Enforcing High-Level Protocols in Low-Level Software

Robert DeLine and Manuel Fähndrich

2018-04-09

Problem

- Safe languages not applied to low-level infrastructure software
- Need to manage resource management protocols:
  - resource references (dangling pointers, leaks, race conditions)
  - usage rules (order of operation)

Solution

- Vault language (C-like) to specify domain-specific resource management protocols
- Keys
  - Compile-time tokens representing run-time resources (neither duplicated nor lost)
- Held-key set
  - Set of keys in possession, their state, at a program point
- Type guards
  - Auxiliary condition on use of a value (describes *when* operations are valid)
  - Atomic predicate: key is in held-key set
- Can encode
  - Accessibility of resource
  - State of resource

Key Manipulation

- Tracked types: provide one-to-one correspondence between compile-time key and run-time object
  - `tracked(K) point p = new tracked point {x=3; y=4;}; K:int x = 4; free(p);`
- Functions: use *effect-clause*, pre- and post-conditions of how it changes key set
  - `void fclose(tracked(F) FILE f) [-F];`

Syntax	Pre on K	Post on K
`[K@a->b]`	held in state a	held in state b
`[K@a]`	held in state a	held in state a
`[-K@a]`	held in state a	not held
`[+K@b]`	not held	held in state b
`[new K@b]`	nonexistent	fresh key held in state b

Key Manipulation (continued)

- Types parameterized by keys
  - `void foo(tracked(F) FILE f, guarded_int<F> gi) [F];`
- Keyed variants (i.e. algebraic data types)
  - `variant opt_key<key K> [ 'NoKey | 'SomeKey {K} ];`
  - Constructing `'SomeKey` 1) requires `K` in key set 2) removes `K` from set
  - Used to enable typechecker to safely move between static and dynamic knowledge regarding held-key set

Keyed Variant Example

```
void foo(tracked(F) FILE f) [-F] {
  tracked opt_key<F> flag;
  if (close_early) {
    fclose(f);
    flag = 'NoKey;
  } else {
    flag = 'SomeKey{F}; // consume F
  }
  // F is gone here
  switch (flag) {
    case 'NoKey:
     // stuff
    case 'SomeKey: // get F back
     fclose(f);  // consume F
  }
}
```

More Examples (Regions)

- Regions = named heap subsets, deallocated as a whole
- Can catch dangling references and memory leaks

```
extern module Region : REGION;
void okay() {
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=1; y=2}; // object of type R:T, accessible only
  pt.x++;
  Region.delete(rgn); // remove from key set
}
void dangling() {
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=1; y=2};
  Region.delete(rgn);
  pt.x++; // R not in held-key set
}
void leaky() { // signature: assumes pre- and post-key set are same
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=1; y=2};
  pt.x++; // extra key in held-key set
}
```

More Examples (Sockets)

- Use keys' states to enforce steps to create socket ready to receive messages

```
interface SOCKET {
  type sock;
  variant domain ['UNIX | 'INET]; variant comm_style ['STREAM | 'DGRAM];
  variant status<key K> [ 'OK {K@named} | 'Error(error_code){K@raw} ];
  struct sockaddr { ... };

  tracked(S@raw) sock socket(domain, comm_style, int);
  tracked status<S> bind(tracked(S) sock, sockaddr) [-S@raw];
  void listen(tracked(S) sock, int) [S@named->listening];
  tracked(N) sock accept(tracked(S) sock, sockaddr) [S@listening, new N@ready];
  void receive(tracked(S) sock, byte[]) [S@ready];
  void close(tracked(S) sock) [-S];
}
tracked (@raw) sock mysocket = socket('UNIX, 'INET, 0);
switch (bind(mysocket, mysockaddr)) { case 'OK: ... case 'Error(code) ... }
```

