Enforcing High-Level Protocols in Low-Level Software

Robert DeLine Manuel Fähndrich
Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
vault@microsoft.com

ABSTRACT

The reliability of infrastructure software, such as operating systems and web servers, is often hampered by the mismanagement of resources, such as memory and network connections. The Vault programming language allows a programmer to describe resource management protocols that the compiler can statically enforce. Such a protocol can specify that operations must be performed in a certain order and that certain operations must be performed before accessing a given data object. Furthermore, Vault enforces statically that resources cannot be leaked. We validate the utility of our approach by enforcing protocols present in the interface between the Windows 2000 kernel and its device drivers.

1. INTRODUCTION

The past several years have witnessed the wide-spread acceptance of safe programming languages, due mostly to the popularity of Java. A safe language uses a combination of exhaustive static analysis and run-time checks and management to ensure that a program is free from entire classes of errors, like type errors and memory management errors. Ironically, one class of software in which the safe language movement has not made many inroads is low-level "infrastructure" software that needs to be highly reliable, like operating systems, database management systems, and Internet servers. The exhaustive analysis that a safe language provides seems a promising way to increase the reliability of this class of software, for which inexhaustive methods, like testing, have previously proved useful but incapable by themselves of achieving the goal of high reliability.

Such infrastructure software manipulates many resources, like memory blocks, files, network connections, database transactions and graphics contexts. The correctness of such software depends both on correctly managing references to these resources (no dangling references, no leaks, no race conditions) and on obeying resource specific usage rules (for example, the order in which operations on the resource must be applied). Together, we refer to these as *resource management protocols*. Today, such protocols are typically recorded in documentation and enforced through testing. The Vault programming language provides a new feature called *type guards*, with which a programmer can specify domain-specific resource management protocols. Such a protocol can specify that operations must be performed in a certain order, that certain operations must be performed before accessing a given data object, and that an operation must be in a thread's computational future[1]. Vault's type checker exhaustively seeks and reports any violation of such a protocol. In short, we move the description of resource management protocols from a software project's documentation to its source code, where it can be automatically enforced at compile time. To validate the utility of type guards, we have used them to describe and enforce resource management protocols in the existing interface between the Windows 2000 kernel and its device drivers.

This paper describes the resource management features of Vault, *keys* and *type guards*, and their application to Windows 2000 device drivers. In Section 2, we discuss the general framework of type guards and its instantiation in Vault's current design. We informally introduce keys through two widely known examples: memory regions and Unix sockets. Section 3 explains how Vault's type system enforces resource management protocols. Section 4 describes some of the protocols in the interface between the Window 2000 kernel and its drivers and how we enforce them. Section 5 discusses related work, and Section 6 concludes.

2. DESCRIBING RESOURCE PROTOCOLS

To check resource management protocols, Vault uses an extended notion of type checking. A typical type checker uses types to discriminate the values that the program manipulates to ensure that each operation is applied only to appropriate values. Vault's type system extends a type with a predicate called a *type guard*, which is an auxiliary condition on the use of a value of a given type. The Vault type checker tracks an abstraction of the computation's global state at each program point. For a program to access[2] a value at a given program point, the value's type guard must be true in the computation's global state at that point. In this light, we say that a type describes *which* operations are valid and a type guard describes *when* operations are valid.

2.1 Using Keys to Track Resources

In the current design of Vault, the abstract global state of the computation and the predicate language of type guards are intentionally

[1]Statically checking whether a thread actually reaches a given operation is clearly undecidable.

[2]Accessing a value means applying a primitive operation such as reading or writing through a pointer, or using an arithmetic operation on a numerical value.

PLDI 2001 6/01 Snowbird, Utah, USA

kept simple to enable an efficient decision procedure. The global state, called the *held-key set*, consists of a set of keys, which are simply compile-time tokens representing run-time resources. Each distinct key represents a unique run-time resource in each context, ie., two distinct keys represent two distinct resources. Keys can be neither duplicated nor lost, thereby providing the fundamental mechanism for tracking resource management at compile-time. Keys are purely compile-time entities that have no impact on run-time representations or execution time.

The atomic predicate of type guards is whether a given key is in the held-key set. A type guard is either `true` or a conjunction of one or more of these atomic predicates. Thus, a data object is guarded by zero or more keys; at a given program point, all the object's keys must be in the held-key set in order for the program to access the data object at that point. The type checker evaluates these predicates at compile-time. Type guards have no impact on run-time representation or execution time.

Vault's statement and expression syntax is based on the C programming language [11], hence the declaration

```
FILE input;
```

declares the variable `input` to be of type `FILE`. The declaration

```
K:FILE input;
```

declares the variable `input` to be of the guarded type `K:FILE`, which means that the variable has type `FILE` and that the key `K` must be in the held-key set at any point in the program at which the variable `input` is accessed.

For further expressiveness, the held-key set actually tracks a local state called a *key state* for each key. Key states are simply names. For instance, the programmer may choose to describe files as having two local states, "open" and "closed". A variation on the previous declaration

```
K@open:FILE input;
```

declares that a key `K` in local state `open` guards the variable `input`. In order to access this variable, the key `K` must be in the held-key set and `K`'s local state must be `open`. In examples where local key states are of no importance we simply omit them. Depending on syntactic context, omitted key states default to a fixed unique state or represent any possible state.

In summary, keys model two properties of resources: (1) whether a resource is accessible (whether the key is in the held-key set), and (2) what conceptual state a resource is in (the key's local state).

There are four features in Vault that are used to manipulate keys: *tracked types* associate keys with resources; functions filter keys in the held-key set; types with key parameters specialize types to particular keys; and *keyed variants* turn keys into values and static checks into dynamic checks.

Tracked types. Since Vault keeps track of the availability and state of individual run-time objects, the Vault type checker needs a way to distinguish the identities of run-time objects (that is, not to confuse one for another). The challenge is that the program text may contain many names for the same run-time object (aliases). In Vault, a key provides a symbolic name for an object's identity, and a tracked type provides a one-to-one correspondence between a compile-time key and a run-time object. The declaration `tracked(R) T x` states, as usual, that the variable `x` names some run-time object of type `T` (call it `O`). Further, the declaration provides alias information that the type-checker checks: within the scope of the key `R`, all program names of type `tracked(R) T` refer to the same object `O`, and no other program names refer to the object `O`. Vault's type rules guarantee that calls to other functions that manipulate the object `O` reflect state changes through key `R` as well. In short, the benefit of giving an object a tracked type is that the Vault type checker can trace the availability and state of that object throughout the program's text; the cost is that there are limitations on how program names may alias that object.

