Information Flow Control-by-Construction for an Object-Oriented Language

Tobias Runge[1,2] [0000-0002-9154-7743], Alexander Kittelmann[1,2] [0000-0002-8804-7051], Marco Servetto[3], Alex Potanin[4] [0000-0002-4242-2725], and Ina Schaefer[1,2]

[1] TU Braunschweig, Braunschweig, Germany
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
[3] Victoria University of Wellington, Wellington, New Zealand
[4] Australian National University, Canberra, Australia
{tobias.runge,alexander.kittelmann,ina.schaefer}@kit.edu, marco@ecs.vuw.ac.nz, alex.potanin@anu.edu.au

Abstract. In security-critical software applications, confidential information must be prevented from leaking to unauthorized sinks. Static analysis techniques are widespread to enforce a secure information flow by checking a program after construction. A drawback of these systems is that incomplete programs during construction cannot be checked properly. The user is not guided to a secure program by most systems. We introduce IFbCOO, an approach that guides users incrementally to a secure implementation by using refinement rules. In each refinement step, confidentiality or integrity (or both) is guaranteed alongside the functional correctness of the program, such that insecure programs are declined by construction. In this work, we formalize IFbCOO and prove soundness of the refinement rules. We implement IFbCOO in the tool CorC and conduct a feasibility study by successfully implementing case studies.

Keywords: correctness-by-construction, information flow control, security-by-design

1 Introduction

For security-critical software, it is important to ensure *confidentiality* and *integrity* of data, otherwise attackers could gain access to this secure data. For example, in a distributed system, one client `A` has a lower privilege (i.e., a lower security level) than another client `B`. When both clients send information to each other, security policies can be violated. If `A` reads secret data from `B`, confidentiality is violated. If `B` reads untrusted data from `A`, the integrity of `B`'s data is no longer guaranteed. To ensure security in software, mostly static analysis techniques are used, which check the software after development [28]. A violation of security is only revealed after the program is fully developed. If violations occur, an extensive and repetitive repairing process of writing code and checking the security properties with the analysis technique is needed. An alternative is to check the security with language-based techniques such as type systems [28] during the development. In such a secure type system, every expression is assigned to a type, and a set of typing rules checks that the security policy is not violated [28]. If violations occur, an extensive process of debugging is required until the code is type-checked.

To counter these shortcomings, we propose a constructive approach to directly develop functionally correct programs that are secure by design without the need of a *post-hoc* analysis. Inspired by the correctness-by-construction (CbC) approach for functional correctness [18], we start with a security specification and refine a high-level abstraction of the program stepwise to a concrete implementation using a set of refinement rules. Guided by the security specification defining the allowed security policies on the used data, the programmer is directly informed if a refinement is not applicable because of a prohibited information flow. With IFbCOO (Information Flow control by Construction for an Object-Oriented language), programmers get a local warning as soon as a refinement is not secure, which can reduce debugging effort. With IFbCOO, functionally correct and secure programs can be developed because both, the CbC refinement rules for functional correctness and the proposed refinement rules for information flow security, can be applied simultaneously.

In this paper, we introduce IFbCOO which supports information flow control for an object-oriented language with type modifiers for mutability and alias control [13]. IFbCOO is based on IFbC [25] proposed by some of the authors in previous work, but lifts its programming paradigm from a simple imperative language to an object-oriented language. IFbC introduced a sound set of refinement rules to create imperative programs following an information flow policy, but the language itself is limited to a simple while-language. In contrast, IFbCOO is based on the secure object-oriented language SIFO [27]. SIFO's type system uses immutability and uniqueness properties to facilitate information flow reasoning. In this work, we translate SIFO's typing rules to refinement rules as required by our correctness-by-construction approach. This has the consequence that programs written in SIFO and programs constructed using IFbCOO are interchangeable. In summary, our contributions are the following. We formalize IFbCOO and establish 13 refinement rules. We prove soundness that programs constructed with IFbCOO are secure. Furthermore, we implement IFbCOO in the tool CorC and conduct a feasibility study.

2 Object-Oriented Language SIFO by Example

SIFO [27] is an object-oriented language that ensures secure information flow through a type system with precise uniqueness and (im)mutability reasoning. SIFO introduces four type modifiers for references, namely `read`, `mut`, `imm`, and `capsule`, which define allowed aliasing and mutability of objects in programs. While, `mut` and `imm` point to mutable and immutable object respectively, a `capsule` reference points to a mutable object that cannot be accessed from other `mut` references. A `read` reference points to an object that cannot be aliased or mutated. In this section, SIFO is introduced with examples to give an overview of the expressiveness and the security mechanism of the language. We use in the examples two security levels, namely `low` and `high`. An information flow from `low` to `high` is allowed, whereas the opposite flow is prohibited. The security levels can be arranged in any user-defined lattice. In Section 4, we introduce SIFO formally. In Listing 1, we show the implementation of a class `Card` containing a `low` immutable `int number` and two `high` fields: a mutable `Balance` and an immutable `Pin`.

```
1  class Card{low imm int number; high mut Balance blc;
2    high imm Pin pin;}
3  class Balance{low imm int blc;}
4  class Pin{low imm int pin;}
```

