Verifying Security Properties of the Source Code of Access Control Tools Using Frama-C

1st Andrey M. Kanner
Department of Cryptology and Cyber Security (42)
National Research Nuclear University MEPhI
Moscow, Russia
kanner@mail.ru

2nd Tatiana M. Kanner
Department of Cryptology and Cyber Security (42)
National Research Nuclear University MEPhI
Moscow, Russia
sheikot@mail.ru

***Abstract*—The paper discusses the features of the approach most commonly used to verify access control tools, where a high-level specification (security model) is generated. Serious problems are revealed in the applied approach, since there is a discrepancy between the object being verified and the real object requiring verification. As an alternative approach to verifying access control tools, it is proposed to verify security properties at the source code level, for example, for ANCI C code using Frama-C and special theorem provers. The paper considers the features of the Frama-C core, plug-ins and theorem provers. Using a synthetic example of a system, where subject-object interaction is arranged within the Bell-LaPadula Model, a possibility is demonstrated to generate the properties of a security model in the form of low-level contracts for several functions. A possibility of forming new function conditions by analyzing covert channels for the Bell-LaPadula mandate policy is demonstrated, taking into account the hierarchical structure of the system objects, as well as the corresponding change in the source code to meet the new conditions. In conclusion, the advantages of verifying access control tools at the source code level in comparison with the previously used approach are given. The features are listed that arise in case of a change in the source code and in case of verifying access control tools using standard library functions. A possible direction is proposed for further research on dynamic verification of the source code of access control tools.**

***Index Terms*—verification, access control tools, security properties, Frama-C**

I. Introduction

As a rule, access control tools implement a certain security model in practice [1] (Take-Grant, Bell-LaPadula, etc.) and, accordingly, can ensure implementation of formal security properties in the target system. However, in order to implement theoretical security properties, it is required to justify conformity of access control implementation to a particular model. To justify such conformity, access control tools are usually (formally) verified using special tools.

In the Russian Federation, like in other countries, access control tools are verified at the moment they are certified for compliance with the requirements of regulatory documents governing information security [2]. Despite the fact that the processes of modeling and verifying access control tools will soon become standardized, at the moment there is no single approach to their implementation.

In [3], [4] the authors described one of the approaches to the verification of access control tools using Lamport's Temporal Logic of Actions (TLA) and the Model Checking method [5], [6]. In this approach, a high-level specification is compiled from the source code of the access control tool in a formal language suitable for verification, and such a specification is further verified for compliance with security properties. Based on the results of such verification, however, it is only possible to confirm that such a specification of the access control tool meets the given security properties. However, an important fact overlooked is the need to substantiate the conformity of the specification to the source code; otherwise the access control tool may not be deemed as complying with the given security properties.

Unfortunately, a similar approach is now used everywhere – each access control tools manufacturer creates its own specification (security model); testing laboratories and government certification agencies somehow check such a specification and confirm that successful verification of such a security model implies compliance of the access control tool with the generated security properties. In fact, however, the source code of the access control tool is not analyzed in this process, and its compliance with the specification is shown only informally.

In most cases, a high-level system specification does not cover all the implementation nuances of a large source code, since the system is simplified and entities are abstracted to a level at which it is convenient to describe security properties. As a result, using such an approach, an entity is verified, other than that very entity, which really requires verification, but only some derivative thereof.

Thus, it seems logical to verify not some derived specification of the access control tool, but exactly its source code. In practice, testing laboratories analyze source code using static analyzers and heuristic tools that find only a fixed set of source code flaws (incorrect memory handling, errors in arithmetic operations, etc.) and are unable to substantiate the complete absence of errors.

However, in addition to static analyzers, there are more advanced static analysis tools for the source code of various programming languages, such as Frama-C[1] for ANCI C [7]–[9], which allow to use special provers to prove that the code conforms to some properties described in its specification.

[1] https://frama-c.com/

Such properties can include various conditions that must be met by the source code or its components (functions):

- functional properties – to check the correctness of functions in accordance with their specifications;
- safety properties – to check and make sure that the functions and the program will not fail (undefined behavior, etc.);
- termination – to check the work termination process (at the moment when it is supposed to be).

