.

Our approach

Limit the power of the processor, to gain better static predictability of inputs' effects. Compile to it from a language that is **deliberately** not Turing-complete.

An input-handler/parser that is accidentally TC on a complex data format will (and **should**) be hard to write.

Least Computational Privilege

DJB in *"10 Years of Qmail"*: Least Privilege is a distraction.

Updated *Least Privilege Principle*

Computational power exposed to attacker is privilege. Minimize it.

This is not to inveigh on general-purpose computing. LCPP belongs at communication boundaries between TC systems:

"Your CPU needs to be able to perform arbitrary computation. ICMP ECHO does not. So that's an important distinction, and do please keep it in mind."

Meredith L. Patterson, "Science of Insecurity", 28C3

Sub-Turing Execution
Like Language, Like Processor

The sub-Turing programming language must make it hard to express what's *hard to analyze*.

- It must compile to an execution model where hard to analyze is hard to compile to.
- It must still resemble C well enough, because programmers don't fall from the sky.

Peter Pan of Programming

Example

A crocodile (allegedly) never turns back. How about a processor that never takes a backward jump. No loops => no TC

Combine this with an upper limit on execution steps. Captain Hook would have a much easier time evading a time-limited crocodile.

A Sub-Turing Language

- These questions led us to a six-month DARPA seedling under I2O
- Designed with minimum power needed to perform most programming tasks
- Provably terminating in countable time (Walther recursion)
- No issues with Halting Problem
- Forbids unbounded loops, unbounded [co]-recursion
- Targets LLVM compiler tool-chain for ease of integration
- Easy to develop in and small learning curve

Language Limitations

Obviously, Crema is not the right tool for every task, there are some programming tasks that *require* the full computational expressiveness:

- Cannot support unbounded looping
- Not for scheduling loops or REPL/server loops, e.g.:
    - Apache server listen loop
    - OS scheduler
    - User-driven programs/UI
- Can be used as dispatch tasklets, still needs a TC controller

However, most programs are the composition of a very few TC components and a lot of parsers and data-analysis methods. By replacing those with a sub-TC environment, your attack surface is minimized.

Language Fundamentals

- Strongly-typed, C-like language
- Can use LLVM FFI to call into (or be called from) other languages
- Can express what is known as a parser or a *transducer*, converting input from one format to another
- Transducers should take a polynomial function of time w.r.t input length, should not be undecidable

Sample Crema Program

```
int hundred[] = crema_seq(1, 100)

foreach(hundred as i) {
  int_print(i)
  str_print(" ")
  if (i % 3 == 0) {
    str_print("Fizz")
  }
  if (i % 5 == 0) {
    str_print("Buzz")
  }

  str_println(" ")
}
```

Not so bad, huh

Designed to be approachable for developers and familiar-looking

- Supports `structs` and arrays
- Automatically manage
- Common boolean and bitwise operators
- Looping construct is the `foreach` loop, to iterate through a finite list or sequence generated with `crema_seq` (e.g., `crema_seq(1, 3) = [1, 2, 3]`)

Future Work

Crema is very young still (looking for your input!) and as such has some rough edges and missing features. Future work on improving Crema includes:

- Objects and classes
- Integration of a parser generator as the *only* method for reading input
- Stronger standard library
- Syntactic sugar for cleaning up duck typing and conversion

Formal Model

Based on a classical Turing Machine model with a modified transition function:

Modified Transition Function

Transition function δ is limited in such a way that it cannot return to an already-visited state:

$$\delta : (Q \setminus F) \times \Gamma \rightarrow Q' \times \Gamma \times \{L, R\}$$

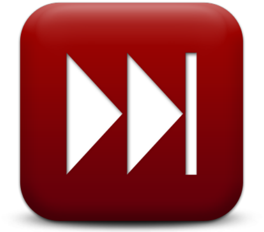
Where:

- Q is the finite set of states
- F is the set of terminating states
- Γ is the symbol alphabet
- $\{L, R\}$ denote moving the tape reader head left or right
- Q' is the new set of states $Q' : Q \setminus q_c$ where q_c is the current state

Forward-only Execution

Great, but what does that mean?

- Imagine a CPU that can only execute *forward* (i.e., to higher memory addresses)
- Naturally, program will terminate in finite time
- Enforces bound on state-space explosion to verify (number of branches in program)
- But this removes looping and function calls...



JIT Unrolling

- Using the notion of “just-in-time compilation”, a program can be instanciated from an abstract program
- Loops and funtions are unrolled and inserted JIT
- Using the program input as guide for number of iterations to unroll
- Supports Walther recursion
- Program analysis can still operate in the forward-only execution model, abstracted to develop full-featured programs