Listing 1: Class declarations

In Listing 2, we show allowed and prohibited field assignments with immutable objects as information flow reasoning is the easiest with these references. In a secure assignment, the assigned expression and the reference need the same security level (Lines 6,7). This applies to mutable and immutable objects. The security level of expressions is calculated by the least upper bound of the accessed field security level and the receiver security level. A `high int` cannot be assigned to a `low blc` reference (Line 8) because this would leak confidential information to an attacker, when the attacker reads the `low blc` reference. The assignment is rejected. Updates of a `high` immutable field are allowed with a `high int` (Line 9) or with a `low int` (Line 10). The `imm` reference guarantees that the assigned integer is not changed, therefore, no new confidential information can be introduced and a promotion in Line 10 is secure. The promotion alters the security level of the assigned expression to be equal to the security level of the reference. As expected, the opposite update of a `low` field with a `high int` is prohibited in Line 11 because of the direct flow from higher to lower security levels.

```
 5  low mut Card c = new low Card();//an existing Card reference
 6  high mut Balance blc = c.blc;//correct access of high blc
 7  high imm int blc = c.blc.blc;//correct access of high blc.blc
 8  low imm int blc = c.blc.blc;//wrong high assigned to low
 9  c.blc.blc = highInt;//correct field update with high int
10  c.blc.blc = c.number;//correct update with promoted imm int
11  high imm int highInt = 0;//should be some secret value
12  c.number = highInt;//wrong, high int assigned to low c.number
```

Listing 2: Examples with immutable objects

Next, in Listing 3, we exemplify which updates of mutable objects are legal and which updates are not. We have a strict separation of mutable objects with different security levels. We want to prohibit that an update through a higher reference is read by lower references, or that an update through lower references corrupt data of higher references. A new `Balance` object can be initialized as a `low` object because the `Balance` object itself is not confidential (Line 12). The association to a `Card` object makes it a confidential attribute of the `Card` class. However, the assignment of a `low mut` object to a `high` reference is prohibited. If Line 13 would be accepted, Line 14 could be used to insecurely update the confidential `Balance` object because the `low` reference is still in scope of the program. Only an assignment without aliasing is allowed (Line 16). With `capsule`, an encapsulated object is referenced to which no other `mut` reference points. The `low capsBlc` object can be promoted to a `high` security level and assigned. Afterwards, the `capsule` reference is no longer accessible. In the case of an immutable object, the aliasing is allowed (Line 18), since the object itself cannot be updated (Line 19). Both `imm` and `capsule` references are usable to communicate between different security levels.

```
12 low mut Balance newBlc = new low Balance(0);//ok
13 c.blc = newBlc;//wrong, mutable secret shared as low and high
14 newBlc.blc = 10;//ok? Insecure with previous line
15 low capsule Balance capsBlc = new low Balance(0);//ok
16 c.blc = capsBlc;//ok, no alias introduced
17 low imm Pin immPin = new low Pin(1234);//ok
18 c.pin = immPin;//ok, pin is imm and can be aliased
19 immPin.pin = 5678;//wrong, immutable object cannot be updated
```

Listing 3: Examples with mutable and encapsulated objects

3 IFbCOO by Example

With IFbCOO, programmers can incrementally develop programs, where the security levels are organized in a lattice structure to guarantee a variety of confidentiality and integrity policies. IFbCOO defines 13 refinement rules to create secure programs. As these rules are based on refinement rules for correctness-by-construction, programmers can simultaneously apply refinements rules for functional correctness [18, 26, 12] and security. We now explain IFbCOO in the following examples. For simplicity, we omit the functional specification. IFbCOO is introduced formally in Section 4.

In IFbCOO, the programmer starts with a class including fields of the class and declarations of method headers. IFbCOO is used to implement methods in this class successively. The programmer chooses one abstract method body and refines this body to a concrete implementation of the method. A starting IFbCOO tuple specifies the typing context Γ and the abstract method body eA. The expression eA is abstract in the beginning and refined incrementally to a concrete implementation. During the construction process, local variables can be added. The refinement process in IFbCOO results in a method implementation which can be exported to the existing class. First, we give a fine-grained example to show the application of refinement rules in detail. The second example illustrates that IFbCOO can be used to implement larger methods.

The first example in Listing 4 is a setter method. A field `number` is set with a parameter `x`. We start the construction with an abstract expression $eA : [\Gamma; \texttt{low imm void}]$ with a typing context $\Gamma = \texttt{low mut}\ C\ \texttt{this}, \texttt{low imm int x}$

extracted from the method signature (C is the class of the method receiver). The abstract expression eA contains all local information (the typing context and its type) to be further refined. A concrete expression that replaces the abstract expression must have the same type `low imm void`, and it can only use variables from the typing context Γ. The tuple $[\Gamma;$`low imm void`$]$ is now refined stepwise. First, we introduce a field assignment: $eA \rightarrow eA_1$`.number` $= eA_2$. The newly introduced abstract expressions are $eA_1 : [\Gamma;$`low mut` $C]$ and $eA_2 : [\Gamma;$`low imm int`$]$ according to the field assignment refinement rule. In the next step, eA_1 is refined to `this`, which is the following refinement: eA_1`.number` $= eA_2 \rightarrow$ `this.number` $= eA_2$. As `this` has the same type as eA_1, the refinement is correct. The last refinement replaces eA_2 with `x`, resulting in `this.number` $= eA_2 \rightarrow$ `this.number` $=$ `x`. As `x` has the same type as eA_2, the refinement is correct. The method is fully refined since no abstract expression is left.

```
low mut method low imm void setNumber(low imm int x) {
  this.number = x; }
```

Listing 4: Set method

To present a larger example, we construct a check of a signature in an email system (see Listing 5). The input of the method is an `email` object and a `client` object that is the receiver of the email. The method checks whether the key with which the `email` object was signed and the stored public key of the `client` object are a valid pair. If this is the case, the `email` object is marked as verified. The fields `isSignatureVerified` and `emailSignKey` of the class `email` have a `high` security level, as they contain confidential data. The remaining fields have `low` as security level.

```
static low imm void verifySignature(
  low mut Client client, low mut Email email) {
  low imm int pubkey = client.publicKey;
  high imm int privkey = email.emailSignKey;
  high imm boolean isVerified;
  if (isKeyPairValid(privkey, pubkey)) {
    isVerified = true;
  } else {
    isVerified = false;
  }
  email.IsSignatureVerified = isVerified;
}
```