Tracked allocation acts as the primitive key granting mechanism:

```
tracked(K) point p = new tracked point {x=3; y=4;};
K:int x = 4;
```

At run time, the `new` operation allocates a fresh `point` object on the heap. At compile time, the compiler generates a fresh key (named `K` here[3]) associated with the fresh data object, and adds this key to the held-key set. The key `K` represents the availability of a memory resource, namely a heap-allocated `point` data structure. The example above also uses key `K` to guard the integer `x`; that is, the programmer has chosen to tie the availability of the variable `x` to the availability of `p`. At those program points at which key `K` is in the held-key set, the program may access both `p` and `x`; at those points at which the key is not in the set, the program may access neither.

Sometimes, the local name of a key is not important. In those cases, the programmer may let the compiler manage the key names.

```
tracked point p = new tracked point {x=3; y=4;};
```

This code is similar to the one above except that we can't refer to the key name in this scope directly. However, since we can pass the value of `p` to other functions, the key may be named in other scopes.

The primitive key revoking mechanism in Vault is the `free` operation.

```
tracked(K) point p = new tracked point {x=3;y=4;};
free(p);
```

In this example, the `free` operation takes an argument of type `tracked(K) T`[4] and requires that key `K` be in the held-key set. At compile-time, after the operation, key `K` is no longer in the held-key set. At run-time, the operation deletes the given heap-allocated data structure.

Functions. In Vault, a function's type has a *pre- and postcondition*, which respectively state which keys must be in the held-key set to call the function and which keys are in the held-key set when the function returns. For brevity's sake, the pre- and postconditions are written together as an *effect clause*, which states how the function changes the key set. An effect clause is written within square brackets. For a given key `K`, the effect clause `[K@a->b]` states that the key must be held before (in state `a`) and is held when the function returns (in state `b`). The effect clause `[-K@a]` states that the key must be held before in state `a` but won't be held when the function returns. The effect clause `[+K@b]` states that the key is not held before but is held after the function call. Finally, the effect clause `[new K@b]` states that on return, a fresh key (unknown to the context) is held in state `b`. As a shorthand, we write `[K@a]` for the common case `[K@a->a]`, and we omit key states altogether when they are of no importance. For example, the following function signature (akin to a function prototype in C)

```
void fclose(tracked(F) FILE f) [-F];
```

describes a function that takes a tracked file parameter whose key

[3] Key names such as `K` are bound when first referenced and have the same scope as a program variable bound at that point.

[4] There are some restrictions on the type `T`, such as `T` must not be abstract in the context of the operation. Other restrictions have to do with keys embedded within `T`.

is consumed by calling the function.

Types with key parameters. Vault supports *parameterized types*. The familiar case is to parameterize types by other types. For instance, a two-dimensional array that can be used at many types of data is declared as follows[5]:

```
type array2d<type T> = T[][];
```

Given this type definition, `array2d<float>` is the type of a two-dimension array of floating-point numbers. Less familiarly, a Vault type may be parameterized by a key set. (Currently, we restrict key set parameters to singleton sets.) For instance, the type declaration

```
type guarded_int<key K> = K:int;
```

declares a type abbreviation for a single integer that is guarded with the key on which the type is instantiated. For example, the signature

```
void foo(tracked(F) FILE f, guarded_int<F> gi) [F];
```

describes a function that takes two parameters: a tracked file `f`, whose key is called `F`; and a record `gi` whose field `x` is guarded by the same key `F`. Since guards and keys are purely compile-time entities, the function `foo` will be compiled into a function taking an ordinary `FILE` parameter and an ordinary `int` parameter.

Keyed variants. Vault supports *algebraic data types*, called *variants*, as found in most functional languages. For example, an optional integer is described with the following type declaration

```
variant opt_int [ 'NoInt | 'SomeInt(int) ];
```

This variant has two *constructors*, the constant constructor `'NoInt` (called "constant" because it takes no parameters) and the constructor `'SomeInt` which takes an integer parameter. The values of type `opt_int` are `'NoInt` and `'Int(n)` for any value `n` of type `int`. Vault's `switch` statement supports pattern matching over variants.

Variants are important for key management in Vault because constructors may have key parameters (written in braces). For example, given the variant declaration

```
variant opt_key<key K> [ 'NoKey | 'SomeKey {K} ];
```

constructing a value of type `opt_key<K>` with the constructor `'SomeKey` both requires that the key `K` be in the held-key set and removes the key from the set. Pattern matching against a value of type `opt_key<K>` restores the key to the held-key set in the `'SomeKey` case. For example, given the following code template

```
void foo(tracked(F) FILE f) [-F] {
  tracked opt_key<F> flag;   // F held on entry
  if (close_early) {
    fclose(f);               // F not held
    flag = 'NoKey;           // don't need F here
  } else {
    flag = 'SomeKey{F};      // need F, consume F
  }
  // whatever branch was taken, F is gone here
  // code A
  switch (flag) {
    case 'NoKey:   // we don't get F in this case
      // code B
    case 'SomeKey: // we get F in this case
      // code C
      fclose(f);   // consume F
  }
}
```

[5]The default bit-width of a type parameter is 32bits. Other widths must be explicitly declared in Vault.

```
interface REGION {
  type region;
  tracked(R) region create() [new R];
  void delete(tracked(R) region) [-R];
}
```

Figure 1: A Vault interface describing a region abstraction.

the key F would appear in the held-key set as follows. The key is held on entry to foo. The code then determines using `close_early` whether or not to close the file early. After the `fclose` in the true branch, key F is no longer held, and we record this fact in the `flag` variable. In the false branch, we record the fact that we still hold the key. Note that creating the value `'SomeKey{F}` removes key F from the held-key set by conceptually attaching it to the flag value (there is no run-time representation for keys). Thus in code section A, key F is not in the held-key set. But we can recover it by testing the value of `flag`. In the `'NoKey` case, code section B still does not hold key F. In the `'SomeKey` case however, the type checker knows that during code section C, key F is held again. Assuming code section C does not consume key F, the call to `fclose` after section C is valid, and all code paths of `foo` end in the state where key F is not held, which corresponds to `foo`'s declared effect.

