Dependent Types for Enforcement of Information Flow and Erasure Policies in Heterogeneous Data Structures

Gordon Stewart
Princeton University
jsseven@cs.princeton.edu

Anindya Banerjee
IMDEA Software Institute
anindya.banerjee@imdea.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

ABSTRACT

We consider verification of information flow and erasure properties in programs with *heterogeneous* heap-based data structures, in the presence of procedures with local state. A heterogeneous data structure, such as a hash table implementing a medical record database, may store both secret and public data simultaneously. In contrast, extant work primarily focuses on homogeneous data structures which store data of a uniform security level. Heterogeneity, however, does not come for free. For example, standard implementations of hash tables do not support heterogeneity, and may leak sensitive information easily owing to hash collisions. In this paper we identify *unique representation* as a sufficient condition for a heterogeneous data structure to be leak-free, while simultaneously supporting abstraction and modularity in verification. As a case study, we implement and verify a novel uniquely-represented variant of heterogeneous hash tables. Furthermore, we demonstrate modular reasoning by showing how specifications of the hash table methods can be used in a client application; we thereby obtain abstract and concise formal proofs of erasure. We formalize our work in Relational Hoare Type Theory (RHTT), an expressive, higher-order imperative language and program logic embedded in the Coq proof assistant.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory — Semantics; D.4.6 [**Security and Protection**]: Access Controls, Information Flow Controls, Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions.

General Terms

Security, Verification, Languages

Keywords

Dependent Type Theory, Verification, Information Flow, Erasure, Data Structures

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PPDP '13, September 16 - 18 2013, Madrid, Spain. Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2154-9/13/09 ...$15.00.
http://dx.doi.org/10.1145/2505879.2505884.

1. INTRODUCTION

This paper considers verification of information flow and erasure policies [10, 11, 18, 23, 32] in programs that manipulate mutable, heap-based, data structures. One of the challenges for programming and enforcement of security policies in this setting is that data structures often need to be *heterogeneous*, i.e. store data of different security levels. For example, a patient record in a medical database may contain public (or *low* security) data, such as patient name and address, as well as confidential (or *high* security) data, such as diagnosis.

Extant type systems and logics for information flow security primarily target *homogeneous* data structures, where the data stored in the structures is either all low, or all high. A medical database, as in the example above, therefore must be factored into two separate parts, one for storing low fields and the other for storing high fields of the patient records. Moreover, the relationship between the two parts must be maintained in order to match patient names and addresses with diagnoses. Even if this can be done using current database technology, it leads to code which violates data abstraction: the factoring propagates throughout *client programs*. For example, a client program that stores a patient record into a hash table also must be factored. It has to introduce *two* hash tables, one for the low and another for high security fields, and then somehow maintain the connection between the two hash tables.

In this paper we study heterogeneous data structures in order to circumvent the above problems. Heterogeneity, however, does not come for free. To illustrate the difficulties that arise, consider the following scenario with two program modules A and B. A has some private state, a salary, which it is prepared to share with module B in order for B to compute A's tax rate via a method called `compute_tax`. B would like to store A's salary into a *heterogeneous* hash table in B's local state. A allows this storage, but wants to make sure that upon termination of `compute_tax`, no information about its salary can be leaked from B's local state. A's policy therefore allows B to access the secret salary provided B proves (i.e., provides evidence) that `compute_tax` *erases* A's salary and *all results that depend on the salary*, upon return.

However, forcing B to remove the salary from the hash table, and to erase any results that may depend on it, is itself insufficient to prevent leakage of information about the salary. Due to hash collisions, the insertion of the salary into the hash table may influence the positions of subsequently inserted *low* entries. Thus even if the salary is eventually removed, an attacker that can traverse the hash table, say via a computation that follows pointers into the data structure or via pointer arithmetic, may be able to reverse-engineer useful information about the salary simply by inspecting the relative positions of the surviving low elements. For example, in an open-addressing hash table with linear probing, collision of a newly

inserted low security element with a high security entry already in the table will change the index at which the low element is placed. A concrete attacker program could follow pointers into the hash table array in order to determine the index of the low security entry, and thus determine the existence of the high element, and perhaps its hash value, even after the high element is deleted.

To address the above problem, one might consider encapsulating the internals of the hash table by means of type abstraction. Then, the only way to access the hash table is by invoking its API methods for lookup, insertion and removal of elements. The relevant aspects of the layout of the structure, e.g. the pointer linkage or the distance of elements from the origin of the open-addressing hash table, cannot be directly obtained, and thus cannot be a source of information leak. However, even type abstraction does not quite suffice in the above scenario. A has to make a decision on whether to grant B access to its secret, and we want B to provide a certificate of lack of leaks. Thus, B must *prove* that it only uses data structures which are *well-encapsulated* by type abstraction. While recently there has been a flurry of work on developing logics for parametricity [5, 6], the goals of which include reasoning about encapsulation via type abstraction, we are unaware of a system in which parametricity has been reconciled with information flow (i.e., absence of leaks) and mutable state.

In this paper, we therefore advance *unique representation*, a notion that applies irrespective of whether a data structure is well-encapsulated by type abstraction, as an alternative for stating and certifying the absence of leaks in heterogeneous data structures. Unique representation (UR, [15, 16, 30]) ensures that if the logical representations of B's hash table—e.g., as a set of key-value pairs—at any two program points are indistinguishable, then the concrete heap layouts will be indistinguishable as well. In other words, information leakage through the layout of a UR data structure, as in the linear probing hash table example described above, is impossible. Moreover, we show that we can formally prove using a proof assistant (in our case, Coq) that programs respect such information flow policies. We can do so modularly, verifying clients of hash tables separately from the implemenation of hash tables, while exposing only that the hash table satisfies the UR property. Our results apply not just to hash tables, but to many other heterogeneous data structures as well.

The technical details of our usage of UR data structures are as follows. We specify and statically enforce A's policy above using *higher-order* methods [24]. A's interface exposes a higher-order method, endorse, which takes a method (e.g. B.compute_tax) as an argument and grants the argument method access to A's salary, provided that the argument method comes with a proof that its local state, upon termination, is independent of the salary. This proof is statically checked at link time, and is developed in a higher-order relational variant of Hoare logic. The logic is relational because independence is a relational notion: semantically it means that any two runs of compute_tax executed from indistinguishable initial states result in indistinguishable final states. With this in mind, we make the following technical contributions:

1. We observe that, in a language with pointers such as RHTT, the proof required by A.endorse guarantees proper erasure and absence of leaks only if B carries it out with respect to the *concrete* states (i.e. concrete heaps), as it is the leakage through the layout, or linkage, of the data structure that we want to prevent. Unfortunately, proofs about concrete states are necessarily low-level. They are overly specific to the data structure implementation, thus preventing different implementations of one and the same library interface from being interchanged in the client code. We show that with UR data structures (Sec. 5), one may reason about the high-level logical representation of states, and due to the uniqueness of representation, the results directly transfer to the level of concrete heaps (Sec. 6 and Sec. 7).
2. To demonstrate that nontrivial, formally verified UR data structures are feasible, we develop a UR variant of hash tables called multilevel UR hash tables (Sec. 6) and verify its correctness and unique representation formally in Coq. Sec. 7 demonstrates how the multilevel UR hash tables of Sec. 6 can be used in a medical database application. To the best of our knowledge, this is the first formal *static* (as opposed to run-time) verification of a security property for a non-trivial data structure (but see Sec. 8 for discussion).

Uniquely represented data structures have been considered before in the algorithms and complexity communities, by Golovin [15], Naor and Teague [25] and others [8, 16, 20, 30]. In each of these cases, the data structures were motivated by the need to prevent an observer with access to the final state of a computation from reconstructing its history, and thus uncovering information that should remain private. In this paper, we show a different application of unique representation, namely, as a tool for *verifying programs* that manipulate heterogeneous heap-based data structures for conformance with information flow and erasure policies.

We use Relational Hoare Type Theory (RHTT, [24]) for both programming and proof throughout this paper. RHTT is a programming language and logic for specifying and enforcing expressive, state-based access control and information flow policies of imperative programs. Although RHTT is implemented as a domain-specific language in the Coq interactive theorem prover, we do not assume previous knowledge of either interactive theorem proving or of Coq. Sec. 3 presents all the features of RHTT necessary to understand the code in this paper. Our code and proofs are available at `http://www.cs.princeton.edu/~jsseven/papers/ur`.

2. ATTACK MODEL

For the purposes of this paper, attackers are well-typed RHTT programs that can read from the public components of initial and final (but not intermediate) states of a computation. For example, in the above case with A and B, we will consider B's hash table to be public *in the end*, on account of erasure of A's secret. Therefore, the attacker can observe the layout of the hash table, e.g., by following pointers into the data structure, after compute_tax has terminated, although during the tax computation the hash table may still contain secret data.