Listing 5: Program of a secure signature verification

In Figure 1, we show the starting IFbCOO tuple with the security level of the variables (type modifier and class name are omitted) at the top. In our example, we have two parameters `client` and `email`, with a `low` security level. To construct the algorithm of Listing 5, the method implementation is split into three parts. First, two local variables (private and public key for the signature verification) are initialized and a Boolean for the result of the verification is

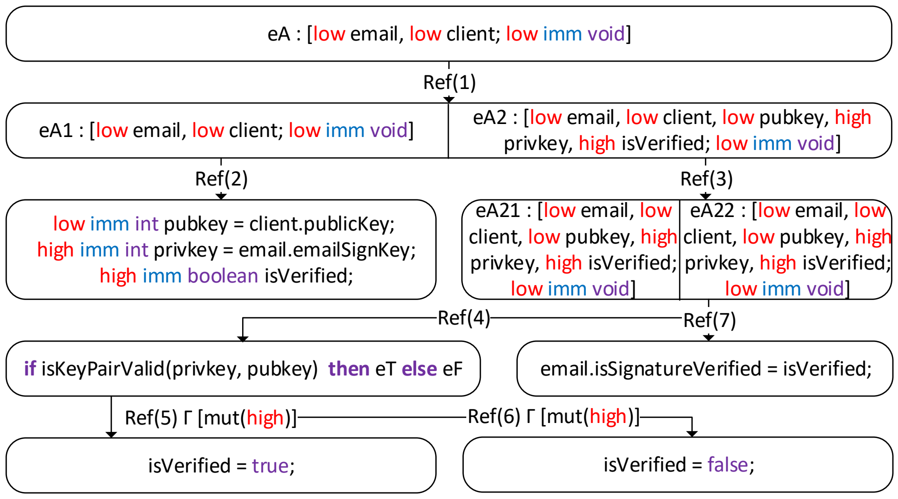
eA : [low email, low client; low imm void]
Ref(1)
eA1 : [low email, low client; low imm void]
eA2 : [low email, low client, low pubkey, high privkey, high isVerified; low imm void]
Ref(2)
Ref(3)
low imm int pubkey = client.publicKey;
high imm int privkey = email.emailSignKey;
high imm boolean isVerified;
eA21 : [low email, low client, low pubkey, high privkey, high isVerified; low imm void]
eA22 : [low email, low client, low pubkey, high privkey, high isVerified; low imm void]
Ref(4)
Ref(7)
if isKeyPairValid(privkey, pubkey) then eT else eF
email.isSignatureVerified = isVerified;
Ref(5) Γ [mut(high)]
Ref(6) Γ [mut(high)]
isVerified = true;
isVerified = false;

Fig. 1: Refinement steps for the signature example

declared. Second, verification whether the keys used for the signature form a valid pair takes place. Finally, the result is saved in a field of the `email` object.

Using the refinement rule for composition, the program is initially split into the initialization phase and the remainder of the program's behavior (Ref(1)). This refinement introduces two abstract expressions $eA1$ and $eA2$. The typing contexts of the expressions are calculated by IFbCOO automatically during refinement. As we want to initialize two local variables by further refining $eA1$, the finished refinement in Figure 1 already contains the local `high` variables `privkey` and `isVerified`, and the `low` variable `pubkey` in the typing context of expression $eA2$.

In Ref(2), we apply the assignment refinement[5] to initialize the integers `pubkey` and `privkey`. Both references point to immutable objects that are accessed via fields of the objects `client` and `email`. The security levels of the field accesses are determined with the field access rule checked by IFbCOO. The determined security level of the assigned expression must match the security level of the reference. In this case, the security levels are the same. Additionally, it is enforced that immutable objects cannot be altered after construction (i.e., it is not possible to corrupt the private and public key). In Ref(3), the next expression $eA2$ is split with a composition refinement into $eA21$ and $eA22$.

Ref(4) introduces an if-then-else-expression by refining $eA21$. Here, it is checked whether the public and private key pair is valid. As the `privkey` object has a `high` security level, we have to restrict our typing context with $\Gamma[mut(\texttt{high})]$. This is necessary to prevent indirect information leaks. With the restrictions, we can only assign expressions to at least `high` references and mutate `high`

[5] To be precise, it would be a combination of composition and assignment refinements, because an assignment refinement can only introduce one assignment expression.

$$
\begin{array}{lll}
T & ::= & s\ mdf\ C \\
s & ::= & \texttt{high} \mid \texttt{low} \mid \ldots \text{(user defined)} \\
mdf & ::= & \texttt{mut} \mid \texttt{imm} \mid \texttt{capsule} \mid \texttt{read} \\
CD & ::= & \texttt{class}\ C\ \texttt{implements}\ \overline{C}\ \{\overline{F}\ \overline{MD}\ \} \mid \texttt{interface}\ C\ \texttt{extends}\ \overline{C}\ \{\overline{MH}\} \\
F & ::= & s\ \texttt{mut}\ C\ f; \mid s\ \texttt{imm}\ C\ f; \\
MD & ::= & MH\ \{\texttt{return}\ e;\} \\
MH & ::= & s\ mdf\ \texttt{method}\ T\ m(T_1\ x_1,\ \ldots,\ T_n\ x_n) \\
e & ::= & eA \mid x \mid e_0.f = e_1 \mid e.f \mid e_0.m(\overline{e}) \mid \texttt{new}\ s\ C(\overline{e}) \mid e_0; e_1 \\
 & & \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{while}\ e_0\ \texttt{do}\ e_1 \mid \texttt{declassify}(e) \\
\Gamma & ::= & x_1 : T_1 \ldots x_n : T_n \\
\mathcal{E} & ::= & [] \mid \mathcal{E}.f \mid \mathcal{E}.f = e \mid e.f = \mathcal{E} \mid \mathcal{E}.m(\overline{e}) \mid e.m(\overline{e}\ \mathcal{E}\ \overline{e}) \mid \texttt{new}\ s\ C(\overline{e}\ \mathcal{E}\ \overline{e})
\end{array}
$$

Fig. 2: Syntax of the extended core calculus of SIFO

objects (mut(`high`)) in the then- and else-expression. If we assign a value in the then-expression to a `low` reference that is visible outside of the then-expression, an attacker could deduce that the guard was evaluated to true by reading that `low` reference.