A detail we glossed over in the code above is that the `opt_key` type of the `flag` variable is itself tracked. This is necessary, since the variant type may hold a key. If we allowed the `flag` variable to be copied without tracking aliases, then key F might be extracted multiple times from `flag`, or worse, it might never be extracted and thus lost. The code above does not show the key associated with the `flag` variable, but forgetting to test the flag would manifest itself by an extra key at the end of the function.

Using keyed variants, the programmer can turn static knowledge (whether a particular key is held) into a dynamic value (the variant). Pattern matching on keyed variants enables the programmer to help the compiler recover static knowledge (whether a particular key is held) from dynamic values. The variant type acts as an invariant that enables the type checker to safely move between static and dynamic knowledge regarding the held-key set. Variant types are also useful for expressing correlations between different state changes and return values of functions. This aspect of variants is illustrated in Section 2.3 to encode failure conditions.

Together, these four features allow the programmer to describe a useful variety of resource management protocols. To provide further introduction to these features, the remainder of this section applies them to two simple examples. Section 4 later shows how we used these features to check some of the resource protocols in the interface between a Windows 2000 device driver and the kernel.

2.2 Checking Memory Regions With Keys

A typical C program uses the functions `malloc` and `free` to allocate and deallocate individual heap objects. An alternative is to use regions [18, 8], also called arenas or heaps. A region is a named subset of the heap. A program individually allocates objects from a region, but it deallocates the region as a whole rather than deallocating individual objects. Based on the work of Crary, Walker, and Morrisett [3], we can create a safe region abstraction in Vault, as shown in Figure 1.

This interface declares an abstract type[6], called `region`. The

[6]An abstract type is one whose representation is private to the module that implements the interface.

```
extern module Region : REGION;

void okay() {
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=1; y=2;};
  pt.x++;
  Region.delete(rgn);
}

void dangling() {
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=1; y=2;};
  Region.delete(rgn);
  pt.x++; // error: key R not in held-key set
}

void leaky() {
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=1; y=2;};
  pt.x++;
  // error: extra key R in held-key set
}
```

Figure 2: A Vault program that uses the region abstraction. The function `okay` correctly uses it; the function `dangle` accesses a dangling reference; and the function `leaky` contains a region memory leak.

function `create` creates a new region, which is individually tracked. The function `delete` deletes the region and removes its key from the key set. To allocate an object in a region, Vault provides a primitive `new` operation taking the following form:

```
new(rgn) T [init]
```

Given a tracked region `rgn` with key `R`, the new construct returns an object of type `R:T`, that is, an object guarded by key `R`.[7] Thus, this object is accessible as long as the region is accessible. After a call to delete, all objects allocated within a region are inaccessible.

Figure 2 shows three functions that use this region abstraction, two of which have errors. The function `okay` correctly uses the region abstraction. Calling `Region.create` creates a new region whose key we label `R`. The point object `pt` is allocated from this region and is guarded with the key `R`. The increment to `pt`'s field `x` requires `pt`'s guarding key `R` to be in the held-key set, which it is. Finally, the call to `Region.delete` deletes the region and removes its key from the held-key set, thereby invalidating access to the variables `rgn` and `pt`. The function `dangling` reverses the region delete operation and the increment of `pt`'s field `x`. Because the effect of calling `Region.delete` removes the key `R` from the held-key set, the increment expression is incorrect since it requires the key `R`. The function `leaky` contains a more subtle error. Because this function has no explicit effect clause, it promises that the pre and post key set will be the same (no keys added, no keys removed). Because there is no call to `Region.delete`, the function has one extra key (`R`) in the held-key set at the end of the function than it did at the beginning. Hence the function's implementation violates its (implicit) effect clause, which is an error. This region interface thus catches both dangling references and memory leaks.

[7] We ignore the possibility of allocation failure here. In practice, `new` returns a variant indicating success or failure.

```
interface SOCKET {
  type sock;

  variant domain [ 'UNIX | 'INET ];
  variant comm_style [ 'STREAM | 'DGRAM ];
  tracked(@raw) sock socket(domain, comm_style, int);

  struct sockaddr { ... };
  void bind(tracked(S) sock, sockaddr)
    [S@raw->named];
  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];
}
```

Figure 3: A Vault interface that describes a socket abstraction.

2.3 Checking Sockets With Keys

Connection-oriented sockets are a popular software abstraction for inter-process and inter-machine communication in client/server architectures. Developing a server for such an architecture can be error-prone because setting up the socket to accept connections and communicate through them involves several steps—omitting one or more of these steps is a common beginner's mistake. To prevent such mistakes, we can create a Vault interface to a socket library like that in Figure 3. This interface uses the ability for keys to have states to enforce the necessary steps to create a connection-oriented socket that is ready to receive messages. The function `socket` creates a new socket whose key is in the "raw" state. We can see that in order to receive a message on a socket, its key must be in the "ready" state. The effect clauses for the functions `bind` and `listen` show how these functions change the state of the key from "raw" to "named" and from "named" to "listening," respectively. Finally, the function `accept` takes a tracked socket whose key `S` is in the "listening" state and returns a new tracked socket whose key `N` is in the "ready" state, the state needed to receive a message.

This interface to sockets is somewhat naive, in that it ignores the possibility of failure. To describe, for example, the fact that the `bind` operation can fail, we can change its function signature to the following:

```
variant status<key K> [ 'Ok {K@named} |
                        'Error(error_code){K@raw} ];
tracked status<S> bind(tracked(S) sock, sockaddr)
                  [-S@raw];
```

The `bind` function now consumes the tracked socket's key in the "raw" state and returns a variant. This variant has two constructors: the `'Error` constructor describes the failure case and its parameter provides an error code that explains the error; the `'Ok` constructor describes the success case. Both constructors have attached the key `K`, but in different states. In the `'Ok` case, the socket has correctly changed to the "named" state, whereas in the `'Error` case, the socket remains in the "raw" state.

The use of this variant type forces the programmer to check the status result of calling `bind`. Consider a program that forgets to check the status result:

```
tracked(@raw) sock mysocket = socket('UNIX,'INET,0);
```

```
bind(mysocket,mysockaddr);
listen(mysocket,0); // error!
```

Here, the call to `bind` removes the socket's key from the held-key set, hence the precondition for `listen` is violated. In order to call `listen`, the programmer must first check the status result from `bind`:

```
tracked(@raw) sock mysocket = socket('UNIX,'INET,0);
switch (bind(mysocket,mysockaddr)) {
  case 'Ok:
    listen(mysocket,0); // Okay now.
    ...
  case 'Error(code):
    // Report an error.
    ...
}
```

By checking the return status, the `'Ok` case puts the socket's key back in the held-key set in state "named", which makes the call to `listen` legal. In the `'Error` case, we have the key in the "raw" state and can for example try another `bind` operation.

