Lattice-Based Information Flow Control-by-Construction for Security-by-Design

Tobias Runge
TU Braunschweig
Germany
tobias.runge@tu-bs.de

Alexander Knüppel
TU Braunschweig
Germany
a.knueppel@tu-bs.de

Thomas Thüm
University of Ulm
Germany
thomas.thuem@uni-ulm.de

Ina Schaefer
TU Braunschweig
Germany
i.schaefer@tu-bs.de

ABSTRACT

Many software applications contain confidential information, which has to be prevented from leaking through unauthorized access. To enforce confidentiality, there are language-based security mechanisms that rely on information flow control. Typically, these mechanisms work post-hoc by checking whether confidential data is accessed unauthorizedly after the complete program is written. The disadvantage is that incomplete programs cannot be interpreted properly and information flow properties cannot be built in constructively. In this work, we present a methodology to construct programs incrementally using refinement rules to follow a lattice-based information flow policy. In every refinement step, confidentiality and functional correctness of the program is guaranteed, such that insecure programs are prohibited by construction. Our contribution is fourfold. We formalize refinement rules for the constructive information flow control methodology, prove soundness of the refinement rules, show that our approach is at least as expressive as standard language-based mechanisms for information flow, and implement it in a graphical editor called CorC. Our methodology is also usable for integrity properties, which are dual to confidentiality.

KEYWORDS

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

ACM Reference Format:
Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In *8th International Conference on Formal Methods in Software Engineering (FormaliSE '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3372020.3391565

1 INTRODUCTION

Today, customers have a high demand for secure software. An important security property of data is *confidentiality*, which means that no confidential information is leaked to unauthorized or external systems. Another important property is *integrity* to ensure that critical software is functionally correct and is not influenced by other untrusted software parts. To improve the process of developing secure software, security-by-design techniques have been proposed. These techniques provide guidelines for the overall development process to design and implement secure software. For example, a well-known process is the *Security Development Lifecycle* (SDL) by Microsoft [16]. At implementation level, SDL relies on post-hoc program analysis techniques (i.e., techniques applied after the creation of the program) to ensure confidentiality and integrity [13].

The information flow between variables on source code level is mostly analyzed with language-based static analysis techniques [28]. Such techniques specify security policies to determine the permitted information flow between variables in the program. For example, we may define a policy with two confidentiality levels, high and low, arranged in a lattice where variables are categorized into either of the two. To prevent information leaks, the lattice-based information flow policy prohibits a flow from high to low variables. The same applies for trusted and untrusted variables with a policy that prohibits an information flow from untrusted to trusted variables (i.e., to preserve integrity). As shown by Biba [12], integrity can be seen as a dual to confidentiality, which means that either of them can be checked with the same information flow analysis techniques. Standard information flow analyses are based on security types systems [28, 31]. Such a type system assigns to every variable and expression an explicit security type. A set of typing rules describes the allowed information flow and discards programs that violate the security policy.

In contrast to post-hoc analyses that cannot ensure information flow properties during program construction, but only check programs after their creation, we propose to develop programs that are secure by construction analogous to the correctness-by-construction (CbC) approach for functional correctness [18]. Guided by a pre-/postcondition specification, an abstract program is refined stepwise to a concrete implementation. By applying a sound set of refinement rules, the resulting program is correct by construction. In this paper, we propose Information Flow Control-by-Construction (IFbC) to create functionally correct programs, which also satisfy a lattice-based information flow policy for capturing confidentiality and integrity. The information flow policy can be specified in any bounded upper semi-lattice (i.e., security levels are arranged in a partially ordered set representing the allowed direction of information flow).

In every step of the program construction in IFbC, the security levels associated with variables are updated according to our refinement rules, and therefore prevent a violation of the information flow policy. Furthermore, the current status of all variables can be observed in (partial) programs after each refinement step. To give programmers more flexibility while constructing a program, we allow to reverse the information flow in appropriate cases. We introduce a *declassify* operation, which can be used if the programmer encrypts or otherwise disguises the confidential information. As the refinement rules also take functional correctness into account, programmers using our methodology create programs that meet two properties, namely functional correctness and security.

In this paper, we demonstrate the strengths of a constructive methodology to develop secure and correct programs. We give two examples to emphasize the advantage of ensuring confidentiality and integrity throughout the development process, rather than having to check this property afterwards. With a sound set of refinement rules, developers can never construct an insecure program, contrary to security type systems that only discard insecure programs. Therefore, IFbC can reduce the post-hoc analysis effort or even make it obsolete, as developers are guided by constructive rules to an already secure program [32]. The IFbC approach contributes to the security-by-design paradigm to close the gap of a constructive process at implementation level. It can be used supplementary to existing processes and analyses for security-critical programs during development.

IFbC presented in this paper extends C14bC by Schaefer et al. [29]. C14bC uses a confidentiality specification with only two levels, high and low, and refinement rules to ensure the confidentiality of programs written in a simple while-language without method calls. Moreover, Schaefer et al. [29] discussed potential tool support for this approach. Finally, we list the four contributions of this work.

- We create an IFbC methodology to construct functionally correct and secure programs regarding a lattice-based confidentiality and integrity policy. Confidentiality, integrity, and functional correctness are ensured simultaneously while constructing the program. The underlying language of IFbC is also extended by method calls to support more meaningful programs.
- We prove the soundness of the proposed refinement rules, such that a program constructed by IFbC never violates our information flow policy.
- We show that IFbC is at least as expressive as a type system for lattice-based information flow control to justify that IFbC can be used supplementary in a program development process.
- We implement the IFbC methodology in a tool called CorC and discuss applicability of our approach.

$$
\begin{array}{lll}
\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\} & \textit{can be refined to} & \\
\textit{Skip}: & \{\mathtt{P}\}\ \textit{skip}\ \{\mathtt{Q}\}\ \textit{iff}\ \mathtt{P}\ \textit{implies}\ \mathtt{Q} & (1)\\
\textit{Assignment}: & \{\mathtt{P}\}\ x := E\ \{\mathtt{Q}\}\ \textit{iff}\ \mathtt{P}\ \textit{implies}\ \mathtt{Q}[\mathtt{x} := \mathtt{E}] & (2)\\
\textit{Composition}: & \{\mathtt{P}\}\ \mathtt{S1}\ ;\ \mathtt{S2}\ \{\mathtt{Q}\}\ \textit{iff there is}\ \mathtt{M}\ \textit{such that} & (3)\\
 & \{\mathtt{P}\}\ \mathtt{S1}\ \{\mathtt{M}\}\ \textit{and}\ \{\mathtt{M}\}\ \mathtt{S2}\ \{\mathtt{Q}\} & \\
\textit{Selection}: & \{\mathtt{P}\}\ \mathbf{if}\ \mathtt{G}\ \mathbf{then}\ \mathtt{S1}\ \mathtt{else}\ \mathtt{S2}\ \mathbf{fi}\ \{\mathtt{Q}\}\ \textit{iff} & (4)\\
 & \{\mathtt{P} \land \mathtt{G}\}\ \mathtt{S1}\ \{\mathtt{Q}\}\ \textit{and}\{\mathtt{P} \land \neg\mathtt{G}\}\ \mathtt{S2}\ \{\mathtt{Q}\} & \\
\textit{Repetition}: & \{\mathtt{P}\}\ \mathbf{do}\ \mathtt{G} \longrightarrow\ \mathtt{S}\ \mathbf{od}\ \{\mathtt{Q}\}\ \textit{iff there is an} & (5)\\
 & \textit{invariant}\ \mathtt{I}\ \textit{and a variant}\ \mathtt{V}\ \textit{such that} & \\
 & (\mathtt{P}\ \textit{implies}\ \mathtt{I})\ \textit{and}\ (\mathtt{I} \land \neg\mathtt{G}\ \textit{implies}\ \mathtt{Q}) & \\
 & \textit{and}\ \{\mathtt{I} \land \mathtt{G}\}\ \mathtt{S}\ \{\mathtt{I}\}\ \textit{and} & \\
 & \{\mathtt{I} \land \mathtt{G} \land \mathtt{V} = \mathtt{V_0}\}\ \mathtt{S}\ \{\mathtt{I} \land 0 \leq \mathtt{V} < \mathtt{V_0}\} & \\
\textit{Weaken pre}: & \{\mathtt{P'}\}\ \mathtt{S}\ \{\mathtt{Q}\}\ \textit{iff}\ \mathtt{P}\ \textit{implies}\ \mathtt{P'} & (6)\\
\textit{Strengthen post}: & \{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q'}\}\ \textit{iff}\ \mathtt{Q'}\ \textit{implies}\ \mathtt{Q} & (7)\\
\textit{Method call}: & \{\mathtt{P}\}\ \mathtt{M}(\mathtt{a_1} \ldots \mathtt{a_n})\ \{\mathtt{Q}\}\ \textit{for a method} & (8)\\
 & \{\mathtt{P'}\}\ \mathtt{M}(\mathtt{z_1} \ldots \mathtt{z_n})\ \{\mathtt{Q'}\}\ \textit{iff}\ \mathtt{P} = \mathtt{P'}[\mathtt{z_i} \backslash \mathtt{a_i}] & \\
 & \textit{and}\ \mathtt{Q} = \mathtt{Q'}[\mathtt{z_i^{old}}, \mathtt{z_i} \backslash \mathtt{a_i^{old}}, \mathtt{a_i}] &
\end{array}
$$

Figure 1: Correctness-by-construction refinement rules [18]

2 FOUNDATIONS

In this section, we provide the background on correctness-by-construction and information flow in order to introduce IFbC in the subsequent section. We also introduce lattices as underlying mathematical structure for lattice-based information flow policies.