Technically, by "states" we mean the source-level representation of heaps given by the semantics of standard sequential program execution; we do not address erasure in compiled code, nor low-level erasure in hardware, or from cache lines, etc. in this paper. Following Nanevski et al. [24], we assume heaps are indistinguishable iff they are exactly equal, i.e., they contain equal addresses, storing equal values. Thus we also guard against attackers that can distinguish concrete representations of pointers as integer addresses, and perform pointer arithmetic.

3. BACKGROUND: RHTT

We briefly recapitulate the main components of the RHTT framework introduced by Nanevski et al. [24]. Fundamentally, RHTT is based on the following aspects of dependent type theory: dependent function types, inductive types and module systems. Dependent function types describe how a function body depends on its arguments; inductive types are needed to specify data structures

such as lists, trees, graphs, etc.; module systems including abstract types and predicates are needed for information hiding. Nanevski et al. show that these aspects jointly give RHTT the power to specify and verify flexible security policies: for example, one can specify the erasure-dependent access policy "A grants B the right to read its salary provided B proves that its code will erase all copies of the salary before termination." In Sec. 4, we adapt a medical records system from Borgström et al. [9] to incorporate erasure policies such as the one above. First we provide basic intuitions.

Noninterference, types, specifications. The specification of flexible security policies crucially depends on a relational specification of noninterference (NI) which says low outputs of a computation are independent of high inputs [12]. It is independence that is a relational property: Consider a function $f{:}A^2{\to}A^2$, where $A^2 = A \times A$. Let $e.1$ and $e.2$ denote resp. the first and the second component of the ordered pair e. Then, mathematically, f's first output is independent of f's second argument iff

$$\forall x_1\ x_2\ y_1\ y_2.\, x_1 = x_2 \to f(x_1, y_1).1 = f(x_2, y_2).1$$

In other words, in two runs of f, equal x inputs, lead to equal $f(x, y).1$ outputs. This relational statement of independence can be viewed as a statement about NI between f's second argument and f's first output [2, 4]. The consequence is this: rather than employ security lattices and orderings on variables to determine what is low security and what is not [3, 13], we consider inputs and outputs related by *equality* in the two runs of f above as *low* (x and $f(x, y).1$ above). The unconstrained values (y and $f(x, y).2$) are implicitly considered *high*. These ideas can be lifted to security lattices with multiple levels (see discussion in [24]), but we forgo such development here.

This reading of low security immediately lends itself to a type-theoretic interpretation and thus motivates RHTT's use of dependent type theory. The primary observation is that if information that x is low is *absent* at a module interface then x is possibly high. But this is precisely a notion of information hiding that is explained using standard constructs of type theory such as abstract types and abstract predicates [21].

The type $\mathsf{STsec}\ A\ (p, q)$ specifies heap-manipulating, potentially diverging RHTT computations. The type A gives the return type of an RHTT computation (int, string, etc.) whereas p and q define the program's precondition and postcondition respectively, as in Floyd-Hoare logic. Following Separation Logic [26], the precondition p is a predicate on heaps that defines the subset of memories in which it is safe to run the command. In particular, if a program typechecks in RHTT with precondition p, then every heap location accessed by the program (besides those allocated by the program itself) will be accessible in every heap described by p.

In order to specify relational properties such as NI, the postcondition $q{:}A^2 \to \mathsf{heap}^2 \to \mathsf{heap}^2 \to \mathsf{prop}$ relates *pairs* of return values (y_1, y_2) of type A^2 with pairs of initial heaps (i_1, i_2) and pairs of final heaps (m_1, m_2). Here heap is the type of *heaps*, modeled semantically as finite partial maps from locations to values, and prop can roughly be read as bool. Intuitively, the relational postcondition of the (termination-insensitive) STsec judgment says that if two runs from initial states i_1 and i_2 terminate in final states m_1 and m_2, returning values y_1 and y_2 as results, then the triple $((y_1, y_2), (i_1, i_2), (m_1, m_2))$ is in q. NOTATION: Examples in the sequel will often use the notation, yy, ii, mm to stand for pairs $(y_1, y_2), (i_1, i_2), (m_1, m_2)$ respectively. Also, we will view a predicate p on A as a characteristic function of a set of A-typed elements, and frequently write $a \in p$ instead of $p\ a$, where $a : A$.

An example erasure policy. Consider the following specification of an erasure policy, and a program conforming to the specification, adapted from Hunt and Sands [18], both presented here in a somewhat stylized RHTT notation. We consider two integer heap pointers x and y that are arguments to the program. The contents of x is initially low, and the contents of y is initially high. The policy is: the program must erase the high value stored in y. In RHTT, we express this policy with the type

$$\Pi x\ y{:}\mathsf{ptr}.\, \mathsf{STsec}\ [\,]\ \mathsf{unit}\ (p, q)$$

where the precondition, p, is

$$\mathsf{fun}\ i.\, \exists u\ v{:}\mathsf{nat}.\ i = (x \mapsto u \bullet y \mapsto v)$$

and the postcondition, q, is

$$\begin{array}{l}\mathsf{fun}\ rr\ ii\ mm.\, \forall uu\ vv.\\ \quad ii = ((x,x) \Mapsto uu \bullet\bullet (y,y) \Mapsto vv) \ \to\ u_1 = u_2 \to\\ \quad \exists uu'\ vv'.\, mm = ((x,x) \Mapsto uu' \bullet\bullet (y,y) \Mapsto vv') \wedge v_1' = v_2'\end{array}$$

(Recall our notation: uu is (u_1, u_2), vv' is (v_1', v_2'); We will explain the local context $[\,]$ momentarily.) The above STsec type is an instance of a *dependent function type*; it describes functions with arguments x and y of type ptr (pointers), as indicated by the variables following the Π-symbol. Such functions produce computations of type STsec with the listed pre- and postconditions; the type is dependent, because the arguments appear in the assertions in order to describe the policy.

The precondition p states that the program must start with an initial heap i containing two pointers x and y, with appropriately-typed contents. Here $x \mapsto u$ is the singleton heap containing only the location x storing value u; while $\bullet$ is *disjoint* heap union. As i is the disjoint union of smaller singleton heaps, there is no aliasing between the two pointers x, y. The postcondition binds over three variables $rr{:}\mathsf{unit}^2$ and $ii, mm{:}\mathsf{heap}^2$, which are, respectively, the pair of return values, the pair of initial heaps, and the pair of final heaps for the two runs of the program.

The notation $zz \Mapsto ww$ denotes a pair of singleton heaps $(z_1 \mapsto w_1, z_2 \mapsto w_2)$. Similarly $\bullet\bullet$ lifts $\bullet$ to pairs; that is, $jj\ \bullet\bullet\ kk = (j_1 \bullet k_1, j_2 \bullet k_2)$. Thus, the postcondition q states that if the contents of x in the two initial heaps are equal (hence low), then the contents of y are low in the end. This is only possible if the value stored in y is erased, by overwriting it with a constant, or with some other value computed from the initial (low) contents of x, *but not* y, as the initial contents of y are not declared low.

One program satisfying the above policy (i.e. type) is the following.

$$\begin{array}{l}\mathsf{fun}\ x\ y.\, \mathsf{do}\ (u \leftarrow \mathsf{read}\ x;\ v \leftarrow \mathsf{read}\ y;\\ \qquad\qquad \mathsf{if}\ v = 0\ \ \mathsf{then\ write}\ x\ (u+1)\ \mathsf{else\ skip};\\ \qquad\qquad \mathsf{write}\ y\ (u+2))\end{array}$$

In this program, in addition to purely functional constructs such as anonymous functions fun, we use side-effecting primitives such as write $x\ n$, which stores the value of n into the location x; read x, which returns the contents of x; and $x \leftarrow e_1; e_2$, which sequentially composes e_1 and e_2, substituting the return value of e_1 for x in e_2.

The above program satisfies the policy because the value stored into y at the end is low (the initial contents of x incremented by 2). The ending contents of x are not low, as they depend on the initial contents of y. Our policy, which does not specify the ending security level of the contents of x, allows this flow. Indeed, the program illustrates that the contents of memory cells may change their security level during the program run—a key characteristic of our semantic definition of NI.

Therefore, in RHTT, memory addresses are classified separately from their contents. For example, we can have a low pointer address whose contents is high. While the STsec type of the above program classifies the security of the *contents* of x and y, it doesn't classify the pointer addresses themselves, as the latter requires discerning the address names in two different runs. RHTT provides for this possibility by endowing the STsec type with a *local context*. This is a list of types for the arguments of the computation, whose significance we illustrate by an example.

