THALES
Building a future we can all trust

Université d'ORLÉANS | UFR Sciences & Techniques

Runtime Verification for High-Level Security Properties: Case Study on the TPM Software Stack

Yani ZIANI[1,2], Nikolai KOSMATOV[1],
Frédéric LOULERGUE[2], Daniel GRACIA PEREZ[1]

[1] Thales Research & Technology
[2] University of Orleans
www.thalesgroup.com

Sept 9, 2024 @ TAP 2024

Outline

> Context and Motivation

- Formal Verification of security properties of trusted layers of software
- Runtime verification with Frama-C
- Trusted Platform Module (TPM) and TPM Software Stack (TSS)

> Runtime Verification for High-Level Security Properties

- Verification methodology
- Companion Memory Model for Sensitive Data
- Defining and Verifying High-level Security Properties
- Summary of results
- Evaluation and key lessons

> Conclusion and Future Work

Formal verification of security properties of trusted software

> The Trusted Platform Module has become a key security component

- used by OS and applications through the TPM Software Stack (TSS)
- **tpm2-tss** is a popular open-source implementation of this stack

> Formal verification of the tpm2-tss library is important

- **vulnerabilities** could allow an attacker to recover sensitive data or take control of the system

> Motivation

- Proof of **global security properties in Frama-C**, with **MetAcsl and Wp**, on **large security-critical code** **[Djoudi et al. FM'21]**
  - Can be challenging on **large real-life code not designed for verification**
- Verification of **functional properties** and **absence of runtime errors** for a subset of functions of **tpm2-tss** involved in communications with the TPM **[ZIANI et al., iFM'23]**
  - **Several limitations** of deductive verification with Frama-C/WP identified (e.g. dynamic allocation, reasoning at byte-level)

Our goal: runtime verification of security properties of trusted software

> Explore an alternative approach: runtime verification for a set of test cases with Frama-C/E-ACSL

- **More features of C are supported** (e.g. the ability to reason at byte-level and dynamic allocation)

> Contributions of this work

- runtime verification of high-level properties in **tpm2-tss** using the Frama-C platform
- case study on a function call on a high-level layer to a TPM command
- **Integrity** and **confidentiality** of sensitive data verified
- proposed **methodology** for the verification high-level **security properties** over **sensitive data**

> Target application

- **Secure import** of an object onto the TPM using the TSS (TPM Software Stack)
- How to specify and verify security properties on **real-life** safety-critical code ?
- Code not written with verification in mind

Frama-C Verification Platform

> Plugin-based open-source verification platform for C code analysis:

- ACSL (ANSI/ISO C Specification Language) to specify functional properties of programs
- Developed by CEA List

> Wp plugin for deductive verification

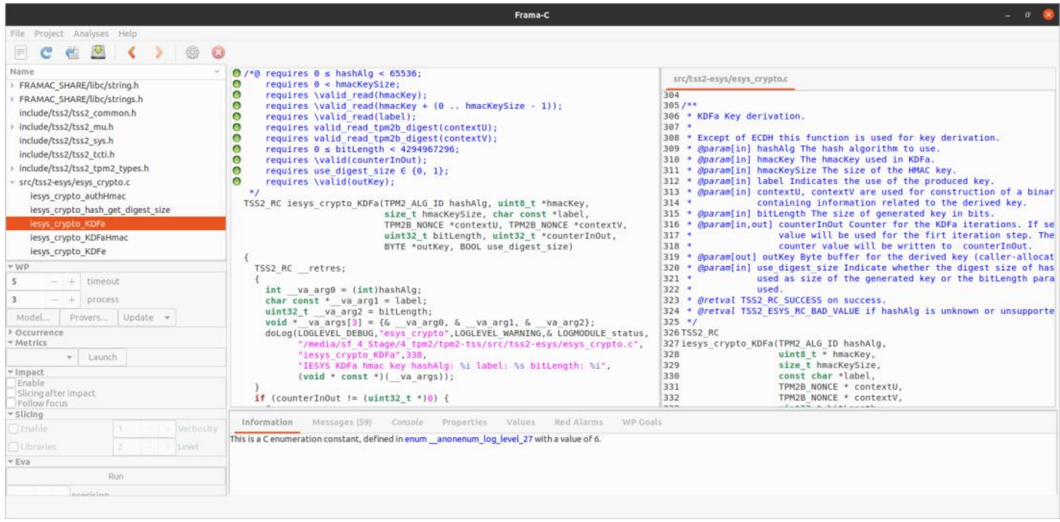
- **Formal verification** of functional properties
- **Generates proof obligations** to be proved by solvers
- Recognized by ANSSI for highest levels of certification