2.1 Functional Correctness-by-Construction

Correctness-by-construction (CbC) [18] is an approach to construct programs guided by a pre-/postcondition specification. CbC starts with an abstract Hoare triple {P} S {Q} consisting of a precondition P, an abstract statement S, and a postcondition Q. This triple is successively refined using a set of refinement rules to a concrete implementation, which satisfies the specification. For simplicity in this paper, we consider the guarded command language introduced by Dijkstra [15]. Each of the refinement rules takes an abstract statement and replaces it with a more concrete guarded command language statement. Every refinement rule preserves the correctness of the program if a discharged side condition is proven correct [25].

The eight considered refinement rules are shown in Fig. 1. As concrete instructions, we have *skip*, *assignment*, and *method call* with call by value-result. *Composition* is used for a sequence of statements. *Selection* and *repetition* are used for the control flow of the program. In Fig. 1, the side conditions for applicability of a refinement rule are shown. For example, when refining to a method call, it has to be proven that the pre-/postcondition specification of the refined triple is equal to the specification of the called method. This refinement rule also requires that the parameters of the method are correctly passed where $\mathtt{a_i}$ are the actual parameters and $\mathtt{z_i}$ are the formal parameters. The parameters with superscript old refer to parameters before the execution of the method.

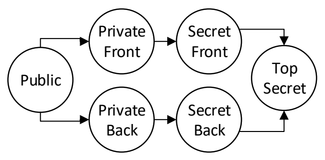


Figure 2: Example of a lattice of confidentiality levels

2.2 Information Flow Control

Information flow control mechanisms [28, 31] are used to specify programs with respect to a security policy. The policy can establish confidentiality of processed information to prevent leaks of unauthorized information, or it can guarantee integrity of the processed information by ensuring that trusted parts are not influenced by untrusted parts. Both properties can be analyzed by considering the information flow, as confidentiality can be modeled as dual to integrity [12, 28]. Confidentiality requires that information flow to specific destinations is prevented. Similarly, integrity requires that a flow from specific sources is prevented to ensure that the system is not harmed by untrusted sources. Integrity also requires a functionally correct program because incorrect methods can violate the integrity by computing wrong data. Correctness can be achieved with the presented CbC approach of Section 2.1.

To give an example of a security policy for confidentiality, we consider a company with two different departments. A front office that should know personal information of customers (e.g., their name and age) and a back office that should know critical financial information of customers. The back office does not know other personal information to make unbiased decisions. The front office on the other hand should treat the customers without being influenced by their financial status. To establish this policy, a lattice as in Fig. 2 can be used. This lattice also includes `Public` data for general access, and `Top Secret` data for access by the management. The front and back office are also divided into two levels, `Private` and `Secret`, for data with different confidentiality levels. The arrows in the graph show the allowed flow directions.

Lattice. Bell, LaPadula [11], and Denning [14] were the first to arrange confidentiality levels in a lattice. This arrangement of confidentiality levels in our example fulfills the requirements of a bounded upper semi-lattice. A lattice is a structure $\langle L, \leq, lub, \top, \bot \rangle$ where L is a set of levels and $\leq$ is a partial order (e.g., `Public` $\leq$ `Private Front`). The relation operator is reflexive, antisymmetric, and transitive, but per definition not every pair of elements need to be comparable. An upper bound in the lattice is defined as follows: for a set of elements $X \subseteq L$, an upper bound y exists if $\forall x \in X : x \leq y$. The element u is the least upper bound ($lub : \mathcal{P}(X) \rightarrow X$), of all $x \in X$ if $u \leq y$ for all upper bounds y. We restrict the lattice to be a bounded upper semi-lattice, which has the greatest element $\top$ and the least element $\bot$, (i.e., $\bot \leq a \leq \top$ for every $a \in L$). For every combination of levels, a unique least upper bound (lub) must exist. The lub is used to calculate the least security level such that violations of the information flow policy are prevented (e.g., that no financial information flows to a member of the front office).

We distinguish between two information flow types. Information can flow *directly* through an assignment statement, which assigns data to another variable. Here, we have to ensure that the assigned variable gets a security level of at least the *lub* of all variables used in the expression to prevent a leak. Information can also flow *indirectly* through conditional or loop statements. If confidential data is used in a guard of a conditional statement, the chosen branch gives information about the variables in the guard. Therefore, the confidentiality level in the branches must be the least upper bound of the levels in the guards, too.

$$(1) \vdash \mathsf{x} : \tau\ \mathsf{var} \qquad (2)\ \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'} \qquad (3)\ \frac{\vdash \mathsf{p} : \rho \quad \vdash \rho \subseteq \rho'}{\vdash \mathsf{p} : \rho'}$$

$$(4)\ \frac{\vdash \tau \subseteq \tau'}{\vdash \tau'\ cmd \subseteq \tau\ cmd} \qquad (5)\ \frac{\vdash \mathsf{x} : \tau\ \mathsf{var} \quad \vdash \mathsf{e}' : \tau}{\vdash \mathsf{x} := \mathsf{e}' : \tau\ cmd}$$

$$(6)\ \frac{\vdash \mathsf{c} : \tau\ cmd \quad \vdash \mathsf{c}' : \tau\ cmd}{\vdash \mathsf{c}; \mathsf{c}' : \tau\ cmd} \qquad (7)\ \frac{\vdash \mathsf{e} : \tau \quad \vdash \mathsf{c} : \tau\ cmd}{\vdash \textbf{while}\ \mathsf{e}\ \textbf{do}\ \mathsf{c} : \tau\ cmd}$$

$$(8)\ \frac{\vdash \mathsf{e} : \tau \quad \vdash \mathsf{c} : \tau\ cmd \quad \vdash \mathsf{c}' : \tau\ cmd}{\vdash \textbf{if}\ \mathsf{e}\ \textbf{then}\ \mathsf{c}\ \textbf{else}\ \mathsf{c}' : \tau\ cmd}$$

Figure 3: Security type system [31]

Security Type System. A security type system [28] ensures the compliance of a program with an information flow policy. A set of typing rules determine the allowed information flow and discard programs, which violate the security policy. An excerpt of a type system by Volpano et al. [31] is shown in Fig. 3. Here, we have security levels τ that are arranged in a lattice $L\ \langle L, \leq \rangle$ with $\tau \in L$. The language consists of statements c that are typed with $\tau\ cmd$, expressions e that are typed with a security level τ, and variables x that are typed with $\tau\ var$ (Rule 1). The typing of expressions should prevent a leak through direct information flow, and the typing of statements is used for the indirect information flow. Variables are expressions. Expressions and statements are both phrases p. The different types $\tau\ cmd$, $\tau\ var$, and τ are all phrase types ρ. The partial order of confidentiality levels ($\leq$) is extended to a subtype relation $\subseteq$ (rules 2–4) to use subtyping in the other typing rules 5–8. Typing Rule 5 shows a secure assignment. To assign expression e′ to x, both expressions must agree on their security level. Through subtyping (2–4), an assignment from a lower to a higher security level is allowed. The rules 6–8 describe the standard program flow constructs for sequence, conditional, and repetition. Here, the security levels of the commands c, c′, and guards e have to be equal or subtyping has to be used.

3 INFORMATION FLOW CONTROL-BY-CONSTRUCTION

To motivate the IFbC approach, we give two examples. The first example creates a confidential program, and the second example uses an information flow policy to ensure integrity of a program.