Ref(5) introduces an assignment of an immutable object to a `high` reference, which is allowed in the restricted typing context. As explained, the assignment to `low` references is forbidden. The assigned immutable object `true` can be securely promoted to a `high` security level. In Ref(6), a similar assignment is done, but with the value `false`. Ref(7) sets a field of the `email` object by refining $eA22$. We update the `high` field of the `email` object by accepting the `high` expression `isVerified`. With this last refinement step, the method is fully concretized. The method is secure by construction and constitutes valid SIFO code (see Listing 5).

4 Formalizing Information Flow Control-by-Construction

In this section, we formalize IFbCOO for the construction of functionally correct and secure programs. Before, we introduce SIFO as the underlying programming language formally.

4.1 Core Calculus of SIFO

Figure 2 shows the syntax of the extended core calculus of SIFO [27]. SIFO is an expression-based language similar to Featherweight Java [17]. Every reference and expression is associated with a type T. The type T is composed of a security level s, a type modifier mdf and a class name C. Security levels are arranged in a lattice with one greatest level $\top$ and one least level $\bot$ forming the security policy. The security policy determines the allowed information flow. Confidentiality and integrity can be enforced by using two security lattices and two security annotations for each expression. Each property is enforced by a strict separation of security levels. In the interest of an expressive language, we allow the information flow from lower to higher levels (confidentiality or integrity security levels) using promotion rules while the opposite needs direct interaction with the programmer by using the `declassify` expression. For convenience, we will use only one lattice of confidentiality security levels in the explanations.

The type modifier *mdf* can be `mut`, `imm`, `capsule`, and `read` with the following subtyping relation. For all type modifier *mdf* : `capsule` $\leq$ *mdf*, *mdf* $\leq$ `read`. In SIFO, objects are *mutable* or (deeply) *immutable*. The reachable object graph (ROG) from a mutable object is composed of mutable and immutable objects, while the ROG of an immutable object can only contain immutable objects. A `mut` reference must point to a mutable object; such an object can be aliased and mutated. An `imm` reference must point to an immutable object; such an object can be aliased, but not mutated. A `capsule` reference points to a mutable object. The object and the mutable objects in its ROG cannot be accessed from other references. As `capsule` is a subtype of `imm` and `mut` the object can be assigned to both. Finally, a `read` reference is the supertype that points to an object that cannot be aliased or mutated, but it has no immutability guarantee that the object is not modified by other references. These modifiers allow us to make precise decisions about the information flow by utilizing immutability or uniqueness properties of objects. For example, an immutable object cannot be altered, therefore it can be securely promoted to a higher security level. For a mutable object, a security promotion is insecure because an update through other references with lower security levels can corrupt the confidential information.

Additionally, the syntax of SIFO contains class definitions CD which can be classes or interfaces. An interface has a list of method headers. A class has additional fields. A field F has a type T and a name, but the type modifier can only be `mut` or `imm`. A method definition MD consists of a method header and a body. The header has a receiver, a return type, and a list of parameters. The parameters have a name and a type T. The receiver has a type modifier and a security level. An expression e can be a variable, field access, field assignment, method call, or object construction in SIFO. In the extended version presented in the paper, we also added abstract expressions, sequence of expressions, conditional expression, loop expression, and declassification. With the `declassify` operator a reverse information flow is allowed. The expression eA is abstract and typed by $[\Gamma; T]$. Beside the type T a local typing context Γ is used to have all needed information to further refine eA. We require a Boolean type for the guards in the conditional and loop expression. A typing context Γ assigns a type T_i to variable x_i. With the evaluation context $\mathcal{E}$, we define the order of evaluation for the reduction of the system. The typing rules of SIFO are shown in the report [24].

4.2 Refinement Rules for Program Construction

To formalize the IFbCOO refinement rules, in Figure 3, we introduce basic notations, which are used in the refinement rules.

L is the lattice of security levels to define the information flow policy and *lub* is used to calculate the least upper bound of a set of security levels. The functional and security specification of a program is defined by an IFbCOO tuple $\{P; Q; \Gamma; T; eA\}$. The IFbCOO tuple consists of a typing context Γ, a

L	Bounded upper semi-lattice $(L, \leq)$ of security levels
$lub : \mathcal{P}(L) \rightarrow L$	Least upper bound of the security levels in L
$\{P; Q; \Gamma; T; eA\}$	Starting IFbCOO tuple
$eA : [P; Q; \Gamma; T]$	Typed abstract expression eA
$\Gamma[mut(s)]$	Restricted typing context
$sec(T) = s$	Returns the security level s in type T

Fig. 3: Basic notations for IFbCOO

type T, an abstract expression eA, and a functional pre-/postcondition, which is declared in the first-order predicates P and Q. The abstract expression is typed by $[P; Q; \Gamma; T]$. In the following, we focus on security, so the functional specification is omitted.

The refinement process of IFbCOO starts with a method declaration, where the typing context Γ is extracted from the arguments and T is the method return type. Then, the user guides the construction process by refining the first abstract expression eA. With the notation $\Gamma[mut(s)]$, we introduce a restriction to the typing context. The function $mut(s)$ prevents mutation of mutable objects that have a security level lower than s. When the user chooses the lowest security level of the lattice, the function does not restrict Γ. The function $sec(T)$ extracts the security level of a type T.