> E-ACSL plugin for runtime verification

- Translates **ACSL** properties into **executable code**

> MetAcsl plugin for high-level and global properties

- Translates them into low-level annotations, to be verified by other tools



```
/*@ requires 0 ≤ hashAlg < 65536;
    requires 0 < hmacKeySize;
    requires \valid_read(hmacKey);
    requires \valid_read(hmacKey + (0 .. hmacKeySize - 1));
    requires \valid_read(label);
    requires valid_read_tpm2b_digest(contextU);
    requires valid_read_tpm2b_digest(contextV);
    requires 0 ≤ bitLength < 4294967296;
    requires \valid(counterInOut);
    requires use_digest_size ∈ {0, 1};
    requires \valid(outKey);
 */
TSS2_RC iesys_crypto_KDFa(TPM2_ALG_ID hashAlg, uint8_t *hmacKey,
                          size_t hmacKeySize, char const *label,
                          TPM2B_NONCE *contextU, TPM2B_NONCE *contextV,
                          uint32_t bitLength, uint32_t *counterInOut,
                          BYTE *outKey, BOOL use_digest_size)
{
  TSS2_RC __retres;
  {
    int __va_arg0 = (int)hashAlg;
    char const *__va_arg1 = label;
    uint32_t __va_arg2 = bitLength;
    void *__va_args[3] = {& __va_arg0, & __va_arg1, & __va_arg2};
    doLog(LOGLEVEL_DEBUG,"esys_crypto",LOGLEVEL_WARNING,& LOGMODULE_status,
          "/media/sf_4_Stage/4_tpm2/tpm2-tss/src/tss2-esys/esys_crypto.c",
          "iesys_crypto_KDFa",338,
          "IESYS KDFa hmac key hashAlg: %i label: %s bitLength: %i",
          (void * const *)(__va_args));
  }
  if (counterInOut != (uint32_t *)0) {
```

```

/**
 * KDFa Key derivation.
 *
 * Except of ECDH this function is used for key derivation.
 * @param[in] hashAlg The hash algorithm to use.
 * @param[in] hmacKey The hmacKey used in KDFa.
 * @param[in] hmacKeySize The size of the HMAC key.
 * @param[in] label Indicates the use of the produced key.
 * @param[in] contextU, contextV are used for construction of a binar
 *            containing information related to the derived key.
 * @param[in] bitLength The size of generated key in bits.
 * @param[in,out] counterInOut Counter for the KDFa iterations. If se
 *                value will be used for the firt iteration step. The
 *                counter value will be written to  counterInOut.
 * @param[out] outKey Byte buffer for the derived key (caller-allocat
 * @param[in] use_digest_size Indicate whether the digest size of has
 *            used as size of the generated key or the bitLength para
 *            used.
 * @retval TSS2_RC_SUCCESS on success.
 * @retval TSS2_ESYS_RC_BAD_VALUE if hashAlg is unknown or unsupporte
 */
TSS2_RC
iesys_crypto_KDFa(TPM2_ALG_ID hashAlg,
                  uint8_t * hmacKey,
                  size_t hmacKeySize,
                  const char *label,
                  TPM2B_NONCE * contextU,
                  TPM2B_NONCE * contextV,
```

This is a C enumeration constant, defined in enum __anonenum_log_level_27 with a value of 6.