Auction Example for Confidentiality. To illustrate IFbC for confidentiality, we construct a program for an auction. The input is an array of bids for an item, and the goal is to find the maximum bid that wins the auction. The array of bids is traversed to find this maximum, which is published. We assume three security levels `public`, `private`, and `secret` with `public < private < secret` and each variable is labeled with one of these security levels.

```
pre: publishBid = 0
post: \forall int x; ((x >= 0
  & x < bids.length)
  -> (publishBid >= bids[x]))
void auction(private int[] bids,
  public int publishBid) {
  public int i = 0;
  secret int highestBid = 0;
  do (i < bids.length) {
    if (highestBid < bids[i]) {
      highestBid = bids[i];
    else {
      skip
    } fi
    i = i + 1;
  } od
  publishBid = declassify(highestBid);
}
```

Listing 1: Program of the auction example

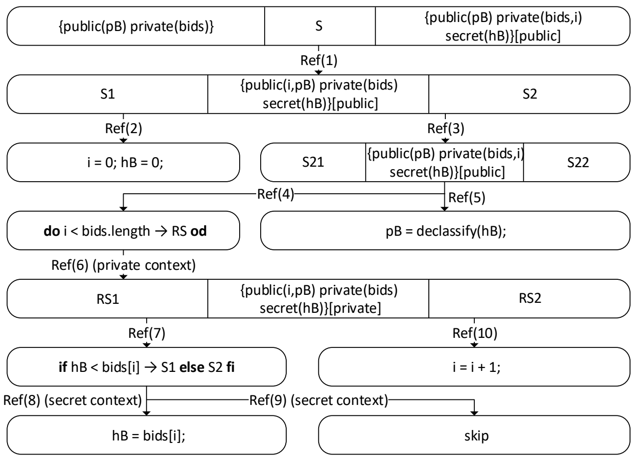

{public(pB) private(bids)}
S
{public(pB) private(bids,i) secret(hB)}[public]
Ref(1)
S1
{public(i,pB) private(bids) secret(hB)}[public]
S2
Ref(2)
Ref(3)
i = 0; hB = 0;
S21
{public(pB) private(bids,i) secret(hB)}[public]
S22
Ref(4)
Ref(5)
do i < bids.length → RS od
pB = declassify(hB);
Ref(6) (private context)
RS1
{public(i,pB) private(bids) secret(hB)}[private]
RS2
Ref(7)
Ref(10)
if hB < bids[i] → S1 else S2 fi
i = i + 1;
Ref(8) (secret context)
Ref(9) (secret context)
hB = bids[i];
skip

Figure 4: Refinement steps for the auction example

The auction method is specified such that it gets as input a `private` array `bids` and a `public` variable `publishBid` (`pB`). The method sets `pB` to the maximum bid of the auction. In IFbC, parameters are passed by value-result. The security levels of other local variables used in the code are not specified yet. If needed, programmers can add additional variables with an initially chosen security level while constructing the program, where the resulting security level of the variables can be changed in the program to prevent leaks. Additionally, a functional specification of the program can be given to construct a functionally correct program. The refinement rules of Fig. 1 are used to guarantee the functional correctness. Simultaneously, refinement rules of IFbC are used to ensure the specified confidentiality policy.

To construct the program with IFbC, we start with a provided IFbC triple $\{\mathcal{V}^{pre}, \mathtt{P}\}\ \mathtt{S}\ \{\mathcal{V}^{post}, \mathtt{Q}\}[\eta]$. This specification indicates the security levels of variables before (labeling function $\mathcal{V}^{pre}$) and after ($\mathcal{V}^{post}$) program execution. An instance would be the specification of the auction problem as above: $\mathcal{V}^{pre}, \mathcal{V}^{post}$:= `public pB`, `private bids`. The security context η is used to reason about indirect information flow. It tracks the security level of guards used in conditional or loop statements. Furthermore, the triple includes the abstract statement `S` that is refined to a concrete program. The functional specification is provided as logical precondition `P` and postcondition `Q` (cf. `pre` and `post` in Listing 1). In the following, we construct the program and refer to the functional refinement rules that are applied. By refining the program, we can also guarantee that the security specification is met by construction.

In Fig. 4, we show the refinement steps for the auction example in a graphical notation. Here, we omit the functional specification to focus on the information flow. The postcondition contains the `public` variable `publishBid` (indicated by the predicate `public(pB)` in the graphic), the `private` variables `bids` and `i` (a control variable of the loop), and the secret variable `highestBid` (`hB`) (a temporary variable for the maximum bid). The precondition specifies that `publishBid` is `public` and `bids` has a `private` security level. The additional variables `i` and `hB`, which do not occur in the specification above, are added by the programmer while constructing the program. Their resulting confidentiality levels are determined through the application of the refinement rules.

To construct the algorithm as in Listing 1, we want to divide the problem into three parts, an initialization of some temporary variables, the loop through the array of `bids` to search for the highest bid, and the assignment of the highest bid to the public variable `pB`. The first split into the initialization and the rest of the program is done with Refinement (1). It introduces a composition statement (cf. Rule 3 in Fig. 1), splitting the problem in two abstract subproblems `S1` and `S2` with an intermediate specification, which is calculated by IFbC while refining the program. The intermediate specification presents the security level of variables between statements.

In Refinement (2), the initialization of the temporary variables is done with the assignment statement `i = 0; hB = 0;` (cf. Rule 2, indeed it is a multi-step refinement with Rule 3 and 2). The statement initializes `i` as `public` and `highestBid` as `secret` variable, as declared by the programmer. In the declarations, the variables are initially labeled by the programmer, and further refinement rules ensure that these labeled variables are correctly adjusted during the refinements. In the postcondition of statement `S2`, which is the postcondition of the starting triple, the variable `i` has a `private` security level. Here, we can see that the security level of `i` is updated from `public` to `private` in the program to prevent a leak.

In Refinement (3), we further split the second part of the program with the composition rule to iterate through the array of bids first, and then, to assign the highest bid to a `public` variable with the use of a declassify operation. Refinement (5) assigns the highest bid and is explained after the refinement of the loop.

Refinement (4) creates the loop to iterate through the array of bids searching for the maximum as long as the control variable `i` is smaller than the length of the array (cf. Rule 5 for functional correctness). As the variable `bids` in the guard of the loop (`i < bids.length`) has a `private` security level, the security context of the loop body has to be increased to `private` to prevent leaks through indirect information flow. That means, sub-statements of the repetition can only assign information to variables of at least the security level of the new security context. For example, if we would assign to a `public` variable in the loop body, an attacker could deduce that the guard was evaluated to true by reading that `public` variable (e.g., `bids.length` is bigger than `i`).

The refinements (6)–(10) create the loop body which compares the next element of the array with the current highest bid. If the next element is greater, we update the highest bid. Refinement (6) splits the loop body with a composition into a check of the next bid and the increment of the loop variable. The refinements (7)–(9) establish the selection to check whether the next bid is higher than the current highest bid. As `hB` is used in the guards, the security context inside the selection statement is increased to `secret`. In Refinement (8), we assign a new highest bid to our variable (`hB = bids[i]`;). As `hb` is already `secret`, the security level stays the same, otherwise the security level of the variable has to be increased to `secret` because of the `secret` security context. In the case that the next bid is smaller or equal to the highest bid, Refinement (9) introduces a skip statement to not alter the program state.

The assignment in Refinement (10) increments the loop counter. Here, we increase the security level of `i` through the `private` security context. The security level of the `public` variable `i` is set to `private`. This increase of the security level propagates through the program, and therefore the security level of `i` is `private` in the initial triple of the program.

In Refinement (5), we construct the last part of the program, the assignment to the variable `pB`. Normally, by assigning `secret` data to a `public` variable, the `public` security level has to be increased to `secret`. This prevents a leak through direct information flow, as `secret` data would be accessible through a `public` variable. With a declassify operation, programmers can prevent the increase of the security level (e.g., if they are sure that the confidential data is allowed to be published, or the data is encrypted beforehand).

Banking Example for Integrity. In Fig. 5, we show a second example demonstrating how IFbC works for integrity. A user withdraws money from a bank account and the balance should be updated if the withdrawal is trustworthy. In the other case, the balance is not updated to secure the integrity of the bank. The precondition of the program specifies that it gets two `trusted` variables `balance` and `checked` (`checked` is used as parameter to return the result of the method), and an untrusted variable `withdraw` as input. The postcondition specifies that these security levels must not be altered. The allowed flow is from `trusted` to `untrusted`. The complete program with a functional specification is shown in Listing 2. The `balance` is reduced by the value of `withdraw` if the value of variable `checked` is true. In the other case, the `balance` is not altered. The Boolean variable `checked` is set by a method call to `check`.

