Tool Support for Confidentiality-by-Construction

Extended Abstract

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

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

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

Loek Cleophas
TU Eindhoven
Stellenbosch University
loek@fastar.org

Derrick Kourie
Stellenbosch University
CAIR, Stellenbosch
derrick@fastar.org

Bruce W. Watson
Stellenbosch University
CAIR, Stellenbosch
bruce@fastar.org

ABSTRACT

In many software applications, it is necessary to preserve confidentiality of information. Therefore, security mechanisms are needed to enforce that secret information does not leak to unauthorized users. However, most language-based techniques that enable information flow control work post-hoc, deciding whether a specific program violates a confidentiality policy. In contrast, we proposed in previous work a refinement-based approach to derive programs that preserve confidentiality-by-construction. This approach follows the principles of Dijkstra's correctness-by-construction. In this extended abstract, we present the implementation and tool support of that refinement-based approach allowing to specify the information flow policies first and to create programs in a simple while language which comply to these policies by construction. In particular, we present the idea of confidentiality-by-construction using an example and discuss the IDE C-CorC supporting this development approach.

ACM Reference Format:
Tobias Runge, Ina Schaefer, Alexander Knüppel, Loek Cleophas, Derrick Kourie, and Bruce W. Watson. 2018. Tool Support for Confidentiality-by-Construction: Extended Abstract. In *Proceedings of HILT Workshop 2018 (HILT'18).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern software applications must be developed to preserve the confidentiality of personal data, like personal information, such as passwords or credit card data. This information has to be secured from unauthorized access. Language-based security mechanisms [21] are used to control access to confidential information. In information flow control approaches [21, 24], *security policies* can be specified to define how information can flow in the program. Therefore, variables are classified as *high* or *low* variables. For example, a simple information flow policy can allow information flow from low to high variables, but prohibit the other direction. The high/private information should be separated from low/public information. Until now, information flow control is mostly checked post-hoc rejecting programs that do not comply to the specified information flow control policy.

In previous work [22], we have presented an approach to construct programs such that they satisfy a given information flow policy by construction. The approach is called *confidentiality-by-construction (C14bC)*[1], and it is based on the concept of correctness-by-construction to create correct programs incrementally using refinement rules. Correctness-by-construction (CbC) [15] is an approach to construct programs beginning with an abstract statement and a specification (pre-/postcondition). The statement is refined by using refinement rules to a concrete program that satisfies the specification. This classical CbC approach was proposed by Dijkstra [9] and others [10, 16] to create programs constructively. It should not be confused with the CbyC process by Hall and Chapman [11] which is a software development process where formal modeling techniques and analyses such as the Z-notation are used for all development phases from requirements analysis to maintenance. The CbC approach reduces the effort of post-hoc verification because the developer pays attention to program quality from the beginning [25]. In the CorC tool[2], programs can be constructed incrementally using the Dijkstra CbC style.

In this paper, we present the tool C-CorC[3] for building programs satisfying confidentiality properties in the sense of information flow control following the C14bC approach [22]. In C14bC, we use pre-/postcondition specifications—as traditionally used for classical functional correctness—with confidentiality specifications, expressing which variables are classified as high, thus, containing secrets. Then, we provide rules for each possible program construct[4] to refine an abstract program. The programmer can introduce such a program construct by using a refinement rule. Side conditions that have to be valid for the sound application of the refinement rule are checked automatically using static analysis such that no secrets are leaked with respect to the given information flow policy. We then derive a new information flow specification determining

HILT'18, November 2018, Boston, MA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] The numeronym C14bC abbreviates confidentiality as C14, as there are 14 letters after the first C.
[2] https://github.com/TUBS-ISF/CorC, CorC is an acronym for Correctness by Construction
[3] https://github.com/TUBS-ISF/CorC, C-CorC is an acronym for Confidentiality and Correctness by Construction
[4] We currently restrict ourselves to a simple while language which will be extended in future work.

which information is confidential after introducing the program construct. The C-CorC tool comprises a graphical editor visualizing the program, its specification and the refinements in a tree-like structure. The refinement rules are implemented in the editor to only allow refinements which satisfy the confidentiality policy. The C-CorC tool combines program development using CbC and C14bC, and hence is a proper extension of the CorC tool. In this paper, we focus on the confidentiality part.