Refinement Rules. The refinement rules are used to replace an IFbCOO tuple $\{\Gamma; T; eA\}$ with a concrete implementation by concretizing the abstract expression eA. This refinement is only correct if specific side conditions hold. On the right side of the rules, all newly introduced symbols are implicitly existentially quantified. The rules can introduce new abstract expressions eA_i which can be refined by further applying the refinement rules.

Refinement Rule 1 (Variable)
eA is refinable to x if $eA : [\Gamma; T]$ and $\Gamma(x) = T$.

The first IFbCOO rule introduces a variable x, which does not alter the program. It refines an abstract expression to an x if x has the correct type T.

Refinement Rule 2 (Field Assignment)
eA is refinable to $eA_0.f := eA_1$ if $eA : [\Gamma; T]$ and $eA_0 : [\Gamma; s_0\ \texttt{mut}\ C_0]$ and $eA_1 : [\Gamma; s_1\ mdf\ C]$ and $s\ mdf\ C\ f \in fields(C_0)$ and $s_1 = lub(s_0, s)$.

We can refine an abstract expression to a field assignment if the following conditions hold. The expression eA_0 has to be `mut` to allow a manipulation of the object. The security level of the assigned expression eA_1 has to be equal to the least upper bound of the security levels of expression eA_0 and the field f. The field f must be a field of the class C_0 with the type $s\ mdf\ C$. With the security promotion rule, the security level of the assigned expression can be altered.

Refinement Rule 3 (Field Access)
eA is refinable to $eA_0.f$ *if* $eA : [\Gamma; s\ mdf\ C]$ *and* $eA_0 : [\Gamma; s_0\ mdf_0\ C_0]$ *and* $s_1\ mdf_1\ C\ f \in fields(C_0)$ *and* $s = lub(s_0, s_1)$ *and* $mdf_0 \triangleright mdf_1 = mdf$*.*

We can refine an abstract expression to a field access if a field f exists in the class of receiver eA_0 with the type $s_1\ mdf_1\ C$. The accessed value must have the expected type $s\ mdf\ C$ of the abstract expression. This means, the class name of the field f and C must be the same. Additionally, the security level of the abstract expression eA is equal to the least upper bound of the security levels of expression eA_0 and field f. The type modifiers must also comply. The arrow between type modifiers is defined as follows. As we allow only `mut` and `imm` fields, not all possible cases are defined: $mdf \triangleright mdf' = mdf''$
- $\texttt{mut} \triangleright mdf = \texttt{capsule} \triangleright mdf = mdf$
- $\texttt{imm} \triangleright mdf = mdf \triangleright \texttt{imm} = \texttt{imm}$
- $\texttt{read} \triangleright \texttt{mut} = \texttt{read}$.

Refinement Rule 4 (Method Call)
eA is refinable to $eA_0.m(eA_1, \ldots, eA_n)$ *if* $eA : [\Gamma; T]$ *and* $eA_0 : [\Gamma; T_0] \ldots eA_n : [\Gamma; T_n]$ *and* $T_0 \ldots T_n \rightarrow T \in methTypes(class(T_0),\ m)$ *and* $sec(T) \geq sec(T_0)$ *and forall* $i \in \{1, \ldots, n\}$ *if* $mdf(T_i) \in \{\texttt{mut}, \texttt{capsule}\}$ *then* $sec(T_i) \geq sec(T_0)$*.*

With the method call rule, an abstract expression is refined to a call to method m. The method has a receiver eA_0, a list of parameters $eA_1 \ldots eA_n$, and a return value. A method with matching definition must exist in the class of receiver eA_0. This method definition is returned by the *methTypes* function. The function *class* returns the class of a type T. The security level of the return type has to be greater than or equal to the security level of the receiver. This condition is needed because through dynamic dispatch information of the receiver may be leaked if its security level is higher than the security level of the return type. The same applies for `mut` and `capsule` parameters. The security level of these parameters must also be greater than or equal to the security level of the receiver. As the method call replaces an abstract expression eA, the return value must have the same type (security level, type modifier, and class name) as the refined expression. In the technical report [24], we introduce multiple methods types [27] to reduce writing effort and increase the flexibility of IFbCOO. A method can be declared with specific types for receiver, parameters and return value, and other signatures of this method are deduced by applying the transformations from the multiple method types definition, where security level and type modifiers are altered. All these deduced method declarations can be used in the method call refinement rule.

Refinement Rule 5 (Constructor)
eA is refinable to $\texttt{new}\ s\ C(eA_1 \ldots eA_n)$ *if* $eA : [\Gamma; s\ mdf\ C]$ *and* $fields(C) = T_1\ f_1 \ldots T_n\ f_n$ *and* $eA_1 : [\Gamma; T_1[s]] \ldots eA_n : [\Gamma; T_n[s]]$*.*

The constructor rule is a special method call. We can refine an abstract expression to a constructor call, where a mutable object of class C is constructed with a security level s. The parameter list $eA_1 \dots eA_n$ must match the list of declared fields $f_1 \dots f_n$ in class C. Each parameter eA_i is assigned to field f_i. This assignment is allowed if the type of parameter eA_i is (a subtype of) $T_i[s]$. $T[s]$ is a helper function which returns a new type whose security level is the least upper bound of $sec(T)$ and s. It is defined as: $T[s] = lub(s, s')\ mdf\ C$, where $T = s'\ mdf\ C$, defined only if $s' \leq s$ or $s \leq s'$. By calling a constructor, the security level s can be freely chosen to use parameters with security levels that are higher than originally declared for the fields. In other words, a security level s is used to initialize lower security fields with parameters of higher security level s. This results in a newly created object with the security level s [27]. As the newly created object replaces an abstract expression eA, the object must have the same type as the abstract expression. If the modifier promotion rule is used (i.e., no mutable input value exist), the object can be assigned to a `capsule` or `imm` reference.