To construct the program, we use a composition Refinement (1) to split the problem into a check whether the withdrawal is allowed and the update of the `balance` variable. Refinement (2) introduces a method call to `check` whether the system can trust the input variable `withdraw`. If this is the case, the variable `checked` is set to a true value. When calling the method, all parameters are passed by value-result, and therefore their security level can be changed. For example, the method `check` could be specified that it returns `balance` with an `untrusted` security level, as we allow an update of the security levels from `trusted` to `untrusted`. Then, the bank method would have to proceed with an `untrusted` variable. In our case, the security levels stay the same because we assume that

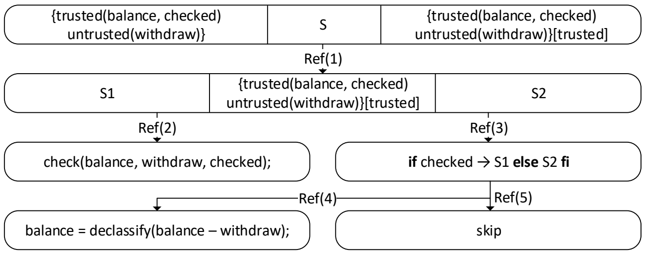

{trusted(balance, checked) untrusted(withdraw)}
S
{trusted(balance, checked) untrusted(withdraw)}[trusted]
Ref(1)
S1
{trusted(balance, checked) untrusted(withdraw)}[trusted]
S2
Ref(2)
Ref(3)
check(balance, withdraw, checked);
if checked → S1 else S2 fi
Ref(4)
Ref(5)
balance = declassify(balance – withdraw);
skip

Figure 5: Refinement steps for the banking example

```
pre: true
post: (!checked -> balance ==
  \old(balance)) & (checked -> balance
  == \old(balance) - withdraw);
void bank(trusted int balance,
  trusted boolean checked,
  untrusted int withdraw) {
  check(balance, withdraw, checked);
  if (checked) {
    balance = declassify(balance - withdraw);
  } else {
    skip
  } fi
}
```

Listing 2: Program of the banking example

the method `check` is specified that way. To verify that the method `check` fulfills its specification, it would be created with IFbC.

Refinements (3)–(5) introduce the selection statement to set the new balance of the bank account. As a `trusted` variable is used in the guard, the security context stays the same. With the declassify operation, we can calculate the new balance in the then-branch. Without declassify, it is not permitted to assign an `untrusted` value to the `trusted` variable `balance`. In the else-branch, a skip statement is used that does not alter the program.

For clarity, we give individual examples for confidentiality and integrity, but both policies can be ensured in the same program simultaneously by construction, as the IFbC refinement rules operate on any lattice of security levels. Practically, the variables would be labeled with a confidentiality and an integrity level, which are updated individually. Another possibility is to create the power set of both lattices and label every variable with a combination of security levels [31].

4 FORMALIZING INFORMATION FLOW CONTROL-BY-CONSTRUCTION

In this section, we formalize IFbC for the construction of functionally correct and secure programs. With this approach, programmers can incrementally construct programs, where the security levels are organized in a lattice structure to guarantee a variety of confidentiality and integrity policies. IFbC defines seven refinement rules to create secure programs. As these rules are based on refinement rules for correctness-by-construction, programmers can create programs that are functionally correct and secure.

$Vars$	Set of program variables
S	Statement (from the GCL [15])
$\mathsf{x} \in Vars$	Program variable
E	Expressions over the program variables in $Vars$
$vars(\mathsf{E}) \subseteq Vars$	Set of variables occurring in expression E
L	Bounded upper semi-lattice $(L, \leq)$ of security levels
$\mathcal{V}^{pre}, \mathcal{V}^{post}, l : Vars \to L$	Labeling function to map a variable to a security level
$lub_L : \mathcal{P}(L) \to L$	Least upper bound of the security levels in L
$\eta \in L$	Security context
$\{\mathcal{V}^{pre}, \mathsf{P}\}\ \mathsf{S}\ \{\mathcal{V}^{post}, \mathsf{Q}\}[\eta]$	IFbC triple

Figure 6: Basic notations for IFbC

4.1 Refinement Rules for Program Construction

To formalize the IFbC rules, we introduce in Fig. 6 basic notations for variables and security levels, which are used in the refinement rules. Every variable of the program is associated to one security level. Levels are arranged in a bounded upper semi-lattice with one greatest and one least level. The functional and security specification of a program is defined by an IFbC triple $\{\mathcal{V}^{pre}, \mathsf{P}\}\ \mathsf{S}\ \{\mathcal{V}^{post}, \mathsf{Q}\}[\eta]$. As a Hoare triple, the IFbC triple consists of a precondition $\{\mathcal{V}^{pre}, \mathsf{P}\}$, an abstract statement S, and a postcondition $\{\mathcal{V}^{post}, \mathsf{Q}\}$. The functional specification is declared in the logical formulas P and Q. In the following, we focus on security, so the functional specification will be omitted. The labeling function $\mathcal{V}^{pre}$ assigns security levels to all variables before the statement S is executed and $\mathcal{V}^{post}$ assigns security levels to all variables after the execution. The label η is used to capture the security context of the IFbC triple. This security context is used to reason about implicit information flow by tracking the security levels of guards in conditional or loop statements. The refinement rules replace an abstract statement by a more concrete statement. In the refined triple, the security levels of the variables are updated to implement the security policy of the program.

Skip. The first IFbC rule introduces a `skip` statement, which does not alter the program. It refines an IFbC triple $\{\mathcal{V}^{pre}\}\ \mathsf{S}\ \{\mathcal{V}^{post}\}[\eta]$ to a skip. The rule is applicable if the variables and their associated security levels stay the same.

Rule 1 (Skip).
$\{\mathcal{V}^{pre}\}\ \mathsf{S}\ \{\mathcal{V}^{post}\}[\eta]$ *is refinable to* $\{\mathcal{V}^{pre}\}\ \mathtt{skip}\ \{\mathcal{V}^{post}\}[\eta]$ *iff* $\mathcal{V}^{post}(\mathsf{x}) = \mathcal{V}^{pre}(\mathsf{x})$ *for all* $\mathsf{x} \in Vars$.

Assignment. With the assignment rule, an abstract statement S is refined to an assignment of the form x := E. This represents explicit information flow from the variables in the expression E to the variable x on the left-hand side. This direct flow can cause a leak if data with a higher security level is assigned to x. We can prevent this leak by enforcing the security level of x.

To apply the refinement, the labeling function $\mathcal{V}^{post}$ has to be altered. The new security level of the variable x is determined by the least upper bound of the security levels of all variables in the expression, the security level of the context to consider the indirect information flow and the security level of x itself. This new security level is assigned to x in the labeling function $\mathcal{V}^{post}$ in the postcondition of the IFbC triple. So, the only difference of $\mathcal{V}^{pre}$ and $\mathcal{V}^{post}$ is the update of the security level of variable x.

Rule 2 (Assignment).
$\{\mathcal{V}^{pre}\}\ \mathsf{S}\ \{\mathcal{V}^{post}\}[\eta]$ *is refinable to* $\{\mathcal{V}^{pre}\}\ \mathsf{x} := \mathsf{E}\ \{\mathcal{V}^{post}\}[\eta]$ *iff* $\mathcal{V}^{post}(\mathsf{y}) = \mathcal{V}^{pre}(\mathsf{y})$ *for all* $\mathsf{y} \in Vars \setminus \{x\}$, *and* $\mathcal{V}^{post}(\mathsf{x}) = lub(\{\mathcal{V}^{pre}(\mathsf{v}) \mid \mathsf{v} \in vars(\mathsf{E})\} \cup \{\mathcal{V}^{pre}(\mathsf{x}), \eta\})$.

Composition. With the composition rule, an abstract IFbC triple $\{\mathcal{V}^{pre}\}\ \mathsf{S}\ \{\mathcal{V}^{post}\}[\eta]$ is refined to two triples $\{\mathcal{V}^{pre}\}\ \mathsf{S1}\ \{\mathcal{V}'\}[\eta]$ and $\{\mathcal{V}'\}\ \mathsf{S2}\ \{\mathcal{V}^{post}\}[\eta]$, which are executed sequentially. Both triples can be further refined. To apply the rule, a labeling function $\mathcal{V}'$ is introduced, which assigns a security level to all program variables after the execution of the first statement and before the execution of the second statement. The exact labeling function $\mathcal{V}'$ is determined by refining S1 and S2 to concrete statements. The labeling functions $\mathcal{V}^{pre}$ and $\mathcal{V}^{post}$ and the security context η are not changed. For all variables, the security level can only be increased by the program. To capture a reverse information flow in specific cases, the declassify operation and new variables are used.

Rule 3 (Composition).
$\{\mathcal{V}^{pre}\}\ \mathsf{S}\ \{\mathcal{V}^{post}\}[\eta]$ *is refinable to* $\{\mathcal{V}^{pre}\}\ \mathsf{S1}; \mathsf{S2}\ \{\mathcal{V}^{post}\}[\eta]$ *iff there exists a labeling function* $\mathcal{V}' : Vars \to L$ *such that* $\{\mathcal{V}^{pre}\}\ \mathsf{S1}\ \{\mathcal{V}'\}[\eta]$ *and* $\{\mathcal{V}'\}\ \mathsf{S2}\ \{\mathcal{V}^{post}\}[\eta]$ *and for all* $\mathsf{v} \in Vars : \mathcal{V}^{pre}(\mathsf{v}) \leq \mathcal{V}'(\mathsf{v}) \leq \mathcal{V}^{post}(\mathsf{v})$.