The remainder of the paper is structured as follows: In Section 2, we review related tool-based approaches for information flow control and functional correctness. In Section 3, we introduce C14bC giving an example. Afterwards in Section 4, the C-CorC tool is explained to create programs which preserve confidentiality. Section 5 concludes the paper.

2 RELATED WORK

Language-based information flow security is a broad field of research [21]. Most approaches use static or dynamic program analysis to check that policies are satisfied [18]. For example, type systems are popular [24]. As in our approach, Andrews and Reitman [6] combined information flow control with Hoare-style program logics. In contrast to our approach, the check is post-hoc at compile-time. The language JFlow [17] is an extension to Java which statically checks information flow properties. Annotations are added to the code to allocate read and write permissions. These annotations are checked at compile time. UMLS [13] is an extension of the Unified Modeling Language to generate programs preserving confidentiality. Diagrams with security information are translated to code and validated with JFlow. In contrast to the other tools, JSFlow [12] is an interpreter that dynamically checks whether Java Script programs violate information flow policies. Rifle [23] is another tool to dynamically check information flow of programs. It translates a program binary into a binary that is checked on a special processor architecture. Lift [20] is an information flow tracking system which also analyses binaries, but without special hardware requirements. The focus of the tool is on security attacks, but with small modifications it can also be used to detect confidentiality leakage. All these approaches and tools work post-hoc and therefore can be distinguished from our approach to create secure programs during construction.

There are several tools to post-hoc verify correctness of programs which are specified in a functional Hoare style logic. The Tecton Proof System [14] is a related tool to structure and interactively prove Hoare logic specification. The proof or rather multiple proofs are represented as set of linked trees. This tool has a different focus in comparison to our tool, we try to hide the proofs in the background and focus on the code to construct. Other tools to prove programs interactively are the theorem provers Coq [8], Isabelle/HOL [19] or KeY [4]. For KeY, also an extension to check information flow policies in a post-hoc fashion exists [4].

SPARK [7] has similar goals compared to our approach presented here. SPARK is a subset of the programming language Ada and supports verification and information flow analysis. Code can be specified by contracts, and the correctness of the program is checked by SPARK tools. The contracts include the used data, information flow, pre-, postconditions and assertions. However, while SPARK uses post-hoc analysis of the annotations with respect to the implementation, we are following a constructive approach with C14bC.

The Event-B framework [1] is also a related correctness-by-construction approach. Here, automata-based system containing a specification are refined to a concrete implementation. Atelier B [2] implements the B method by providing an automatic and interactive prover. Rodin [3] is another tool implementing the Event-B method. We distinguish from these tools by working directly on code and specification rather than automata-based system.

3 C14BC BY EXAMPLE

To introduce the approach of C14bC, we explain it using an example. In Listing 1, we show a small program to show the credit card number of a user who has inserted his card. We have annotated the variables with high and low and assume a simple information flow policy that high information may not influence low information, neither directly by assignment or indirectly in loop-guards or if-conditions. In this artificial example, the user enters a password and that password is checked as long as tries are left. In every of the three iterations, if the password is correct, either the credit card number is shown or the password is asked again. While admittedly somewhat artificial, this is code will be retained for the sake of simplicity.

```
1  int low tries := 0;
2  String high password := getPassword();
3  while (tries < 3) {
4    if (correct(password)) {
5      showCreditCardNumber();
6    } else {
7      password := getPassword();
8    }
9    tries := tries + 1;
10 }
```

Listing 1: C14bC example program

To construct the program in C14bC, we start with an abstract Hoare triple $\{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S}\ \{\mathcal{H}^{\mathtt{post}}\}[low]$ including an abstract statement S and two empty sets of high variables ($\mathcal{H}^{\mathtt{pre}}, \mathcal{H}^{\mathtt{post}}$). These sets track the confidential data which should not be revealed. The context after the triple $[low/high]$ is used to control indirect information flow in loop-guards and if-conditions. By applying refinement rules, the abstract program is refined to the concrete program.