Embedding Crema in Programs

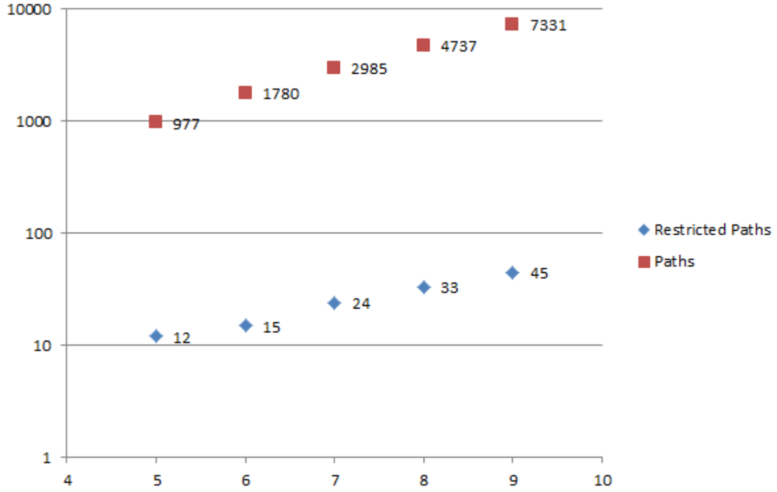
- `cremacc` creates LLVM IR “assembly”
- Ideal is to use Crema for parser and input-driven handlers
- Can call into other languages, can be called from other languages (can break sub-TC guarantees)
- LLVM IR can be optimized and analyzed by existing tools

QMail

- DJB’s mail daemon, designed for security, has open security bounty (one award in years)
- Target of Halvar Flake and Julien Vanegue’s automatic exploit generation research to find bugs
- Parser is highly isolated from main program logic
- Parser was analyzed by KLEE (symbolic execution engine) to measure code coverage & running time
- Re-wrote parser in Crema, and repeated analysis

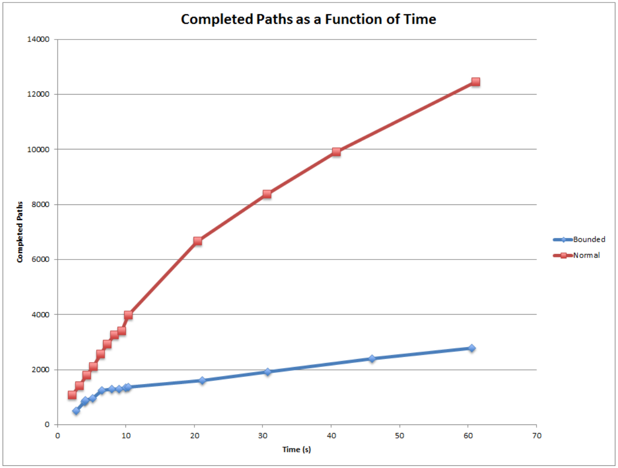
Results

Crema greatly reduced the state-space explosion inherent to program analysis:



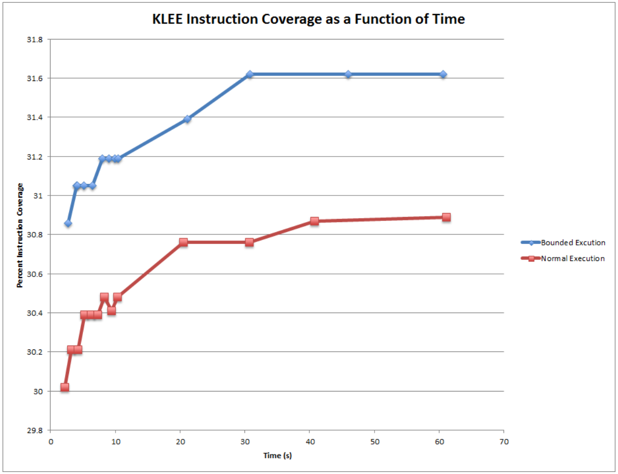
Results II

Bounded state-space to search grows much more slowly than unbounded:



Results III

Bounded execution provides higher code coverage by naive verifier:

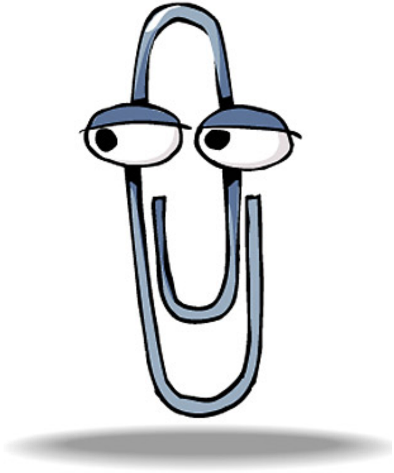


Mark Dowd’s Sendmail Bug

- Address parser to ensure that parentheses and brackets are not nested and the email address is valid
- Reserved space in output buffer to ensure that no overflow could occur if they were unmatched
- Mark found 2003 that a pointer failed to be decremented, leading to an overflow, Halvar used as a verification challenge problem
- WOOT 2012 Julien Vanegue et al proposed this as a solvable verification problem if the analyst knew the nature of the bug
- Hard to generically see “badness” on unbounded loops (while loop)

How Crema Could Help

- Crema would limit the bounds on the loop to a function of email address length
- Program verification tools could detect if the program would write outside its bounds
- Programmer's intent is more naturally expressed as a function of the input length rather than a `while` loop
- This is a perfect use-case for Crema, a parser that under all circumstances should terminate (*transducer*)



Use-cases

- Useful for “reducing the length of the rope programmers can hang themselves with”
- A computationally bounded attacker is a weaker attacker
- Parsing and other input-driven routines should have no need for full unbounded loops/TC
- The Crema model makes program analysis easier automatically, through state-space reduction and easier constructs to analyze

Wrap-up

- Crema is open source (thanks DARPA!)
- Code, examples and documentation available at `http://www.crema-lang.org`
- We hope you will check it out, hack on it, submit patch requests and start using it
- By using Crema, your software will “magically” be easier to analyze and safer

Questions?

Thanks for your time! Don’t hesitate to reach out to us on Twitter!