Selection. The selection rule refines an abstract statement S to an if statement `if(G) → S1 else S2 fi`. Here, an implicit leak can occur as the selected branch reveals information about the guard. To prevent this, the statements in the branches have to be labeled with at least the security level of the guard. As selection statements can be nested, a security context is used to track the current security level that is needed to prevent an implicit leak.

By applying the refinement rule, the security context of the sub-statements have to be adjusted to the least upper bound of the security level of the if-guard and the security context of the outer selection statement. Both sub-statements with the new security context can be further refined.

Rule 4 (Selection).
$\{\mathcal{V}^{pre}\}\ \mathsf{S}\ \{\mathcal{V}^{post}\}[\eta]$ *is refinable to* $\{\mathcal{V}^{pre}\}$ **if** $\mathsf{G} \to \mathsf{S1}$ **else** $\mathsf{S2}$ **fi** $\{\mathcal{V}^{post}\}[\eta]$ *iff* $\{\mathcal{V}^{pre}\}\ \mathsf{S1}\ \{\mathcal{V}^{post}\}[\eta']$ *and* $\{\mathcal{V}^{pre}\}\ \mathsf{S2}\ \{\mathcal{V}^{post}\}[\eta']$ *with* $\eta' = lub(\{\mathcal{V}^{pre}(\mathsf{v}) \mid \mathsf{v} \in vars(\mathsf{G})\} \cup \{\eta\})$.

Repetition. The repetition rule introduces a classic while loop. By executing the loop, information about the guard is revealed. To prohibit this indirect leak, the security context of the inner loop statement is adjusted to the least upper bound of the security levels of the loop-guard and the security context of the outer repetition statement.

Rule 5 (Repetition).
$\{\mathcal{V}^{pre}\}\ \mathsf{S}\ \{\mathcal{V}^{post}\}[\eta]$ *is refinable to* $\{\mathcal{V}^{pre}\}$ **do** $\mathsf{G} \to \mathsf{S1}$ **od** $\{\mathcal{V}^{post}\}[\eta]$ *iff* $\{\mathcal{V}^{pre}\}\ \mathsf{S1}\ \{\mathcal{V}^{post}\}[\eta']$ *with* $\eta' = lub(\{\mathcal{V}^{pre}(\mathsf{v}) \mid \mathsf{v} \in vars(\mathsf{G})\} \cup \{\eta\})$

4.2 Method Call Rule

In a method call, all variables are passed by value-result and appear in the specification of the method. The security level of these passed variables may change, while the security level of other variables remains the same. By calling the method, the parameters of the caller are assigned to the parameters of the called method and the reverse assignment is done when returning from the method. It has to be ensured that in the beginning the security levels of variables of the called method are higher than or equal to the security levels of variables of the caller to prevent flows from higher to lower security levels. It also has to be ensured that the security levels in the postcondition of the caller are higher than or equal to the security levels of the called method for the same reason. For example, a secure value of the method has to be assigned to a variable with at least this security level in the program of the caller. Additionally, the called method has to satisfy its specification, which can be shown in a separate IFbC refinement.

Rule 6 (Method Call).
$\{\mathcal{V}^{pre}\}$ S $\{\mathcal{V}^{post}\}[\eta]$ is refinable to $\{\mathcal{V}^{pre}\}$ $\mathsf{M(a_1, \ldots, a_n)}$ $\{\mathcal{V}^{post}\}[\eta]$ iff for a method $\{\mathcal{V}^{pre}_{call}\}$ $\mathsf{M(z_1, \ldots, z_n)}$ $\{\mathcal{V}^{post}_{call}\}[\eta]$ and for all parameters: $\mathcal{V}^{pre}(\mathsf{a_i}) \leq \mathcal{V}^{pre}_{call}(\mathsf{z_i}) \wedge \mathcal{V}^{post}_{call}(\mathsf{z_i}) \leq \mathcal{V}^{post}(\mathsf{a_i})$ where $\mathsf{a_i}$ are the actual parameters and $\mathsf{z_i}$ are the formal parameters.

4.3 Declassification

With our information flow policy, we are not allowed to assign an expression with a higher security level to a variable with a lower security level without increasing the security level of the variable. This restricts the possibility to develop meaningful programs; in some cases the information flow from a more secure variable to a less secure one should be possible. For example, if a password is saved into a secure variable, an encrypted or hashed version of the password should be assignable to a less confidential variable. A declassification operator [22, 33] can be used to allow the assignment, but it should only be used if the programmer is sure that no secure information is leaked.

The declassification rule is a specialized assignment rule, where an expression E assigned to variable x is surrounded by the `declassify` operator. With this rule, the security level of the assigned variable is only set to the least upper bound of its security level and the security context. The difference to the standard assignment rule is that the security levels of variables of the assigned expression are not used to determine the new security level. The declassification refinement rule is only meaningful if the assigned expression would increase the security level of the assigned variable. If the security levels of all variables of the expression are lower than the security level of the assigned variable, the standard assignment rule and the declassification rule behave the same.

Rule 7 (Declassification Assignment).
$\{\mathcal{V}^{pre}\}$ S $\{\mathcal{V}^{post}\}[\eta]$ is refinable to $\{\mathcal{V}^{pre}\}$ $\mathsf{x := declassify(E)}$ $\{\mathcal{V}^{post}\}[\eta]$ iff $\mathcal{V}^{post}(\mathsf{y}) = \mathcal{V}^{pre}(\mathsf{y})$ for all $\mathsf{y} \in Vars \setminus \{x\}$, and $\mathcal{V}^{post}(\mathsf{x}) = lub(\{\mathcal{V}^{pre}(\mathsf{x}), \eta\})$.

5 PROOF OF SOUNDNESS AND EXPRESSIVENESS OF IFBC

We want to ensure that programs constructed with IFbC are secure. We assume that declassify is correctly used by the programmer because IFbC can detect the use of declassify, but it can not prevent an inappropriate application. In the following, we prove soundness of our IFbC rules.

Definition 1 (Secure program).
Let S be an IFbC program and $\{\mathcal{V}^{pre}\}$ $\mathsf{x := E}$ $\{\mathcal{V}^{post}\}[\eta]$ be an arbitrary IFbC triple in program S. Moreover, let G be the (possibly empty) set of all defined guards along the refinements from the root to that triple (i.e., in conditional statements and loops). We say that program S is secure (denoted by $secure(\mathsf{S})$) iff for all such triples the following two conditions hold:
$\forall \mathsf{v} \in vars(\mathsf{E}) : \mathcal{V}^{post}(\mathsf{x}) \geq \mathcal{V}^{pre}(\mathsf{v})$ (No direct leak)
$\forall \mathsf{v} \in vars(\mathsf{G}) : \mathcal{V}^{post}(\mathsf{x}) \geq \mathcal{V}^{pre}(\mathsf{v})$ (No indirect leak)

The variable x must have a security level that is greater than or equal to all security levels of variables that are in the expression E to prevent an assignment of secure information to an insecure variable. Indirect information flow leaks are prevented if no information can be deduced by analyzing the guards of conditional statements or loops. The variable x must have at least the security level of all guards used in the refinement branch.

To verify the soundness of IFbC, we start with a lemma to reason about indirect information flow. By assigning an expression E to a variable x, we know that x has at least the security level of the security context η that captures the current security level to prevent indirect leaks (i.e., η is used to track the security levels of guards used in the refinement branch).

Lemma 1 (Confinement).
Let $\{\mathcal{V}^{pre}\}$ $\mathsf{x := E}$ $\{\mathcal{V}^{post}\}[\eta]$ be an IFbC triple, then $\mathcal{V}^{post}(\mathsf{x}) \geq \eta$.

Proof. Confinement is proven by the definition of the refinement Rule 2 (Assignment). The new security level of x is computed by $\mathcal{V}^{post}(\mathsf{x}) = lub(\{\mathcal{V}^{pre}(\mathsf{v}) \mid \mathsf{v} \in vars(\mathsf{E})\} \cup \{\mathcal{V}^{pre}(\mathsf{x}), \eta\})$, and therefore $\mathcal{V}^{post}(\mathsf{x}) \geq \eta$ by the definition of the least upper bound. □

Soundness. With this lemma, we can prove the soundness theorem, which states that a program is secure, if it is constructed using our refinement rules.

Theorem 1 (Soundness).
If an IFbC triple $\{\mathcal{V}^{pre}\}$ S $\{\mathcal{V}^{post}\}[\eta]$ is refined to $\{\mathcal{V}^{pre}\}$ C $\{\mathcal{V}^{post}\}[\eta]$ with the IFbC refinement rules without declassify, and C is a concrete program, then $secure(\mathsf{C})$ holds.