TPM (Trusted Platform Module)

> Standard for a secure cryptoprocessor:

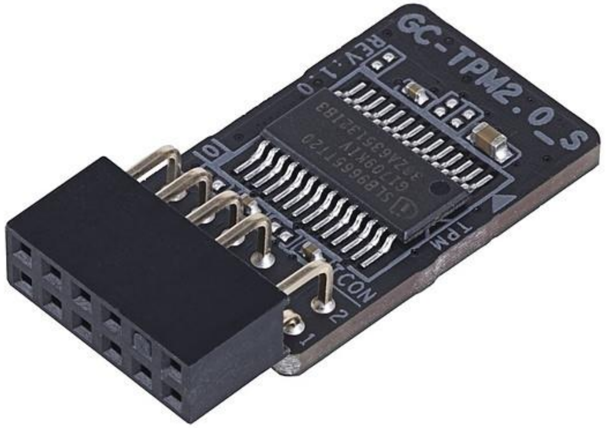
- Platform integrity (during boot), disk encryption (dm-crypt, Bitlocker), protection and enforcement of software licenses

> Different types of TPM 2.0 implementations

- Discrete, Integrated, Firmware, Hypervisor, Software (implemented by several vendors : Infineon, ST, etc)

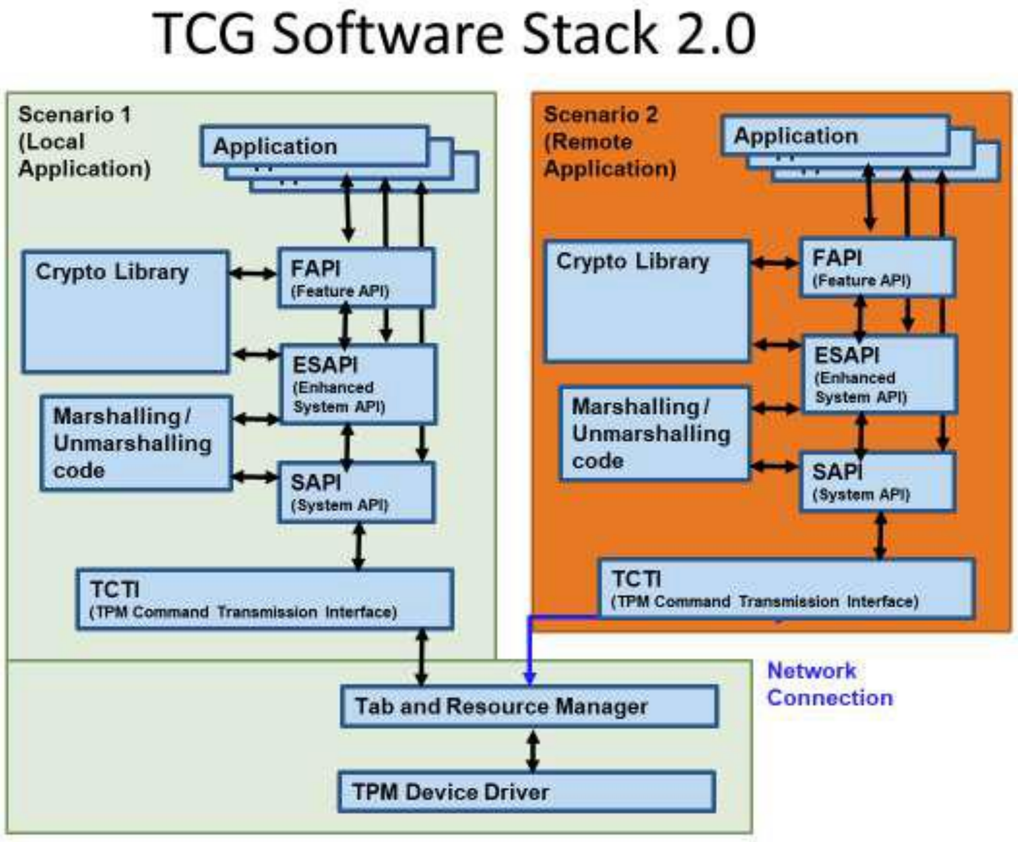
> TPM2 Software Stack (TSS) :

