Pattern Matching Information Flow using GADT

Eric Lindahl, Victor Winter
Dept. of CSCI, University of Nebraska at Omaha, Omaha, Nebraska
eric@ericlindahl.com
vwinter@mail.unomaha.edu

Abstract: Integrating security policies into security assurance mechanisms to ensure end-to-end behavior is still a challenge. Information flow analysis and type checking are effective methods for the analysis and verification of secure communications and processing. Language-based information flow security models use programming-language for specifying and enforcing security policy. Dependently typed programming is an efficient and powerful way to concisely communicate, represent, and then reason over security policies. In this paper we demonstrate an integration of policy elements in a subset of a language-based information flow security model implemented using dependent type programming. We illustrate how recent advances in type theory in secure domains make available enabling technologies for developing policy aware secure computing.

Keywords: GADT, information flow, non-interference, static analysis, type systems, security policies

1. Introduction

Secure communications and processing are becoming a ubiquitous computing need in our mobile and connected culture. People and systems increasingly enter into more relationships with security aspects, such as electronic commerce and multi-national business or military coalitions, resulting in a proliferation of complex security relationships and constraints. This trend will continue and the job of enforcing these numerous and complex security policies rests partially or wholly on automated security policy enforcement systems. We need and expect our computer resources to help protect our information and maintain our privacy. Automated security systems are instructed how to enforce information security though an executable form of the security policy, but correctly specifying and translating the semantics of a security policy into an executable form is a difficult and open research problem. Without the semantics represented in a security policy a program can't distinguish whether the following statements are violations or not.

```
chat_text := social_security_number
mail( competitor, salaries )
```

Without semantics and typing, a program is unable to meaningfully apply general principles of secure computing. There is a clear need to improve the effectiveness and reach of secure computing resources. Important factors of this effectiveness are the capture, representation, and automation of security knowledge (Butler 2002), roles (Yaigiie, Gallardo, & Mafia 2005), and policies (Walker 2000) . Secure computing systems that are ineffective in representing and enforcing security policies multiplicatively reduce the effectiveness of the overall system. For example, one of the basic methods of managing secure information is to apply labels to secure or *high* information and attempt to prevent that information from being 'lost', which usually means being available to non-secure or *low* entities. In an imperative language, this is represented as an assignment from a *high* labeled variable to a *low* (or at least non-*high*) labeled variable.

low := high

Determining what memory is considered *high* may be a very difficult task, as this requires decomposing knowledge according to security policy, and both of these domains are still open for research. Knowledge representation (KR) and ontology design deal with understanding, labeling, translating, and conserving meaning between peoples and systems. Labeling data/variables with *high* or *low* becomes nonsensical if there isn't an underlying knowledge representation, if even informal or by convention. Indeed, a secure program can be seen as the embodiment of a particular KR or ontology, able to label and conserve secure information according to a hierarchy of types in that KR. The *Curry-Howard isomorphism* (Sheard 2005) states that types are propositions and programs are proofs and so translating a security policy into a well-typed program provides an avenue for formal methods to assist in assuring secure computing. Type theory and type checking have had great success in bringing the power of formal methods into programming language and compiler design. Type checking is an effective method of proving that a program meets certain criteria, but only up to as much meaning as was captured and made available through a language type system and operations. Leveraging type theory allows the use of the Curry-Howard isomorphism to verify a secure program. In this model, there are no 'simple' or primitive types, as all types must participate in a type policy. Is it a 'low' or 'high' integer? What 'type' of role is assigned to the subject? In this paper we demonstrate the application of a imperative language information flow model using dependent type programming with generalized algebraic dependent types (GADT) or “guarded recursive data types” (Sheard, Hook, & Linger 2005, Jones, Vytiniotis, Weirich, & Washburn 2006). This dependent typed program represents a subset of an information flow method for enforcing the semantics of a security policy.

