Refinement Types for Secure Implementations

Research Seminar in Cryptography

Tiit Pikma
Supervised by Dominique Unruh

Spring 2014

Introduction

Implementing cryptographic protocols and access control is difficult and error-prone. Manually verifying the security of those implementation is equally so. Moreover, even small changes to the code can introduce flaws that break the security of the implementation. Static analysis has been used for years to check programs for certain types of errors. It can also be used to verify security properties of cryptographic protocols.

Bengston et al. presented in [1] the design and implementation of a typechecker, which can be used to verify security properties for implementations of cryptographic protocols and access control mechanisms. These security properties are encoded in the types used in a language, and static typechecking can detect errors in the implementations.

The authors presented a formal language, which contains refinement types used to constrain the values in a type and express pre- and post-conditions for expressions. The description of this language is given in Section 1.

Cryptographic protocols are modelled as functions in this language and security properties of the protocol are expressed as type refinements or explicit assumptions and assertions in the language. If typechecking the functions succeeds, then the protocol's imeplementation is secure. The idea behind this approach and related theorems are presented in Section 2.

Bengston et al. implemented a typechecker for the language F#, which constructs type proofs from the implementation source code and annotated interfaces using an external theorem prover. The working principles of the typechecker are presented in Section 3 along with a simple example for static file access control.

Section 4 draws conclusions.

1 A language with refinement types

Bengston et al. propose a language [1], which is essentially Fixpoint Calculus [2] (FPC) with the addition of refinement types and concurrency via forking and message passing. They call this language Refined Concurrent FPC (RCF).

1.1 Syntax and expression safety

The syntax of RCF is as follows:

x, y, z		**variable**
a, b, c		**name of a channel**
h ::=		**value constructor**
	inl	left constructor of a sum
	inr	right constructor of a sum
	fold	constructor of recursive type
M, N ::=		**value**
	x	variable
	$()$	unit
	fun $x \to A$	function
	(M, N)	pair
	$h\, M$	construction
A, B ::=		**expression**
	M	value
	MN	function application
	$M = N$	equality
	let $x = A$ **in** B	binding
	let $(x, y) = M$ **in** B	pair split
	match M **with** $hx \to A$ **else** B	constructor match
	$(\nu a)A$	fresh channel
	$A \Rsh B$	fork
	$a!M$	send M on channel a
	$a?$	receive value from a
	assume C	assumption
	assert C	assertion
C		**first-order logic formula with equality**

Let's explain every piece of syntax, but first some notation: if ϕ is a phrase of syntax, then $\phi\{M/x\}$ is the result of substituting each free occurence of

the variable/type variable x in ϕ with the value/type M.

Variables are placeholders for values. Names are identifiers given to message channels. Constructors are used to create instances of algebraic data types: RCF has constructors for disjoint sums and recursive types (explained in Section 1.2).

Possible values are

- variables, placeholders for values;
- unit, the only possible value for a unit type;
- functions, which map a value to an expression;
- pairs, a binary grouping of values; and
- constructions of type instances.

Expressions are computations which return a value. Possible basic expressions are

- values M, which just return M;
- function applications MN, where $M = \mathbf{fun}\ x \to A$, which evaluate $A\{N/x\}$;
- equalities $M = N$, which return **true** if M and N are the syntactically equal, and **false** otherwise (**true** and **false** will be defined in Section 1.2);
- bindings $\mathbf{let}\ x = A\ \mathbf{in}\ B$, which first evaluate A and if it returns a value M, evaluate $B\{M/x\}$;
- pair splits $\mathbf{let}\ (x, y) = M\ \mathbf{in}\ A$, where $M = (N_1, N_2)$, which evaluate $A\{N_1/x\}\{N_2/y\}$; and
- constructor matches $\mathbf{match}\ M\ \mathbf{with}\ h\,x \to A\ \mathbf{else}\ B$, which, if $M = h\,N$ for some N, evaluate $A\{N/x\}$, and otherwise evaluate B.