- Specification by the TCG, providing an API/access layer
- Several open-source libraries
- **Goal** : tpm2-tss (by the tpm2-software community)
- **Target**: import of a sensitive information (e.g. an encryption key) onto the TPM, from higher-level layers (`TPM2_Create` TPM command, `Esys_Create` function on ESAPI layer)



TPM Software Stack

- **FAPI (Feature API) :** designed to capture most common use cases TPM *(tss2-fapi)*
- ***ESAPI (Enhanced System API)* :** Session management, support for cryptographic capabilities ***(tss2-esys)***
- ***SAPI (System API)* :** access to all the functionnality of the TPM ***(tss2-sys)***
- ***TCTI (TPM Command Transmission Interface) (tss2-tcti)***
- **TAB (TPM Access Broker) & Resource Manager**
- **Device Driver**
- **TPM**



Related Work

> TPM related safety and security

- Formal analysis of key exchange [Zhang & Zhao, 2020]
- Proof of cryptographic support using CryptoVerif [Wang et al., 2016]
- Analysis of HMAC authorization [Shao et al., 2018]
- Study of usability and security of TPM library APIs [Rao et al., 2022]

> Formal verification of high-level properties and real-life code

- Study of the correctness of OpenJDK's TimSort using the KeY tool [de Gouw et al., 2015]
- Verification of traffic tunnel control system verification software with VerCors [Oortwijn et al., 2019]
- Verification of a TCP stack using SPARK and KLEE [Cluzel et al., 2021]
- Proof of security properties on the JavaCard Virtual Machine with Frama-C [Djoudi et al., 2021]
- Deductive Verification of Smart Contracts with Dafny [Cassez et al., 2022]
- …

Outline

> Context and Motivation

- Formal Verification of security properties of trusted layers of software
- Runtime verification with Frama-C
- Trusted Platform Module (TPM) and TPM Software Stack (TSS)

> Runtime Verification for High-Level Security Properties

- Verification methodology
- Companion Memory Model for Sensitive Data
- Defining and Verifying High-level Security Properties
- Summary of results
- Evaluation and key lessons

> Conclusion and Future Work

Verification methodology

1. **Define the memory representation to be used for sensitive data**

2. **Identify the target pieces of sensitive data, and add them into the model**
   - data at a high-level of abstraction
   - data whose security has to be ensured

3. **Define security properties over sensitive data**
   - e.g. integrity and confidentiality as previously shown

4. **Run verification with MetAcsl and E-ACSL**
   - If not defined, define a main function/entry point
   - Parse with MetAcsl, instrument with E-ACSL, compile with E-ACSL/GCC, execute the output

5. **Use the verification results to iteratively refine the previous definitions**
   - A detected violation of integrity (resp. confidentiality) indicates either an “illegal” write (resp. read), or that a sensitive data should not be considered as sensitive at that point, or a “legal” write (resp. read) not yet handled by the current definitions

6. **Repeat Steps 4 and 5**
   - Until all detected violations of integrity or confidentiality correspond to security flaws