1.2 Security Classes and Inductive Families of Data Types

Security classes may considered types restrictions on values like applying a security label to an object. Lattice-based models of access control (Denning 1976, Sandhu 1993) rely on operations such as *can-flow* and *join* to verify high confidence doesn't flow towards a low through the use of a lattice model. GADT and dependent types have been used for years as inductive families of data types whose recursive type inference represent classes of lattice models (Jones, Vytiniotis, Weirich, & Washburn 2006). In a type theoretic framework the same type inference and checking mechanisms may be used for verification of subject and object labels, and so user security levels are taken into account via type parameters, inference, and verification. Fully leveraging *Curry-Howard Isomorphism* should mean there are no 'primitive' types such as *integers* or *strings*, but default types in the current trace, such as those provided by the user rights. The concept of a *reference monitor* can then be viewed as a *type checker*, verifying the types of operations, such as writing information to a variable or port.

The following sequence of statements is executed under some user, which has a role, and thus a set of security classes. When establishing communications with other_user, the other_user may (under our type theoretic approach, must) also have a role, and consequently a set of security classes. All the information necessary to verify the statements is available, or can be available via types.

```
1) chat_text_out := connect_chat( other_user )
2) chat_text_out := social_security_number
```

GADT type parameters are useful in compiled and interpreted systems where a type verification engine is available and so can 'dynamically' verify some levels of dynamic changes to security policy via changes in GADT type parameters. However, full type verification requires the full power of a theorem prover any secure system may require 'high' side security policies (such as via their equivalent types) to be reverified.

1.3 Dependent Type Programming

Dependent types are types that depend on their data to fully develop their meaning. Types represent the criteria for the usage of the data, like a security policy determining whether a user can read or write to a network directory, for example. A dependently typed translation (such as GADT) of information flow techniques can take advantage of the powerful and flexible type checking provided by the compiler, in this case it’s Haskell. The GADT calculus of constructors provides an expressive representation for the constraints and semantics of a security policy. There is a great benefit in having a strong correlation between a policy document and an equivalent policy proof, validation, and active enforcement. Lowering the barrier to policy translation and enforcement grants a larger possible deployment and effective usage of secure computing practices. A large imperative language software support structure may be able to provide the guarantees, but at a cost of complexity and maintenance. In contrast, a well-typed and succinct representation can be validated, proved type safe and secure at compile time, for the life of the program. Some of this type safety comes from the calculus of constructors provided by a dependently typed language. GADT doesn’t provide the full capability of other dependently typed languages, but has an excellent design and implementation in Haskell, which is a well studied language with many implantations of multiple platforms. A GADT implementation can provide many of the security proofs necessary to express fairly complex security policies, and has the benefit of being available for implementation with minimal dependency on young technologies.

1.4 Preserving Semantics

Meaningful assignment of confidentiality to data requires some sort of policy. Assigning a security labeling and enforcement without clear reference to policy may likely introduce weaknesses and security violations. Preserving the semantics represented in the security policy is a critical link between secure systems designers and a trusted computing base.

Security policies often can be translated into an associated automata for controlling operations on sensitive data . As we will demonstrate, it is possible to clearly see how security policy language can be interpreted as automata states and events and used to ensure policy is enforced. Having a working and expressing target implementation to represent and enforce the automata is a crucial step in developing security specification translators to transform human readable policy into computer executable policy enforcement. We won't discuss policy translation (and other program transformation issues) but will show a possible legal requirement policy regarding contract state, and representation in a GADT information flow program.

Policy may be interpreted as restricted operations such as invalid operations or secure operations. GADT mechanisms represent and track secure labeling and are used in the declaration or interpretation of the policy as a GADT evaluator. This doesn't prevent expressions or functionally equivalent representations of secure operations but allows for simple specification of secure labeling via more semantically rich policy and patterns. A more complete implementation of the information flow methodology should remain uncompromised, but pattern based specification o f security labels may be a more semantically rich intermediate mechanism of policy specification for secure applications.