To reflect that the pointer y is high at the beginning, in addition to having high contents, we change the original STsec type as follows.

$$
\begin{array}{l}
\Pi x{:}\mathsf{ptr}.\, \mathsf{STsec}\ [\mathsf{ptr}]\ \mathsf{unit} \\
\quad (\mathsf{fun}\ y\ i.\, \exists u\ v{:}\mathsf{nat}. \\
\qquad i = (x \mapsto u \bullet y \mapsto v), \\
\quad\ \mathsf{fun}\ yy\ rr\ ii\ mm.\, \forall uu\ vv. \\
\qquad ii = ((x, x) \Mapsto uu \bullet\bullet\ yy \Mapsto vv)\ \rightarrow\ u_1 = u_2 \rightarrow \\
\qquad \exists uu'\ vv'.\, mm = ((x, x) \Mapsto uu' \bullet\bullet\ yy \Mapsto vv') \wedge v'_1 = v'_2)
\end{array}
$$

Whereas in the original type, the local context was the empty list [ ], now it contains the element ptr, which is the type of the argument y. Correspondingly, the variable y is pushed inside the precondition, and similarly, yy is pushed into the postcondition, so that y_1 and y_2 refer to the addresses of y in two different runs (contrast with (y, y) in the postcondition of the original type). Since the postcondition does not include an equation $y_1 = y_2$, the address y is high. The program syntax changes too, as the local variables now have to be bound within the scope of do. In other words, our program now looks like:

$$\mathsf{fun}\ x.\, \mathsf{do}\ (\mathsf{fun}\ y.\, u \leftarrow \mathsf{read}\ x; \cdots)$$

In general, variables bound by Π are always low, whereas variables declared in the local context may be low or high, depending on the policy described by the type.

4. ERASURE-DEPENDENT ACCESS CONTROL

The erasure policies we described in the introduction grant one method access to some module's private data, provided the method furnishes a proof that it will eventually erase the private data and any derived secrets out of its local state. In this section, we demonstrate, through a concrete application to data storage, the difficulties that arise when the method's local state consists of mutable, heap-allocated data structures.

In order to do so, we first survey a database application containing medical records and a database client, both built in RHTT, and inspired by an example from Borgström et al. [9]. We do not describe the complete database system here, though it is fully developed in our Coq scripts. The primary motivation of this section is rather to to illustrate how the verification effort required to prove erasure policies scales as the local state maintained by modules evolves, from local scalar variables to heap-allocated data structures such as arrays or hash tables. This section also serves to motivate the use of unique representation.

4.1 Example: Data Store for Medical Records

Our medical database application grants access to patient records via the method read_emr (Fig. 1), which stands for "read electronic medical record." The access policy for this method requires that its caller owns a token, called can_read_emr, for the patient whose records it wishes to access. Access to this token is limited to methods of modules from an appropriate group of users (e.g., doctor) who have previously requested (and been granted) permission by

```
Module A.
  read_emr : ... //requires can_read_emr token
  endorse : ∀ (R        : type)
              (G        : type)
              (ccshape : R² → G² → heap² → prop)
              (cmp      : G → G)
      ⎧ //input: STsec program requiring can_read_emr
      ⎪ (e : STsec [database, R, user] int
      ⎪      (fun db r pat i.
      ⎪         ∃(g:G) (u:user) (j t:heap).
      ⎪           can_read_emr (get_uid u) (get_uid pat) ∧
      ⎪           i = j • t ∧
      ⎪           j ∈ shape db u  ∧  t ∈ cshape r g,
    T ⎨       fun ddb rr ppat yy ii mm.
      ⎪         ∀gg uu jj tt. ii = jj •• tt →
      ⎪           jj ∈ sshape ddb uu →
      ⎪           tt ∈ ccshape rr gg →
      ⎪         ∃tt':heap². mm = jj •• tt' ∧
      ⎪           tt' ∈ ccshape rr (cmp g1, cmp g2) ∧
      ⎪           //no leakage of high data
      ⎩           (t1 = t2 → t1' = t2'))).

      ⎧ //output: equivalent program w/o can_read_emr
      ⎪ STsec [database, R, user] int
      ⎪   (fun db r pat i.
    S ⎨      ∃(g:G) (u:user) (j t:heap).
      ⎪        i = j • t ∧
      ⎪        j ∈ shape db u  ∧  t ∈ cshape r g,
      ⎩    fun ddb rr ppat yy ii mm.  ··· ) ≙
  do (fun db r pat. e db r pat)
  ...
End Module A.
```

Figure 1: Erasure in a database application. The function A.read_emr **provides access to medical records, but only to client methods that possess the** can_read_emr **token. The function** A.endorse **takes as input an** STsec **computation requiring the** can_read_emr **token, and produces as result a new computation that does not require** can_read_emr**. Client methods that call** endorse **must prove that they do not leak confidential information ($t_1 = t_2 \rightarrow t'_1 = t'_2$). Thus it is safe for such client methods to access private information in the database using** endorse **even if they are not pre-authorized to do so.**

```
Module B.
  R ≙ ptr
  G ≙ int
  ccshape (rr:R²) (gg:G²) (ii:heap²) ≙ ii = rr ⤇ gg
  cmp : G → G ≙ fun g. g + 1
  prog : T ≙ do (fun db r pat.
      emr ← A.read_emr db pat;
      g ← read r; write r (f1 emr); write r (g + 1);
      ret (f2 emr))
  counting_client : S ≙ A.endorse R G ccshape cmp prog
End Module B.
```

Figure 2: Method B.counting_client **calls** A.endorse **in order to access confidential database records without pre-authorization. On each call to** endorse**,** counting_client **stores a hash of the confidential patient record** emr **into its local integer state (**write r $(f_1\ emr)$**), then overwrites this hash with the incremented count** $g + 1$ **(**write r $(g + 1)$**). In order to typecheck** counting_client**, the programmer must prove that** prog **leaks into its final local state no data derived from** emr **(this is indeed the case).** f_1 **and** f_2 **here are integer hash functions.**

the patient. With this in mind, consider the following generalization of the policy, along the lines of the policy on A's salary from the introduction: a client method may read confidential patient records even without the can_read_emr token, but only if it erases any data derived from these records before it terminates.

To conservatively enforce that a client method does not steal patient records, the medical records database could require that the clients deallocate all local state on function return, and thereby ensure that the client methods steal nothing, but this is prohibitively expensive. If a client makes many thousands of calls to the database, allocating and then deallocating local state on each call becomes infeasible. Requiring deallocation also prohibits the client methods from maintaining persistent, but innocuous, local state, e.g., statistics about the CPU usage. In general, any client local state not derived from the confidential data should be allowed to escape a call to the database. In RHTT, higher-order STsec types allow us to formulate such permissive policies.

Fig. 1 gives the database-specific definitions required to support erasure-dependent access control: namely, the function endorse; while Fig. 2 presents a client method that calls endorse in order to access confidential patient medical records. The endorse function is parametric in several arguments, which the client methods instantiate to their choosing.

- Type R circumscribes the shape of the client method's local heap, typically by encoding the number of required root pointers (e.g., R equals ptr if the client's local heap requires a single root pointer, ptr^2 for two root pointers, etc.)
- Type G defines what kind of values are stored in the client method's local state. For example, G may equal int if the local state encodes a single integer value, or G may be list int, if the local state stores integers in a linked list, etc.
- Predicate ccshape describes the shape in which the data (of type G) is layed out in the client method's local heap, starting from the root pointers as given by the type R. The predicate ccshape is relational (i.e., its arguments are "squared"), so that it can specify the security levels of the pointer addresses and their contents in the client's local heap. In Fig. 1 we also make use of the relational predicate sshape which represents the shape and security levels of the local state of the *database*, similar to how ccshape does so for the clients. The sshape predicate is local to the database module, but is not a parameter of endorse, as it is not something that the clients can choose.
- Function cmp defines how the values given by G evolve after each call to the database. For example, letting $G = \mathsf{int}$ and $\mathsf{cmp} = \mathsf{fun}\ g.\ g+1$ corresponds to the client method storing an integer count g as its local state which increments g after each call.

The body of endorse's specification puts everything together. It takes as argument an STsec computation of type T and produces a computation of type S. The precondition in T demands that the computation works over the local states of the database and the client only. It does so by requiring that the initial heap i can be split into disjoint parts j and t. j "belongs" to the shape predicate of the database, and t "belongs" to the shape predicate of the client. Here shape and cshape are non-relational versions of sshape and ccshape, respectively. For example

$$\mathsf{cshape}\ r\ g = \mathsf{fun}\ i.\ (i, i) \in \mathsf{ccshape}\ (r, r)\ (g, g)$$

Importantly, the precondition in T requires a proof of the abstract predicate can_read_emr, which serves as a permission, or a token, necessary to call the database function read_emr. The precondition of S is similar to T, except that it omits the token can_read_emr. Thus, while the input to endorse is a method that requires a permission, and could ordinarily be executed only by a call from a method of a user (e.g., a doctor) that has been granted the token, the output is a method which needs no permission.

