Mixing Type Checking and Symbolic Execution

Khoo Yit Phang (UMD College Park)
Bor-Yuh Evan Chang (CU Boulder)
Jeffrey S. Foster (UMD College Park)

June 9, 2010
Problem with static analysis

- Recently, there have been many successes in static analysis
  - static analysis tools analyze program source code and discover errors
- However, all practical (sound) static analyses have false warnings
- Minimizing false warnings complicates the design of static analysis tools
Example

• A standard dataflow analysis for data races:

```c
if (multithreaded) fork();
    ... some code ...
if (multithreaded) lock();
    ... shared_var++; ...
if (multithreaded) unlock();
```

≥ 1 thread

lock may not be held

possible data race?

unlocking a
lock not held?

• Need path sensitivity to be precise
  • too much path sensitivity leads to slowdown
Static analysis designers trade off...

- Efficiency: analysis must not take too long to run
- Precision: analysis should minimize false warnings
Pitfalls of ad hoc precision tradeoff

- Leads to complicated implementations
- Difficult to understand analysis results
- Hard to know implementation is correct
- May work for some programs but not others
Observation: only need precision sometimes

```c
if (multithreaded) fork();
if (multithreaded) lock();
shared_var++;
if (multithreaded) unlock();
```
**Mix**: apply analyses of *different precision* on *different parts of code* and combine the results
Overview of Mix

Typed block \{t, t\}

Off-the-shelf analysis within blocks

Standard type checker

Mix rules translate between blocks

Symbolic block \{s, s\}

Assume typed/symbolic blocks are given
Our symbolic executor

- Symbolic executor: an “interpreter” that explores all feasible paths
  - uses *symbolic variables* to represent unknown inputs
- Our symbolic executor checks for type errors
  - if no type errors are found on any path, then none are possible at runtime
- Formalism and soundness proof in paper
How Mix works

Wrap in *typed block* to begin with type checker

```javascript
{t var a = random_bool(), x = 500;

if (a) {
    if (a) x = 1;
    else x = 1 + true;
} else {
    a = x + 1;
    if (a > 0) a = (fib(a) < 100) ;
    else a = true;
}
}
How Mix works

Wrap in *typed block* to begin with type checker

```javascript
{t
  var a = random_bool(), x = 500;
}
```

Wrap in *symbolic block* to switch to symbolic executor