Type System (based on Crary's Capability Calculus)

kinds	$\kappa ::= \textbf{Type} \mid \textbf{Key} \mid \textbf{KeySet} \mid \textbf{State}$	
variables	$\eta ::= \alpha, \rho, \epsilon, \sigma$	
contexts	$N ::= \bullet \mid N, \eta : \kappa \mid N$	
key set	$C ::= \epsilon \mid \emptyset$	key set variable, empty key set
	$\mid \{r@st \mapsto \sigma\}$	key mapping
	$\mid C_1 \oplus C_2$	key set union
key	$r ::= \rho$	key token
state	$st ::= \sigma$	state variable
	$\mid \sigma \leq sname$	state token
	$\mid sname$	state token
	$\mid \top$	default state
existentials	$\sigma ::= \exists[N \mid C].\sigma \mid \tau$	existential type, regular type
types	$\tau ::= \forall[N].\tau$	univeral type
	$\mid C \triangleright \tau$	guarded type
	$\mid p$	named type
	$\mid \mathrm{s}(r)$	singleton type
	$\mid \langle \tau_1, ..., \tau_n \rangle$	tuple type
	$\mid \alpha$	type
	$\mid (C, \sigma) \rightarrow (C', \sigma')$	function type
	$\mid [V_1(\sigma_1) \mid ... \mid V_n(\sigma_n)]$	variant type

Noteworthy Notes on Type System

- Concrete syntax `tracked T` translates to singleton type $\mathsf{s}(r)$
  - every alias for the resource is given same singleton type
  - `tracked region rg1 = Region.create();`
  - `tracked region rgn2 = rgn1;`
- Keys never duplicated
- Functions are polymorphic over rest of key set not mentioned
  - `void fclose(tracked(F) FILE f) [-F];`
  - $\forall \rho_{\mathsf{F}}.\forall \sigma.\forall \epsilon.(\epsilon \oplus \{\rho_{\mathsf{F}}@\sigma \mapsto \texttt{FILE}\}, \mathsf{s}(\rho_{\mathsf{F}})) \to (\epsilon, \texttt{void})$.

Existential Types ($\exists[N|C].\tau$)

- For encoding values carry capabilities with them
- `tracked region create();` has type
- $\forall\epsilon(\epsilon, \texttt{void}) \to (\epsilon, \exists[r : \textbf{Key}|r@\top \mapsto \texttt{region}].\texttt{s}(r))$
- access value of existential type via *unpacking*: creating fresh names for existentially-bound variables, acquirng capability carried

Case Study: Windows 2000 Drivers

- Translating 4900 lines of C to 5200 lines of Vault, then back to C
- Wrote wrapper around C to link back in to original kernel

Case Study: I/O Request Packets

- Use tracked types to reflect IRP ownership model
- `DSTATUS<I> Read(DEVICE_OBJECT, tracked(I) IRP) [-I];`
- Service routine owns IRP param, doesn't return ownership, must call one of three functions for further action
  - `DSTATUS<I> IoCompleteRequest(tracked(I) IRP, NTSTATUS) [-I];` // etc. ...
  - only way to generate *DSTATUS* < *I* > value

Case Study: Thread Coordination

- Coordinate access to whatever data key protects
- Can only access object after acquiring lock
- Detects missing lock releases
- Detect acquiring already-held lock

```
type KSPIN_LOCK<key K>;
KSPIN_LOCK<K> KeInitializeSpinLock<type T>(tracked(K) T) [-K];
void KeAcquireSpinLock(KSPIN_LOCK<K>) [+K];
void KeReleaseSpinLock(KSPIN_LOCK<K>) [-K];
```

Case Study: I/O Request Completion Routines

```
variant COMPLETION_RESULT<key I> [
  'MoreProcessingRequired | 'Finished(NTSTATUS) {I} ] ;

type COMPLETION_ROUTINE<key K> =
  tracked COMPLETION_RESULT<K> Routine(DEVICE_OBJECT, tracked(K) IRP) [-K];

void IoSetCompletionRoutine(tracked(I) IRP, COMPLETION_ROUTINE<I>);

NTSTATUS PnpRequest(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [-I] {
  KEVENT<I> IrpIsBack = KeInitializeEvent(Irp);
  COMPLETION_RESULT<I> RegainIrp(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [-I] {
    KeSignalEvent(IrpIsBack);
    return 'MoreProcessingRequired;
  }
  IoSetCompletionRoutine(Irp, RegainIrp);
  status = IoCallDriver(nextDriver, Irp);
  // key I no longer in held-key set
  KeWaitForEvent(IrpIsBack);
  // key I is back in held-key set
  ...
```