The four expressions to enable concurrent, message-passing computations are

- $(\nu a)A$, which creates a fresh channel with the unique name c and evaluates $A\{c/a\}$ (c can only be used inside A);
- $A \Rsh B$, which starts a new concurrent thread to evaluate A discarding its result value, and evaluates B in the current thread;
- $a!M$, which sends the value M on the channel a and immediately returns (); and
- $a?$, which blocks until some message N is available on the channel a—if such a message appears, the expression removes N from the channel and returns it.

The state of a program in RCF consists of *a*) a multiset of concurrent expressions being evaluated in parallel; *b*) a multiset of messages sent on channels, but not yet read; and *c*) a multiset of assumed first-order logic fomulas with equality, called the *log*. The first two are usual for concurrent, message-passing languages, but the log is not. It is used to specify correctness

properties of a program. The expression **assume** C adds the formula C to the log, indicating that C is assumed to hold. The expression **assert** C is used to assert that the formula C logically follows from the log: it checks if $S \vdash C$, where S is the set of formulas in the log. Both **assume** C and **assert** C always return () (regardless if the assertion succeeds).

Definition (Expression safety)**.** *A closed expression (an expression without free variables) A is* safe *iff in all evaluations of A all assertions succeed.*

Alternatively, an expression is unsafe iff there exists an evaluation of A which leads to an **assert** C, where C does not follow from the log. So we are not considering other errors like deadlocks, infinite loops, etc.

1.2 Types

Every value in RCF has a type. The fact that a value x has the type T is written as $x : T$. The types in RCF have the following syntax:

T, U ::=		**type**
	α	type variable
	unit	unit type
	$\Pi x : T.U$	dependent function type
	$\Sigma x : T.U$	dependent pair type
	$T + U$	disjoint sum type
	$\mu\alpha.T$	iso-recursive type
	$\{x : T \mid C\}$	refinement type
$\{C\}$		**ok-type**
bool		**Boolean type**

A type variable is a placeholder for a type. A unit type is a type which can only hold the value ().

A value of type $\Pi x : T.U$ is a unary function M such that if N is a value of the type T, then function application MN has type $U\{N/x\}$. This is called a dependent function, because the return type depends on the *value* of N, not just its type (x in U is replaced by N, not T). Nullary functions are functions which take a unit argument, and n-ary functions are created by chaining n unary functions.

A value of type $\Sigma x : T.U$ is a pair (M, N), where M has type T and N has type $U\{M/x\}$. This is called a dependent pair, because the second element's type depends on the first element's *value*. Longer tuples are created by pairs, where the second element is another pair, etc.

A disjoint sum $T + U$ is a variant type, where a value with this type is

either **inl** M, where M has type T, or **inr** N, where N has type U.

A value with the recursive type $\mu\alpha.T$ is a construction **fold** M, where M has the unfolded type $T\{\mu\alpha.T/\alpha\}$. Being iso-recursive means that the type and it's unfolding are distinct types, i.e. $\mu\alpha.T$ and $T\{\mu\alpha.T/\alpha\}$ are *not* equal types.

A value M with the refinement type $\{x : T \mid C\}$ has the type T such that the formula $C\{M/x\}$ can be deduced from the log. Among other things, this can be used to apply restrictions to the value M which has this type.

The ok-type $\{C\}$ is just syntactic sugar for $\{x : \textbf{unit} \mid C\}$, where x is *not* a free variable in C. This is used as a token during typechecking to show that C holds.

Similarly, the Boolean type **bool** is just syntactic sugar for the sum type **unit** + **unit**. It can only have two values: **inl** () and **inr** (), which we define as **false** and **true** respectively.

The type system also supports subtyping: we denote that T is a subtype of T'—i.e., a value of type T can be safely used in a context where a value of type T' is expected—as $T <: T'$. In relation to refinement types, if $T <: T'$ and $C \vdash C'$, then $\{x : T \mid C\} <: \{x : T' \mid C'\}$. Also, if the formula C in the refinement type always holds, then $\{x : T \mid C\}$ is type equivalent ($<:>$) to T.