Within this paper, it is proposed to consider a possibility of verifying security properties of the source code of access control tools using Frama-C.

II. Materials and Methods

Frama-C allows generating special comments to source code functions with a specification (contract of the function) – Hoare style pre- and postconditions, invariants (loop invariants, etc) [8], [9].

The Frama-C core parses the source code and each function contracts generating an intermediate representation with an ACSL functional specification (ANSI/ISO C Specification Language, specifying behavioral properties of C source code), from which an abstract syntax tree (AST) with ACSL annotations is subsequently generated. Such annotations contain verification conditions (VC), which are passed to special provers that check their validity. Based on the results of the work of such provers, a conclusion is made about compliance of the function implementation with its contract and the formal evidence of the source code correctness in terms of its compliance with the functional specification. Frama-C supports many plug-ins for various purposes of source code analysis [8], [9]:

- Eva – for abstract interpretation and value analysis;
- WP – for functional verification;
- E-ACSL – for runtime verification;
- MetACSL – for expressing high-level properties;
- SecureFlow – for information flow analysis;
- etc.

Moreover, using Frama-C, it is possible to use various provers by using the Why3[2] platform – Alt-Ergo, Z3, etc.

Initially, it may seem that only a low-level specification may be described in ACSL, that is, only some low-level properties for individual functions of the source code may be verified, without a possibility to verify high-level properties of some security model. However, security properties such as the Bell-LaPadula mandate policy rules may also be described at the specification level for several source code functions.

Let us consider a system consisting of subjects and objects using a synthetic ANCI C example, where subjects can perform read, write and create operations in relation to objects – Fig. 1.

In such a system, for the read, write, and create operations, we can implement the corresponding simplified restrictions mand_access_read and mand_access_write (Fig. 2) within the Bell-LaPadula mandate policy, after which we will need to verify compliance of the read, write, and create functions with this policy.

[2]http://why3.lri.fr

```
struct subject {
        int     uid;
        uint8_t access_level;
};

struct object {
        int     oid;
        uint8_t privacy_level;
};

int read(struct subject *s, struct object *o)
{
        if (!s || !o)
                return -1;
        return 0;
}

int write(struct subject *s, struct object *o)
{
        if (!s || !o)
                return -1;
        return 0;
}

int create(struct subject *s, struct object *o)
{
        if (!s || !o)
                return -1;
        return 0;
}
```

Fig. 1. A prototype of a system of abstract subjects and objects with read, write and create operations.

```
/** Mandatory NRU property */
#define mand_access_read(subj_label, obj_label) ((subj_label) >= (obj_label))

/** Mandatory NWD property (star-property) */
#define mand access write(subj label, obj label) ((subj label) <= (obj label))
```

Fig. 2. Mandate access control policy restrictions for system operations.

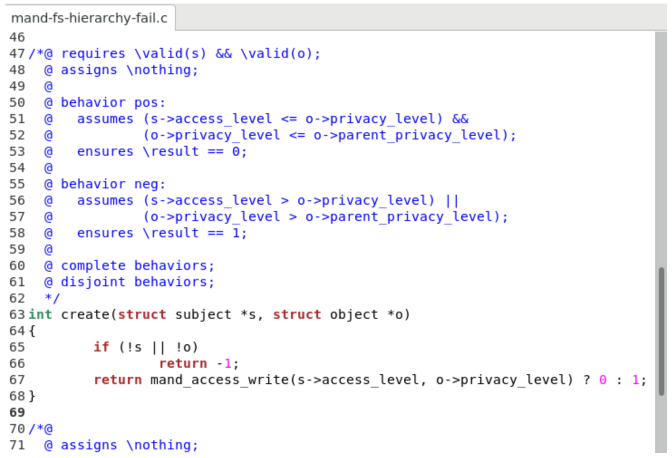
For verification purposes, it is required to create contracts for the read, write, create functions – Fig. 3.