Crucially, endorse will only accept as input those client methods which *come with a proof* that the client's final local state *does not depend* on any secrets that the method computed during the call (the "no leakage of high data" condition). We express this requirement in the postcondition of R: if the client method's local heaps t_1 and t_2 were equal initially (low), then the client method's final heaps t_1' and t_2' must also be equal in the two runs ($t_1 = t_2 \to t_1' = t_2'$).

The proof of such a property is derived by applying relational Hoare-style reasoning (omitted here, but present in our Coq scripts) on the code of the client method. In case the client method contains outgoing calls to functions from its own module, or potentially other modules, we use the specification of each function to verify that the client method satisfies the erasure property, and separately prove that each of the functions that is called satisfies its specification.

Counting Client. Fig. 2 defines a client which maintains as local state a count of the number of times it has been called. The type $R = \mathsf{ptr}$ declares a single root pointer, which is a pointer to this count. The shape invariant ccshape specifies that the heaps i_1 and i_2 in the two runs (recall that $\Mapsto$ operates on pairs of pointers and values) are singleton heaps from r_1 and r_2 to the integer counts g_1 and g_2 respectively. The function cmp just increments the count after each call.

The client program prog operates as follows: it first reads the medical record of a patient, pat, from the database and stores the resulting value in the variable emr. Next, it dereferences the pointer r into a second local variable g, for safekeeping. It now stores an integer hash of the medical record emr into r. If prog were to return at this point, it would not satisfy the required noninterference property (i.e., $t_1' = t_2'$) since emr is confidential data and therefore not known to be equal in the two runs. To ensure that t_1' *does* equal t_2'—and that the count is properly updated—prog overwrites the value in the client local heap a second time with $g + 1$, the incremented count. It then returns $f_2\ emr$, a second integer hash of pat's medical record.

The program counting_client applies endorse to prog. On its own, prog would have had to request permission from a patient in order to call read_emr, thereby establishing the function's precondition. By endorsing prog, we avoid the need to establish this precondition, but at the expense of the additional proof obligation $t_1 = t_2 \to t_1' = t_2'$. In this case, demonstrating that prog meets this specification is straightforward: even though the program stores the confidential value $f_1\ emr$ into local state at an intermediate point during the computation, it overwrites this value with a low value ($g + 1$) before function return. Because counting_client's local state consists of a single pointer to a scalar value, proving that $t_1' = t_2'$ is relatively easy. Indeed, this proof in RHTT is only a few lines.

Issues with Extending to Hash Tables. Now consider the situation in which r is not an integer pointer but a hash table, and the sequence write r ($f_1\ emr$); write r ($g + 1$) is replaced by

$$\mathsf{insert}\ r\ (f_1\ emr); \mathsf{insert}\ r\ (g+1); \mathsf{remove}\ r\ (f_1\ emr).$$

If emr is not known to be equal in the two runs, then ($f_1\ emr$) may equal $g + 1$ in one run—leading to a collision—but not the

other, leading to two different hash table layouts in the two runs. In general, operation sequences on data structures that are invertible at the level of the data structure interface—such as insertions of key-value pairs into a hash table followed by their removal—can leave traces in the underlying heap representation, and therefore make it difficult to prove noninterference.

For the enforcement of erasure properties, it therefore becomes important to impose additional conditions on the data structure implementation. In particular, we must know that any series of invertible data structure operations leaves the underlying heap representation unchanged. Otherwise, the mere erasure of items from the structure can leak information through the layout, as we disscussed previously. In the next section, we focus on *unique representation* as a particularly simple property to enforce and verify of a data structure, which guarantees the above behavior.

Unique representation alleviates the problem just described by making it possible to reason *symbolically* about (the absence of) such flows: by ensuring that *logical* equality of the data structure states in two runs implies *actual* equality of the underlying heaps, it reduces noninterference proofs to arguments about equalities of symbolic, mathematical representations. This reasoning can often be performed elegantly, through the application of algebraic laws. Moreover, proofs constructed in this way are portable to future implementations satisfying the mathematical interface.

In the next two sections, we first introduce unique representation more formally, then present a UR variant of open-addressing hash tables which guarantees that the heap leaves no trace of the history of operations performed. This "history independence" property—a key consequence of unique representation—makes our hash table variant amenable to proofs of erasure properties such as those required by endorse, and to noninterference proofs more broadly.

5. UNIQUE REPRESENTATION

In this section, we formally define unique representation and explore its implications. We then describe an implementation of insert-only bounded multisets as an example of a naturally UR data structure. Sec. 6 presents our implementation of a UR variant of hash tables. First, we develop general definitions.

Data Structures. A data structure ties a logical interface (an abstract type and operations on that type) to a concrete implementation of the interface. One can model a data structure mathematically as a tuple consisting of the type G—of logical or abstract states—and the type of concrete states. For us, the concrete states will always be heaps, whereas logical states may range from integers (as in Sec. 4), to vectors that model arrays, to finite key-value maps that model data structures such as hash tables. As in Sec. 4, we also need a third type, R, to store the root pointers of the structure.

Each pair of a data structure's logical states and root pointers defines a *set* of concrete states that implement it. When a root pointer r and logical state g are implemented by heap h, we say that the *shape* of r and g is h. The predicate

$$\mathsf{shape} : R \to G \to \mathsf{heap} \to \mathsf{prop}$$

on representation types R, logical states G, and heaps, gives the formal definition of the shape relation in our development.

Unique Representation. A UR data structure [15, 16, 25] in this context is one for which the relation shape uniquely determines the heap h.

DEFINITION 1 (UNIQUE REPRESENTATION). *A data structure is* uniquely represented *iff for all root pointers r, logical states g and heaps h_1 and h_2, if $h_1 \in \mathsf{shape}\ r\ g$, and $h_2 \in \mathsf{shape}\ r\ g$, then $h_1 = h_2$.*

As an example of a shape predicate that is naturally UR, consider the definition of shape for standard imperative arrays. Arrays are the simplest UR data structures, and will be the basic building block for all the other UR structures in the paper.

$$\begin{aligned}&\mathsf{shape}\ (r{:}\mathsf{array}\ n\ T)\ (g{:}I_n \to T)\ (h{:}\mathsf{heap})\ \widehat{=}\\ &\quad h = (r + 0 \mapsto g(0),\\ &\qquad\quad r + 1 \mapsto g(1),\\ &\qquad\quad \ldots\\ &\qquad\quad r + (n-1) \mapsto g(n-1))\end{aligned}$$

n is the size of the array, and is a parameter of the definition, as is the type T of elements stored in the array. The finite type $I_n = \{0..n-1\}$ enumerates the indices of the array. The root type $R = \mathsf{array}\ n\ T$ is implemented as a single pointer r storing the base of the array. The type of logical states G is $I_n \to T$, so that $g : I_n \to T$ logically represents the contents of the array as a finite function (i.e., vector) of n, T-typed elements. The predicate shape on heaps asserts that the array consists of contiguous memory blocks rooted at a pointer r; h is the heap that contains value $g(0)$ at location $r + 0$, $g(1)$ at location $r + 1$, and so on. This shape predicate is naturally UR: for a given r and g, the heap h that results is always uniquely determined.

Security Implication of UR. Why is unique representation a useful property for security? The answer is: An attacker with access to the concrete representation of a UR data structure can learn no more than an attacker constrained by the data structure's public interface. In other words, *different* sequences of operations that result in the *same* logical state leave no trace of the history of operations that got them there. In particular, any sequence of operations that is then reverted (e.g., a series of insertions in an array followed by removals of all these inserted elements) is equivalent, at the level of concrete heaps, to having never performed the operations in the first place.

5.1 Example: Insert-only Bounded Multisets

Molnar et al. [22] describe the architecture of a vote storage unit that achieves operation-order independence, and thus voter anonymity, by storing votes in insert-only bounded multisets. In this section, we illustrate an RHTT implementation of insert-only bounded multisets as another example of a UR datastructure, which builds on arrays.

We implement multisets whose elements are integers bounded by n, as arrays with n cells, each containing a natural number. The value of the natural number at each index i in the array denotes the number of times element i appears in the multiset, or its *multiplicity*.