1.3 Safety by typing

Typechecking an expression is done by constructing a logic proof. We denote the judgement that an expression A has type T as $E \vdash A : T$, where E is the typing environment, which tracks the type of all variables in scope. If we can construct a proof which shows that this judgement holds, then typechecking succeeds.

Theorem 1 (Safety)**.** *If $\varnothing \vdash A : T$, then the expression A is safe.*

Proof. See Appendix C in [1]. □

What this theorem essentially means is that if typechecking a closed expression A is successful, then that expression is safe (as defined in Section 1.1).

All the typing rules used to construct the proof are given in Section 4 of [1]: let's only look at the ones regarding refinement types, **assume**, and **assert**. The judgement $E \vdash C$ denotes that the formula C follows from the formulas contained in refinement types in E.

Refinement types If $E \vdash M : T$ and $E \vdash C\{M/x\}$, then $E \vdash M : \{x : T \mid C\}$.

assume Since assumption is only used to introduce new formulas to the environment, then we typecheck them as $E \vdash \textbf{assume}\ C : \{C\}$, where all free variables in C must exist in E.

assert If $E \vdash C$, then $E \vdash \textbf{assert}\ C : \textbf{unit}$.

As an example, let's typecheck the simple expression $\textbf{let}\ x = 10\ \textbf{in}\ \textbf{assert}\ (x > 5) : \textbf{unit}$. We can see that the assertion succeeds (10 is indeed greater than 5), so the expression is safe. But now we want to prove that the expressions is safe via typechecking.

For this we will additionally need the typing rule for **let** expressions, which is as follows: if $E \vdash A : T$ and $E, x : T \vdash B : U$, then $E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : U$.

$$\dfrac{\dfrac{\varnothing \vdash 10 : \textbf{integer} \qquad \varnothing \vdash 10 = 10}{\varnothing \vdash 10 : \{x : \textbf{integer} \mid x = 10\}} \qquad \dfrac{x : \{x : \textbf{integer} \mid x = 10\} \vdash x > 5}{x : \{x : \textbf{integer} \mid x = 10\} \vdash \textbf{assert}(x > 5) : \textbf{unit}}}{\varnothing \vdash \textbf{let}\ x = 10\ \textbf{in}\ \textbf{assert}\ (x > 5) : \textbf{unit}}$$

The left branch is obvious given the typing rule for refinement types. The right branch concludes with an axiom, because $x > 5$ directly follows from the formula contained in the type of x (i.e., $x = 10$).

Since this **let** expression typechecked succesfully, the theorem states it is safe, which—as we already concluded—it is.

2 Modelling cryptographic protocols

Next we will try to see how we can specify security properties for cryptographic protocols using safety by typing. We model each party of the protocol (e.g. a client and a server) as a function which communicates with others on channels that represent the public network. This is done similarly as in [3].

2.1 Authentication properties

Authentication properties of a cryptographic protocol are presented as **assume** and **assert** expressions in its functions. For example, when a client sends a message, it adds to the log a predicate (via **assume**), which states that a client sent that message. On receiving a message, the server first asserts that the message was sent by a client and not an adversary.

2.1.1 Opponents

Our goal is to ensure that these properties hold in the presence of an active opponent. The opponent is modelled as an arbitrary expression O, which knows the protocol functions and has access to the channels representing the public network. This expression can fork new instances of the protocol functions or just execute (almost) arbitrary expressions, and can send or intercept messages from the public channels, with the goal of forcing the failure of an **assert** expression, demonstrating that the asserted authentication property does not hold.

Definition (Opponent)**.** *A closed expression O is an* opponent *iff O contains no occurence of* ***assert****.*

The opponent can not contain an **assert** expression, otherwise it would be trivial for it to force a failing assertion.