2.4 Limitations of the Approach

Extending a type system to track compile-time names for resources has two limitations to consider: resources kept in collections become "anonymous;" and the types of values must agree at program join points.

Tracking arbitrary numbers of resources. Given that Vault uses compile-time names to track resources, how can Vault statically track an unbounded number of resources? In previous examples, the programs dealt with fixed numbers of resources whose keys were given names statically bound in the program's text. A programmer obviously cannot write down static names for an unknown number of resources. Using anonymous tracked types gets around this problem. For instance, we can declare the type of a list that contains an unbounded number of tracked regions:

```
variant reglist ['Nil|'Cons(tracked region,
                            tracked reglist)];
```

The type `reglist` allows the program to store an unbounded number of regions. However, placing a region on such a list makes it "anonymous"–that is, we lose track of exactly which key guards which region.

For example, the program in Figure 4 creates a region and allocates a point data structure out of the region. To access this point, its guarding key `R` must be in the held-key set. Once we put the region in the list, we lose the key `R`. (Since keys cannot be duplicated, we cannot both put the region with its key on the list and retain the key.) We then take the same region back out of the list by pattern matching. However, by placing the region on the list, its key becomes "anonymous" – the type checker knows that *some* key is associated with the tracked region but does not know that this key is the same as the key `R`. Hence, incrementing `pt`'s `x` field is illegal since this requires the key `R` and instead the held-key set contains some fresh key. (To fix the error in this program, we could use a list of pairs of regions and points, that is a list of type

```
type regptpair = (tracked(R) region, R:point);
variant regptlist ['Nil|'Cons(tracked regptpair,
                              tracked regptlist)];
```

which maintains the correlation between the region's key and the point's key guard.) In short, the Vault type checker can track both a fixed number of resources whose keys have statically bound names and an arbitrary number of resources whose keys are anonymous.

```
void main() {
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=4;y=2;};
  tracked reglist list = 'Cons(rgn,'Nil);
  // We lost key R.
  ...
  switch(list) {
    case 'Cons(rgn2,_):  // We got some key back.
      pt.x++;            // Bug! We need key R.
    ...
  }
}
```

Figure 4: An illegal Vault program that illustrates the "anonymizing" aspect of tracked collections.

```
void main() {
  tracked(R) region rgn = Region.create();
  R:point pt = new(rgn) point {x=4;y=2;};
  if (pt.x > 0) {
    pt.y = 0;
    Region.delete(rgn);
  } else
    pt.y = pt.x;
  // Bug! join point inconsistent
  if (pt.x <= 0)
    Region.delete(rgn);
}
```

Figure 5: An illegal Vault program that uses data correlation to encode a region's availability.

The type theory behind this "anonymity" is discussed in the next section.

Type agreement at join points. Consider the program in Figure 5, which correlates the value of the variable `pt.x` and the region `rgn`'s deletion status. Although this program is, in fact, memory-safe, the Vault type checker will reject it. At the commented join point, the held-key set either does or does not contain the key `R`, depending on which branch is taken. As a result, the type checker cannot know whether the precondition for the subsequent call to `Region.delete` is satisfied. The limitation that types must agree at program join points is a common limitation of the type checking approach to program verification. In order to make the example acceptable to Vault, the corelation between the sign of `pt.x` and whether or not we hold key `R` needs to be made explicit using a keyed variant, similarly to the example in Section 2.1. Instead of correlating the two tests with the condition on `pt.x`, the second branch would switch on a variant initialized in the branches of the first test.

3. VAULT'S TYPE SYSTEM

Vault's type system is based on the Capability Calculus [3] and alias types [15, 20]. The language's complete typing rules are lengthy. Here we only sketch how tracked types and type guards are represented in the underlying type language and the overall structure of the type checker. Because Vault's typing rules are very similar to the Capability Calculus, we rely on their soundness proof to ensure the soundness of Vault's type checking.

The type checker's job is to translate the Vault surface syntax

kinds	κ	::=	**Type** | **Key**	
		|	**KeySet** | **State**	
variables	ν	::=	$\alpha, \rho, \epsilon, \delta$	
contexts	N	::=	$\cdot \mid N, \nu : \kappa \mid N$	
key set	C	::=	ϵ	key set variable
		|	$\emptyset$	empty key set
		|	$\{r@st \mapsto \sigma\}$	key mapping
		|	$C_1 \oplus C_2$	key set union
key	r	::=	ρ	key token
state	st	::=	δ	state variable
		|	$\delta \leq sname$	bounded variable
		|	$sname$	state token
		|	$\top$	default state
existentials	σ	::=	$\exists[N \mid C].\sigma$	existential type
		|	τ	
types	τ	::=	$\forall[N].\tau$	universal type
		|	$C \rhd \tau$	guarded type
		|	p	named type
		|	$\mathrm{s}(r)$	singleton type
		|	$\langle \tau_1, \ldots, \tau_n \rangle$	tuple type
		|	α	type variable
		|	$(C, \sigma) \to (C', \sigma')$	function type
		|	$[\, V_1(\sigma_1) \mid \cdots \mid V_n(\sigma_n) \,]$	variant type

Figure 6: Underlying type language

types to the internal type language in Figure 6 and to assign internal types to the program's terms (statements and expressions). We discuss the roles that these various types play below. Part of the type checking is standard: for each lexical scope, the type checker keeps an environment that maps program names to types, keys, etc., and checks inductively that each program term is applied to subterms of the correct type.

In addition, the type checker ensures that no type guards are violated. To do this, the type checker forms a control flow graph for each function and computes the held-key set before and after each node in the graph. The held-key set before the function's entry node is the precondition key set from the function's effect clause. The type checker ensures that the held-key set at each of the function's exit nodes is the postcondition key set from the function's effect clause.

On control-flow join points, we abstract over the actual names of local keys in incoming key sets so as to analyze the remainder of the control-flow graph only for distinct alias relationships of local variables. Imperative loops may require declared loop invariants, unless the invariant can be inferred in a fixed number of iterations. Loop invariants take the form of a function type with multiple outcomes, one for each possible loop exit. Inputs to the function are the variables that are used within the loop. For all of the loops in our device driver case study, the type checker automatically infers the loop invariants, since they are trivial.

