Hypertypes: Types for Hyperproperties

Project Description — Practical Work

Ramon Wick
Supervised by Thibault Dardinier

April 7, 2025

1 Introduction

1.1 Hyperproperties

To check a system's correctness, one may look at individual executions of the program. With this approach, one can verify, for example, that for any possible input list, the output is correctly sorted. Conceptually, it is shown that every possible *trace*, i.e., every possible program execution, yields a correctly sorted list.
With this approach, not every property of a system can be checked. *Hyperproperties* [CS10] formally capture properties across multiple program executions. One such hyperproperty is determinism, i.e., with the same starting values, always the same result is returned. By just looking at individual traces in isolation, one cannot check such a property. Multiple traces need to be taken into account for verifying a property like this.

To verify a system's security, such hyperproperties are essential. To give more intuition on hyperproperties, we will have a closer look at the *noninterference* [GM82] hyperproperty.

1.2 Noninterference

Noninterference is a confidentiality property stating that there is no information flow from a highly confidential variable to a less confidential variable. For this, we divide all the data into two security levels. The data can be *high*, meaning it is confidential, or it can be *low*, meaning it is public.

The noninterference property states that no information must flow from a high variable to a low variable. Or stated differently, an observer who can only read low input variables and the output must be unable to infer anything about the value of a high variable.

For the Examples 1 and 2, the argument l will contain a low integer, while the argument h contains a high integer. In both examples, we want to check if information flows from the high variable to the returned value.

The method in Example 1 has no information flow from h to the return variable. Hence, it satisfies the noninterference property. For any input combination, the low observer will see that the result is always one bigger than the low input l. This is the case regardless of the value of the high argument h. Hence, the low observer cannot infer any information about the variable h. And therefore, the method satisfies the noninterference property.

Example 1.

```
1  method secure(l, h) {
2      var o
3      if h > 2 {
4          o := l + 1
5      } else {
6          o := l + 1
7      }
8      return o
9  }
10
```

On the other hand, the method in Example 2 is insecure. This is because the low observer can make the following observation about the high variable h. The high variable h is strictly positive exactly when the difference between the input variable l and the returned value is one. Hence, in this example, some information flows from the high variable h to the return variable. The insecure function leaks information.

Example 2.

```
1  method insecure(l, h) {
2      var o
3      if h > 0 {
4          o := l + 1
5      } else {
6          o := l + 2
7      }
8      return o
9  }
10
```

1.3 Verification

To verify such hyperproperties, two main approaches exist. The first involves designing a *hypertype system* tailored to a specific hyperproperty, which annotates variables with types to facilitate verification. The second approach relies on SMT-based verifiers. In the following, we examine how each method can be applied to verify Example 1.

1.3.1 Hypertype Systems

With a hypertype system, we add security level declarations to the variables. With $\text{Int}[\text{low}]$ we refer to an integer variable that contains public information. Similarly, we write $\text{Int}[\text{high}]$ to denote the type of a high integer variable. To illustrate this method, we propose a simple type system inspired by [Bar20] for a simple lambda calculus. We have the following rules:

$$\frac{}{\Gamma \vdash n :: Int[\sigma]}\ \text{Int} \qquad \frac{\sigma \leq \sigma'}{\Gamma, x : \tau[\sigma] \vdash x :: \tau[\sigma']}\ \text{Var} \qquad \frac{\Gamma \vdash e_1 :: Int[\sigma] \quad \Gamma \vdash e_2 :: Int[\sigma]}{\Gamma \vdash e_1 + e_2 :: Int[\sigma]}\ \text{Op}$$

$$\frac{\Gamma, x : \tau_1[\sigma_1] \vdash t : \tau_2[\sigma_2]}{\Gamma \vdash \lambda x.t :: \tau_1[\sigma_1] \to \tau_2[\sigma_2]}\ \text{Abs} \qquad \frac{\Gamma \vdash c :: \tau[\sigma]}{\Gamma \vdash \text{if } b \text{ then } c \text{ else } c :: \tau[\sigma]}\ \text{Same-if-else}$$

$$\frac{\Gamma \vdash b :: \tau[\sigma] \quad \Gamma \vdash c_1 :: \tau[\sigma] \quad \Gamma \vdash c_2 :: \tau[\sigma]}{\Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 :: \tau[\sigma]}\ \text{Diff-if-else}$$

Figure 1: Simple Typing System for Secure Information Flow

We have two different rules for if-else branches. The Same-if-else rule can be applied whenever both branches contain the same expression. Intuitively, it does not matter what the condition evaluates to, because the same expression is evaluated anyway. Hence, it does not matter if the condition contains high variables. The other if-else rule can be

applied when the branches contain different expressions. Here we must type the condition, because this could reveal information as was illustrated in Example 2.