The refinement steps to arrive at the program in Listing 1 are shown in Figure 1 (based on the rules introduced by Schaefer et al. [22]). The first refinement splits the abstract Hoare triple into two triples $\{\}\ \mathtt{S1}\ \{\}[low] \wedge \{\}\ \mathtt{S2}\ \{\}[low]$. The refinement is called composition. To introduce line 1, we refine the first triple with statement S1 to an assignment `tries := 0;` (ref. 2). The variable `tries` is `low`, and therefore we do not alter the sets of high variables. The second triple is refined to the rest of the program. To introduce line 2, the second Hoare triple is split with the composition rule $\{\}\ \mathtt{S21}\ \{\}[low] \wedge \{\}\ \mathtt{S22}\ \{\}[low]$ (ref. 3). The first triple is refined to the assignment in line 2 (ref. 4). We introduce a high variable,

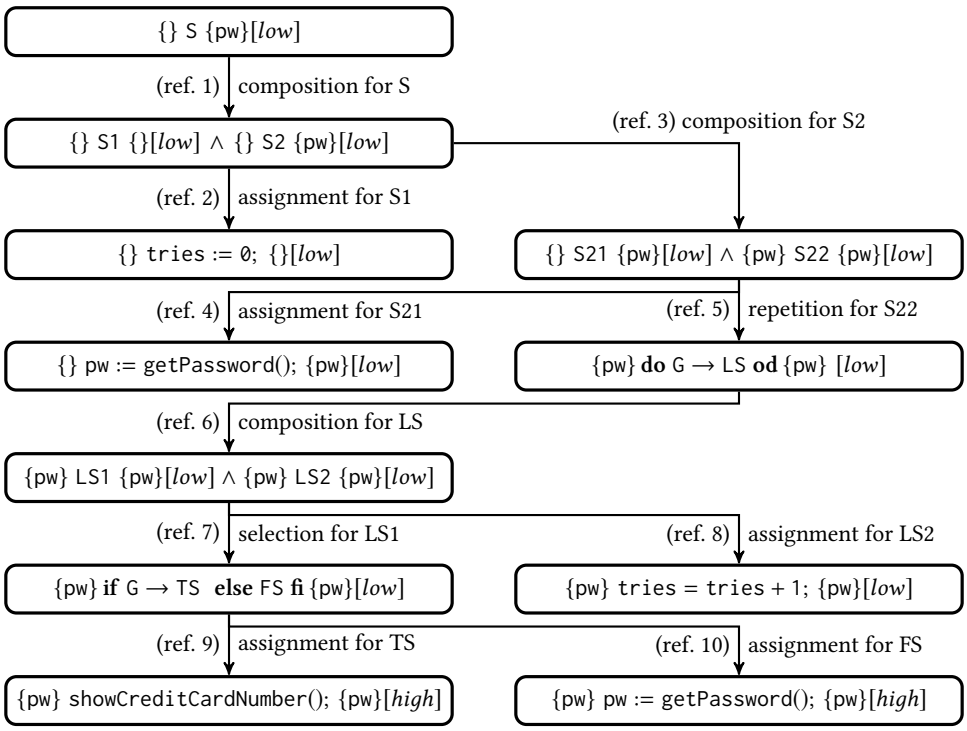

Figure 1: Refinement steps for the credit card example

therefore, we have to ensure that `password` (`pw` in Figure 1) is in the post set of high variables. The high variable propagates up and is added to the composition statements and the starting triple, so that the confidentiality in the program is correctly captured, and the remaining program contains the added high variable. The called method `getPassword` contains an input operation and therefore can not be checked by our approach. The same goes for the method `showCreditCardNumber` which contains an output operation. We use these methods in the example to simplify the code.

The while loop of line 3–10 is added with a repetition refinement (ref. 5). In the guard of the while loop, we have only low variables, therefore, the context stays low. If a high variable is used in the guard, the context has to be raised to high. In a high context, an assignment to a low variable is forbidden because an attacker could conclude the value of the high variable in the guard. To implement the loop body, another composition is needed. The second triple is refined to the assignment in line 9 using only low variables (ref. 8). The first triple of the composition is refined to a selection statement (ref. 7). The selection statement contains a high variable in the guard, so the context for all statements inside is raised to high. We introduce an assignment in refinement 9 which calls the process to show the credit card number. Here, the sets of high variables does not have to be altered, assuming that `showCreditCardNumber` does not have side effects on the variables of the calling method.