Fig. 3 gives the signature and RHTT implementation of the insert-only bounded multisets module. The type mset is an alias for arrays with size n and cells of type nat. The predicate shape takes a multiset r—a pointer to the base of the array—and a finite function g as arguments. The latter encodes the multiplicity of each multiset element. The shape predicate is implemented by calling the corresponding shape predicate of the array module, which specifies that r is the base of a contiguous memory region of size n with contents given by g.

The precondition of insert defines when it is safe to execute the method, i.e., $\mathsf{shape}\ r\ g$ must hold of the input heap i and the base

```
Module MultiSet.
  mset ≙ array n nat
  shape (r:mset) (g:I_n → nat) ≙ Array.shape r g
  bump (k:I) (g:I_n → nat) ≙ g[k ↦ g(k) + 1]
  new : STsec [ ] mset ...
  insert (r:mset) :
    STsec [I_n] unit
      (fun k i. ∃g. i ∈ shape r g,
       fun kk yy ii mm. ∀gg.
         ii ∈ sshape (r, r) gg →
         mm ∈ sshape (r, r) (bump k₁ g₁, bump k₂ g₂)) ≙
    do (fun k. x ← Array.read r k;
               Array.write r k (x + 1))
End Module MultiSet.
```

Figure 3: Signature and implementation of insert-only multisets. The module is parameterized by $n : \mathsf{nat}$, which is the bound on the number of elements of the multiset. The specification and implementation of the method new are elided.

pointer r for some finite function g. In insert's postcondition, the proposition $mm \in \mathsf{sshape}\ (r, r)\ (\mathsf{bump}\ k_1\ g_1,\ \mathsf{bump}\ k_2\ g_2)$ states that in the two runs, the output heaps mm contain the local state of the multiset r, the contents (i.e., multiplicities) of whose indices k_1 and k_2 have been incremented by 1. In the case of arrays and multisets, the relational predicate sshape is defined as a "duplication" of shape, that is $\mathsf{sshape}\ rr\ gg\ hh = \mathsf{shape}\ r_1\ g_1\ h_1 \wedge \mathsf{shape}\ r_2\ g_2\ h_2$.

The code for insert first reads the contents of the array at index k into x by calling the array read operation Array.read. Next, it writes $x + 1$ at index k by calling the array write operation Array.write. In our RHTT implementation of arrays, Array.read and Array.write are implemented via pointer arithmetic on the base address.

The proof (in the Coq scripts) that bounded multisets are UR follows from the fact that multisets are implemented as arrays, and the latter are UR as explained previously.

6. MULTILEVEL UR HASH TABLES

Open-addressing hash tables—those that use probing to resolve collisions—are a key imperative data structure with wide-ranging applications. However, such hash tables are unfortunately not UR: key collisions can create different memory layouts depending on the order in which keys are inserted, and thus leak information. When modules use hash tables to store their local state, it therefore becomes quite difficult to prove the sorts of information flow and erasure specifications that appeared in earlier sections of the paper.

Inspired in part by a particular kind of open-addressing hash table, called a *filter hash* (cf. Fotakis et al. [14]), in this section we present a variant of standard open-addressing hash tables, called *multilevel UR hash tables*, that *is* uniquely represented. Although we will describe below the filter hash tables of Fotakis et al. in order to introduce our UR multilevel variant, we note at the outset that our UR multilevel hash tables do not have the same complexity profile as the Fotakis et al. filter hash tables. Indeed, the primary purpose of this section is not to demonstrate that UR data structures present efficient alternatives to standard data structures; Golovin, in his Ph.D. thesis [15], has already demonstrated this point by presenting a variety of efficient UR data structures (though the particular hash table variant we implement in this section has not been analyzed before). The purpose of this section is, rather, to demonstrate that it is *possible*, in a fully mechanized system such as RHTT, to prove nontrivial data structures uniquely represented. Section 7 then revisits how unique representation can be used in practice to prove erasure and information flow properties of modules that employ multilevel UR hash tables in their local state.

6.1 Canonical Representation

Standard filter hash tables were first proposed by Fotakis et al. [14] as a simple yet efficient variant of m-level cuckoo hash tables. They consist of m levels, or *filters*, each of which has an associated hash function hash and an array for storing key-value pairs, just as in standard open-addressing hashing. When a collision occurs in the top filter of the table (i.e., level 0), insertion is reattempted at a lower level (e.g., level 1) until either the insertion succeeds or the table runs out of filters. In the latter case, the item is placed in a secondary key-value map. As we mentioned above, filter hash tables unfortunately suffer from the same kinds of information leaks as standard open-addressing hash tables. It is therefore difficult to prove erasure and information flow specifications of modules that use filter hash tables to define their local state.

Multilevel UR hash tables are just like filter hash tables except that (1) they impose a *canonicity* invariant on the layout of the keys at each level of the table; and (2) they employ a single hash function across all levels, whereas filter hash tables use n hash functions for n levels. Condition (1), the canonicity invariant, is what makes our multilevel hash tables uniquely represented. We explain canonicity in more detail below. Condition (2) is slightly more *ad hoc*. If our hash table variant did not implement deletion, then condition (2) would be unnecessary. However, as it currently stands condition (2) enables the implementation of efficient deletion, while most likely adversely affecting the space usage of the data structure. We are not aware of a better UR algorithm; this is an open research question.

It is also possible to define uniquely represented *chaining* hash tables, by ensuring that the linked list that implements each bucket in the table remains sorted at all times. Although RHTT supports dynamic allocation, reconciling allocation with information flow and erasure is quite tricky (cf. Golovin [15]). For example, pointer comparisons make even the *order* in which memory blocks are allocated observable. Considerations such as these prompted our investigation of the multilevel alternative tables we describe in this section.

To get an intuition for the canonicity invariant, imagine an m-level hash table is split into two parts: (1) the top level d_0 and (2) the remaining levels $d_1 \ldots d_{m-1}$, together with the auxiliary store b, which we call the "bucket". Let $<$ be a total order on keys.

DEFINITION 2 (CANONICITY). *An m-level hash table is* canonical *when the following properties hold for all $0 \leq i < m$:*

- *For all keys k and k', if (a) k is stored in level d_i, (b) k' is stored either in a strictly lower level or in the bucket, b, and (c) $\mathsf{hash}(k) = \mathsf{hash}(k')$, then $k < k'$.*
- *For all indexes j, if $d_i[j]$ is empty, then there is no key k such that (a) $\mathsf{hash}(k) = j$, and (b) k is stored either in a strictly lower level or in the bucket, b.*

In other words, canonicity requires that keys be placed as high up in the multilevel hash table as possible and that whenever there is a collision during insertion, a suitable strict order $<$ be used to resolve the collision. Of course, deletion must preserve the canonicity invariant as well. One consequence of this property is that all occupied slots in the top level d_0 always contain the minimum key among all keys currently in the table that would also have hashed

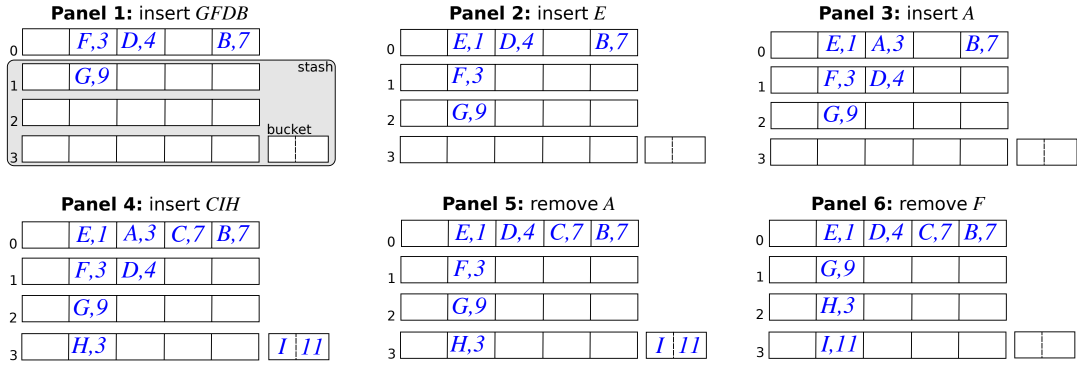

Figure 4: 4-level UR hash table containing keys A-I. Any two insertion orders, e.g., $ABCDEFGHI$ and $GFDBEACIH$, produce the same memory layout. Each additional level added to the structure implements, along with those beneath it, the kv-map interface, yielding a modular construction. For example, in Panel 1, levels 1-3 and the auxiliary kv-map, bucket, together form the stash for the top level 0.

to that slot. Nonminimum keys that collided with the minimum key upon insertion must be placed in lower levels of the hash table. Intuitively, canonicity implies unique representation since: (1) the keys in every hash collision path are uniquely ordered by the $<$ relation; and (2) empty slots are always filled in order, topmost first.