Companion Memory Model for Sensitive Data

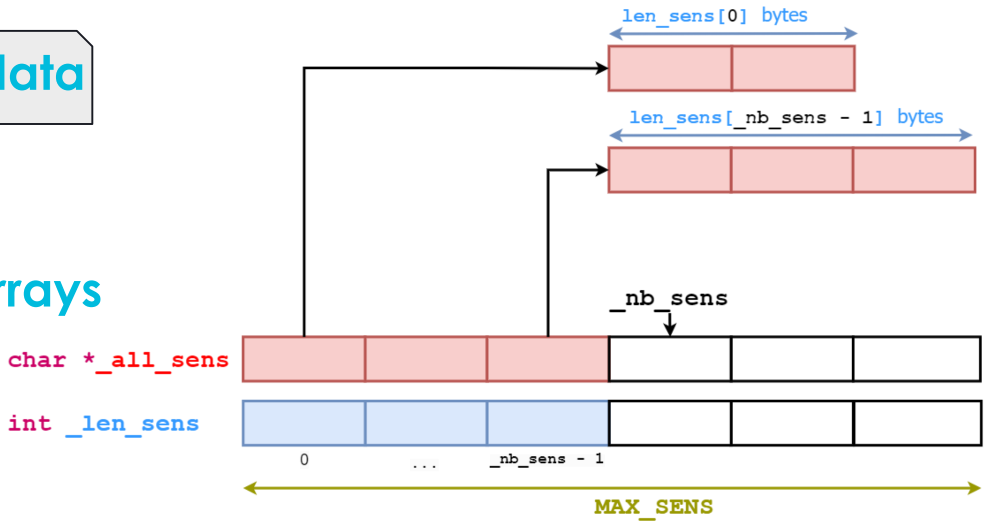
> Step 1: Define memory representation to be used for sensitive data

> Memory representation of the companion model with global arrays

- **_all_sens** used to store addresses of pieces of sensitive data
- **_len_sens** used to store their size in memory in bytes
- **_nb_sens** used as a tracking index of the next available slot
- 3 helper functions
  - **remove_sens** to remove the piece of data at a given index
  - **add_as_sens** to add a piece of data into the model
  - **is_sens** to check if a given address and size correspond to a recorded data in the model

> Target subset of functions

- A simplified integration test for Esys_Create (to import an object onto the TPM without parameter encryption)
- Removed dependencies to other calls to TPM commands, to external libraries, and simplified the command transmission interface



```
#define MAX_SENS 100
int _len_sens[MAX_SENS];
char * _all_sens[MAX_SENS];
int _nb_sens;
// define _sens_##data##_idx
int _sens_exData1_idx = -1;
int _sens_exData2_idx = -1;
int _sens_exData3_idx = -1;
bool is_sens(void *ptr, int size, int idx)
void remove_sens(int idx){_len_sens[idx] = 0;}
int add_as_sens(void *ptr, int size)
{
  int ret_idx;
  if(_nb_sens ≥ MAX_SENS ∨ _nb_sens < 0) {ret_idx =-1;}
  else {
    _all_sens[_nb_sens] = (char*) (ptr);
    _len_bank[_nb_sens] = size;
     idx = _nb_sens++; }
  return ret_idx;
}
```

THALES
Building a future we can all trust

Defining and Verifying High-level Security Properties

> Step 2: Identify sensitive data whose security has to be ensured, and add them into the model.

> Common global view of sensitive information

- tpm2-tss avoids the usage of global state variables
- Necessary to render the data visible at a global level for MetAcsl properties

> Identify and add sensitive data to the model

- Assuming the sensitive data `inSensitive` is already in the representation, any copy of said data should be added as well
- `store_input_parameters` is a tpm2-tss function that copies the data into the `esysContext` context
- We add the data copied into the `esysContext` to the representation