For the assignment in line 7 to get another password (ref. 10), we do not have to alter the set of high variables. The high variable `password` is already in the set of high variables. We are in a high context, so an assignment to a low variable is forbidden. Therefore, the increase of the low variable `tries` (line 9) has to be done outside the if-else construct. The complete program is created with refinements as shown in Figure 1. The specification is {} `S` {pw}[*low*]. We start with an empty set of high variables and end with `password` as the only high variable.

Another important aspect is, whether the user wants to assign a high variable to a low variable. For example, the assignment `int low output := encrypt(password);` is not allowed, although the password is encrypted and no secret information can be leaked. The user of C14bC can use the keyword `declassify` to explicitly allow this assignment. This function should be used with care to not violate confidentiality policies.

4 TOOL SUPPORT

The C-CorC tool for the C14bC approach extends the graphical editor of the CorC tool for constructing functionally correct programs in a correct-by-construction fashion. The editor uses Hoare triples to specify the different statements, like assignments or selections. These statements are connected in a tree-like structure to represent the refinement steps (cf. Figure 1). Every abstract statement has to be refined so that a concrete program is developed.

To support C14bC, we extend the Hoare triples capturing functional specifications to also contain information on high variables. We also extend the editor to preserve confidentiality. Therefore, we implement the C14bC refinement rules of Schaefer et al. [22]. We statically analyze variables in expressions and reason about their usage. The classification as high and low variables is maintained during the program flow, and violations of confidentiality

are reported. The static analysis has to be improved if we want to support richer programming languages including side effects. To summarize the features of the C-CorC tool:

- C-CorC has five statements to construct programs: assignment, skip, composition, selection, and repetition.
- C-CorC supports integers, chars, strings, arrays, and method calls without side effects or I/O.
- Programs are written as Hoare triple specifications, including pre-/postcondition specifications containing sets of high variables and the associated program statement.
- Refinements of statements are automatically checked whether confidentiality is preserved.
- C-CorC includes a declassify operation to allow information flow from high to low if no secrets are leaked.
- The refinements are visualized graphically in a tree-like structure.
- High/private variables can be tracked through the program (shown in pre-/postconditions).
- Functional verification can be done by the CorC tool.

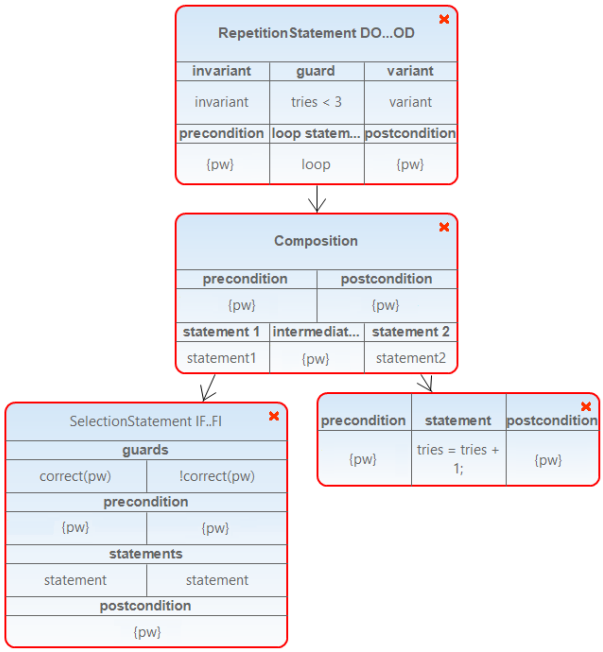

Figure 2: Graphical representation in the C14bC editor

In Figure 2, we show how a program is visualized in the C-CorC editor. In the figure, the while loop in line 3–10 is shown, excluding the statements in line 5 and 7. The tree structure in the editor is the same as in Figure 1 (ref. 6,7 and 8 are shown). Every node represents one Hoare triple, and the connections represent the refinements. We have the repetition statement at the top. The loop body is refined to a composition containing a selection statement and an assignment (`tries := tries + 1;`) which is inserted by the user. In the areas of pre- and postconditions, the high variables are specified. The user can specify the pre- and postconditions of CbC in the same area, but they are not shown here for clarity. We can see, that the high variable `pw` is propagated through the lower nodes. The high variables in the pre- and postconditions are not specified by the user, but are automatically inferred by the C14bC rules. If a security policy is violated, the editor shows the problem. For example, an assignment to a low variable in a high context or an assignment of a high variable to a low variable is prohibited. The repetition statement also has an invariant, a guard and a variant for the CbC part of the editor. In the selection statement the guard for every sub-statement can be specified.

