Methodology for Specification and Verification of High-Level Requirements with MetAcsl

Virgile Robles*, Nikolai Kosmatov*†, Virgile Prevosto*, Louis Rilling‡ and Pascale Le Gall§
* Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
firstname.lastname@cea.fr
† Thales Research & Technology, Palaiseau, France
nikolaikosmatov@gmail.com
‡ DGA, France, louis.rilling@irisa.fr
§ Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupélec, Université Paris-Saclay, Gif-Sur-Yvette, France
pascale.legall@centralesupelec.fr

***Abstract*—Specification and formal verification of high-level properties (such as security properties, like data integrity or confidentiality) over a large software product remains an important challenge for the industrial practice. Recent work introduced METACSL, a plugin of the FRAMA-C verification platform, that allows the user to specify high-level properties, called HIgh-Level ACSL REquirements or HILARE, for C programs and transform them into assertions that can then be verified by classic deductive verification. This paper presents a methodology of specification and verification of a wide range of high-level properties with METACSL and illustrates it on several examples. The goal is to provide verification practitioners with detailed methodological guidelines for common patterns of properties in order to facilitate their everyday work and to avoid some frequent pitfalls. The illustrating examples are inspired by very usual kinds of properties and illustrated on two use cases. One of them—on the real-life code of the bootloader module of the secure storage device Wookey—was fully verified using the described approach, demonstrating its capacity to scale to real-life code. The other one—on a microkernel of an OS—was added to illustrate other common properties, where the description of the system was intentionally left very generic.**

I. INTRODUCTION

The last two decades have seen significant progress in deductive verification of programs. *Deductive verification* is a source-code-based verification technique that allows users to formally prove that a given program satisfies a set of specified requirements. Such requirements are usually formalized as program annotations in the form of *function contracts*, including pre- and post-conditions for each function. A deductive verification tool can be applied to prove that each function satisfies its contract. For C programs, it can be done using for instance the WP plugin of the FRAMA-C verification platform [1], the requirements being formalized with the ACSL language [2].

While this technique has proved to be suitable to specify and verify many functional properties, it is less suitable to address high-level program properties, such as security requirements like data integrity or confidentiality. Software security has become a major concern today, and the assessment of security properties on critical software even more so. Yet formal specification and verification of such properties over a large software product remains an important challenge for the industrial practice.

For instance, in a microkernel entrusted with the management of tasks and their respective memory regions, a desirable property for the integrity of data is that the content of a region can be modified only when its owner is the currently executed task. When expressed as function contracts, such properties necessarily span over many functions, and must be encoded into the contracts of each of them. Given the annotated code, it remains difficult for the user to ensure that those properties were correctly specified in all relevant functions. In some cases, such properties can also be difficult to express as function contracts. An example is a confidentiality property in a microkernel that a memory region can be read only when its owner is the current task. More generally, stating constraints on all read and write operations or function calls would amount to peppering the code with many annotations, an extremely tedious and error-prone task if done manually.

Recent work [3, 4] proposed METACSL, a FRAMA-C plugin[1] that allows the user to specify high-level properties, called HIgh-Level ACSL REquirements (HILARE)[2], and automatically transform them into local annotations, effectively automating the annotation peppering. A HILARE basically defines three main components: a predicate to be instantiated as local annotations, a target set of functions in which it should be done, and a context defining the situations (e.g. reading or writing operations) in which this instantiation must be performed. The resulting annotations can then be verified by classic deductive verification, using the WP plugin. The purpose of this paper is to present a methodology of specification and verification of a wide range of high-level requirements with METACSL and to illustrate it on several examples. The goal is to provide verification practitioners with detailed methodological guidelines for various common patterns of properties in order to facilitate their specification and verification. The provided patterns can be followed by

[1] Available at https://git.frama-c.com/pub/meta
[2] Referred to as *metaproperties* in [3, 4]

less experienced verification engineers to avoid logical errors in the specification of a HILARE. We also emphasize some good practices showing how to avoid some frequent pitfalls.