- `_sens_exData1_idx` to keep track

```c
/** Store sensitive data inside the ESYS_CONTEXT */
static void store_input_parameters (
    ESYS_CONTEXT *esysContext, const TPM2B_SENSITIVE_CREATE *inSensitive)
{
  ...
    esysContext->in.Create.inSensitiveData = *inSensitive;
    esysContext->in.Create.inSensitive =
      &esysContext->in.Create.inSensitiveData;
    /* We add the part of context containing the imported sensitive
       object to our representation of sensitive data. */
     _sens_exData1_idx =
     add_as_sens( esysContext->in.Create.inSensitive,
            (int) sizeof(esysContext->in.Create.inSensitiveData) );
  }
}
```

```c
#define MAX_SENS 100
int _len_sens[MAX_SENS];
char * _all_sens[MAX_SENS];
int _nb_sens;
// define _sens_##data##_idx
int _sens_exData1_idx = -1;
int _sens_exData2_idx = -1;
int _sens_exData3_idx = -1;
bool is_sens(void *ptr, int size, int idx)
void remove_sens(int idx){_len_sens[idx] = 0;}
int add_as_sens(void *ptr, int size)
{
  int ret_idx;
  if(_nb_sens ≥ MAX_SENS ∨ _nb_sens < 0) {ret_idx =-1;}
  else {
    _all_sens[_nb_sens] = (char*) (ptr);
    _len_bank[_nb_sens] = size;
     idx = _nb_sens++; }
  return ret_idx;
}
```

THALES
Building a future we can all trust

Defining and Verifying High-level Security Properties

> **Step 2: Identify sensitive data whose security has to be ensured, and add them into the model.**

> **Identify and add sensitive data to the model on <u>lower-level layers</u>**

- Assuming the sensitive data `inSensitive` is already in the model, any copy of said data should be added as well
- `Tss2_MU_TPM2B_SENS_CREATE_Marshal` is a TSS function that copies the data into a byte buffer in the `sysContext` context
- We add the data copied into the `sysContext` to the model
  - The address of the data in the buffer is given by **`ctx->cmdBuffer + sens_offset`**
  - The size is computed using the **`size`** subfields, following the TCG specification
- `_sens_exData2_idx` to keep track

```c
TSS2_RC Tss2_Sys_Create_Prepare(
  TSS2_SYS_CONTEXT *sysContext, const TPM2B_SENS_CREATE *inSensitive, ...)
{
...
    size_t sens_offset = ctx->nextData;
    rval = Tss2_MU_TPM2B_SENS_CREATE_Marshal(
      inSens, ctx->cmdBuffer, ctx->maxCmdSize, &ctx->nextData);
  /* We add the part of the command buffer containing the imported
    sensitive object to our representation of sensitive data. */
  int sys_sens_size = (int) sizeof(inSensitive->size) +
                 (int) sizeof(inSensitive->sens.auth.size) +
                        (int) inSensitive->sens.auth.size  +
                 (int) sizeof(inSensitive->sens.data.size) +
                        (int) inSensitive->sens.data.size;
 _sens_exData2_idx =
      add_as_sens(ctx->cmdBuffer + sens_offset, sys_sens_size);
}
```