To maintain the canonicity invariant, we ensure that the minimum key for each index is always stored in d_0 (and so on for levels 1 through $m-1$, for the remaining keys). This requires checking whether a newly inserted key k' with $\mathsf{hash}(k') = j$ is less than the current key in slot j of d_0. If it is, then k' is placed in d_0 and the current key k in slot j is evicted to a lower level. If $k' > k$ then k' is inserted into a lower level as usual. When $k = k'$, the new value v' associated with k' overwrites the current value v (we do not maintain duplicate keys). Deletion is somewhat trickier: when a key k is removed from slot j in level d_0, we must find the smallest key k' at level d_1 such that $\mathsf{hash}(k') = j$ and recursively move this key up into level d_0.

Fig. 4 illustrates insertion of the keys $GFDBEACIH$, paired with their values, into a 4-level UR hash table. Key G is inserted first. Then key F is inserted, causing a collision with G in the topmost level. Since $F < G$ in the usual lexicographic order, G is evicted from d_0 and re-inserted into d_1. Keys D and B are inserted next, without incident, into level d_0. The state of the table after G, F, D and B have all been inserted is shown in Panel 1 of the figure.

Next, key E is inserted in slot 1 of d_0, causing a second collision with F. The eviction of F and its re-insertion into level d_1 causes a chained collision with G, which is evicted now for the second time into level d_2. The state of the table after E has been inserted is shown in Panel 2. Key A's insertion causes the eviction of D into a free slot in level d_1 (Panel 3). The rest of the keys are inserted in a similar manner, resulting in the multilevel UR hash table shown in Panel 4 of the figure. Note that the table that results after all keys have been inserted is equivalent to the table resulting from insertion of the keys in order, $ABCDEFGHI$, or in any other permutation. In Panel 4, key I is placed in the auxiliary key-value map, bucket, since it collided with other keys at every level of the table during insertion. Panels 5 and 6 illustrate deletions. In Panel 5, key A is deleted, causing D to be moved up into level 0. In Panel 6, key F is deleted, causing a chain of three reorderings: key G is moved from level 2 to level 1, key H is moved from level 3 to level 2 and key I is moved from the bucket into level 3.

```
Module MultilevelURHashPrelim.
 hashmap ≙ (array n (option (K × V)) × kvmap)
 minkey (g:K ⇀fin V) (ix:I_n) ≙
   min (filter (fun k. hash k = ix) (keys_of g))
 hashtab (f:K ⇀fin V) ≙ fun (ix:I_n).
   if minkey g ix is ⌊k⌋ then
       if fnd k g is ⌊v⌋ then⌊(k, v)⌋ else None
   else None
 stash (g:K ⇀fin V) ≙ foldr (fun ix g₀.
   if hashtab g ix is ⌊(k, v)⌋ then rem k g₀
   else g₀) g [0. . n − 1]
 shape (r:hashmap) (g:K ⇀fin V) (i:heap) ≙
   ∃j k:heap. i = j • k ∧
       Array.shape r.1 (hashtab g) j ∧
       KVMap.shape r.2 (stash g) k
 sshape (rr:hashmap²) (gg:(K ⇀fin V)²) (ii:heap²) ≙
   shape r₁ g₁ i₁  ∧  shape r₂ g₂ i₂
End Module MultilevelURHashPrelim.
```

Figure 5: Preliminary specifications and definitions for multilevel UR hash tables. The module is parameterized by the types K and V, n : nat, and a hash function hash : $K \to I_n$. The functions hashtab and stash define the reference functional implementation of the partitioning of key-value pairs into the top level and the stash. Functions fnd, ins and rem are for look up, insertion and removal of a key in a finite map such as g.

6.2 Preliminary Definitions for Multilevel UR Hash Tables

Now we turn to the definition of multilevel UR hash tables in RHTT. Fig. 5 gives the preliminary RHTT definitions used in our implementation and in the correctness and unique representation proofs. We present these definitions first, before the imperative RHTT code for hash table lookup, insertion and deletion, because the definitions are important for understanding the specifications of these functions.

The function hash of type $K \to I_n$ maps keys $k{:}K$ to indices of type I_n. Here K is a type with a built-in order operation. For

example, K could be instantiated as the type of integers with the usual $<$ relation as the order. Indexes of type I_n are integers in the range 0 to $n-1$, inclusive. These indexes define the slots in the arrays that are used to implement each level of the hash table. Fixing the n parameter, as the above definitions do, forces each level in the hash table to have the same size.

Hashmaps (hashmap) are pairs of objects, an array defining the uppermost level in the hash table (level 0 in Fig. 4) and a "hashable" key-value map called the stash, defined by the type kvmap (highlighted in gray in Fig. 4). The stash includes all of the levels besides the topmost one, as well as the bucket. The type option $(K \times V)$ defines optional pairs of keys and values. These pairs can either be None, meaning no pair at all, or $\lfloor(k, v)\rfloor$, pronounced "some", meaning an actual pair (k, v). Hashable key-value maps are just like standard key-value maps except that they expose—in addition to methods for lookup, insertion and deletion of key-value associations—a hash function on keys. We use this hash function parameter to fix the hash function for each level. Our construction of multilevel UR hash tables is modular in that they are parameterized by a stash and themselves implement the hashable key-value map interface. This means we can construct multilevel UR hash tables of depth m by instantiating the stash with a second multilevel hash table of depth $m-1$. For the bucket, we use a simple functional implementation of the hashable key-value map interface as sorted association lists. The only requirement is that the bucket be UR, for which we sort the key-value pairs and remove duplicates.

The next four functions in the figure together define the shape predicate for multilevel UR hash tables. The function minkey takes two arguments, a finite map g from keys to values and an index ix, and returns the minimum key in g, if one exists, that hashes to ix. To find the keys that hash to ix, it first filters the keys of g (keys_of g) by the predicate (fun k. hash $k = ix$). The hashtab and stash functions partition the key-value pairs in g into the toplevel array and the stash. The hashtab function constructs g's toplevel array by building a new (finite) function that returns at index ix, $\lfloor(k, v)\rfloor$, whenever k is the minkey for g at index ix. Otherwise, hashtab returns None at that index. The stash function operates by iteratively removing any keys already in the toplevel table. For each index ix in the range $[0..n-1]$, it removes k from g whenever the hashtab of g already contains a key-value pair at index ix. Thus the stash includes all key-value pairs that were not already selected for inclusion in the toplevel. Finally, shape (r:hashmap) (g:$K \rightharpoonup_{\mathsf{fin}} V$) asserts that the heap, i, can be split into two subheaps, j and k, such that j has the shape of the array given by hashtab g (Array.shape $r.1$ (hashtab g) j) and k has the shape of the key-value map given by

stash g (KVMap.shape $r.2$ (stash g) k).

Note that our implementation of multilevel UR hash tables requires that KVMap.shape be UR.

6.3 Implementation

Fig. 6 presents our RHTT implementation of multilevel UR hash tables, parameterized by a key-value map module Stash, the signature of which is given in Fig. 7. Function new is the shortest of the four functions in the interface. It calls the new functions from the array module and from the key-value map module to allocate a new array, r_1, and a new stash, r_2. The array is initialized to contain values of type option $(K \times V)$, with default value None. The stash is initially empty.

The lookup function takes a key k as its sole local argument. It first looks up the value in the top level, $r.1$, at position hash k using the array read method. Array.read returns an option, o, so

```
Module MultilevelURHashTab.
 new ≙ do (r₁ ← Array.new n None;
            r₂ ← Stash.new;
            ret (r₁, r₂))

 lookup (r:hashmap) ≙
   do ( fun k. o ← Array.read r.1 (hash k);
        if isNone o then ret None
        else  if (o is ⌊(k′, v′)⌋ && k = k′) then ret ⌊v′⌋
              else Stash.lookup r.2 k)

 insert (r:hashmap) ≙
   do (fun k v. o ← Array.read r.1 (hash k);
       if  (isNone o || (o is ⌊(k′, v′)⌋ && k = k′))
       then Array.write r.1 (hash k) ⌊(k, v)⌋
       else  let ⌊(k′, v′)⌋ = o in
             if k < k′ then Stash.insert r.2 (k′, v′);
                            Array.write r.1 (hash k) ⌊(k, v)⌋
             else Stash.insert r.2 k v

 remove (r:hashmap) ≙
   do (fun k. o ← Array.read r.1 (hash k);
       if (o is ⌊(k′, v′)⌋ && k = k′)
       then  k_min ← Stash.minkey r.2 (hash k);
             if isNone k_min then Array.write (hash k) None
             else  let ⌊k′_min⌋ = k_min in
                   ⌊v′_min⌋ ← Stash.lookup r.2 k′_min;
                   Stash.remove r.2 k′_min;
                   Array.write r.1 (hash k) ⌊(k′_min, v′_min)⌋
       else Stash.remove r.2 k
End Module MultilevelURHashTab.
```

Figure 6: RHTT implementation of multilevel UR hash tables. The module is parameterized by n : nat, which is the size of the levels in the hash table, and a key-value map Stash.

lookup must first check whether o is None—meaning the toplevel array is empty at position hash k—before proceeding. If o is None, then lookup immediately returns None. It can do so because the canonicity invariant implies that k is not present: if k *were* present, it would have been hashed into the empty slot.