Proof. We prove the soundness with structural induction. *Skip*, *assignment*, and *method call* are the basis steps because they are the leaves of the refinement tree, and *selection*, *repetition*, and *composition* are proven in the induction step.
Induction Base:

- Assignment: $\{\mathcal{V}^{pre}\}$ S $\{\mathcal{V}^{post}\}[\eta]$ is refined to $\{\mathcal{V}^{pre}\}$ $\mathsf{x := E}$ $\{\mathcal{V}^{post}\}[\eta]$. By using the assignment rule, the new security level of x is $\mathcal{V}^{post}(\mathsf{x}) = lub(\{\mathcal{V}^{pre}(\mathsf{v}) \mid \mathsf{v} \in vars(\mathsf{E})\} \cup \{\mathcal{V}^{pre}(\mathsf{x}), \eta\})$. We have to show the absence of direct and indirect information flow leaks.

  - Case direct information flow: We have to ensure that $\forall \mathtt{v} \in \mathit{vars}(\mathtt{E}) : \mathcal{V}^{post}(\mathtt{x}) \geq \mathcal{V}^{pre}(\mathtt{v})$. The assignment rule sets the security level of x at least to $\mathit{lub}(\{\mathcal{V}^{pre}(\mathtt{v}) \mid \mathtt{v} \in \mathit{vars}(\mathtt{E})\})$. With the definition of *lub*, we know that $\forall \mathtt{v} \in \mathit{vars}(\mathtt{E}) : \mathcal{V}^{post}(\mathtt{x}) \geq \mathcal{V}^{pre}(\mathtt{v})$, and therefore no leaks can occur.
  - Case indirect information flow: We have to ensure that $\forall \mathtt{v} \in \mathit{vars}(\mathtt{G})$ of guards G in the refinement branch: $\mathcal{V}^{post}(\mathtt{x}) \geq \mathcal{V}^{pre}(\mathtt{v})$. As we are at the start of the induction no refinement rule is used so far and no guards G exist. Using Lemma 1, we know that $\mathcal{V}^{post}(\mathtt{x}) \geq \eta$, so no leaks can occur.
- Skip: $\{\mathcal{V}^{pre}\}\ \mathtt{S}\ \{\mathcal{V}^{post}\}[\eta]$ is refined to $\{\mathcal{V}^{pre}\}\ \mathtt{skip}\ \{\mathcal{V}^{post}\}\ [\eta]$. As the skip statement has no assignment to a variable, no direct or indirect information flow can exist.
- Method Call: Given a method $\{\mathcal{V}^{pre}_{call}\}\ \mathtt{M}(\mathtt{z}_1, \ldots, \mathtt{z}_n)\ \{\mathcal{V}^{post}_{call}\}[\eta]$, the method call rule assigns the parameters $\mathtt{z}_i$ to the actual parameters and ensures that the security levels are only increased. Therefore, with the assumption that the method itself satisfies its IFbC triple, the method call does not violate the security policy.

Induction Hypothesis: For each IFbC triple $\{\mathcal{V}^{pre}\}\ \mathtt{T}\ \{\mathcal{V}^{post}\}[\eta]$ that was created in n refinement steps from an abstract IFbC triple $\{\mathcal{V}^{pre}\}\ \mathtt{S}\ \{\mathcal{V}^{post}\}[\eta]$ (denoted as $\mathtt{T} = \mathit{refined}(\mathtt{S})$), $\mathit{secure}(\mathtt{T})$ holds.
Induction Step:

- Repetition: $\{\mathcal{V}^{pre}\}\ \mathtt{S}\ \{\mathcal{V}^{post}\}[\eta]$ is refined to $\{\mathcal{V}^{pre}\}$ **do** $\mathtt{G} \rightarrow \mathtt{S1}$ **od** $\{\mathcal{V}^{post}\}[\eta]$ with $\{\mathcal{V}^{pre}\}\ \mathtt{S1}\ \{\mathcal{V}^{post}\}[\eta']$. By using the repetition rule, the security context for the statement S1 is set to $\eta' = \mathit{lub}(\{\mathcal{V}^{pre}(\mathtt{v}) \mid \mathtt{v} \in \mathit{vars}(\mathtt{G})\} \cup \{\eta\})$. By using the induction hypothesis, we know that $\mathit{secure}(\mathtt{S1})$ holds before introducing the loop. We have to show that the refinement preserves security.
  - Case direct information flow: Since the repetition statement does not introduce an assignment, no direct leak can occur.
  - Case indirect information flow: The repetition statement introduces a guard G. To prevent an indirect leak, each assigned variable x in the refinement branch of S1 needs at least the security level of G ($\forall \mathtt{v} \in \mathit{vars}(\mathtt{G}) : \mathcal{V}^{post}(\mathtt{x}) \geq \mathcal{V}^{pre}(\mathtt{v})$). Therefore, the repetition rules sets the security context from η to η' as shown above, where η' is greater than or equal to every security level of variables in the guard G. With the correctly updated security context η' and the Confinement Lemma 1 ($\mathcal{V}^{post}(\mathtt{x}) \geq \eta'$), we know that every assignment in the refinement tree of S1 has at least the security level of η', and therefore the complete program with the repetition statement has no leaks.
- Selection: Selection is similar to repetition. A new guard is introduced and the security context is correctly adjusted. The difference is that the adjusted security context applies for two sub-statements.
- Composition: $\{\mathcal{V}^{pre}\}\ \mathtt{S}\ \{\mathcal{V}^{post}\}[\eta]$ is refined to $\{\mathcal{V}^{pre}\}\ \mathtt{S1};\mathtt{S2}\ \{\mathcal{V}^{post}\}[\eta]$ with an intermediate labeling function $\mathcal{V}'$. From the induction hypothesis we know that both triples $\{\mathcal{V}^{pre}\}\ \mathtt{S1}\ \{\mathcal{V}'\}\ [\eta]$ and $\{\mathcal{V}'\}\ \mathtt{S2}\ \{\mathcal{V}^{post}\}[\eta]$ are secure. Since the following applies for all $\mathtt{v} \in \mathit{Vars} : \mathcal{V}^{pre}(\mathtt{v}) \leq \mathcal{V}'(\mathtt{v}) \leq \mathcal{V}^{post}(\mathtt{v})$, the security levels can only be increased. No assignment or guard is introduced, so no new direct or indirect leak can occur and the rest of the program is secure through the induction hypothesis. Therefore, we can deduce that $\mathit{secure}(\mathtt{S})$ holds.

□

Expressiveness. We prove that IFbC is at least as expressive as the information flow type system by Volpano et al. [31]. The type system was already introduced in Section 2.2. Now, we prove that every program, which is type safe (denoted as $\mathit{typesafe}(\mathtt{C})$), can also be constructed using IFbC. Note that the statement c of the typing rules and our statements S are analogous constructs for abstract statements. The security context η of our IFbC approach is also analogous to the type $\tau\ \mathit{cmd}$ of the statement in the type system.

Theorem 2 (Expressiveness).
For all programs C, *if* $\mathit{typesafe}(\mathtt{C})$ *holds, then there exists* $\{\mathcal{V}^{pre}\}\ \mathtt{S}\ \{\mathcal{V}^{post}\}[\eta]$ *as a starting IFbC triple which is refined to the same program* C ($\mathit{refined}(\mathtt{S}) = \mathtt{C}$) *and* $\mathit{secure}(\mathtt{C})$ *holds.*

Proof. We prove the expressiveness with structural induction on the type derivation. The typing rule for assignments (cf. typing Rule 5 in Fig. 3) is the typing rule for the start of the induction and typing rules 6, 7, and 8 are proven in the induction step.
Induction Base:

- Assignment: If $\mathtt{x} := \mathtt{e}'$ is type safe, where x is of type $\tau\ \mathit{var}$ and $\mathtt{e}'$ is of type τ, then we can refine a triple $\{\mathcal{V}^{pre}\}\ \mathtt{S}\ \{\mathcal{V}^{post}\}[\eta]$ to $\{\mathcal{V}^{pre}\}\ \mathtt{x} := \mathtt{e}'\ \{\mathcal{V}^{post}\}[\eta]$, where x and $\mathtt{e}'$ have the same security levels. Subtyping is allowed through typing Rule 2, which is analogous to our lattice-based definition of *lub*.