With this small type system, we can show that Example 1 is secure. The method can be expressed as follows:

$$\lambda l.\lambda h.\text{if } h > 2 \text{ then } l + 1 \text{ else } l + 1$$

We claim that the type of the expression is Int[low] → Int[high] → Int[low], i.e., it returns a low value. Which is indeed the case as can be seen in Fig. 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{l : \text{Int[low]}, h : \text{Int[high]} \vdash l :: \text{Int[low]}}\,\text{Var} \quad \dfrac{}{l : \text{Int[low]}, h : \text{Int[high]} \vdash 1 :: \text{Int[low]}}\,\text{Int}}{l : \text{Int[low]}, h : \text{Int[high]} \vdash l + 1 :: \text{Int[low]}}\,\text{Op}}{l : \text{Int[low]}, h : \text{Int[high]} \vdash \text{if } h > 2 \text{ then } l + 1 \text{ else } l + 1 :: \text{Int[low]}}\,\text{same-if-else}}{l : \text{Int[low]} \vdash \lambda h.\text{if } h > 2 \text{ then } l + 1 \text{ else } l + 1) :: \text{Int[high]} \to \text{Int[low]}}\,\text{Abs}}{\vdash \lambda l.\lambda h.\text{if } h > 2 \text{ then } l + 1 \text{ else } l + 1 :: \text{Int[low]} \to (Int[high] \to \text{Int[low]})}\,\text{Abs}$$

Figure 2: Proof for Example 1

1.3.2 SMT-based Verification

In contrast, another approach is to use modern SMT solvers to verify the desired properties. One such formal verification tool is Hypra [DLM24], which itself is based on Viper [MSS16]. Hypra can verify conditions for Hyper Hoare Logic [DM24]. Hyper Hoare Logic is similar to Hoare Logic; however, it makes statements over multiple program traces. With Hypra, we are also able to prove that Example 1 satisfies the noninterference property. For this, we need to annotate the program.

In Hypra, we can annotate functions with a contract. Here we state that all input states must have the same value for the low variable l. If this condition is met, then Hypra can verify that all the traces have the same output value o. Hence, the output is not influenced by a high variable and will therefore not leak any information.

```
1  method secure(l : Int, h : Int) returns (o : Int)
2  requires forall <_s1>, <_s2> :: (_s1[l] == _s2[l])
3  ensures  forall <_s1>, <_s2> :: (_s1[o] == _s2[o])
4  {
5
6      if (h > 2) {
7          o := l + 1
8      } else {
9          o := l + 1
10     }
11 }
```

1.4 Goal

Both approaches have their advantages and disadvantages. hypertype systems are much faster but have a deficit in their expressiveness. Simple expressions like `(h + l) − h`, where l is a low variable and h is a high variable, may become challenging for a type system to type correctly.

SMT-based verification tools, like Hypra, have no problem proving such statements. However, they come with a significant computational cost. Also, one may need to manually add annotations for the verifier to succeed.

The goal of this project is to bridge the gap between these approaches by extending Hypra to support hypertypes. This will allow the use of efficient type-checking when possible, while leveraging formal verification when necessary. In Example 3, we

illustrate how a hypertype system could enhance the verification capabilities of Hypra.

Example 3.

```
1  method secure(l : Int, h : Int) returns (o : Int)
2    requires forall <_s1>, <_s2> :: (_s1[l] == _s2[l])
3    ensures forall <_s1>, <_s2> :: (_s1[o] == _s2[o])
4  {
5    // Start Part 1
6    var z : Int
7    var x : Int
8    z := 1
9    x := l
10   while (x > z)
11     invariant
12       forall <_q1>,<_q2>:: _s1[x] == _s2[x]
13       && (_s1[x] + _s1[o] == _s2[x] + _s2[o]
14   {
15       x := x − 1
16       z := z + 1
17   }
18   // End Part 1
19   unfold (low) x
20   // Start Part 2
21   o := 0
22   while (x > 0)
23     invariant
24         forall <_s1>, <_s2> :: _s1[x] == _s2[x]
25         && (_s1[x] + _s1[o] == _s2[x] + _s2[o])
26   {
27       o := o + h
28       o := o + 1
29       x := x − 1
30       o := o − h
31   }
32   // End Part 2
33 }
34
```