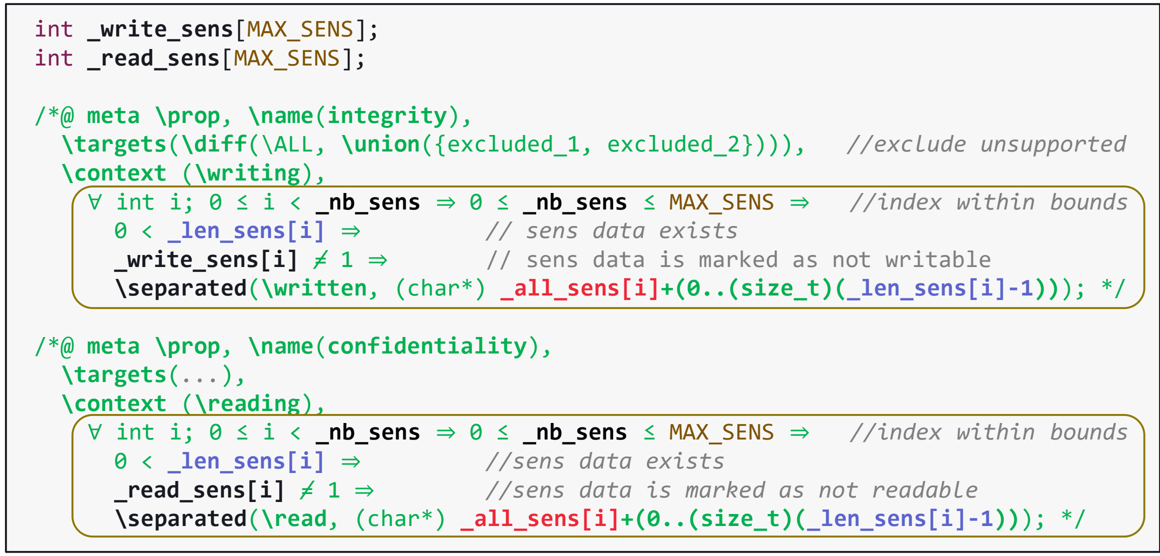
```c
#define MAX_SENS 100
int _len_sens[MAX_SENS];
char * _all_sens[MAX_SENS];
int _nb_sens;
// define _sens_##data##_idx
int _sens_exData1_idx = -1;
int _sens_exData2_idx = -1;
int _sens_exData3_idx = -1;
bool is_sens(void *ptr, int size, int idx)
void remove_sens(int idx){_len_sens[idx] = 0;}
int add_as_sens(void *ptr, int size)
{
  int ret_idx;
  if(_nb_sens ≥ MAX_SENS ∨ _nb_sens < 0) {ret_idx =-1;}
  else {
    _all_sens[_nb_sens] = (char*) (ptr);
    _len_bank[_nb_sens] = size;
     idx = _nb_sens++; }
  return ret_idx;
}
```

Defining and Verifying High-level Security Properties

> Step 3: Define security properties

> Defining integrity and confidentiality

- `_write_sens` and `_read_sens` arrays used to determine whether a piece of sensitive data can be written or read
- We define integrity as the separation between written location (`\written`) and any non-writable sensitive data.
- We define confidentiality as the separation between any read location (`\read`) and any non-readable sensitive data.
- Properties defined as MetAcsl global properties :
  - the `\name` provides a name
  - the `\targets` defines the set of functions in which the property shall be verified
  - `\context(\writing)` (resp. `\context(\reading)`) means the property must hold whenever a memory location is written (resp. read)



```
int _write_sens[MAX_SENS];
int _read_sens[MAX_SENS];

/*@ meta \prop, \name(integrity),
  \targets(\diff(\ALL, \union({excluded_1, excluded_2}))),   //exclude unsupported
  \context (\writing),
    ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒    //index within bounds
      0 < _len_sens[i] ⇒          // sens data exists
      _write_sens[i] ≠ 1 ⇒        // sens data is marked as not writable
      \separated(\written, (char*) _all_sens[i]+(0..(size_t)(_len_sens[i]-1))); */

/*@ meta \prop, \name(confidentiality),
  \targets(...),
  \context (\reading),
    ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒    //index within bounds
      0 < _len_sens[i] ⇒          //sens data exists
      _read_sens[i] ≠ 1 ⇒         //sens data is marked as not readable
      \separated(\read, (char*) _all_sens[i]+(0..(size_t)(_len_sens[i]-1))); */
```

Defining and Verifying High-level Security Properties

> Step 4: Run verification with MetAcsl and E-ACSL

- If not defined, define a main function/entry point
- Parse with MetAcsl, instrument with E-ACSL, compile with GCC, execute the output

> Step 5: Use verification results to refine previous definitions

- A detected violation of integrity (resp. confidentiality) indicates either:
  - an “illegal” write (resp. read)
  - that a sensitive data should not be modeled at the reported program point,
  - or a “legal” write (resp. read) not yet handled by the current definitions