Verification of such contracts using Frama-C (EVA/WP plug-in with the Alt-Ergo prover) will be successful. However, covert channel are often analyzed in case of the mandate security policy verification. Within such an analysis, the hierarchical structure of system objects (for example, the OS file system) can be taken into account. A classic example of a covert leakage channel for the Bell-LaPadula model is information leakage through the names of nested objects (if it is possible to create objects of a higher level of confidentiality in containers). If we add hierarchical logic to the system under consideration for objects and the create operation, as well as a new condition for the create operation contract in Fig. 4, verification will end in an error, as shown in Fig. 5.

To correct verification errors in a system with hierarchical

```
/*@ requires \valid(s) && \valid(o);
  @ assigns \nothing;
  @
  @ ensures (s->access_level <= o->privacy_level ==> \result == 0) &&
  @         (s->access_level >  o->privacy_level ==> \result == 1);
  */
int write(struct subject *s, struct object *o)
{
        if (!s || !o)
                return -1;
        return mand_access_write(s->access_level, o->privacy_level) ? 0 : 1;
}
```

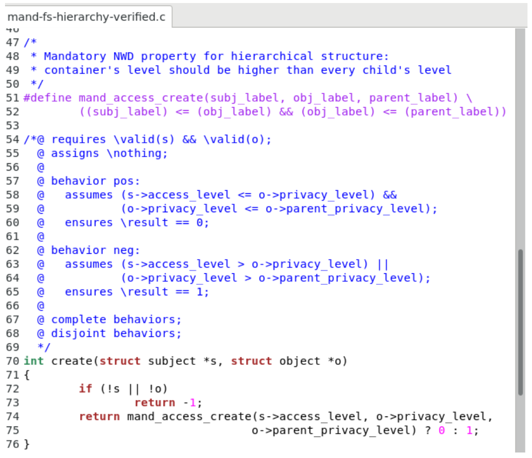
Fig. 3. Write function contract within the mandate access control policy.



mand-fs-hierarchy-fail.c

```
/*@ requires \valid(s) && \valid(o);
  @ assigns \nothing;
  @
  @ behavior pos:
  @   assumes (s->access_level <= o->privacy_level) &&
  @           (o->privacy_level <= o->parent_privacy_level);
  @   ensures \result == 0;
  @
  @ behavior neg:
  @   assumes (s->access_level > o->privacy_level) ||
  @           (o->privacy_level > o->parent_privacy_level);
  @   ensures \result == 1;
  @
  @ complete behaviors;
  @ disjoint behaviors;
  */
int create(struct subject *s, struct object *o)
{
        if (!s || !o)
                return -1;
        return mand_access_write(s->access_level, o->privacy_level) ? 0 : 1;
}

/*@
  @ assigns \nothing;
```

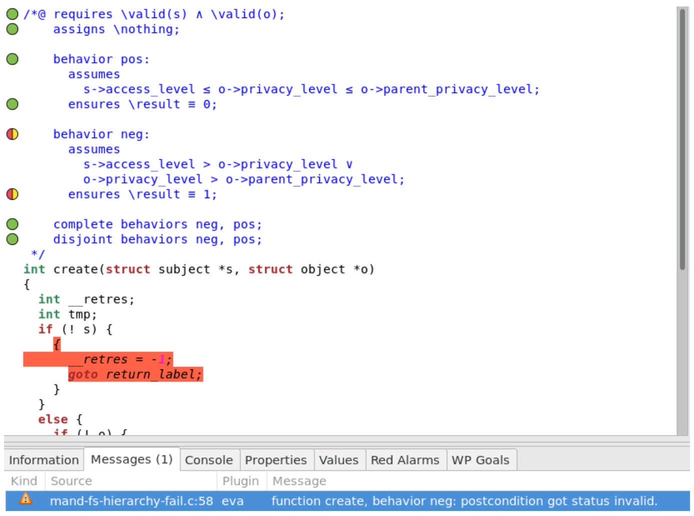
Fig. 4. Contract of the create function with the new conditions for hierarchical objects.