The insert function is somewhat more complicated. It takes two local arguments, a key, k, and a value, v. Like lookup, its first step is to look up the current key-value pair, o, at position hash k in the toplevel array. If o is None then insert writes the pair $\lfloor(k, v)\rfloor$ into the array at that position. The canonicity invariant again implies that there are no keys elsewhere in the table that hash to the same slot as k and yet are less than k in the order. Likewise, if o is an actual key-value pair (k', v') such that $k = k'$, then insert can just overwrite the current value of k' with the new value, v. Otherwise, o is a key-value pair (k', v') such that $k \neq k'$. Now there are two cases: either (1) $k < k'$, or (2) $k' < k$. In case (1), lookup inserts k' recursively into the key-value map, then overwrites k' at position hash k in the toplevel array with $\lfloor(k, v)\rfloor$. Case (2) is even simpler: k' stays where it is and k is inserted recursively into the stash.

The remove function is similar in structure to insert. Like both insert and lookup, it first reads the current key-value pair, o, located at position hash k in the top level. If o is $\lfloor(k', v')\rfloor$ such that $k = k'$, then remove must delete k from the toplevel array and replace it with the next smallest key that would have hashed to that slot, if one exists. To do so, remove calls Stash.minkey on the stash, returning a value k_{min} of type option K. If k_{min} is an actual key, then it and its associated value, v_{min}, are removed

```
Module KVMapSig.
 sshape : R^2 → (K ⇀fin V)^2 → heap^2 → prop
 shape : R → (K ⇀fin V) → heap → prop
 shape_fun :
     ∀r g i1 i2.
     i1 ∈ shape r g →
     i2 ∈ shape r g → i1 = i2
 sshape_shape :
     ∀rr gg i1 i2.
     ii ∈ sshape rr gg →
     i1 ∈ shape r1 g1 ∧ i2 ∈ shape r2 g2
 lookup (r:kvmap) : STsec [K] (option V)
   (fun k. ∃g. i ∈ shape r g,
    fun kk yy ii mm.∀gg.
      ii ∈ sshape (r, r) gg →
      mm = ii ∧ yy = (fnd k1 g1, fnd k2 g2))
 insert (r:kvmap) : STsec [K, V] unit
   (fun k v. ∃g. i ∈ shape r g,
    fun kk vv yy ii mm.∀ff.
      ii ∈ sshape (r, r) gg →
      mm ∈ sshape (r, r) (ins k1 v1 g1, ins k2 v2 g2))
 ...
End Module KVMapSig.
```

Figure 7: Specifications of key-value map lookup and insert. The module is parameterized by the types R, K and V. The hash table functions of Fig. 6 meet these interfaces, along with similar ones for new, remove, and minkey (elided). The type kvmap is a parameter to the specifications. The predicates shape and sshape are related by laws like sshape_shape above. In addition, shape is required to be a function (shape_fun), thus enforcing that modules matching the key-value map interface, such as the multilevel hash table library of this section, be uniquely represented.

from the stash and written into the toplevel array at position hash k. (Stash.minkey guarantees that hash k = hash k'.) Otherwise, None is written into the array at that position, effectively clearing the slot. When o is None or $\lfloor(k', v')\rfloor$ such that $k \neq k'$, k is removed recursively from the secondary map.

Unique Representation. The unique representation theorem for multilevel UR hash tables follows from the canonicity invariant we defined in Sec. 6.1 (Definition 2) and the definition of the multilevel UR hash table shape predicate (cf. Fig. 5).

THEOREM 1 (UR FOR MULTILEVEL UR HASH TABLES). *For all multilevel UR hash tables r, finite maps $g : K \rightharpoonup_{\mathsf{fin}} V$ and heaps m_1 and m_2, if $m_1 \in$ shape r g and $m_2 \in$ shape r g then $m_1 = m_2$.*

6.4 Specifying Multilevel UR Hash Tables

Our multilevel hash table implementation satisfies the key-value map interface we first described informally in the previous section. We forgo presenting the entire interface here since most of the specifications are straightforward (please see our Coq scripts for more details, file `kvmaps.v`). Below we focus on a few of the interface functions.

The lookup function, specified by the STsec type given in Fig. 7 and implemented in the previous section by multilevel hash table lookup, takes two arguments, a function argument r defining a key-value map and a local argument of type K providing the key to be looked up. The precondition of lookup states that the heap i satisfies the key-value map shape predicate, which is required to be a function by the interface.

In our multilevel hash table implementation of the key-value map interface, shape is instantiated with the multilevel hash table shape predicate defined in the previous section. The requirement that shape be a function (law shape_fun in Fig. 7) is satisfied by the unique representation proof for multilevel hash tables (Theorem 1). The postcondition of lookup states that lookup leaves the heap unchanged ($mm = ii$) and that the option values y_1 and y_2 that lookup returns are those discovered when keys k_1 and k_2 are looked up, via fnd, in the mathematical finite maps g_1 and g_2 that represent the key-value map r ($ii \in$ sshape (r, r) gg).

Key-value map insert (also shown in Fig. 7) takes three arguments, the map r, a key k, and its new value v. As in lookup, r is a function argument that we assume to be low, whereas k and v are function arguments that are implicitly high. insert's postcondition states that in the output heap mm, k_1 is mapped to v_1 in one run and k_2 is mapped to v_2 in the other. We model these updates mathematically using the finite map insertion function, ins, which updates a key with a new value in a key-value map.

The interactions among ins, fnd and rem are governed by several algebraic laws, e.g., rem k (ins k v g) = rem k g. As we described in the introduction, clients of the multilevel UR hash table module will reason using laws of this nature. In particular, client reasoning is entirely independent of the number of levels in the table and other implementation details.

7. ERASURE-DEPENDENT ACCESS CONTROL REVISITED

In Sec. 4, we described an RHTT implementation of a medical database application. One of the distinguishing features of the system was that it supported expressive conditional information flow policies. In particular, the higher-order function endorse granted a client unrestricted access to confidential patient records provided the client could first prove that any confidential data it stored in local state during the computation was erased before its function exited. In Sec. 4, our example client's local state consisted of a single pointer to an integer; the simplicity of this local state made it quite easy to prove that the client program met the erasure policy required by endorse.

In this section, we revisit the client of Sec. 4 in order to prove erasure policies for more complicated data structures such as the multilevel UR hash tables of Sec. 6. Unique representation is the key property that enables these proofs.

Hashing Client. Fig. 8 presents the RHTT code for our enhanced medical database client. In the original client of Sec. 4, the client's local state, given by the type R, was a pointer to an integer. Here R is the type of 4-level UR hash tables, defined by the function build_map. This function constructs a multilevel UR hash table with an arbitrary number of levels (4 in this case). The hash function hash is a parameter. The predicate ccshape is defined wrt. the multilevel UR hash table's relational shape invariant, FilterHashPrelim.sshape. This shape invariant applies the multilevel UR hash table's unary shape predicate shape (Sec. 6) to heaps i_1 and i_2 respectively. The function cmp defines how the client's local state evolves after each call to hashing_client. Here we specify that after each call, the client will have inserted the low key-value pair $(18, v)$, for some v, into (and removed key 17 from) its local hash table.

The hashing_client function is similar to the counting_client function of Sec. 6. Within the do block, prog first reads pat's elec-

```
Module B′.
 R ≙ build_map hash 4
 G ≙ finMap K V

 ccshape (rr:R^2) (gg:G^2) (ii:heap^2) ≙
   MultilevelURHashPrelim.sshape rr gg ii

 cmp (g:G) : G ≙ ins 18 v (rem 17 g)

 prog (r:R) : T ≙ do (fun db pat.
   emr ← read_emr db pat;
   MultilevelURHashTab.insert r 17 (f1 emr);
   MultilevelURHashTab.insert r 18 v;
   MultilevelURHashTab.remove r 17;
   ret (f2 emr))

 hashing_client (r:R) : S ≙ A.endorse R G
   MultilevelURHashPrelim.sshape cmp prog
End Module B′.
```

Figure 8: A medical database client with a 4-level UR hash table as local state. The module is parameterized by the types K and V.

tronic medical record, then inserts a confidential integer hash of this record ($f_1\ emr$) into the multilevel UR hash table, associated with key 17. prog next inserts the *low* key-value pair $(18, v)$ into the hash table. Finally, it removes the confidential data it previously inserted at key 17 by calling FilterHashTab.remove r 17.