Induction Hypothesis: For each type safe program C that was typed by n typing rules, the following holds: $\mathtt{C} = \mathit{refined}(\mathtt{S})$ and $\mathit{secure}(C)$.
Induction Step:

- Typing Rule 6: The rule ensures that if C and $\mathtt{C}'$ are type safe, $\mathtt{C};\mathtt{C}'$ is also type safe. With the induction hypothesis, we know that C and $\mathtt{C}'$ are type safe and also the triples $\{\mathcal{V}^{pre}\}\ \mathtt{C}\ \{\mathcal{V}'\}[\eta]$ and $\{\mathcal{V}'\}\ \mathtt{C}'\ \{\mathcal{V}^{post}\}[\eta]$ are secure. By using the composition refinement rule, also $\{\mathcal{V}^{pre}\}\ \mathtt{C};\mathtt{C}'\ \{\mathcal{V}^{post}\}[\eta]$ is secure as the refinement rule ensures that $\mathcal{V}'$ is the same in both triples.
- Typing Rule 7: The rule ensures that if C and e are type safe, (**while** *e* **do** $C : \tau\ \mathit{cmd}$) is also type safe. With the induction hypothesis, we know that C is type safe and also the triple $\{\mathcal{V}^{pre}\}\ \mathtt{C}\ \{\mathcal{V}^{post}\}[\eta]$ is secure. To prove that the triple $\{\mathcal{V}^{pre}\}$ **do** $\mathtt{G} \rightarrow \mathtt{C}$ **od** $\{\mathcal{V}^{post}\}[\eta']$ is secure, we review the adjustment of the security context in the type system and in our approach. The type $\tau\ \mathit{cmd}$ of the statement C can have any type that is more secure than the type of e (cf. typing rules 4 and 7). This relation is analogously ensured by our repetition rule, which sets the security context η to $\mathit{lub}(\{\mathcal{V}^{pre}(\mathtt{v}) \mid \mathtt{v} \in \mathit{vars}(\mathtt{G})\} \cup \{\eta'\})$. The security level of the context has at least the security level of the guard.
- Typing Rule 8: This rule is similar to repetition. The only difference is that this rule needs two sub-statements C and $\mathtt{C}'$.

□

6 TOOL SUPPORT AND APPLICATIONS

Instead of proving post-hoc that a program is secure, we create with IFbC programs that are secure by construction. IFbC is at least as expressive as standard type systems and security is guaranteed through the sound set of refinement rules. In order to make IFbC amenable for programmers, we implemented tool support. We discuss the applicability of IFbC at the end.

Tool Support. We implemented tool support for IFbC, so that programmers can construct programs, which follow a lattice-based information flow policy. The tool guides a programmer to a secure program with the help of the IFbC refinement rules. In every step of the program refinement, a violation of the information flow policy is prevented by updating the security levels of variables. Simultaneously, refinement rules for correctness-by-construction guarantee the functional correctness of the program.

IFbC is implemented in a graphical editor CorC.[1] The editor is implemented in Java as an Eclipse modeling project. By tracking variables and their security levels, programs can be constructed that are secure with respect to the information flow policy. CorC represents the refinement hierarchy of an IFbC program in a tree structure. Every node represents an IFbC triple consisting of a pre-/postcondition specification and a statement; a leaf is a concrete statement and intermediate nodes are abstract statements. A refinement is visualized as an edge between two nodes. If the program is fully refined, it can be exported as Java code.

In Fig. 7, we show on the left-hand side an excerpt of the auction example in CorC (cf. Line 14 in Listing 1). We zoomed in to focus on the main features of the editor rather than showing the complete program. The leaf node is selected, which contains the assignment `i = i + 1;`. In the properties view, we show the security levels of the variables in the pre- and postcondition and the context. As we can see, we have an assignment to the `public` variable `i`. The calculated least upper bound is `private`, as we are in a `private` context, and therefore the security level of `i` is updated to `private` in the postcondition. The outcome of the tool is equal to our calculated security levels in the example above (cf. Fig. 4). In the middle of the graphic, the palette of CorC is shown to add refinements per drag and drop. On the right-hand side, the constructed program in textual form is shown, which is generated automatically by CorC.

This IFbC implementation extends the CorC [25] tool for correctness-by-construction. Besides information flow, CorC reasons about the functional correctness using a functional specification. By refining a program, the pre-/postcondition specification is updated according to the refinement rules and the side conditions are discharged automatically. To separate the functional conditions and the security levels graphically, we decide for a properties view to visualize the information flow at each step in the program. This fits the separation of concerns because the conditions for the functional correctness can be altered by the user, but the information flow is calculated automatically by CorC. By using the refinement rules and analyzing the declared variables, the security level of each variable at each step in the program can be computed. Users do not have to find invariants for loops or other specifications to ensure compliance with the information flow policy. If the user detects an inconsistency in the program, the exact spot where a variable deviates from the intended behavior can be pinpointed.

Applicability of IFbC. To demonstrate applicability of IFbC, we have conducted smaller case studies. Users already familiar with CorC were able to create secure programs while ensuring functional correctness simultaneously. As the IFbC rules are applied automatically, users only noticed the security mechanisms whenever they were prevented from writing insecure code.

[1]https://github.com/TUBS-ISF/CorC

We emphasize that correctness-by-construction is intended to be used in correctness-critical applications [18]. Therefore, the scope of this extension to prevent information leaks is the same. Mostly small security-critical programs will be constructed with IFbC. However, the approach also supports constructing larger programs by splitting them into smaller ones using method calls.

An advantage of IFbC is the constructive nature. Instead of checking that the information flow policy is not violated after writing the program, users can directly construct programs to comply with the policy. In every step of the program, even in partial programs, all variables and their security levels can be observed without executing the program. Another advantage is that the security (confidentiality as well as integrity) and functional correctness are guaranteed simultaneously, as a secure program that does not have the intended behavior is insufficient for the users. Functional correctness is also a mandatory requirement for integrity.

IFbC can be used supplementary to existing standard quality control mechanisms (e.g., a type system, provided that the type system has the same expressiveness, or testing) to increase trust in the created program. A program is constructed with IFbC, and afterwards or at certain points, other mechanisms are used to cross-check the correctness of the program. Overall, the IFbC approach is feasible for creating critical software. As finished programs can be automatically exported to Java, IFbC can be embedded into existing concepts or processes for secure Java development.

The functional CbC tool without information flow was already evaluated qualitatively with a user study [26]. In comparison to a post-hoc verification tool, the participants appreciated the good feedback of CorC to find defects in the code. The additional effort for using the refinement rules, was not mentioned negatively.

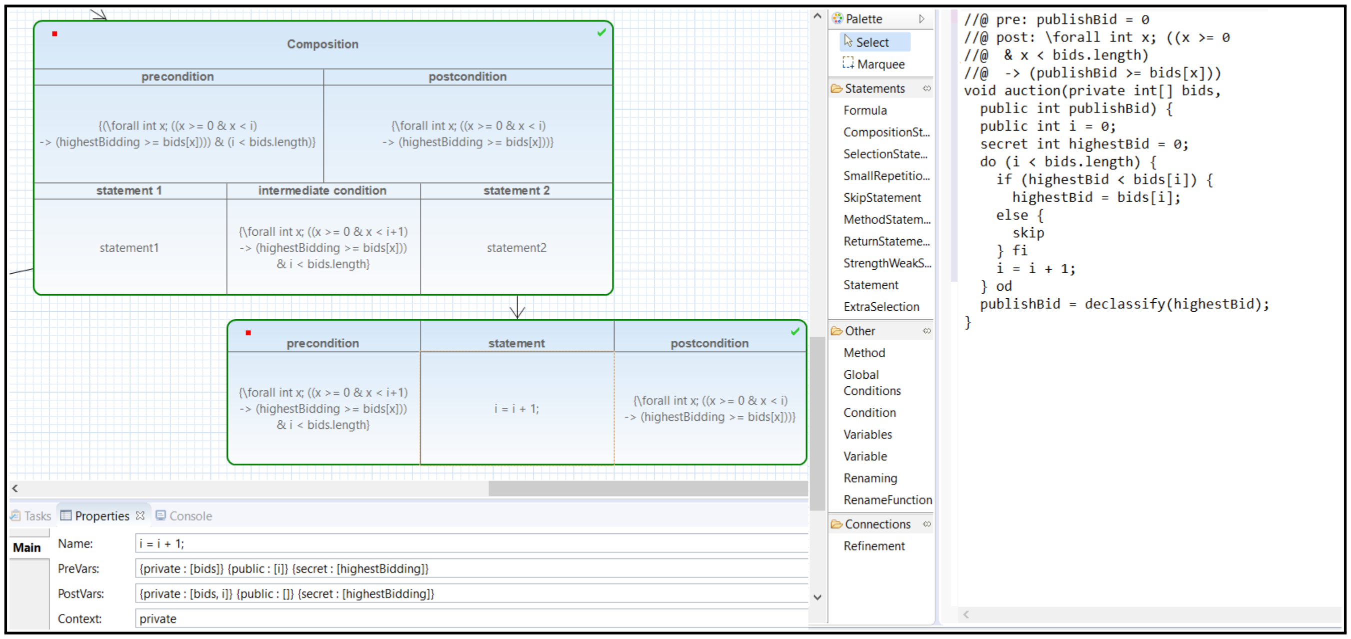
7 RELATED WORK

In the following, we discuss the differences to prior work and distinguish IFbC from other Hoare-style logics for information flow control. We also discuss information flow type systems and functional correctness-by-construction.