Refinement Rule 6 (Composition)
eA is refinable to $eA_0; eA_1$ if $eA : [\Gamma; T]$ and $eA_0 : [\Gamma; T_0]$ and $eA_1 : [\Gamma; T]$.

With the composition rule, an abstract expression eA is refined to two subsequent abstract expression eA_0 and eA_1. The second abstract expression must have the same type T as the refined expression.

Refinement Rule 7 (Selection)
eA is refinable to `if` *eA_0* `then` *eA_1* `else` *eA_2 if $eA : [\Gamma; T]$ and $eA_0 : [\Gamma; s$* `imm Boolean`*$]$ and $eA_1 : [\Gamma[mut(s)]; T]$ and $eA_2 : [\Gamma[mut(s)]; T]$.*

The selection rule refines an abstract expression to a conditional `if-then-else`-expression. Secure information can be leaked indirectly as the selected branch may reveal the value of the guard. In the branches, the typing context is restricted. The restricted typing context prevents updating mutable objects with a security level lower than s. The security level s is determined by the Boolean guard eA_0. When we add updatable local variables to our language, the selection rule must also prevent the update of local variables that have a security level lower than s.

Refinement Rule 8 (Repetition)
eA is refinable to `while` *eA_0* `do` *eA_1 if $eA : [\Gamma; T]$ and $eA_0 : [\Gamma; s$* `imm Boolean`*$]$ and $eA_1 : [\Gamma[mut(s)]; T]$.*

The repetition rule refines an abstract expression to a `while`-loop. The repetition rule is similar to the selection rule. For the loop body, the typing context is restricted to prevent indirect leaks of the guard in the loop body. The security level s is determined by the Boolean guard eA_0.

Refinement Rule 9 (Context Rule)
$\mathcal{E}[eA]$ is refinable to $\mathcal{E}[e]$ if eA is refinable to e.

The context rule replaces in a context $\mathcal{E}$ an abstract expression with a concrete expression, if the abstract expression is refinable to the concrete expression.

Refinement Rule 10 (Subsumption Rule)
$eA : [\Gamma; T]$ is refinable to $eA_1 : [\Gamma; T']$ if $T' \leq T$.

The subsumption rule can alter the type of expressions. An abstract expression that requires a type T can be weakened to require a type T' if the type T' is a subtype of T.

Refinement Rule 11 (Security Promotion)
$eA : [\Gamma; s\ mdf\ C]$ is refinable to $eA_1 : [\Gamma; s'\ mdf\ C]$ if $mdf \in \{\texttt{capsule}, \texttt{imm}\}$ and $s' \leq s$.

The security promotion rule can alter the security level of expressions. An abstract expression that requires a security level s can be weakened to require a security level s' if the expression is `capsule` or `imm`. Other expressions (`mut` or `read`) cannot be altered because potentially existing aliases are a security hazard.

Refinement Rule 12 (Modifier Promotion)
$eA : [\Gamma; s\ \texttt{capsule}\ C]$ is refinable to $eA_1 : [\Gamma[\texttt{mut}\backslash\texttt{read}]; s\ \texttt{mut}\ C]$.

The modifier promotion rule can alter the type modifier of an expression eA. An abstract expression that requires a `capsule` type modifier can be weakened to require a `mut` type modifier if all `mut` references are only seen as `read` in the typing context. That means, that the mutable objects in the ROG of the expression cannot be accessed by other references. Thus, manipulation of the object is only possible through the reference on eA.

Refinement Rule 13 (Declassification)
$eA : [\Gamma; \bot\ mdf\ C]$ is refinable to $\texttt{declassify}(eA_1) : [\Gamma; s\ mdf\ C]$ if $mdf \in \{\texttt{capsule}, \texttt{imm}\}$.

In our information flow policy, we can never assign an expression with a higher security level to a variable with a lower security level. To allow this assignment in appropriate cases, the `declassify` rule is used. An expression eA is altered to a `declassify`-expression with an abstract expression eA_1 that has a security level s if the type modifier is `capsule` or `imm`. A `mut` or `read` expression cannot be declassified as existing aliases are a security hazard. Since we have the security promotion rule, the declassified `capsule` or `imm` expression can directly be promoted to any higher security level. Therefore, it is sufficient to use the bottom security level in this rule without restricting the expressiveness. For example, the rule can be used to assign a hashed password to a public variable. The programmer has the responsibility to ensure that the use of `declassify` is secure.

4.3 Proof of Soundness

In the technical report, we prove that programs constructed with the IFbCOO refinement rules are secure according to the defined information flow policy. We prove this by showing that programs constructed with IFbCOO are well typed in SIFO (Theorem 1). SIFO itself is proven to be secure [27]. In the technical report [24], we prove this property for the core language of SIFO, which does not contain composition, selection, and repetition expressions. The SIFO core language is minimal, but using well-known encodings, it can support composition, selection, and repetition (encodings of the Smalltalk [14] style support control structures). We also exclude the declassify operation because this rule is an explicit mechanism to break security in a controlled way.

Theorem 1 (Soundness of IFbCOO).
An expression e constructed with IFbCOO is well typed in SIFO.

5 CorC Tool Support and Evaluation

IFbCOO is implemented in the tool CorC [26, 12]. CorC itself is a hybrid textual and graphical editor to develop programs with correctness-by-construction. IFbC [25] is already implemented as extension of CorC, but to support object-orientation with IFbCOO a redesign was necessary. Source code and case studies are available at: https://github.com/TUBS-ISF/CorC/tree/CCorCOO.