To prove that hashing_client meets the erasure policy, we must show that the client's local heap is low after the function is executed, assuming its local state was low before execution. To do so, we reason in two steps: we first show that *symbolically* executing prog from equal initial states results in equal *symbolic*, i.e., *logical*, final states. That is, inserting the confidential key-value pair $(17, (f_1\ emr))$, the public or low key-value pair $(18, v)$ and then removing the confidential pair $(17, (f_1\ emr))$ to g_1 and g_2 results in final key-value maps g'_1 and g'_2 that are equal. That this is true can be seen by considering how insert commutes with remove. For example, one such commutation property is rem k (ins $k\ v\ g$) $=$ rem $k\ g$. Note that this symbolic proof is performed with respect to the mathematical abstraction of hash tables as key-value maps and therefore is entirely independent of our particular hash table implementation (thus achieving portability to future implementations).

Once we have proved that $g'_1 = g'_2$, in the second step we apply the unique representation theorem for multilevel UR hash tables (Theorem 1) to show that equality of the *logical* states g'_1 and g'_2 implies equality of the final memory states.

8. RELATED WORK

There has been much work on semantics of noninterference and its relaxations [28, 29] as well as enforcement mechanisms based on type systems, logics and other program analyses. Chong and Myers [10, 11] study specification and enforcement of erasure policies in the context of a simple imperative language and show how their framework can be implemented in Jif [23]. Their framework handles declassification policies as well: they describe a duality between declassification and erasure where the former can be viewed as a relaxation of noninterference and the latter as a strengthening of noninterference. Hunt and Sands [18] show the close connection between information erasure policies and noninterference in the context of simple imperative as well as interactive programs.

Russo et al. [27] employ a flow-sensitive *dynamic* type system, to perform run-time enforcement of secure information flow in *non-UR*, but heterogeneous, data structures such as dynamically allocated DOM trees. Further extensions by Hedin and Sabelfeld [17] and by Birgisson et al. [7] consider tracking information flow in heap-based heterogenous data structures in dynamic languages such as JavaScript. In this line of work, the nodes in a linked structure are assigned several security labels. For example, specifically in [17], each node has one security label for inspecting the existence of the node, another label for changing the structure rooted at the node, and another one for the node's contents. In a secret control context, addition of new fields to a node is allowed only if the node's structure label is high. Otherwise, the addition results in run-time error. Similar to this work, RHTT is also a flow-sensitive type system. In contrast, we do not consider security labels, but use a semantic definition of NI. Furthermore, in RHTT we focus on static verification of non-interference and erasure, rather than run-time enforcement.

Swamy et al.'s F* system [31] permits properties of data values to be specified via preconditions and postconditions in the form of dependent types of methods. The verification conditions are discharged automatically by passing them to the SMT solver Z3, though some manual work is usually needed in the course of development, to guide the SMT solver. The assertions on the methods focus on the properties of values only. In this respect, RHTT differs from F*, as we allow arbitrary higher-order logic assertions over heap shapes, and the assertions may, moreover, be relational.

Nanevski et al. [24] consider verification of flexible security policies in possibly heterogeneous linked lists. Such lists may contain mixed high and low data as well as mixed high and low links, although their main example considers low links only. Here we consider more involved data structures such as hash tables and recognize unique representation as a critical property to ensure secure erasure in the presence of adversaries who have access to the concrete memory state.

In the algorithms community, Amble and Knuth [1] studied hash tables without deletions. However, their goal was to develop fast searching algorithms rather than to exploit properties such as history independence. Theoretical work on history independent data structures by Naor and Teague [25], Hartline et al. [16], and Blelloch and Golovin [8, 15] has been a primary impetus for our work, which complements the algorithmic approaches by applying unique representation to achieve provable guarantees about the functional correctness and security of programs.

9. DISCUSSION AND FUTURE WORK

This paper proposes unique representation as a means of enforcing expressive information flow and erasure policies in programs with modules and procedures with local state, in the presence of pointers and mutable data structures such as hash tables. To explore the verification of UR data structures, we present a case study in which we (1) formalize a UR variant of hash tables and (2) show how the UR theorem for this data structure can be used to prove noninterference of a medical database client. We believe this is the first verification case study of involved security properties such as NI and erasure that considers non-trivial heap-based data structures *and* their clients.

Much work remains. One direction to explore is the connection between enforcement of information flow policies in a high-level language (as done in this paper) and their preservation through the compilation toolchain. For example, an information-flow aware compiler must not optimize away operations introduced by the programmer to prevent information leakage (e.g., zeroing out of confidential local variables in the stack frame before function return) just because these operations do not affect a program's dataflow.

Another direction involves studying interactions between garbage collection and erasure guarantees, especially in the context of timing and other attacks over covert channels. Proper engineering of an incremental garbage collector could ensure that private data is erased, and prevent some timing attacks. However, any end-to-end erasure guarantee of programs composed with the garbage collector would depend on the correctness of the garbage collector itself (cf. McCreight et al. [19]).

The resolution of both of the above issues may require generalizing our specifications so that they express not only properties of the beginning and the end of the computation, but of intermediate points as well. This may require reasoning about traces and temporal properties.

Another direction for generalization involves studying the verification of *history independent* data structures [16]. History independent structures cannot distinguish different but logically equivalent operation sequences. They generalize UR. Indeed, there are some history-independent data structures that use internal randomization as an implementation strategy, that are not UR. For example, in order to prevent a binary search tree from leaking information through the physical layout, one may consider randomly rebalancing the tree after each operation. Currently, RHTT cannot reason about randomization. In the presence of randomization, we foresee generalizing the notion of indistinguishability of heaps, which feeds into the definition of uniqueness of representation (cf. Definition 1). RHTT currently employs exact equality of heaps, but it may be possible to relax this notion by attaching a random distribution to the set of heaps.

We also plan to study the interplay between UR and parametricity, and consider how internalized reasoning about type abstraction [6] may be used to formally prove that a module interface tightly encapsulates the internal state of the module.

Acknowledgments. We thank Deepak Garg for his comments and encouragement, and the anonymous referees for their suggestions. This research was partially supported by Madrid Regional Government Project S2009TIC-1465 Prometidos; Spanish Ministry of Economy and Competitiveness projects TIN2009-14599-C03-02 Desafios, TIN2010-20639 Paran10, and TIN2012-39391-C04-01 Strongsoft; EU Project NoE-256980 Nessos; Ramon y Cajal grant RYC-2010-07433 and AMAROUT grant PCOFUND-GA-2008-229599.

References

[1] O. Amble and D. E. Knuth. Ordered hash tables. *Comput. J.*, 17(2):135–142, 1974.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006.

[3] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.

[4] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[5] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free — parametricity for dependent types. *JFP*, 22(2), 2012.

[6] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *LICS*, 2012.

[7] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS*, 2012.

[8] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *FOCS*, 2007.

[9] J. Borgström, A. D. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. *JFP*, 21(2):159–207, 2011.

[10] S. Chong and A. C. Myers. Language-based information erasure. In *CSFW*, 2005.

[11] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassification. In *CSF*, 2008.

[12] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*. 1978.

[13] D. Denning. A lattice model of secure information flow. *CACM*, 19(5):236–242, 1976.

[14] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. In *STACS*, 2003.

[15] D. Golovin. *Uniquely Represented Data Structures with Applications to Privacy*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 2008.

[16] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[17] D. Hedin and A. Sabelfeld. Information-flow security for a core of Javascript. In *CSF*, 2012.

[18] S. Hunt and D. Sands. Just forget it - the semantics and enforcement of information erasure. In *ESOP*, 2008.

[19] A. McCreight, T. Chevalier, and A. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ICFP*, 2010.

[20] D. Micciancio. Oblivious data structures: Applications to cryptography. In *STOC*, 1997.

[21] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *TOPLAS*, 10(3):470–502, 1988.

[22] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine (extended abstract). In *IEEE Symp. Security and Privacy*, 2006.

[23] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[24] A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of access control and information flow policies. *TOPLAS*, 35(2):6, 2013.

[25] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *STOC*, 2001.

[26] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.

[27] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.

[28] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, 2003.

[29] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *JCS*, 17(5):517–548, 2009.

[30] R. Sundar and R. E. Tarjan. Unique binary-search-tree representations and equality testing of sets and sequences. *SIAM J. Comput.*, 23(1):24–44, 1994.

[31] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, 2011.

[32] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *JCS*, 4(2/3):167–188, 1996.