Tapir: a language for verified OS kernel probes

Ilya Yanok, Nathaniel Nystrom

4 Oct 2015
Motivation
System debugging and tuning

- Systems are large and complex
- Problems often occur only after deployment
- Need to debug and tune interactions between system components in the field
- => Whole-system dynamic instrumentation
Whole-system dynamic instrumentation

- Inject **probes** into application programs and into the kernel
- Probes inspect the program (kernel) state at their insertion point
DTrace [Cantrill et al. 2004]

- Small C-like language ("D") for writing probes
- Probes can be inserted (almost) anywhere
  - user programs
  - device drivers
  - the kernel
- Probes inserted dynamically
  - zero cost when no probes are active
Probes must not cause panic

- Probes cannot …
  - write to application or kernel data structures
  - access memory-mapped I/O space
  - dynamically allocate memory in the kernel
  - cause hardware traps
    - e.g., division by zero, accessing unmapped pages or unaligned addresses
  - run for a long time (especially in the kernel)
How does DTrace ensure safety?

- Embed trusted VM in the kernel
  - Compile probes into bytecode for a simple VM
  - VM does run-time checks during bytecode interpretation
  - Memory fault handler traps back to the interpreter
- Restrict the DSL
  - Writes to kernel or application memory forbidden
  - Loops forbidden
  - No recursive functions (or really, any functions)
SystemTap [RedHat]

- Similar to DTrace but…
  - DSL supports functions and loops
  - No VM. Probes are compiled to native code
  - Run-time safety checks are inserted by the compiler
  - Used analysis is often undocumented and buried into the C++ source
  - The DSL allows embedded C code which is never checked
• Quote from the SystemTap Specification

There are a number of undocumented but complex safety constraints on concurrency, resource consumption and runtime limits that are applied to code written in the SystemTap language. These constraints are not applied to embedded C code, so use embedded C code with extreme caution. Be especially careful when dereferencing pointers.

– SystemTap Specification
Idea
Our goal

- Whole-system instrumentation solution
  - provides the same level of safety guarantees as DTrace
  - compiles probes to native code
  - no ad-hoc analysis
  - support for loops
Tapir

tapir DSL

compile

safe Core

compile

C

injection

running kernel

probe
Safe Core

- Both DTrace and SystemTap use combination of two approaches
  - Restricted language
  - Run-time checks
- Can we merge the two?

such that all the required run-time checks are guaranteed to present in a correct program?
Type systems could be used to restrict the language.
Richer type system could enforce more interesting properties
Dependent types
Notation

- Logical rules

  \[ \textit{assumptions} \quad \Rightarrow \quad \textit{conclusion} \]

  reads Assuming \textit{assumptions} hold, \textit{conclusion} also holds

- Typing judgements

  \[ \Gamma \vdash t : \tau \]

  reads In the context \( \Gamma \) term \( t \) has type \( \tau \)
Sum types

- Is just a discriminated (tagged) union
- Contains either a value of type $A$ or a value of type $B$
- Denoted by $A \uplus B$

- `case` provides a way to deconstruct a $A \uplus B$ value

```
case aOrB of
  a -> doSomethingWithA;
  b -> doSomethingWithB
```

- Values of contained types are only available in the respective branch
Dependent types

• Allow types to depend on arbitrary terms, for example Vector A n type is a type of lists with a known length
• Ability to depend on an arbitrary term and not just a value is important: allows to give precise type to more functions. For example, vector concatenation:

\[
\Gamma \vdash xs : \text{Vector } A \ n \quad \Gamma \vdash ys : \text{Vector } A \ m
\]

\[
\Gamma \vdash xs \# ys : \text{Vector } A \ (n + m)
\]
Dependent types could be used to represent propositions about terms

- For example, type IsEven $n$ can be used to represent a proposition of evenness of a natural number
- Types with more than one argument could be used to represent logical relations between terms
- Types are propositions, terms are proofs (Curry-Howard isomorphism)
Division by zero
Division by zero must be prevented

• We don't want to cause trap
• Static checks are not enough:
  ▪ may divide by value read from the kernel
  ▪ dynamic checks may be necessary
How to ensure dynamic checks are in place?
Let's make division require a proof of denominator being non-zero

Division now takes an extra argument of type \( \neg (m \equiv 0) \):
\[
\Gamma \vdash n \; m : \text{Word} \quad \Gamma \vdash p : \neg (m \equiv 0)
\]
\[
\Gamma \vdash \text{div}_p[n \; m] \; n \; m : \text{Word}
\]
How to create a proof object?

- Testing operation now returns either proof of check succeeded (failed) instead of just saying "check succeeded (failed)"
  \[ \Gamma \vdash t : \text{Word} \]
  \[ \Gamma \vdash t = 0 : t \equiv 0 \oplus \neg(t \equiv 0) \]
- Recall that when branching on sum type value, the proof of success will be available only in the success branch
Memory safety
Our assumptions

- Limitations of our solution
  - no support for user-space application tracing
  - kernel memory is mapped at statically known range of addresses (holds for Linux)
  - no support for Linux high memory inspection
Same approach as for division

- Dynamic checks might be needed
- `peek` operation requires one more argument, a "Pointer is in range" proof
- Check operation is used to build "less than" proofs, which can be combined to build an "in range" proof:
  \[
  \Gamma \vdash n \; m : \text{Word}
  \]
  \[
  \Gamma \vdash n < m : n < m \uplus \neg(n < m)
  \]
Step counting
Idea: represent computations as data

• Introduce type Timed $A$ $n$. Values of this type are computations returning type $A$ in $n$ steps

• Make all basic operations instances of Timed, e.g.

$$
\Gamma \vdash x \ y : \text{Word} \\
\Gamma \vdash x + y : \text{Timed Word 1}
$$
Timed computations can be combined

- Pure value can be promoted to a timed computation returning in no time
  \[ \Gamma \vdash x : A \rightarrow \Gamma \vdash \text{return } x : \text{Timed } A \ 0 \]

- Timed computation that returns \( A \) and Timed computation parametrized on \( A \) can be combined into a single timed computation
  \[ \Gamma \vdash x : \text{Timed } A \ n \quad \Gamma \vdash f : A \rightarrow \text{Timed } B \ m \]
  \[ \Gamma \vdash x \gg= f : \text{Timed } B \ (n + m) \]
Future work

- Nicer DSL
- Formal safety proofs
- Concurrency
- Other extensions (not just probes)
Questions?
Thanks!
Extra slides
Memory safety: could instead use a modified trap handler

- **Pros**
  - That's what DTrace/SystemTap do
  - Needed for user-space program instrumentation

- **Cons**
  - Can't instrument trap handlers themselves (double-trap)
  - Still need a region based check to avoid reading from memory mapped I/O
Unneeded checks: some things are known statically

- Sometimes we need proof about statically known values, for example, to divide by 5, will need a proof \( \neg (5 \equiv 0) \), which is obviously true.
- Unfortunately, there is no way to get this proof without doing an unnecessary check in Core. The only way to build a proof is to perform an actual check.
- To solve this we added an ability to postulate such "obvious" proofs to our type checker.
Step counting example

```haskell
sum :: (xs : List Int) -> Timed Int (length xs)
sum [] = return 0
sum (x :: xs) = do
    s <- sum xs
    s + x
```