5.1 CorC for IFbCOO

For space reasons, we cannot introduce CorC comprehensively. We just summarize the features of CorC to check IFbCOO information flow policies:

- Programs are written in a tree structure of refining IFbCOO tuples (see Figure 1). Besides the functional specification, variables are labeled with a type T in the tuples.
- Each IFbCOO refinement rule is implemented in CorC. Consequently, functional correctness and security can be constructed simultaneously.
- The information flow checks according to the refinement rules are executed automatically after each refinement.
- Each CorC-program is uniquely mapped to a method in a SIFO class. A SIFO class contains methods and fields that are annotated with security labels and type modifiers.
- A properties view shows the type T of each used variable in an IFbCOO tuple. Violations of the information flow policy are explained in the view.

5.2 Case Studies and Discussion

The implementation of IFbCOO in the tool CorC enables us to evaluate the feasibility of the security mechanism by successfully implementing three case studies [16, 32] from the literature and a novel one in CorC. The case studies are also implemented and type-checked in SIFO to confirm that the case studies are secure. The newly developed *Database* case study represents a secure system that

Name	**#Security Levels**	**#Classes**	**#Lines of Code**	**#Methods in CorC**
Database	4	6	156	2
Email [16]	2	9	807	15
Banking [32]	2	3	243	6
Paycard	2	3	244	5

Table 1: Metrics of the case studies

strictly separates databases of different security levels. *Email* [16] ensures that encrypted emails cannot be decrypted by recipients without the matching key. *Paycard* (`http://spl2go.cs.ovgu.de/projects/57`) and *Banking* [32] simulate secure money transfer without leaking customer data. The *Database* case study uses four security levels, while the others (*Email*, *Banking*, and *Paycard*) use two.

As shown in Table 1, the cases studies comprise three to nine classes with 156 to 807 lines of code each. 28 Methods that exceed the complexity of getter and setter are implemented in CorC. It should be noted that we do not have to implement every method in CorC. If only `low` input data is used to compute `low` output, the method is intrinsically secure. For example, three classes in the Database case study are implemented with only `low` security levels. Only the class `GUI` and the main method of the case study, which calls the `low` methods with higher security levels (using multiple method types) is then correctly implemented in CorC. The correct and secure promotion of security levels of methods called in the main method is confirmed by CorC.

Discussion and Applicability of IFbCOO. We emphasize that CbC and also IFbCOO should be used to implement correctness- and security-critical programs [18]. The scope of this work is to demonstrate the feasibility of the incremental construction of correctness- and security-critical programs. We argue that we achieve this goal by implementing four case studies in CorC.

The constructive nature of IFbCOO is an advantage in the secure creation of programs. Instead of writing complete methods to allow a static analyzer to accept/reject the method, with IFbCOO, we directly design and construct secure methods. We get feedback during each refinement step, and we can observe the status of all accessible variables at any time of the method. For example, we received direct feedback when we manipulated a `low` object in the body of a `high` then-branch. With this information, we could adjust the code to ensure security. As IFbCOO extends CorC, functional correctness is also guaranteed at the same time. This is beneficial as a program, which is security-critical, should also be functionally correct. As IFbCOO is based on SIFO, programs written with any of the two approaches can be used interchangeably. This allows developers to use their preferred environment to develop new systems, re-engineer their systems, or integrate secure software into existing systems. These benefits of IFbCOO are of course connected with functional and security specification effort, and the strict refinement-based construction of programs.

6 Related Work

In this section, we compare IFbCOO to IFbC [25, 29] and other Hoare-style logics for information flow control. We also discuss information flow type systems and correctness-by-construction [18] for functional correctness.

IFbCOO extends IFbC [25] by introducing object-orientation and type modifiers. IFbC is based on a simple while language. As explained in Section 4, the language of IFbCOO includes objects and type modifiers. Therefore, the refinement rules of IFbC are revised to handle secure information flow with objects. The object-orientation complicates the reasoning of secure assignments because objects could be altered through references with different security levels. If private information is introduced, an already public reference could read this information. SIFO and therefore IFbCOO consider these cases and prevent information leaks by considering immutability and encapsulation and only allowing secure aliases.

Previous work using Hoare-style program logics with information flow control analyzes programs after construction, rather than guaranteeing security during construction. Andrews and Reitman [5] encode information flow directly in a logical form. They also support parallel programs. Amtoft and Banerjee [3] use Hoare-style program logics and abstract interpretations to detect information flow leaks. They can give error explanations based on strongest postcondition calculation. The work of Amtoft and Banerjee [3] is used in SPARK Ada [4] to specify and check the information flow.

Type system for information flow control are widely used, we refer to Sabelfeld and Myers [28] for a comprehensive overview. We only discuss closely related type systems for object-oriented languages [11, 9, 31, 20, 30, 10]. Banerjee et al. [9] introduced a type system for a Java-like language with only two security levels. We extend this by operating on any lattice of security levels. We also introduce type modifiers to simplify reasoning in cases where objects cannot be mutated or are encapsulated. Jif [20] is a type system to check information flow in Java. One main difference is in the treatment of aliases: Jif does not have an alias analysis to reason about limited side effects. Therefore, Jif pessimistically discards programs that introduce aliases because Jif has no option to state immutable or encapsulated objects. IFbCOO allows the introduction of secure aliases.

In the area of correctness-by-construction, Morgan [19] and Back [8] propose refinement-based approaches which refine functional specifications to concrete implementations. Beside of pre-/postcondition specification, Back also uses invariants as starting point. Morgan's calculus is implemented in ArcAngel [22] with the verifier ProofPower [33], and SOCOS [6, 7] implements Back's approach. In comparison to IFbCOO, those approaches do not reason about information flow security. Other refinement-based approaches are Event-B [1, 2] for automata-based systems and Circus [21, 23] for state-rich reactive systems. These approaches have a higher abstraction level, as they operate on abstract machines instead of source code. Hall and Chapman [15] introduced with CbyC another related approach that uses formal modeling techniques to analyze the development during all stages (architectural design, detailed design, code) to eliminate defects early. IFbCOO is tailored to source code and does not consider other development phases.