Definition (Robust safety)**.** *A closed expression A is* robustly safe *iff the application OA is safe for all opponents O.*

As an example consider authenticated messages sent from a client to a server. Let the client be the function

$$\mathbf{fun}\, c \to (\mathbf{let}\, x = 10\, \mathbf{in}\, (\mathbf{let}\, \_ = \mathbf{assume}\, \mathit{Send}(x)\, \mathbf{in}\, c!x))$$

and the server the function

$$\mathbf{fun}\, c \to (\mathbf{let}\, x = c?\, \mathbf{in}\, (\mathbf{assert}\, \mathit{Send}(x))),$$

where both functions take as an argument the same channel c.

When the server receives a message from the client, then the assertion succeeds, because all messages x sent on the channel c are preceded by the assumption $\mathit{Send}(x)$. But if an opponent sends a value on the channel which the client has not sent, then the assertion fails. I.e., the opponent

$$\mathbf{fun}\, \mathit{server} \to (\nu c)(\mathbf{let}\, \_ = c!11\, \mathbf{in}\, \mathit{server}\, c),$$

breaks the assertion, because $\mathit{Send}(11)$ can not be deduced from the log. Therefore this protocol implementation is not robustly safe.

2.1.2 The universal type

Since the opponent is an arbitrary closed expression, we do not know what its type is. Because of this, the *universal type* **Un** was introduced. **Un** is type equivalent to

- **unit**,
- $\Pi x : \mathbf{Un}.\mathbf{Un}$,
- $\Sigma x : \mathbf{Un}.\mathbf{Un}$,
- $\mathbf{Un} + \mathbf{Un}$, and
- $\mu\alpha.\mathbf{Un}$.

Using the universal type, we can type $O : \mathbf{Un}$ for all opponents O.

Additionally, we can characterize two kinds[1] of types: *public types*, whose values may flow to the opponent, and *tainted types*, whose values may flow from the opponent.

Definition (Public type)**.** *A type T is* public *iff* $T <: \mathbf{Un}$.

Definition (Tainted type)**.** *A type T is* tainted *iff* $\mathbf{Un} <: T$.

Again, all kinding rules are given in the Section 4 of [1] and we will only look at the ones regarding refinement types.

Public refinement types $E \vdash \{x : T \mid C\} <: \mathbf{Un}$ iff $E \vdash T <: \mathbf{Un}$, i.e., a refinement type is public iff the type it is refining is public.

Tainted refinement types $E \vdash \mathbf{Un} <: \{x : T \mid C\}$ iff $E \vdash \mathbf{Un} <: T$ and $E, x : T \vdash C$, i.e., a refinement type is tainted iff the type it refines is tainted and the refinement formula C deduces from the refinement formulas in E regardless of the value of x. Intuitively, this means that a refinement type is only tainted if the opponent can choose *any* value x of type T.

Note that not all types are public and not all types are tainted ([1] constructs examples of both), meaning that not all types are equivalent to the universal type **Un** and **Un** is not the top-most type.

Now that we can type an arbitrary opponent, we can reason about opponents in cryptographic protocols.

Theorem 2 (Robust safety)**.** *If $\varnothing \vdash A : \mathbf{Un}$, then A is robustly safe.*

Proof. See Appendix C in [1]. □

This means that to verify the authentication properties of a protocol, we must check that each function of the protocol is a safe expression with type T and T is public (a subtype of **Un**).

[1] *Kinds* are metatypes: types of types.

2.2 Secrecy properties

Section 2.1 showed how to check that an opponent can not invalidate the authentication properties given in a protocol. Next we want to consider secrecy properties. For example if one party sends another an encrypted message over the public channels, we do not want an opponent to be able to learn the value of the message. A value is considered secret if no opponent can gain direct access to it.[2]

Definition (Robust secrecy)**.** *Let A be an expression with a free variable s. The expression A* preserves the secrecy of s unless C *iff the expression* ***let*** $s = (\textbf{\textit{fun}}\ \_ \rightarrow \textbf{\textit{assert}}\ C)$ ***in*** A *is robustly safe.*

If the opponent would learn the value of s, it could simply invoke the function application $s()$, triggering the assertion. Because by definition the above expression would be robustly safe, no assertion could fail and therefore C would have to follow from the log.