C14bC. We build on top of existing work. Schaefer et al. [29] introduced C14bC as a constructive approach to reason about information flow. They introduced a Hoare-style confidentiality specification with two levels, high and low, and developed refinement rules to create programs that ensure this specification. The programs are written in a simple while-language without method calls.

IFbC extends the specification of C14bC from high and low confidentiality levels to a lattice of security levels. We adapted the refinement rules to preserve the security of constructed programs for any input of user defined security levels. Confidentiality, integrity, as well as functional correctness can be ensured simultaneously in one program. We also extended the underlying language with a method call to improve the scalability of the approach, and we proved soundness of all refinement rules. Furthermore, IFbC is implemented as tool support that ensures the lattice-based information flow policy.

Hoare-Style Logics for Information Flow. Previous works that use Hoare-style program logics with information flow control analyze the programs after construction, instead of guaranteeing the security during construction. The work of Andrews and Reitman [5] is



Name: i = i + 1;
PreVars: {private : [bids]} {public : [i]} {secret : [highestBidding]}
PostVars: {private : [bids, i]} {public : []} {secret : [highestBidding]}
Context: private

```
//@ pre: publishBid = 0
//@ post: \forall int x; ((x >= 0
//@  & x < bids.length)
//@  -> (publishBid >= bids[x]))
void auction(private int[] bids,
  public int publishBid) {
  public int i = 0;
  secret int highestBid = 0;
  do (i < bids.length) {
    if (highestBid < bids[i]) {
      highestBid = bids[i];
    else {
      skip
    } fi
    i = i + 1;
  } od
  publishBid = declassify(highestBid);
}
```

Figure 7: Excerpt of the auction example in CorC

similar to our approach, but they directly encode the information flow in a logical form. They support multiple security levels for parallel programs. Amtoft and Banerjee [3] also use Hoare-style program logics and abstract interpretation to reason about information flow leaks. They can give failure explanations based on the derivation of strongest postconditions. This work is the basis for specifying and checking information flow in SPARK Ada [4].

Program Analysis for Information Flow. Static and dynamic program analysis is used to enforce information flow policies [23, 27, 28]. Examples are taint analysis [6] or security type systems [10, 17, 19, 31]. IFbC checks the compliance with an information flow policy similar to a type system, but if necessary our approach updates security levels of variables to prevent leaks in the program (cf. the update of the security level of a variable in the assignment rule).

JFlow [21] is a related approach that extends Java to check information flow. In contrast to other languages, JFlow supports language features such as objects, subclassing, and exceptions. With our proposed security-by-construction method, we are more restrictive, but we created a concept to create secure and functionally correct programs by construction that can be extended for richer languages. If a similar expressiveness is given, IFbC can be used supplementary to established program analysis tools to increase the security of programs.

To discover security flaws early, Tuma et al. [30] proposed an approach to analyze the information flow in a system at design level using security data flow diagrams. Their technique is inspired by type systems to detect violations of an information flow policy.

Functional Correctness-by-Construction. Correctness-by-construction is mostly used to ensure functional correctness. A specification is refined stepwise to actual programs. The Event-B framework [1] is an approach to refine specified automata-based systems to concrete and functionally correct implementations. The Event-B method is implemented in the Rodin platform [2]. This approach differs by another abstraction level. Our underlying functional correctness-by-construction approach uses code and logical specification rather than automata-based systems. The CbC approaches of Back [9] and Morgan [20] are also related. Both can be used to refine abstract programs into functionally correct programs. Implementations are ArcAngel [24] for Morgan's refinement calculus and SOCOS for the refinement approach proposed by Back [7, 8]. All discussed approaches are limited to the functional correctness and cannot reason about security.

8 CONCLUSION

In this work, we presented IFbC, a constructive approach for secure lattice-based information flow control. This approach enables security-by-design processes to guarantee the security of programs during construction. We formalized the refinement rules of IFbC and proved their soundness. We also showed that IFbC is at least as expressive as a type system for information flow [31]. IFbC is implemented in the open-source tool CorC. CorC support the information flow rules presented in this paper, and it can also guarantee the functional correctness of a program. With the tool support, we evaluated our methodology by implementing some examples and discussed the applicability of IFbC.

For future work, we can convert IFbC to be flow- and path-sensitive to make it less pessimistic. By transforming the program to eliminate false flow dependencies and by using dependent types to better reason about branches in the control flow, more secure programs can be accepted [19].

REFERENCES

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *STTT* 12, 6 (2010), 447–466.

[3] Torben Amtoft and Anindya Banerjee. 2004. Information Flow Analysis in Logical Form. In *SAS (LNCS)*, Vol. 3148. Springer, 100–115.

[4] Torben Amtoft, John Hatcliff, Edwin Rodríguez, Robby, Jonathan Hoag, and David A. Greve. 2008. Specification and Checking of Software Contracts for Conditional Information Flow. In *FM*. Springer, 229–245.

[5] Gregory R. Andrews and Richard P. Reitman. 1980. An Axiomatic Approach to Information Flow in Programs. *TOPLAS* 2, 1 (1980), 56–76.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *PLDI*, Vol. 49. ACM, 259–269.

[7] Ralph-Johan Back. 2009. Invariant Based Programming: Basic Approach and Teaching Experiences. *FAOC* 21, 3 (2009), 227–244.

[8] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. 2007. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *TAP (LNCS)*, Vol. 4454. Springer, 61–78.

[9] Ralph-Johan Back and Joakim Wright. 2012. *Refinement Calculus: A Systematic Introduction*. Springer Science & Business Media.

[10] Anindya Banerjee and David A Naumann. 2002. Secure Information Flow and Pointer Confinement in a Java-like Language.. In *CSFW*, Vol. 2. 253.

[11] D Elliott Bell and Leonard J La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report. MITRE Corp Bedford MA.

[12] Kenneth J Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report. MITRE Corp Bedford MA.

[13] Bart De Win, Riccardo Scandariato, Koen Buyens, Johan Grégoire, and Wouter Joosen. 2009. On the Secure Software Development Process: CLASP, SDL and Touchpoints Compared. *InfSof* 51, 7 (2009), 1152–1171.

[14] Dorothy E Denning. 1976. A Lattice Model of Secure Information Flow. *CACM* 19, 5 (1976), 236–243.

[15] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice Hall.

[16] Michael Howard, Steve Lipner, U Index, U Part, U Chapter, U Why In, U First Steps, U New Threats, U Windows, U Seeking Scalability, et al. 2006. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press.

[17] Sebastian Hunt and David Sands. 2006. On Flow-Sensitive Security Types. *SIGPLAN Not.* 41, 1 (Jan. 2006), 79–90.

[18] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-By-Construction Approach to Programming*. Springer.

[19] Peixuan Li and Danfeng Zhang. 2017. Towards a Flow-and Path-Sensitive Information Flow Analysis. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 53–67.

[20] Carroll Morgan. 1994. *Programming from Specifications* (2nd ed.). Prentice Hall.

[21] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. ACM, New York, NY, USA, 228–241.

[22] Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy Using the Decentralized Label Model. *TOSEM* 9, 4 (2000), 410–442.

[23] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer.

[24] Marcel Vinicius Medeiros Oliveira, Ana Cavalcanti, and Jim Woodcock. 2003. ArcAngel: A Tactic Language for Refinement. *FAOC* 15, 1 (2003), 28–47.

[25] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W Watson. 2019. Tool Support for Correctness-by-Construction. In *FASE (LNCS)*, Vol. 11424. Springer, 25–42.

[26] Tobias Runge, Thomas Thüm, Loek Cleophas, Ina Schaefer, and Bruce W Watson. 2019. Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study. In *Refine*. Springer. To appear.

[27] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 186–199.

[28] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *J-SAC* 21, 1 (2003), 5–19.

[29] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick Kourie, and Bruce W Watson. 2018. Towards Confidentiality-by-Construction. In *ISoLA (LNCS)*, Vol. 11244. Springer, 502–515.

[30] Katja Tuma, Riccardo Scandariato, and Musard Balliu. 2019. Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In *ICSA*. IEEE, 191–200.

[31] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *JCS* 4, 2/3 (1996), 167–188.

[32] Bruce W. Watson, Derrick G. Kourie, Ina Schaefer, and Loek Cleophas. 2016. Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience?. In *ISoLA (LNCS)*, Vol. 9952. Springer, 730–748.

[33] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *CSFW*. IEEE, 15–23.