7 Conclusion

In this paper, we present IFbCOO, which establishes an incremental refinement-based approach for functionally correct and secure programs. With IFbCOO programs are constructed stepwise to comply at all time with the security policy. The local check of each refinement can reduce debugging effort, since the user is not warned only after the implementation of a whole method. We formalized IFbCOO by introducing 13 refinement rules and proved soundness by showing that constructed programs are well-typed in SIFO. We also implemented IFbCOO in CorC and evaluated our implementation with a feasibility study. One future direction is the conduction of comprehensive user studies for user-friendly improvements which is only now possible due to our sophisticated tool CorC.

Acknowledgments This work was supported by KASTEL Security Research Labs.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer **12**(6), 447–466 (2010)
3. Amtoft, T., Banerjee, A.: Information Flow Analysis in Logical Form. In: International Static Analysis Symposium. LNCS, vol. 3148, pp. 100–115. Springer (2004)
4. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., Greve, D.A.: Specification and Checking of Software Contracts for Conditional Information Flow. In: International Symposium on Formal Methods. pp. 229–245. Springer (2008)
5. Andrews, G.R., Reitman, R.P.: An Axiomatic Approach to Information Flow in Programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **2**(1), 56–76 (1980)
6. Back, R.J.: Invariant Based Programming: Basic Approach and Teaching Experiences. Formal Aspects of Computing **21**(3), 227–244 (2009)
7. Back, R.J., Eriksson, J., Myreen, M.: Testing and Verifying Invariant Based Programs in the SOCOS Environment. In: International Conference on Tests and Proofs (TAP). LNCS, vol. 4454, pp. 61–78. Springer (2007)
8. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer Science & Business Media (2012)
9. Banerjee, A., Naumann, D.A.: Secure Information Flow and Pointer Confinement in a Java-like Language. In: Computer Security Foundations Workshop. vol. 2, p. 253 (2002)
10. Barthe, G., Pichardie, D., Rezk, T.: A Certified Lightweight Non-Interference Java Bytecode Verifier. In: European Symposium on Programming. LNCS, vol. 4421, pp. 125–140. Springer (2007)
11. Barthe, G., Serpette, B.P.: Partial Evaluation and Non-Interference for Object Calculi. In: International Symposium on Functional and Logic Programming. LNCS, vol. 1722, pp. 53–67. Springer (1999)
12. Bordis, T., Cleophas, L., Kittelmann, A., Runge, T., Schaefer, I., Watson, B.W.: Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY. In: The Logic of Software. A Tasting Menu of Formal Methods. Springer (2022)

13. Giannini, P., Servetto, M., Zucca, E., Cone, J.: Flexible Recovery of Uniqueness and Immutability. Theoretical Computer Science **764**, 145–172 (2019)
14. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman Publishing Co., Inc. (1983)
15. Hall, A., Chapman, R.: Correctness by Construction: Developing a Commercial Secure System. IEEE Software **19**(1), 18–25 (2002)
16. Hall, R.J.: Fundamental Nonmodularity in Electronic Mail. Automated Software Engineering **12**(1), 41–79 (2005)
17. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems (TOPLAS) **23**(3), 396–450 (2001)
18. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer Science & Business Media (2012)
19. Morgan, C.: Programming from Specifications. Prentice Hall, 2nd edn. (1994)
20. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 228–241. ACM (1999)
21. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP Semantics for Circus. Formal Aspects of Computing **21**(1), 3–32 (2009)
22. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: A Tactic Language for Refinement. Formal Aspects of Computing **15**(1), 28–47 (2003)
23. Oliveira, M.V.M., Gurgel, A.C., Castro, C.G.: CRefine: Support for the Circus Refinement Calculus. In: 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods. pp. 281–290. IEEE (Nov 2008)
24. Runge, T., Kittelmann, A., Servetto, M., Potanin, A., Schaefer, I.: Information Flow Control-by-Construction for an Object-Oriented Language Using Type Modifiers (2022), https://arxiv.org/abs/2208.02672
25. Runge, T., Knüppel, A., Thüm, T., Schaefer, I.: Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In: Proceedings of the 8th International Conference on Formal Methods in Software Engineering (2020)
26. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool Support for Correctness-by-Construction. In: International Conference on Fundamental Approaches to Software Engineering. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019)
27. Runge, T., Servetto, M., Potanin, A., Schaefer, I.: Immutability and Encapsulation for Sound OO Information Flow Control (2022), Under Review
28. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
29. Schaefer, I., Runge, T., Knüppel, A., Cleophas, L., Kourie, D., Watson, B.W.: Towards Confidentiality-by-Construction. In: International Symposium on Leveraging Applications of Formal Methods. LNCS, vol. 11244, pp. 502–515. Springer (2018)
30. Strecker, M.: Formal Analysis of an Information Flow Type System for MicroJava. Technische Universität München, Tech. Rep (2003)
31. Sun, Q., Banerjee, A., Naumann, D.A.: Modular and Constraint-Based Information Flow Inference for an Object-Oriented Language. In: International Static Analysis Symposium. LNCS, vol. 3148, pp. 84–99. Springer (2004)
32. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based Deductive Verification of Software Product Lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering. pp. 11–20 (2012)
33. Zeyda, F., Oliveira, M., Cavalcanti, A.: Supporting ArcAngel in ProofPower. Electronic Notes in Theoretical Computer Science **259**, 225–243 (2009)