Flow sensitive security typing (Hunt & Sands 2006, Sabelfeld & Myers 2003) 'follows' the confidential data through the system to ensure that observers of the program are unable to read the confidential data. The capability of maintaining separation of influence between high and low confidential information is called *non-interference.*

1.5 Non-interference

Secure programs may use confidential data to provide services to non-confidential users. The owner of the confidential information doesn't want that information to become inadvertently visible through the manipulation of the program or service. A *non-interference policy* states that confidential information should not interfere or affect the public data, and so become visible (Sabelfeld & Myers 2003, Goguen & Meseguer 1992).

The semantics of a program (command) is represented by $[\![C]\!]$ and *s* is an output state. The observable variation on the high input side is represented by the *equivalence relation* $=_L$ which states that the low side must maintain agreement. Furthermore, the equivalence relation should produce a semantic equivalence $\approx_L$ for all output states of the program $[\![C]\!]$ such that an attacker is unable to distinguish a difference .

$$\forall s_1, s_2 \in S.\, s_1 =_L s_2 \Rightarrow [\![C]\!]s_1 \approx_L [\![C]\!]s_2$$

Noninterference for programs essentially means that a variation of confidential (high) input does not cause a variation of public (low) output. This intuition can be rigorously formalized using the machinery of programming-language semantics.

2. Survey of Information Flow

Secure systems can apply policy to prevent unprivileged access to data where access control is being exercised, such as logging in or connecting to a resource, but this is insufficient to enforce secure processing within a program, where secure information may leak to an unprivileged variable or resource. Access control does not assist in determining the flow or propagation of information through the system. Information flow is a method of analyzing end-to-end design of the computer system in accordance with a security policy. Hacking may exploit a weakness in a system, say by changing a variable allowing a path of code access to a high variable. Also, inadvertent or poor design may have code paths with possible loss of confidential information. Information flow analysis annotates the program with security related information, to apply rules preventing possible loss of confidential information. There are strong guarantees with system that is able to essentially run a theorem prover on the program to prove that security policy is being met. Here we will talk about two related methods in Flow Caml and Language-based Information Flow. Our GADT Information Flow demonstration will be in the latter.

2.1 Flow Caml

Information flow in a functional language has been considered before, such as in Flow Caml (Simonet 2003) where standard ML types were annotated with security levels and the Objective Caml type inference engine was specialized for information tracing. Thus Flow Caml handles a large subset of Objective Caml and is not an extension to it, but requires a different implementation of the type inference engine. In this example from of the Flow Caml annotation, there are two definitions with security principals alice and bob.

```
let x1  : !alice int =
11;;
val x1 : !alice int
let x2 : !bob int 22;;
```

In this example, **`alice`** and **`bob`** do not have a relative relationship and so are not comparable may not exchange information. An expression over these variables will summarize this constraint by requiring the security to be greater or equal to both **`alice`** and **`bob`**, or a *symbolic union* of the two principals.

```
x1 + x2;;
: [> !alice, !bob]
```

So the expression accumulates the principals of the sub-expressions, and thus the assignment or function semantics can apply security principal constraints to validate the operation. For example, we would expect the following expression to fail.

```
x1 := x2;;
      This expression generates the following information flow(s):
      from !bob to !alice
      which are not legal.
```

This is a good example of modern programming languages answering the call for secure computing, but Flow Caml is a specialized implementation. We'll examine next a well-known example of an information flow specification that can be demonstrated using GADT.

2.2 Language-based Information Flow

The language-based information flow (LBIF) annotates the language semantics with two carriers for keeping track of secure labels. In particular, LBIF develops the idea of a *pc* or program counter, to deal with the problem of tracking interaction between mixed high and low code. Having a high variable assignment doesn't necessary mean that information is leaking to the low side. Following are the main rules of LBIF. For our purposes, high variables *h* and low variables are *l*. Rules [E1] and [E2] say any expression can by high, but expressions can be low only if they do not contain high variables. [C1] and [C2] skip and high variable assignment are typeable in any context.