3.1 Tracking aliases

The key to ensuring that a program does not reference a resource after that resource has been released is to keep track of the various names by which the program refers to the resource. Without tracking aliases, a program could delete a resource through one name and then reference it through another. In Vault, a key[8] serves as a unique name for a resource; the type checker uses the same key to refer to the resource, no matter how many aliases for this resource the program's text contains.

A type `tracked T` in the surface syntax, which gives rise to a key, is translated to a singleton type $\mathrm{s}(r)$ in the internal type language. Such a singleton type represents the run-time value (handle) used to manipulate the unique resource whose key is r. Every alias for the resource in the program text is given the same singleton type $\mathrm{s}(r)$. Hence, given the pair of assignments

```
tracked region rgn1 = Region.create();
tracked region rgn2 = rgn1;
```

Both the variables `rgn1` and `rgn2` are assigned the same singleton type $\mathrm{s}(r)$ for some fresh r. Calling `Region.delete` on either `rgn1` or `rgn2` deletes the key r from the held-key set, which prevents the region from being referenced under either name after the deletion.

Another important aspect of tracking aliases is ensuring that keys are never duplicated. For instance, if the type system were to allow a region's key to end up twice in the held-key set, then a program that deletes the region twice would type check correctly, but would cause an error at run time.

3.2 Functions

Whereas Vault's surface syntax for functions combines a function's pre- and postcondition into a single effect clause, the internal function type $(C, \sigma) \to (C', \sigma')$ separates them into the key set C that must be held to call the function and the key set C' that is held after the function returns. Functions in Vault are always polymorphic. First a function is polymorphic in the keys of its arguments. For example, the function signature

```
void fclose(tracked(F) FILE) [-F];
```

can be called on any tracked file, regardless of its particular key. Second, since the state of key `F` is omitted here, the function is polymorphic in the state δ of `F`. Third, a function affects only those keys mentioned in its signature; other keys in the held-key set are irrelevant. To make a Vault function callable from many different contexts, we make its type polymorphic over the "rest" of the held-key set not mentioned in the function's signature. Given these three forms of polymorphism, the function `fclose` above is assigned the type
$\forall \rho_F . \forall \delta . \forall \epsilon . (\epsilon \oplus \{\rho_F @ \delta \mapsto \texttt{FILE}\}, \mathrm{s}(\rho_F)) \to (\epsilon, \texttt{void})$.
The variable ϵ refers to the "rest" of the held-key set that the function does not affect, the variable ρ_F refers to key `F`, and the variable δ refers to the state of that key. This function can be called on any value of type $\mathrm{s}(\rho)$ at any program point whose held-key set includes some key ρ associated with type `FILE`.

3.3 Existential types

Existential types are useful for encoding that certain values carry capabilities with them. The existential type $\exists[N \mid C].\tau$ represents a value of type τ, holding on to capabilities C. The existentially bound variables N provide a way to abstract the actual names used for keys, states, and types in C and τ. For instance, consider a function whose signature is

```
tracked region create();
```

Calling this function returns a tracked region, that is, it returns both

[8] In the Capability Calculus, a key is called a *resource* and a key set is called a *capability*.

a new resource and the key needed to access that resource. This function is assigned the type
$\forall\epsilon.(\epsilon, \texttt{void}) \rightarrow (\epsilon, \exists[r : \mathbf{Key} | \{r@\top \mapsto \texttt{region}\}].\, \mathrm{s}(r))$.
The returned existential type binds together both the new region (the singleton type) and the key needed to access that region.

To access a value of an existential type, the type must first be *unpacked*. Unpacking means creating fresh names for the existentially bound variables and acquiring the capability carried by the value. Clearly, since unpacking yields a capability, values of existential types cannot be freely copied by a program, for otherwise capabilities could be duplicated. To control the duplication of existential types which carry capabilities, we maintain the invariant that environments map program variables to unpacked types τ only. Thus, existential types must be unpacked before they are bound in the environment. For example, function parameters are unpacked on entry to a function.

Existential types are the basis for keeping "anonymous" tracked resources in collections. For instance, the type `reglist` in the previous section has the following type in the internal type language:

$$[\,`\mathit{Nil} \,|\, `\mathit{Cons}(\exists[\rho_1 : \mathbf{Key}, \rho_2 : \mathbf{Key} \,|\, \{\rho_1@\top \mapsto \texttt{region}\} \oplus \{\rho_2@\top \mapsto \texttt{reglist}\}].\, \langle \mathrm{s}(\rho_1), \mathrm{s}(\rho_2)\rangle)\,]$$

Each element in this list is of existential type. To use an element from this list, it must first be unpacked, which generates a fresh name for the existentially bound key. This is the technical sense in which these keys are "anonymous."

Our use of existential types is related to the *unique types* of the programming language Concurrent Clean. In Clean, a unique type is written $\tau^{\bullet}$. Its equivalent Vault type is $\exists[\rho : \mathbf{Key} \mid \{\rho@\top \mapsto \tau\}].\, \mathrm{s}(\rho)$, i.e., an anonymous tracked type. However, Clean does not support named tracked types and is thus unable to express alias relationships.

4. CASE STUDY: WINDOWS 2000 DRIVERS

Device drivers pose an important reliability risk to operating systems, since drivers generally execute in the kernel's protected mode. Because a device driver is used in many different machine configurations and sits within a multithreaded kernel, reproducing erroneous behavior in a driver is very difficult. Hence, testing has not proven to be a good way to achieve high reliability in drivers. In this section, we describe how Vault's type checker catches at compile time many of the errors that are difficult to reproduce at run time.

What is the difficulty in writing a correct device driver? Typically, the company that manufactures a device also provides the device driver for it. Because the developer creating the device driver is very familiar with the device itself, the interface between the driver and the hardware, though complex, is not often the source of errors. Instead, faults often lie in the interface between the device driver and the kernel. This interface is quite complex, in part due to the variety of devices that interact with the kernel and in part due to the need for good performance.

One source of complexity in the interface between the kernel and a driver is its asynchronous nature. A driver provides a collection of services to the kernel, like starting the device, reading from the device, writing to the device, and shutting down the device. The driver is implemented as a module with one function per service. However, the lifetime of a request to the driver is not the same as the lifetime of a call to the corresponding service function. To keep the kernel from blocking on a driver request, a driver's service function is expected to return quickly, regardless of whether the driver has completed the request.