In this Example, we again want to show that no information flows from the high argument into the output of the method. We divided the function into two parts. In the first part, the variables x and z are assigned. Before the while loop, both x and z have type Int [low]. Additionally, since all variables that exist within the while loop have a security level of low, the type system would conclude that after the loop, x still has type Int [low].

The second part is too involved for the hypertype system to verify. Without the invariant for the first loop, Hypra is unable to see that the invariant of the second loop does indeed hold. The important fact is that the value of x is the same in all traces after the first loop. The hypertype system could also reach this conclusion, as it follows from the fact that x has type $Int[low]$. The type system could assist Hypra in verifying this program by inserting an unfold expression.

2 Project Goals

In the previous section, we illustrated the noninterference hyperproperty and the possible improvements for Hypra. But noninterference is not the only hyperproperty of interest to us.

2.1 Further Hyperproperties

We will give some intuition for further hyperproperties together with some high-level ideas for hypertype systems:

- Determinism: If a program runs twice with the same inputs, then it returns the same outputs.
  This can be implemented using the same type system as for the noninterference property. Every input and output variable could be labeled low. Random values are typed as high. Hence, if the function returns a low result, it is deterministic.
- Monotonicity: A function is monotonic if it holds that $x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2)$. This idea was translated into a type system by Clancy, Miller, and Meiklejohn [CM17]. This type system could serve as a starting point for an implementation in Hypra.

- Symmetric relation: A function is symmetric if the order of the inputs does not influence the output. For every calculated value in the function, the type system could track to which arguments it is symmetric. For example, given a function with arguments x and y. Then, $x + y$ would have type $\text{Int}[\text{x,y}]$. Because we can permute x and y and it gives the same result.

Further hyperproperties that we are interested in are: Generalized noninterference, idempotence, transitivity, existence of minimum/maximum, anti-symmetry, asymmetry, total relation, associativity, robustness/k-lipschitz, injectivity.

2.2 Project Roadmap

Here we present the planned roadmap on how to extend Hypra with hypertype systems. We differentiate between **core goals** and *extended goals*.

1. **Add an Information Flow Type System to Hypra**
   This goal includes: Deciding on a syntax, updating the parser, extending the type checker, and implementing a translation between the syntactic and semantic worlds.

2. **Add Support for Type Systems for Other Hyperproperties**
   Building on the knowledge gained from the first goal, we implement more complex type systems that capture monotonicity and injectivity.

3. **Benchmarking**
   To analyze the achievement, existing benchmarks need to be rewritten to fit the new type systems. Then, we will compare the performance between basic Hypra and Hypra with hypertype systems in terms of execution time and number of needed annotations.

4. *More Type Systems*
   We will use the results from the third goal, as well as existing benchmarking results [DLM24], to decide which further type systems we will implement.

5. *User-Defined Hypertype Systems*
   We want to develop a domain-specific language (DSL) allowing users to define rules as well as semantic definitions for the type. Revisiting the noninterference example, the semantic definition of $\text{x} : \text{Int}[\text{low}]$ would be $\forall_{\langle s_1 \rangle, \langle s_2 \rangle}. s_1[x] = s_2[x]$.

References

[Bar20] Gilles Barthe. Information Flow and Program Counter Security. In *An Introduction to Relational Program Verification*. Now Publishers, 2020.

[CM17] Kevin Clancy and Heather Miller. Monotonicity Types for Distributed Dataflow. In *Proceedings of the Programming Models and Languages for Distributed Computing*, pages 1–10, Barcelona Spain, June 2017. ACM.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, September 2010.

[DLM24] Thibault Dardinier, Anqi Li, and Peter Müller. Hypra: A Deductive Program Verifier for Hyper Hoare Logic. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1279–1308, October 2024.

[DM24] Thibault Dardinier and Peter Müller. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1485–1509, June 2024.

[GM82] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, Oakland, CA, USA, April 1982. IEEE.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 9583, pages 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. Series Title: Lecture Notes in Computer Science.