mand-fs-hierarchy-verified.c

```
/*
 * Mandatory NWD property for hierarchical structure:
 * container's level should be higher than every child's level
 */
#define mand_access_create(subj_label, obj_label, parent_label) \
        ((subj_label) <= (obj_label) && (obj_label) <= (parent_label))

/*@ requires \valid(s) && \valid(o);
  @ assigns \nothing;
  @
  @ behavior pos:
  @   assumes (s->access_level <= o->privacy_level) &&
  @           (o->privacy_level <= o->parent_privacy_level);
  @   ensures \result == 0;
  @
  @ behavior neg:
  @   assumes (s->access_level > o->privacy_level) ||
  @           (o->privacy_level > o->parent_privacy_level);
  @   ensures \result == 1;
  @
  @ complete behaviors;
  @ disjoint behaviors;
  */
int create(struct subject *s, struct object *o)
{
        if (!s || !o)
                return -1;
        return mand_access_create(s->access_level, o->privacy_level,
                                  o->parent_privacy_level) ? 0 : 1;
}
```

Fig. 6. Modified restriction for the create function which takes into account hierarchical logic.

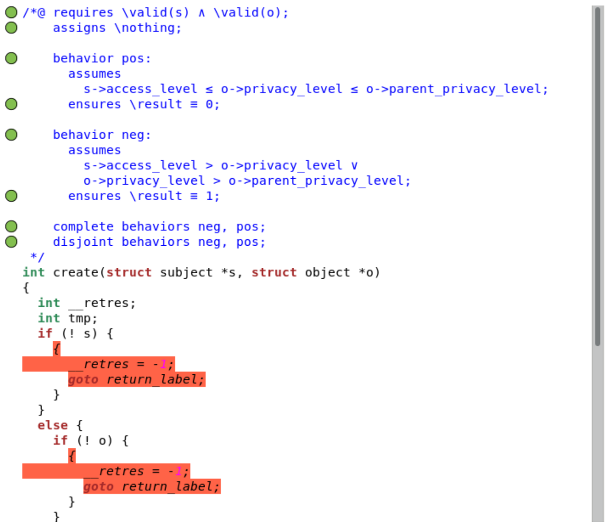


```
/*@ requires \valid(s) ∧ \valid(o);
    assigns \nothing;

    behavior pos:
      assumes
        s->access_level ≤ o->privacy_level ≤ o->parent_privacy_level;
      ensures \result ≡ 0;

    behavior neg:
      assumes
        s->access_level > o->privacy_level ∨
        o->privacy_level > o->parent_privacy_level;
      ensures \result ≡ 1;

    complete behaviors neg, pos;
    disjoint behaviors neg, pos;
 */
int create(struct subject *s, struct object *o)
{
  int __retres;
  int tmp;
  if (! s) {
    {
      __retres = -1;
      goto return_label;
    }
  }
  else {
```

Information | Messages (1) | Console | Properties | Values | Red Alarms | WP Goals

```
Kind  Source                    Plugin  Message
      mand-fs-hierarchy-fail.c:58  eva   function create, behavior neg: postcondition got status invalid.
```

Fig. 5. Create function verification error arisen as a result of using incorrect rules when creating objects in containers.



```
/*@ requires \valid(s) ∧ \valid(o);
    assigns \nothing;

    behavior pos:
      assumes
        s->access_level ≤ o->privacy_level ≤ o->parent_privacy_level;
      ensures \result ≡ 0;

    behavior neg:
      assumes
        s->access_level > o->privacy_level ∨
        o->privacy_level > o->parent_privacy_level;
      ensures \result ≡ 1;

    complete behaviors neg, pos;
    disjoint behaviors neg, pos;
 */
int create(struct subject *s, struct object *o)
{
  int __retres;
  int tmp;
  if (! s) {
    {
      __retres = -1;
      goto return_label;
    }
  }
  else {
    if (! o) {
      {
        __retres = -1;
        goto return_label;
      }
    }
```

Fig. 7. Successful verification of the create function with modified restriction.

logic, it is required to change the restriction for the create operation to mand_access_create – Fig. 6, after which the verification of function contracts will be successful – Fig. 7.

The above synthetic example is based on the source code of a real access control tool analyzed by the authors in [3], [4].