To achieve the desired asynchronous interface in Windows 2000, each request is encapsulated in a data structure, called an I/O Request Packet (IRP). The kernel passes this data structure to the driver when it calls one of its service functions, and the driver handles the request by updating this data structure over time. Whenever the driver completes a request, it calls the function `IoCompleteRequest` on the IRP to signal the completion to the kernel and to return the IRP.

As a further complication, a driver does not work in isolation, but instead sits within a driver stack. For example, in between the kernel and a floppy disk drive would typically sit the following drivers, in order: a file system driver; a driver for a generic storage device; a floppy disk driver; and a bus driver. Each driver in the stack may choose to handle a request itself, to pass the request down to the next driver in the stack, or to pass a new request (or set of requests) down to the next driver in the stack.

Finally, as part of the kernel, a device driver must deal with the contingencies of kernel-level programming. For instance, at any given moment, the processor can be at one of several interrupt levels. The processor's current interrupt level governs both which kernel functions can be called and what memory is available. The kernel's memory space is divided into those pages that the virtual memory system manages and those that are locked down and therefore always accessible. A pointer to a block of paged memory can only be accessed if the particular page is known to be resident or if the current interrupt level is such that the virtual memory system can handle a page fault to make the page resident. If a driver dereferences a pointer to a non-resident paged block when the interrupt level prevents the the virtual memory system from running, the entire operating system deadlocks.

This section shows how each of these aspects of device drivers can be described in Vault. To test our ideas, we wrote a Vault description of the interface between the Windows 2000 kernel and a device driver. We then translated an existing driver for a floppy disk device from C (4900 lines) into Vault (5200 lines).[9] This Vault driver uses the Vault interface to interact with the kernel and is therefore subject to the checking we describe in this section. We then used the Vault compiler to compile the driver's source code into C. In some cases we chose to deviate from the original kernel interface, for example, by choosing to represent a status code with a variant rather than an integer in order to allow static checking. As such, our driver could not be directly linked against the original kernel. Instead, we wrote a thin wrapper in C to make up for these differences in data representation. The driver linked with the wrapper runs successfully under Windows 2000, although it is not of production quality due to incomplete modeling of memory allocations and deallocations. We also have not run any performance measurements on the resulting code.

4.1 I/O Requests Packets

The Windows 2000 documentation describes an "ownership" model for the I/O Request Packets (IRPs). Initially, the IRP "belongs" to the kernel. When the kernel calls a driver's service function, it gives ownership of the IRP to the driver. The driver can then take one of three actions: it can complete the request by calling `IoCompleteRequest`, which gives ownership of the IRP back to the kernel; it can call `IoCallDriver` to pass ownership of the IRP down to the next driver in the stack; or it can call

[9] So far, we have not concentrated on keeping Vault as syntactically close as possible to C. Hence these numbers do not reflect an inherent blow-up in using Vault's type system. Evaluating the programming burden of Vault's annotations is future work.

IoMarkIrpPending to retain ownership of the IRP after the call to the service function. A driver may only legally access an IRP when it has ownership of it.

This IRP ownership model corresponds naturally to tracked types. In Vault, a typical driver service routine is given the following signature:

```
DSTATUS<I> Read(DEVICE_OBJECT, tracked(I) IRP) [-I];
```

The signature states that a service routine obtains the ownership of the parameter IRP and does not pass the ownership back to the caller. Furthermore, the service routine must return a value of type DSTATUS<I>, which we use to enforce that one of the three possible functions mentioned above are called:

```
DSTATUS<I>
  IoCompleteRequest(tracked(I) IRP, NTSTATUS) [-I];
DSTATUS<I>
  IoCallDriver(DEVICE_OBJECT, tracked(I) IRP) [-I];
DSTATUS<I>
  IoMarkIrpPending(tracked(I) IRP) [I];
```

Since we keep the type DSTATUS<I> abstract from the service routine, and since the type of the return status is parameterized by the key I of the IRP request, the only way a service routine can generate a DSTATUS<I> value is by calling one of the above functions in the context of that particular invocation. This avoids the common error that some drivers exhibit code paths on which IRPs are neither completed, passed on, nor pended.

We leave it up to the driver to manage queues of pending IRP requests, thus IoMarkIrpPending does not consume the IRP key. A driver consumes the key by storing the IRP on a pending list, thus anonymizing and packaging the key with the IRP.

4.2 Thread Coordination

The Windows 2000 kernel provides several thread coordination mechanisms, one of which is *events*. An event allows one thread to block until another thread takes some action. Our Vault description of events can be used to pass a key from one thread to another, thereby coordinating access to whatever data that key protects:

```
type KEVENT<key K>;
KEVENT<K> KeIntializeEvent<type T>(tracked(K) T) [K];
void KeSignalEvent(KEVENT<K>) [-K];
void KeWaitEvent(KEVENT<K>) [+K];
```

The initialization function takes a tracked object whose key is to be transferred from one thread to another. In a multithreaded program, there is one key set per thread. To pass the key between threads, the first thread calls KeWaitEvent and blocks until the second thread calls KeSignalEvent. After the call to KeSignalEvent, the second thread no longer has the key in its held-key set, while the first thread unblocks and gains the key in its held-key set. As is typical of Windows 2000 drivers, the floppy driver uses this event mechanism to pass IRP ownership from one driver to another, as described in the next section.

We can similarly describe kernel spin locks in Vault:

```
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];
```

This interface protects against common locking errors. First, once a lock has been created on a tracked data object, the only way to access the object is first to acquire the lock. Second, in the same way that Vault can detect memory leaks (by finding keys in a function's final held-key set that were not promised in its signature's post key set), Vault can similarly detect missing lock releases. Third, since a key cannot appear in the held-key set multiple times, Vault will detect when a program acquires a lock that it already holds, since the second acquire will introduce a key into the held-key set that is already present. This approach however is inadequate to model reentrant locks.

4.3 I/O Request Completion Routines

As mentioned earlier, when a driver passes an IRP down to the next driver in the stack (by calling IoCallDriver), it loses ownership of the IRP. However, a driver often needs first to pass an IRP to the next lower driver and then to regain ownership of the IRP after the lower driver has completed it. To do this, a driver attaches a *completion routine* to the IRP, which is a function that is called on the IRP when the lower driver completes it. If a driver's completion routine returns a status of "more processing required" then the driver once again gains ownership of the IRP.