The provided examples are inspired by very frequent kinds of properties and illustrated on two security-relevant use cases. The first one, based on the bootloader module of the secure storage device WOOKEY[3] [5], was fully verified using the described approach, demonstrating its capacity to scale to real-life code. The second one, on a microkernel of an operating system (OS), is only sketched and was included to illustrate other common properties, where the description of the system was intentionally left generic.

Contributions. The contributions of the paper include:

- a methodology for specification and verification of high-level requirements based on METACSL, including a rich set of common property patterns,
- their application to the verification of a critical module of the secure storage device WOOKEY, whose security properties were fully verified using METACSL,
- an illustration of their application to a microkernel.

Outline. Section II gives a brief overview of ACSL and FRAMA-C. Then, in Section III, we lay down a general methodology for dealing with high-level requirements with HILARE. Section V describes and illustrates several commonly used specification patterns. It relies on two use-cases, that are presented in Section IV and then studied further in Section VI to showcase how the presented patterns can be articulated for complex properties. General guidelines related to the proof of such properties are then mentioned in Section VII. Finally, Sections VIII and IX provide related work and a conclusion.

II. BACKGROUND

We briefly recall in this section the main features of ACSL, upon which HILARE and METACSL are relying. An ACSL [2] annotation is primarily a first-order logic formula that is supposed to hold for any program state reaching some specific point in the program. Notably, an ACSL function contract for a function `f` will specify the *pre-condition* of `f`, that is, the properties that `f` requires from its caller, as well as the *post-condition* of `f`, that is, the properties that `f` ensures when it returns control back to its caller. In addition, it is often important to specify the footprint of `f`, i.e. the set of memory locations that the function might assign to during its execution, either directly or through calls to other functions. Apart from contracts, another important kind of annotation is the assertion, which allows specifying a property locally at a given point within a function body. Finally, ACSL formulas are based on pure C expressions (i.e. without side-effects), as well as logic functions and predicates, either built-ins (notably to speak about pointers) or user-defined. For convenience of the reader, Figure 2 summarizes the meaning of ACSL (and HILARE) constructs used in this paper. For

[3]Available at https://github.com/wookey-project

```
1  int A, B, C;
2
3  /*@ requires validity: \valid(p) ∧ \valid(q);
4      requires separation: \separated(p,q);
5      assigns *p;
6      ensures copied: *p == *q; */
7  void copy(int* p, const int* q) { *p = *q; }
8
9  /*@ requires A == B;
10     assigns A,B;
11     ensures C ≥ 0 ∧ A == C ∧ B == C ∨
12             C < 0 ∧ A == \old(A) ∧ B == \old(B); */
13 void foo(){
14   if ( C ≥ 0 ){
15     copy(&A,&C);
16     copy(&B,&C);
17     /*@ assert same: A == B; */
18   }
19 }
```

Figure 1: Example of C code with ACSL specification

Syntax	Semantics
`&&  ||  ==>  !`	Logical operators
`\forall var; P`	Universal quantification
`\exists var; P`	Existential quantification
`\diff(S1,S2)  \union(S1,S2)`	Set operators $S1 \setminus S2$, $S1 \cup S2$
`\ALL`	Set of all functions
`\at(loc,lab)  \old(loc)`	Value of location at label
`\valid(p)`	Pointer validity
`\separated(loc1, loc2)`	Memory separation
`\overlaps(loc1, loc2)`	Memory overlapping
`\tguard(P)  \fguard(P)`	Guards against typing errors
`\formal(param)`	Refers to function parameter
`\written  \read  \called`	Refers to constrained object

Figure 2: ACSL and HILARE terms and predicates

example, **`\separated`**`(p,q)` indicates that memory locations referred to by pointers `p` and `q` are disjoint. To indicate that two locations are not disjoint, in this work we denote by **`\overlaps`**`(p,q)` the negation of **`\separated`**`(p,q)`. Note that in the code examples in this paper, some C and ACSL constructs (e.g. `>=`, `||`, `&&`, `\forall`, `==>`) are pretty-printed using mathematical notation (e.g. $\geq$, $\vee$, $\wedge$, $\forall$, $\Rightarrow$).