Theorem 3 (Robust secrecy)**.** *If $s : (\Pi x : \{C\}.\textbf{unit}) \vdash A : \textbf{Un}$, then A preserves the secrecy of s unless C.*

Proof. This follows as a simple corollary from robust safety (Theorem 2) using typing rules. The proof is given in Section 3.7 in [1]. □

An example of how to apply this theorem is symmetric encryption: suppose the expression A evaluates to the encryption of s with a symmetric key, and the formula C models the fact that the opponent knows the symmetric key. If typechecking $s : (\Pi x : \{C\}.\textbf{unit}) \vdash A : \textbf{Un}$ succeeds, then we can be sure that encrypting s via A preserves the secrecy of the value of s unless the opponent knows the symmetric key used.

3 Implementing refinement types for F#

To apply refinement types in practice, Bengston et al. implemented an enhanched typechecker for the language F# [4] called F7. Source code that is checked by F7 consists of extended typed interfaces, which contain declarations for types, values and security policies (i.e. assumptions), and concrete modules implemented in F#, which define the values declared in the interface and optionally use other interfaces to call code from other modules.

[2]Note that this form of secrecy does not protect against the secret value implicitly flowing to the opponent, only that the opponent can not access that value directly.

RCF has minimalistic formal syntax, which is not practical for writing module interfaces. Because of this the interfaces are written in F# with additional refinements for F7. An explanation on how to map the different syntaxes from one to another is given in [1].[3] Here we will only look at how interfaces declare values and how modules define those values.

An F7 interface contains *value* declarations with the form

$$\mathbf{val}\, x_1 : T_1 \ldots \mathbf{val}\, x_n : T_n,$$

which is interpreted as a RCF tuple type

$$\Sigma x_1 : T_1. \ldots \Sigma x_{n-1} : T_{n-1}.T_n.$$

A value declaration can also be of the form **private val** $x_i : T_i$, in which case the value is not public and is not made available to the opponent. This also means that those values are excluded when typechecking robust safety (Theorem 2).

If the interface contains any type declarations of the form **type** $x = \ldots$, then those are handled as just synonyms for the types they declare. If the interface contains any security policies (assumptions), then the statement **assume** C is interpreted as a value declaration **val** $x : \{C\}$ and a corresponding definition **let** $x =$ **assume** C, where x is a new, unique variable.

A F# module is a sequence of value definitions of the form

$$\mathbf{let}\, x_1 = A \ldots \mathbf{let}\, x_n = A_n,$$

which is interpreted as the expression

$$\mathbf{let}\, x_1 = A_1\, \mathbf{in} \ldots \mathbf{let}\, x_n = A_n\, \mathbf{in}\, (x_1, (x_2, \ldots (x_{n-1}, x_n) \ldots)$$

in RCF.

If T is the RCF tuple type of value declarations in an interface and A the RCF expression of definitions in a module, then F7 typechecks $A : T$, i.e. if the expression returns a tuple whose type matches the tuple type of the interface.

In addition, F7 does kindchecking: it ensures that each public value (data that can flow to the opponent) has a public type, i.e. the value declaration in the interface is not preceded by the keyword **private**.

When type- and kindchecking, the typechecker must create logic proofs. In trivial cases it does so itself, but for more complicated proofs, it generates a

[3] RCF and F# are not completely compatible, so only a subset of the F# language is supported by F7.

proof obligation in the Simplify format [5] and lets the Z3 prover [6] generate the proof. If Z3 is able to provide a proof, then type- and kindchecking of the module succeeded.

In order to enable writing F# module implementations which utilize these extended typed interfaces, F7 creates a plain F# interface from the RCF one by erasing all refinements. Modules can be written against this stripped-down interface, because the information that was removed is only necessary for F7.

Note that not all source code can be verified, because the program can use some standard or trusted libaries. We can provide a F7 interface for those libraries, but we must trust their implementation, since we do not have access to their source code (or alternatively we have to reimplement them ourselves).