We describe completion routines in Vault with the following definitions:

```
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>);
```

The function IoSetCompletionRoutine sets an IRP's completion routine, which is a function that takes a device object and a tracked IRP and consumes the IRP's key.

The code in Figure 7 shows shows a common idiom for regaining ownership of an IRP after it is passed to a lower driver. The code uses a completion routine to learn when the lower driver has finished and an event to resume processing where it left off before calling the lower driver.

The figure shows a service function for a "plug and play" request, like a request to shut down the device. The function first declares an event, IrpIsBack, which is parameterized by the IRP's key, and declares the local function RegainIrp. The function PnpRequest then sets the IRP's completion routine to the function RegainIrp and passes the IRP down to the next driver with a call to IoCallDriver. After this call, ownership of the IRP has been passed to the next driver, which is reflected in the fact that the key I is no longer in the held-key set. The function then waits for the event IrpIsBack. When the lower driver completes the IRP, the kernel owns the IRP until it calls the completion routine RegainIrp. The completion routine in turn signals the event IrpIsBack, thereby passing ownership back to the PnpRequest function. The completion routine returns the status 'MoreProcessingRequired to tell the kernel that this driver has once again accepted ownership of the IRP.[10] When the call to

[10] A careful reader might be concerned that the completion routine could signal that the driver owns the IRP, but then forget to return 'MoreProcessingRequired–a situation that would lead to a dangling reference. This in fact cannot happen since the only other constructor for this variant ('Finished) takes the IRP's key as a parameter, a key which is no longer in the held-key set after the call to KeSignalEvent. Given the defi-

```
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
  ...
}
```

Figure 7: A driver uses an event and a completion routine to regain ownership of an IRP after passing it to a lower driver.

`KeWaitForEvent` returns, key `I` again appears in the held key set. The code after the call to `KeWaitForEvent` is therefore free to access the variable `Irp`.

4.4 Interrupt Levels and Paging

To represent the processor interrupt level (or IRQL, in Windows 2000 terminology), we use two details of Vault that were previously unmentioned. First, although keys typically arise from tracked types, a programmer can also statically declare a key. Second, we can optionally define a partial order over the states of a key and constrain state variables by states. Using both these features, we represent the current processor interrupt level as a global key `IRQL`:

```
stateset IRQ_LEVEL = [ PASSIVE_LEVEL <
   APC_LEVEL < DISPATCH_LEVEL < DIRQL ];

key IRQL @ IRQ_LEVEL;
```

Given these definitions, we can describe the preconditions of various kernel functions. The function `KeSetPriorityThread` requires the interrupt level to be at `PASSIVE_LEVEL`:

```
KPRIORITY KeSetPriorityThread(KTHREAD,KPRIORITY)
  [ IRQL @ PASSIVE_LEVEL ];
```

The kernel function `KeReleaseSemaphore` is more flexible. It requires the interrupt level to be less than or equal to `DISPATCH_LEVEL`:

```
long KeReleaseSemaphore(KSEMAPHORE,KPRIORITY,long)
  [ IRQL @ (level <= DISPATCH_LEVEL) ];
```

This function is polymorphic in the local state of the key `IRQL` as captured by the explicit state variable `level`, which is upper-bounded by state `DISPATCH_LEVEL`. Finally, the function `KeAcquireSpinLock` is more complicated. It requires that the interrupt level be less than or equal to `DISPATCH_LEVEL` on entry and raises the interrupt level to `DISPATCH_LEVEL` on exit. It also returns as its result a value that represents whatever the interrupt level was on entry. Given the definitions above and a type `KIRQL` that is parameterized by a state (which is similar to having a type parameterized by a key), we can describe this complex behavior:

```
type KIRQL<state S>;

KIRQL<level> KeAcquireSpinLock(KSPIN_LOCK)
  [ IRQL @ (level <= DISPATCH_LEVEL) ->
                              DISPATCH_LEVEL ];
```

Like the previous kernel function, this function uses bounded polymorphism over the local state of the key `IRQL`. Further, it uses the state variable `level` to refer to the state of key `IRQL` at the call site in order to reflect this level in the result type. Finally it uses the arrow notation to state that the function changes the key `IRQL`'s state from the state represented by `level` to the state `DISPATCH_LEVEL`.

The examples above thus make use of constrained state variables of the form `sv <= st`. The type checker uses the partial order specified in `stateset` declarations to determine when such constraints are satisfied.

Using constrained states, we can describe types `T` in paged memory by introducing a type guard on the interrupt level.

```
type paged<type T> = (IRQL @ (level<=APC_LEVEL)):T;
```

A value of a paged type may thus only be accessed at program points where the interrupt request level is at or below `APC_LEVEL`, ensuring that the page handler can service possible page faults. Internally, a paged type `paged<T>` is represented as

$$\forall[\delta : \mathbf{State}].\{\texttt{IRQL@}(\delta \leq \texttt{APC\_LEVEL}) \mapsto \texttt{void}\} \rhd \texttt{T}$$

If a driver accesses data in paged kernel memory at an interrupt level that prevents the virtual memory system from running, the result is unpredictable behavior: if the data's page happens to be resident, then the access is fine; otherwise, the kernel deadlocks when it tries to run the virtual memory system. Such a subtle error is very difficult is reproduce and correct. By using the interrupt level to guard data in paged memory, the Vault type checker finds such errors at compile time.

5. RELATED WORK

Our work is inspired in part by the *typestate* approach provided in the programming language NIL [17, 16]. In NIL, states are attached to objects along with their types. NIL does not allow any aliasing of objects, thus severely restricting the class of programs that can be expressed in NIL.

The work involving the calculus of capabilities by Crary, Walker, Smith, and Morrisett [3, 19, 15, 20] shows how to track states of objects in the presence of aliasing. The essential improvement over the typestate approach is to add a level of indirection between objects and their state through keys. Now the non-aliasing requirement is confined to keys whereas the aliasing relationships among objects are made explicit. This work provides the theoretical basis for Vault's type system. Our system differs in minor details such as using direct pre/post conditions of functions instead of continuation-passing style, which leads us to infer join-point abstractions or reanalyzing path fragments under different key sets. A more fundamental difference introducing a number of complications is exposing this rich type language to the programmer in an intuitive way.