> Refining the handling of sensitive data

- The sensitive data `**outPrivate` is in the model before the call to `free`
- `*outPrivate` should be removed from the model before being freed
- `_sens_exData3_idx` to keep track

```
if (outPrivate != NULL){
    if((*outPrivate) != NULL) {
      /*remove the sensitive data from the model before freeing*/
      if(is_sens(*outPrivate,
            (int) sizeof(TPM2B_PRIVATE),
            _sens_exData3_idx))
        remove_sens(_sens_exData3_idx);
      free((void*) (*outPrivate));
      (*outPrivate)=NULL;
    }
  }
```

```
#define MAX_SENS 100
int _len_sens[MAX_SENS];
char * _all_sens[MAX_SENS];
int _nb_sens;
// define _sens_##data##_idx
int _sens_exData1_idx = -1;
int _sens_exData2_idx = -1;
int _sens_exData3_idx = -1;
bool is_sens(void *ptr, int size, int idx)
void remove_sens(int idx){_len_sens[idx] = 0;}
```

Summary of results

- **Proposed methodology for specification of:**
  - a **shared representation** of sensitive data **usable for MetAcsl** based approaches
  - properties expressing that sensitive data is **never modified**, **never read** when it is not supposed to

- **Successful verification on a (simplified) import operation with TPM2_Create command/Esys_Create function:**
  - 86 functions operations involved in the import operation:
    - 36 “unique” internal TSS operations, 50 marshal functions
  - Approximately 20k lines of code, (12k interfaces, 8k actual function implementations, marshal defined as non unfolded macros)
  - Target : high-level function call simplified by removing dependencies to external libraries such as OpenSSL calls, and replaced the TCTI with a dummy, and no prior or subsequent communications with the TPM
  - Verified both the integrity and the confidentiality of target pieces of sensitive data

Evaluation

> RQ1: Expressiveness

- Our approach renders sensitive data visible at a global level
  - usable for the definition of high-level security properties with MetAcsl, extending its capabilities
- Verifiable with other Frama-C plug-ins such as Wp or E-ACSL.

> RQ2: Effectiveness

- Requires a much smaller specification effort on real-life code than that of deductive verification for ACSL properties
- Goals regarding memory separations are much easier to verify, while deductive verification can require a lot of intermediary specifications

> RQ3: Efficiency

- Processing times for the target code (approx. 19 min) are considerably shorter than what *would* be required for proof with Wp
- Lesser specification effort

> Threats to validity

- E-ACSL has its own limitations wrt. the support of certain parts of ACSL
- Previous conclusions may not hold on a different target, codes with linked data structures or with external dependencies

Outline

> Context and Motivation

- Formal Verification of security properties of trusted layers of software
- Runtime verification with Frama-C
- Trusted Platform Module (TPM) and TPM Software Stack (TSS)

> Runtime Verification for High-Level Security Properties

- Verification methodology
- Companion Memory Model for Sensitive Data
- Defining and Verifying High-level Security Properties
- Summary of results
- Evaluation and key lessons

> Conclusion and Future Work

Main Achievements

- Advanced verification case study for a complex security critical library
- Aimed to extend the capabilities of MetAcsl for the definition of global properties
- Proposed verification methodology of high-level security properties at runtime
- Some tool limitations were identified
  - temporary code simplifications were proposed
  - simplifications should become unnecessary after tool extensions
- Our verification approach is readily available for extension

Artifacts Available EAPLS V1.1

Artifacts Evaluated EAPLS Functional V1.1

Artifact available at:

bitly

Ongoing and future work

> Extend the range of verified properties

- Currently, only the integrity and the confidentiality of sensitive data were verified

> Extend the verification for a larger perimeter of code

- Consider other critical features and functions
- By reintroducing cryptographic capabilities of the TSS, and running the code with a real or simulated TPM

> Improve the automation of the approach

> Combined approaches

- For instance, combine the deductive verification of Wp with the runtime verification of E-ACSL to take the best of both worlds
- To perform a more thorough and more complete verification of high-level security properties.