$\vdash exp : high$ [E1] $\qquad \dfrac{h \notin Vars(exp)}{\vdash exp : low}$ [E2]

$[pc] \vdash \mathbf{skip}$ [C1] $\qquad [pc] \vdash h := exp$ [C2]

Rule C3 is the first explicit inference of the *pc* for a low expression. Rules C5 and C6 transfer the expression labeling to the *pc* labeling specifically to prevent using the state of a high expression from manipulating a low variable, and thus violating the *non-interference principle.*

$\dfrac{\vdash exp : low}{[low] \vdash l := \ exp}$ [C3]

$\dfrac{[pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash C_1 ; C_2}$ [C4]

$$\frac{\vdash exp : pc \quad [pc] \vdash C}{[pc] \vdash \textbf{while } exp \textbf{ do } C} \text{ [C5]}$$

$$\frac{\vdash exp : pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \textbf{if } exp \textbf{ then } C_1 \textbf{ else } C_2} \text{ [C6]}$$

$$\frac{[high] \vdash C}{[low] \vdash C} \text{ [C7]}$$

The subsumption rule C7 is used to reset the *pc* to a low state to prevent the *pc* from getting stuck in a high state and preventing legal high/low code interaction.

3. GADT Information Flow

Some of the proof methods from the referenced information flow method can be translated into GADT type constructors. This demonstration adapts the folklore (Gibbons 2007) GADT example to represent the annotation of the language based information flow. This example doesn't concentrate on rules C1, C4, C7, but rather on a simpler exercise of GADT information flow. We use the unification of the GADT type refinement to associate unbound annotations with Low. This allows High to act as a High signal and mimic a logical or. There are two annotations in this example for the expression (*exp*) and the program counter (*pc*). There are generally two pieces to a GADT specification, one being the GADT data type declaration itself with a list of constructors, and a set of function specifications mapping GADT types to program behavior. By fixing specific constructor patterns, we can ensure that the types for various elements in the LBIF conform to the rules. For example, the **`LSet`** constructor implements rule C3, where setting a low variable with a low expression sets both the *pc* and the *exp* to low. However, according to rule C2, **`HSet`** or high variable set should not change the values of *pc* or *exp*.

```
data Term a pc exp where
 Lit  :: Int -> Term Int pc exp
 IsZ  :: Term Int pc exp -> Term Bool pc exp
 LVar :: Int -> Term Int pc exp
 HVar :: Int -> Term Int pc High
 HSet :: Term a pc exp -> Term a pc exp
 LSet :: Term a pc Low -> Term a Low Low
 If   :: Term Bool pc exp -> Term a exp expt
        -> Term a exp expe -> Term a exp expi
```

The **`HVar`** high variable constructor has the *exp* set to High, which prevents setting a low variable with a high variable, according to E2. GADT type unification will carry type information through the constructor specification, unifying based on type carrier names. This is a limited method, however, as only unification and type equivalence can be used. There are other dependent type implementations that allow for further type manipulations and transformations in addition to equivalence and unification.

```
> :t (LSet (HVar 3))
    Couldn't match expected type `Low' against inferred type
`High'
      Expected type: Term Int pc Low
      Inferred type: Term Int pc High
```

Here is an example of an **`if`** statement with a *high* guard expression and *high* bodies. The **`pc`** is inferred from the GADT type rules to be *high* as expected.

```
> :t (If (IsZ (HVar 4)) (HVar 10) (HVar 32))
(If (IsZ (HVar 4)) (HVar 10) (HVar 32)) :: (Num (Term a High expt), Num (Term a High expe))
=> Term a High expi
> eval(If (IsZ (HVar 4)) (HVar 10) (HVar 32))
32
```