The user can construct this example program by adding the needed nodes with drag and drop and refining the nodes by adding connection arrows. In the respective text fields, conditions and assignments can be concretized. The advantage of the graphical editor is that the user can see at every refinement step the associated high variables, and she can track where the high variables are introduced. The tool offers an overview of the whole program, and it allows to analyze every Hoare triple in isolation. If the program is fully specialized or in between, the code can be extracted as a textual view automatically if needed for clarity reasons. We also work on a textual editor as already implemented for CorC.

In this example, C14bC is considered in isolation. However, the C-CorC tool supports CbC and C14bC. The refinement rules of both approaches are implemented. We can construct programs which are functionally correct and preserve confidentiality. The rules of both approaches are checked, and violations are highlighted for the user. The tool was implemented as an extension of CorC, and therefore, it can reuse much functionality. The background model which stores all information had to be extended to track the high and low variables and the confidentiality context. We implemented a parser to analyze the assignment statements. The hardest part was to implement a reasoning system which updates the high and low variables in the conditions. The tool was tested without problems with small examples, like the one in Section 3, and bigger examples are intended for future work. The scalability of the tool is an open question, but we assume to use the tool for small security critical parts.

5 CONCLUSION AND FUTURE WORK

The presented tool C-CorC is a first version to support program development using C14bC. In this paper, we presented the idea of C14bC by an example and introduced the tool support to construct simple programs which preserve confidentiality. There are several directions to extend this work: The tool should be evaluated with case studies to measure the applicability and scalability. A user study could also be done to measure the usability of the tool. The approach is implemented for a simple while language. We can extend the language with object-orientation, following information flow control approaches for object-oriented languages [5]. The introduction of method calls with side effects should have a positive impact on scalability. Furthermore, the C14bC approach is tailored to high/low security policies. We could generalize the approach to support several security layers, for which we have to revise the refinement rules.

REFERENCES

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

[2] Jean-Raymond Abrial and Jean-Raymond Abrial. 2005. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

[3] 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. *International journal on software tools for technology transfer* 12, 6 (2010), 447–466.

[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001. Springer.

[5] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. 2006. A Logic for Information Flow in Object-Oriented Programs. In *POPL*. 91–102.

[6] Gregory R. Andrews and Richard P. Reitman. 1980. An Axiomatic Approach to Information Flow in Programs. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 56–76.

[7] John Gilbert Presslie Barnes. 2003. *High Integrity Software: The Spark Approach to Safety and Security*. Pearson Education.

[8] Yves Bertot and Pierre Castéran. 2013. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media.

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

[10] David Gries. 1987. *The Science of Programming*. Springer.

[11] A. Hall and R. Chapman. 2002. Correctness by Construction: Developing a Commercial Secure System. *Software, IEEE* 19, 1 (Jan 2002), 18–25. https://doi.org/10.1109/52.976937

[12] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1663–1671.

[13] Rogardt Heldal and Fredrik Hultin. 2003. Bridging Model-Based and Language-Based Security. In *European Symposium on Research in Computer Security*. Springer, 235–252.

[14] Deepak Kapur, Xumin Nie, and David R. Musser. 1994. An Overview of the Tecton Proof System. *Theoretical Computer Science* 133, 2 (1994), 307–339.

[15] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-By-Construction Approach to Programming*. Springer. http://books.google.co.za/books?id=5Ig6ELUQFM4C

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

[17] Andrew C Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 228–241.

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

[19] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Springer Science & Business Media.

[20] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. Lift: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 135–148.

[21] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[22] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. *ISoLA* (2018). To appear.

[23] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. 2004. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. IEEE, 243–254.

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

[25] 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*. 730–748.