3.1 Example: Access control in partially-trusted code

The full paper [1] contains many examples which help gain insight into what is possible with F7. A basic one is included here to at least give a practical idea of what F7 provides—it is a modification of an example given in [1].

The example does static enforcement of file access policies. Suppose we have a file which must not be read by the program, a file which can only be read by the program, and a file which the program can read and write. We have trusted implementations for read and write functions, but now we want to use those functions in code which is typechecked, but not inherently trusted, such as a browser plug-in or a mobile application.

First we create a F7 interface for the read and write functions. Assume we have already defined a type called `string`, which is just a standard F# string. Next we need two unary predicates: `CanRead(x)` which is true if the file x is readable by the partially-trusted code, and `CanWrite(x)`, which is true if x is writable.

These predicates are modelled as variant constructors for an algebraic data type `predicates`, taking a string argument. Using F# syntax, this will be written in the F7 interface as

```
type predicates =
  | CanRead of string
  | CanWrite of string
```

Next specify the security policies for static access control:

```
assume CanRead("/etc/motd")
assume CanWrite("/tmp/writeable")
assume ∀x. CanWrite(x)⇒CanRead(x)
```

Finally include the value declarations for the read and write functions in the interface:

```
val read: file:string{CanRead(file)} → string
val write: file:string{CanWrite(file)} → (string → unit)
```

Here $x : T\{C\}$ in F7 syntax is equivalent to $\{x : T \mid C\}$ in RCF (because F# does not have dependent types) and $x : T \to U$ is a function which takes the argument $x : T$ and returns a value with the type U. So `read` has the RCF type

$$\Pi x : \{x : \texttt{string} \mid \mathit{CanRead}(x)\}.\mathbf{string}$$

and `write` has the RCF type

$$\Pi x : \{x : \mathbf{string} \mid \mathit{CanWrite}(x)\}.(\Pi y : \mathbf{string}.\mathbf{unit}),$$

i.e. it takes as the first argument a string with the file name and returns a function which takes as input a string to write to that file and returns a unit.

Essentially, the formulas in the refinement types of the arguments act as pre-conditions for the functions and are statically checked by F7.

The plain F# interface that F7 generates via erasure would contain just

```
val read: file:string → string
val write: file:string → (string → unit)
```

without any extra types and assumptions. Now an application can be written which uses this plain interface and we can typecheck it against the F7 interface.

If the application contains the function calls

```
read "/etc/motd"
write "/tmp/writeable"
```

then typechecking succeeds, because we have explicitly assumed that `/etc/motd` is readable and `/tmp/writeable` is writeable. The function call

```
read "/tmp/writeable"
```

would also typecheck, because the file is writeable and we have a formula, which states that all writeable files are readable.

However, if the code would attempt

```
read "/etc/passwd"
```

then that would cause a type error, because we can not construct a proof, which states that `CanRead("/etc/passwd")` holds.

The full paper [1] also contains examples of cryptographic protocols, which should be easy to follow using the concepts explained in this report, so we will not duplicate them here.

4 Conclusions

Security properties of cryptographic protocols can be encoded as logical formulas in refined types of a language and a static typechecker can be used to verify that these properties hold in an implementation of the protocol. This allows us to statically check for errors in protocol code by constructing logic proofs and ensure a secure implementation where the specified security properties hold.

References

[1] J. Bengston, K. Bhargavan, C. Fournet, A. D. Gordon, S. Maffeis, *Refinement Types for Secure Implementations*. CSF, 2008.

[2] C. Gunter, *Semantics of programming languages*. MIT Press, 1992.

[3] K. Bhargavan, C. Fournet, A. D. Gordon, S. Tse, *Verified interoperable implementations of security protocols*. ACM TOPLAS, 31:5:15:61, 2008.

[4] *The F# Software Foundation*. `http://fsharp.org/`, last accessed May 27, 2014.

[5] D. Detles, G. Nelson, J. Saxe, *Simplify: A theorem prover for program checking*. J. ACM, 52(3):365-473, 2005.

[6] *Z3*. `http://z3.codeplex.com/`, last accessed May 27, 2014.