The evaluator can actually provide the virtual machine behavior and execute the security language specification. The evaluator is a clear and succinct behavioral specification that can be easily extended with new type constructors. Notice that the evaluator does not fix the type parameter, and so can be used with many different types. We will use this freedom to extend the LBIF implementation with sensitive information policy enforcement.

```
eval :: Term a pc exp -> a
eval (IsZ e)    = (eval e) == 0
eval (Lit i)    = i
eval (HVar i)   = i
eval (LVar i)   = i
eval (LSet i)   = eval i
eval (HSet i)   = eval i
eval (If b t e)
   = if eval b then eval t else eval
```

4. Semantics of Security Policy

Security policies are the basis for assigning meaning or reasoning to high/low labeling. The structure and support for security policies are not the result of automated reasoning, but represent the realities of a given domain and associated knowledge. The designation of confidential information is a human endeavor and represents human values and desires. Translating human interests into a security policy will likely remain a difficult task, but if the semantics of the security policy was written in a semi-rigorous fashion, like a legal document or software specification, then a semi-automated translation may assist the secure systems developers in implementation. Modeling languages, such as UML, have extensions and expressive requirements features to allows visual requirements tools to create policy requirements specifications which may be machine translated into something like a GADT Information Flow.

4.1 Example Policy

Our example policy demonstrates a simple security machine governing the actions with sensitive data. Businesses often conduct business with sensitive data under protection of a non-disclosure agreement (NDA), as is represented by this possible clause in a security policy.

```
Program shall not allow comminications and negotiations to occur when an NDA is not in
effect.
An NDA shall be in effect if an NDA has been signed and NDA shall not be in effect if it has
```

Communications and negotiations with others must have some legal protections in place to provide remedies for misuse of proprietary information. In this example NDA policy, we develop a simple automata representing the policy regarding actions under and NDA state. This example is based on the GADT folklore example from . The state of whether an NDA is in place is kept track of as a singleton type, and used to guard the state changes of signing and revoking and NDA, as well as mailing and negotiating sensitive information. In this graphics of the NDA automata, sensitive operations can be seen as only occurring when an NDA is in place (it is in an NDA state).

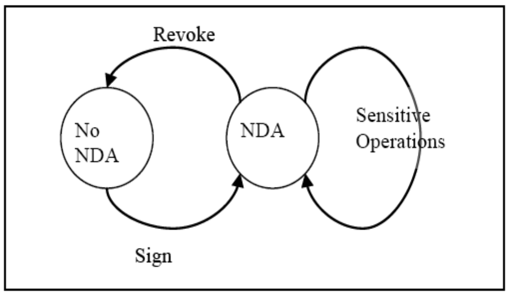

Figure 1: Simple state machine representing non-disclosure policy.

Translating this security policy automata to GADT is fairly straightforward, where the edges are encoded as *acts* moving between states, and the state records which state the system is in and what the last state was. The **`Act`** GADT type constructors happen to know about the NDA states, but notice that these are just singleton types, and could be any of hundreds of possible states that the system is in at any given time. The **`Act`** is dependent on the type parameters.

```
data NDA
data NoNDA
data Act a b where
  Revoke :: Act NoNDA NDA
  Sign :: Act NDA NoNDA
  Mail :: Act NDA NDA
  Negotiate :: Act NDA NDA

data State x y where
  Empty :: State x y
  Commit :: Act x y -> State y z -> State x
```

With a state machine we can implement a protocol for ensuring the behavior of the policy. In our example policy, operations on sensitive information must always be done under a signed NDA. This is instructive in forming the signature for sensitive operations, for example the act of mailing in our policy is a sensitive operation. To simulate a mail operation we **`Commit`** a **`Mail`** act upon the state of a high state variable that contains sensitive information, **`HVar`**.

```
sensitive :: Act a b
      -> Term (State NDA y) pci expi
      -> Term (State NDA NDA) High
High
sensitive (Mail) (HVar past)
```

The high expression signature should be sufficient to ensure that this sensitive operation is always done under a high guard.