III. Results and Discussion

The paper is devoted to the approach to verifying security properties of access control tools at the source code level. Within such an approach, instead of creating a high-level specification independent of any source code, comments are added to some functions with their formal specification, which are used during the verification by Frama-C.

Such an approach to verifying access control tools allows to:

- Reduce the number of entities to be analyzed (certified) – instead of the source code and formal notation, only the source code with additional comments is used.
- Eliminate the need to justify the conformity of the source code to a certain formal notation.
- Eliminate the need to change a certain formal notation when changing the source code.

The proposed approach requires generation of a (low-level) specification at the source code function level. However, in such a case there will be no problems with modifying the source code, since in case of a discrepancy between the source code and the specification the verification process will end in an error. Function specifications must either change along with the source code change, or be corrected at the moment of the next verification.

In addition, when using Frama-C, one can additionally verify the absence of typical errors (memory leaks, buffer overflows, etc.) at the source code level, at a new qualitative level if compared to static analyzers.

When verifying, it is also necessary to take into account that access control tools usually use many standard library functions in their source code that can affect the result of their work. But thanks to the works of other authors [10], such functions have already been verified, so when verifying access control tools, it is possible to only check the correct operation of security functions.

Moreover, it is worth highlighting the dynamic verification capability of Frama-C (E-ACSL) [11]. When generating function specification, errors can occur that are difficult to detect in static analysis. In case of dynamic verification, special assert instructions are added to the source files during their assembly, which are triggered when the specification is violated. Thus, when the first violation is detected during dynamic verification, it becomes possible to identify and correct specification errors. Further research is expected to be done on this topic.

References

[1] J. McLean, R.R. Schell, D.L. Brinkley, "Security models," In Encyclopedia of Software Engineering, 2002, DOI:https://doi.org/10.1002/0471028959.sof297.

[2] International organization for standardization "ISO/IEC 15408-3. Information technology security techniques – Evaluation criteria for IT security – Part 3: Security assurance components," 2008.

[3] A.M. Kanner, T.M. Kanner, "Verification of a model of the isolated program environment of subjects using the Lamport's temporal logic of actions," Proceedings of the VII Engineering & Telecommunication Conference, 2020, pp. 1–5, DOI:https://doi.org/10.1109/EnT50437.2020.9431263.

[4] A.M. Kanner, T.M. Kanner, "Special features of TLA+ temporal logic of actions for verifying access control policies," Proceedings of Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology, 2021, pp. 411–414, DOI: https://doi.org/10.1109/USBEREIT51232.2021.9455090.

[5] A.V. Kozachok, "TLA+ based access control model specification," Proceedings of the Institute for System Programming of the RAS, vol. 30(5), 2018, pp. 147–162, DOI:https://doi.org/10.15514/ISPRAS-2018-30%285%29-9.

[6] L. Lamport, "The temporal logic of actions," ACM Trans. Program. Lang. Syst., vol. 16(3), 1994, pp. 872–923, DOI:http://doi.acm.org/10.1145/177492.177726.

[7] F. Kirchner, et. al, "Frama-C, a software analysis perspective," In Formal Aspects of Computing, vol. 27(3), 2015, pp. 573–609, DOI:https://doi.org/10.1007/s00165-014-0326-7.

[8] N. Kosmatov, J. Signoles, "Frama-C, a collaborative framework for C code verification: tutorial synopsis," In International Conference on Runtime Verification, 2016, pp 92–115, DOI:https://doi.org/10.1007%2F978-3-319-46982-9_7.

[9] J. Signoles, "The Frama-C framework and some applications to code security,“ Lecture Notes in Cyber in Saclay, 2021.

[10] D. Efremov, M. Mandrykin, A. Khoroshilov, "Deductive verification of unmodified Linux kernel library functions," In International Symposium on Leveraging Applications of Formal Methods, 2018, pp. 216–234, DOI:https://doi.org/10.1007/978-3-030-03421-4_15.

[11] K. Vorobyov, N. Kosmatov, J. Signoles, "Detection of security vulnerabilities in C code using runtime verification," In International Conference on Tests and Proofs, 2018, pp. 139–156, DOI:https://doi.org/10.1007/978-3-319-92994-1_8.