```javascript
{s
  if (a) {
    if (a) x = 1;
    else x = 1 + true;
  } else {
    a = x + 1;
    if (a > 0) a = (fib(a) < 100) ;
    else a = true;
  }
}s
```

On entry, convert types to symbolic values

- e.g.: `x` has type `int`, so init `x` to a fresh symbolic `int β`

Symbolic executor skips infeasible paths

- (path sensitive)

Symbolic executor allows temporary type change

- (flow sensitive)

On exit, convert symbolic values to types

- (must have the same types on all paths)

`fib(β + 1)` takes *a really long time* to compute in the symbolic executor
How Mix works

Wrap in *typed block* to begin with type checker
\{
t \text{var } a = \text{random\_bool()}, \ x = 500;
\}

Wrap in *symbolic block* to switch to symbolic executor
\{s\text{if (a) \{}\
\qquad \text{if (a) } x = 1;
\qquad \text{else } \ x = 1 + \text{true};
\}\}

\} \text{else \{}\
\qquad a = x + 1;
\qquad \text{if (a > 0) } a = \{ \text{t(fib(a) < 100)} \};
\qquad \text{else } \ a = \text{true};
\}\s}

\}t\}
Gaining local type refinement

Extend type system with \textit{int} subtypes: \textit{posint}, \textit{zero}, \textit{negint}

\begin{align*}
\{t\} & \quad \text{var } x = \text{some\_int}(); \\
\{s\} & \quad \text{if}(x > 0) \\
\{t\} & \quad \text{some code...} \\
\{s\} & \quad \text{x:posint} \\
\{t\} & \quad \text{else if}(x == 0) \\
\{t\} & \quad \text{some code...} \\
\{s\} & \quad \text{x:zero} \\
\{t\} & \quad \text{else} \\
\{t\} & \quad \text{some code...} \\
\{s\} & \quad \text{x:negint}
\end{align*}

Outer typed block can only determine \texttt{x:int}

Refine type of \texttt{x} locally
Handling limitations of symbolic execution

May have function type, but not source code; can’t check with symbolic execution

\{ s ... \{ t \ unknown\_function() \ t \} ... s \}\n
Instead, wrap in typed block to analyze

\{ s ... \{ t \ while(true) \{ \{ s ... loop\_body ... s \}\} \ t \} s \}\n
Slow to check long-running loops

Wrap in symbolic block to regain within-loop precision
Mix idioms

**Flow sensitivity**

\[
\{ s \ x = 500; \ \{ t \ \ldots \ \text{fib}(x) \ \ldots \ t \} \};
\]

\[
x = \text{true}; \ \{ t \ \ldots \ x \ \&\& \ \text{true} \ \ldots \ t \} \; s\}
\]

\[
x \ \text{initialized to} \; \text{int}
\]

**Path sensitivity**

\[
\{ s \ \text{if(true)} \ \{ t \ \text{fib}(500) \ t \} \ \text{else} \ \{ t \ \text{(true + 3)} \ t \} \; s\}\}
\]

**Context sensitivity**

\[
\text{fun id}(x) \ \{ \ { s \ \text{return} \ x; \ s} \} \}
\]

\[
\{ t \ldots \{ s \text{id}(1)s\} \ldots \{ s \text{id}("foo")s\} \ldots t\}
\]

**Path + context sensitivity**

\[
\text{fun div}(x,y) \ \{ \}
\]

\[
\{ s \text{if(y==0) return "err" else return x/y;s}\}
\]

\[
\{ t \ldots \{ s \text{div}(7,4)s\} \ldots \{ s \text{div}(5,0)s\} \ldots t\}
\]

More details in paper!
Mix soundness

- Formalized and proven sound for an imperative language (not higher order)
  - sound: “if Mix accepts the program, then the program will run without type errors”
  - Mix proof = type system proof as-is + symbolic executor proof as-is + 2 mix block cases
Mixy prototype

• Mix-based null/non-null type qualifier annotation system for C

• Detects null-pointer errors
  • when NULL reaches function argument annotated with non-null
Mixy implementation

- Combines two tools:
  - CilQual: type qualifier inference engine for C (unification-only reimplemention of Foster et. al.’s CQual)
  - Otter: symbolic executor for C (developed at UMD)

- Challenges:
  - aliasing to initialize symbolic pointers, constraint graph
  - recursion between functions annotated as typed and symbolic
  - performance – added caching to blocks
Preliminary evaluation

• Add one annotation to `vsftpd-2.0.7`:
  ```c
  void sysutil_free(void *nonnull p) {
      ... 
  };
  ```

• Eliminated 3 false warnings involving flow, path and context sensitivity
void sysutil_free(void *nonnull p) {
    {t ... t} 
};

void str_alloc_text(struct mystr *str) {
    {t ... t} 
};

const char* sysutil_next_dirent(...) {
    {t ... return NULL; ... t}
}

void str_next_dirent(...) {
    const char* filename = sysutil_next_dirent(...);
    {s ... if (filename != NULL) Path insensitivity (ignores null check)
        str_alloc_text(filename);
    s} }

    {t ... str_alloc_text(str); sysutil_free(str); ... t}
Conclusion

• Mix: a novel system that mixes standard type checking and symbolic execution
  • trade efficiency for precision, only where necessary

• Formlized and proven sound

• Mixy: prototype Mix implementation for C

• Future work: better aliasing, more evaluations, block placement, other mixes

• Preliminary experience is quite promising