4.2 Changing Security Levels

An important aspect of a security policy is in dealing with changing state between multiple levels of security (MLS) within the same system. This is a difficult problem requiring stringent protocols and enforcement. This is another example of a possible requirement forcing the *pc* high to better control MLS concerns.

```
sign :: Term (State NoNDA y) pci expi
      -> Term (State NDA NoNDA) High High
sign (HVar past) = HVar (Commit Sign
```

4.3 Enforcing Sensitive Information Policy

We may wish to guard against sensitive information manipulation by a low side variable. A logical extension to the security policy may state that sensitive information may only be operated on by authorized systems. We can see that the LBIF high/low guard mechanism of rule C6 prevents the inadvertent manipulation of sensitive data by forcing the *pc* to be high for such operations. The semantics of the policy and automata are able to encourage and inform the program specification to be highly aligned with the intended meaning of the policy.

```
:t (If (IsZ (LVar 0))
      (sensitive Mail salaries)
      (sensitive Negotiate salaries))
  Couldn't match expected type `Low' against inferred type `High'
  Expected type: Term (State NDA NDA) Low expt   Inferred type: Term (State NDA NDA) High
  High
```

5. Conclusions

Information flow and dependent programming are powerful paradigms for accelerating and improving secure systems development. These methodologies will benefit from improvements in mechanistic reasoning, but large portions of the purpose, meaning, and structure driving and funding development are not easily rendered mechanistically. The semantics of policy, cost, utility, planning, procurement, etc. are open research problems, and may likely be so for the foreseeable future. Overall, secure systems effectiveness can be increased through improved methods of acquiring and utilizing semantic and human factors, such as policies and valuation (cost and/or utility) in an inclusive security model taking into account the system as a mechanistic and organic whole. We demonstrated here a method for using available software development tools and technologies for integrating and implementing policy elements into an information flow security analysis. Dependent programming may prove to be very effective means of representing and enforcing high resolution yet dynamic security policies that typically were cost prohibitive to design and implement.

References

Butler, S. A. (2002) "Security attribute evaluation method: a cost-benefit approach," *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on,* pp. 232-240.

Denning, D. (1976) “A Lattice Model of Secure Information Flow”, *Comm. ACM,* Vol. 19, No. 5, pp. 236-243

Gibbons, J. (2007) "Generic and Indexed Programming," University of Oxford.

Goguen,J. A. and Meseguer, J. (1982) "Security policies and security models," *Proc. IEEE Symposium on Security and Privacy,* pp. 11–20.

Hunt, S. and Sands, D. (2006) "On flow-sensitive security types," *ACM SIGPLAN Notices, V*ol. 41.

Jones, S. P., Vytiniotis, D., Weirich, S. and Washburn, G. (2006) "Simple unification-based type inference for GADTs," *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming,* pp. 50-61.

Sabelfeld, A. and Myers, A. C. (2003) "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on,* vol. 21, pp. 5-19.

Sandhu, R. (1993) “Lattice-Based Access Control Models”, *Journal of IEEE Computer*, Vol. 26, No. 11, pp. 9-19

Sandhu, R. Coyne, E., Feinstein, H., and Youma, C. (1996) “Role-Based Access Control Models”, *Journal of IEEE Computer,* Vol. 29, No. 2, pp. 38-47

Sheard, T. (2005) "Putting curry-howard to work," *Haskell*.

Sheard, T., Hook, J. and Linger, N. (2005) "GADTs+ Extensible Kinds= Dependent Programming," in *ICFP*, Estonia.

Simonet, V. (2003) "Flow Caml in a nutshell," *Proceedings of the first APPSEM-II workshop,* pp. 152–165.

Sulzmann, M., Schrijvers, T., and Stuckey, P. (2006) "Type inference for GADTs via Herbrand constraint abduction," Technical report, The National University of Singapore.

Walker, D. (2000) "A type system for expressive security policies," *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* pp. 254-267.

Yagiie, M. I., Gallardo, M. D. M. and Mafia, A. (2005) "Semantic access control model: A formal specification," *Lecture notes in computer science,* pp. 24-43.