In order to allow general graph structures, objects cannot always be tracked individually, but must be tracked as groups. The canon-

nition of the variant `COMPLETION_RESULT`, if a completion routine consumes its IRP parameter, it has no choice but to return `'MoreProcessingRequired`, since no other option will type check.

ical example of such group tracking are memory regions for safe explicit deallocation [3, 19]. Region annotations on types are one particular kind of predicate, stating that the named region must not be freed in order to access the data.

A related approach to tracking individual objects is present in the programming language Concurrent Clean [2]. Concurrent Clean uses *unique* types to represent unaliased objects. Operations such as array updates may be performed destructively on objects of unique type even in a purely functional language, since the modification cannot be distinguished from a copy. Clean's unique types correspond to Vault's anonymous tracked types, where the key remains unnamed. More technically, unique types correspond to singleton types where the key is existentially bound [20].

Sagiv, Reps, and Wilhelm provide a framework for intraprocedural shape analysis via 3-valued logic [14]. Their framework can express more detailed alias relations not currently expressible in Vault, as for example a function returning a pointer to the last element of a list, while leaving the list intact.[11]

At first glance, type guards are similar to type qualifiers [7]. However, type qualifiers refine the type (how the object can be manipulated) rather than guarding the access (when the object can be accessed). Furthermore, type qualifiers are constant and cannot change state.

Guarded types can be viewed as a form of qualified types [9, 10] $\forall S.P \Rightarrow \tau$, where the qualification quantifies over the abstract store S. At each use, the type must be instantiable to the current store S_i and the predicate P must be satisfied $S_i \vdash P$. However, the framework of qualified types lacks a notion of state.

Vault shares much of the motivation with the work on Extended Static Checking (ESC) [12]. ESC however starts with a memory safe language (Module-3/Java) and thus precludes its use in low-level system code such as device drivers. Furthermore, ESC takes a pragmatic approach to aliasing, tracking aliasing correctly within a procedure, but it does not consider all possible aliasing relationships created by procedure calls [4]. ESC is based on first-order logic with arithmetic pre and post-conditions. The presence of specification or ghost variables allows for tracking the state of an object, similarly to our local key states. However, this is not enough to describe the creation and disappearance of resources as is possible with keys and key sets, since there is no object available to attach presence information to. On the other hand, Vault's formalism is much less ambitious in terms of expressible pre and post conditions, since it cannot for example express arithmetic relationships. Thus, the techniques described here complement those of ESC.

Flanagan et. al. propose a type system for Java to statically detect data races [6]. In their approach, the compiler tracks held lock sets and checks lock guards on class fields. Although similar to keys and type guards, their system differs from ours in that lock acquire and releases have to be syntactically scoped using a synchronize expression. Thus, a method call cannot change the lock set. Furthermore, locks are not first class values, but a restricted form of syntactic expressions. Thus it is not possible in their system to pass an object and a separate lock protecting that object to a method.

Like the Vault project, the SLAM project at Microsoft Research is also focussed on using exhaustive static analysis to enforce protocols of low-level software [1]. Unlike Vault however, SLAM focusses on existing software written in C. The SLAM tools use an iterative approach: the SLAM tools create an ever more precise abstraction of the C program and use a model checker to search this abstraction for protocol violations. This interative refinement stops when either a violation is found, or no violation is present in the abstration, or a limitation of the tools has been reached.

In the context of the Metal project, Engler et. al. use programmer written compiler extensions to check properties of code at compile time [5]. The properties their system is able to check are similar to the ones described here, e.g., proper matching of lock acquire and release. In contrast to Vault, the Metal approach relies on syntactically recognizing state transitions, such as lock acquire and release, by matching against the names of specific functions. While sufficient for checking stylized properties such as acquiring and releasing a lock within the same function, the approach would require annotations similar to the ones proposed here to check invariants that are established inter-procedurally.

[11] Walker [20] shows how to express lists with pointers to the last element explicitly, but the point here is that such alias relationships would need to be anticipated, whereas they don't in Sagiv et. al.'s work.

6. CONCLUSIONS AND FUTURE WORK

Our case study on Windows 2000 drivers gives us an initial confidence that the resource management features of Vault are sufficient to model "real world" interfaces. Nevertheless, we need to continue validating these features in other domains, like graphic interfaces and other parts of the kernel interface.

Providing resource management features in a new language rather than an existing one allows us to design the language to make type checking tractable. The downside of a new language is the investment in existing languages, both in terms of legacy code and in terms of training. We hope that by basing our syntax on the popular language C, we can leverage some of the training cost. Wrapping Vault interfaces around existing C code allows that legacy code to be reused. However, the wrapper code can be a new source of errors, and we are looking into tool support in this area as well. We are also considering adding keys to the new language C♯ currently being deployed within Microsoft.

Finally, the device driver, while complex, is only a single compilation unit. To ensure that Vault's typing rules are not so restrictive as to prevent useful programs, we are writing a front-end for Vault in Vault. This system is a multi-stage pipeline where each stage's results are stored in its own region. This experience will allow us to evaluate the burden of Vault's annotations and typing restrictions.

7. REFERENCES

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[2] E. Barendsen and J. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *13th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 761, pages 41–51, Dec. 1993.

[3] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In POPL'99 [13].

[4] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report 156, Compaq SRC, jul 1998.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, Oct. 2000.

[6] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, May 1999.

[8] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 33:5 in SIGPLAN notices, pages 313–323, June 1998.

[9] M. P. Jones. A theory of qualified types. *Science of Computer Programming*, 22(3):231–256, June 1994. Selected papers of the Fourth European Symposium on Programming (Rennes, 1992).

[10] M. P. Jones. Simplifying and improving qualified types. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, June 1995.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[12] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In K. Koskimies, editor, *Compiler Construction (CC'98)*, pages 302–305, Lisbon, 1998. Springer LNCS 1383.

[13] *Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 1999.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In POPL'99 [13], pages 105–118.

[15] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the 2000 European Symposium on Programming*, Mar. 2000.

[16] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, 19(5):478–485, May 1993.

[17] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *tose*, SE-12(1):157–171, Jan. 1986.

[18] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the 21st Annual ACM SSymposium on Principles of Programming Languages*, pages 188–201, Jan. 1994.

[19] D. Walker, K. Crary, and G. Morrisett. Typed memory management in a calculus of capabilities. *Transactions on Programming Languages and Systems*, 2001.

[20] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Proceedings of the 4th Workshop on Types in Compilation*, Sept. 2000.