To make things more palatable, let us have a look at the small example shown in Figure 1. It contains two functions with their ACSL contracts. Concretely, ACSL annotations are C comments, beginning with the character `@`.

The contract for `copy`, in lines 3-6, first gives two pre-conditions (given in **`requires`** clauses). Note that any ACSL formula may be given one or several names followed by a column "`:`", which are simply identifiers added for readability and traceability purposes. The `validity` pre-condition indicates that `copy` expects to be given two *valid* pointers, i.e., it must be able to dereference `p` and `q` safely. The `separation` requirement indicates that `p` and `q` must point to two disjoint locations of the memory. Then, the **`assigns`** clause explains that `copy` does not modify anything except the location pointed to by `p`. Finally, `copy` ensures that when it returns, the value of `*p` is equal to the one stored in `*q` (which, as can be deduced by `separation` and the **`assigns`** clause, is left untouched).

The contract for `foo` requires that it starts in a state where global variables `A` and `B` are equal, and says that it may

overwrite both `A` and `B`. Its post-condition is a disjunction, which says that if `C` is non-negative, `A` and `B` get assigned the value of `C`, and if `C` is negative, `A` and `B` are left untouched. The `\old(A)` construct indeed refers to the value of `A` in the pre-state of the contract, allowing the post-condition to relate the values of the post-state and the pre-state. Finally, inside the body of `foo`, we **`assert`** that after the two `copy`, `A` and `B` have the same value.

Within the FRAMA-C framework, verification that the code is correct with respect to its ACSL specification can be done through several means, notably the WP and E-ACSL plugins. The former relies on deductive verification, and generates a set of proof obligations that are sent to automated provers. On the example of Figure 1, command `frama-c -wp -wp-rte example.c` instructs WP to attempt proving the specification itself as well as the absence of runtime errors and immediately succeeds in proving the 12 proof obligations that are generated. The E-ACSL plugin can be used to instrument the code so that each annotation can be checked at runtime, e.g. to be assessed against a test suite, as further explained in [6, 4].

III. GENERAL METHODOLOGY

When confronted with a verification problem that seemingly involves large parts of a code base, using METACSL and its HILARE specification language might be an efficient and expressive way to encode the desired properties with reasonable effort. With experience, the specification and verification process with METACSL usually follows a recurring pattern, which might be useful for new users to know about.

Ensuring the problem is within the scope of METACSL

While it is possible to encode various categories of properties with HILARE, some of them are outside of its scope and could be better addressed by more suitable tools.

Thus, one should check that a desired property:

- *does not relate multiple execution traces*. Indeed if the specification involves the comparison of multiple execution traces, it is a relational property. METACSL is not intended to deal with such properties.[4]
- *is not overly concerned with the order of execution*. While HILARE can be used to write simple temporal properties (e.g. relating two consecutive states of the program), complex properties describing the behavior of a program over time could be better treated with other FRAMA-C plugins.[5]
- *can somehow be reduced to a property on the global state*. Since METACSL is meant to express high-level requirements, properties can only be expressed over data that is visible in the global scope. While there are some facilities to pry into the state of individual functions when necessary, a property that is overly specific to a local state might be hard to verify, or even to specify, as a HILARE. In fact, a plain ACSL annotation would probably be sufficient in this case.

[4]One could use e.g. the dedicated RPP plugin for that purpose.

[5]Such as CAFE or AORAÏ.

Identifying the working subset of the global state

As mentioned before, HILARE can only express properties over data that are visible at global level. Hence, it is necessary to identify where to anchor the properties in the global state. There are several such "anchor points":

Global variables This is the simplest and most desirable way. Global variables can just be referred to by name in any HILARE.

Common parameters Sometimes data are not global but passed around as a parameter in a large number of functions. If the naming of the formal parameter is consistent across those functions, it can be referred to.

Heap data Memory dynamically allocated on the heap[6].

Once the relevant elements of the state of the program have been identified and made available, the properties themselves can be formulated as HILARE. The set of variables (or more generally, memory locations) used to specify a property is referred to as its *memory footprint*.

Formulating the problem as HILARE

One should first identify the *target set* of the property: what set of functions should uphold the requirement. It is often helpful to define named function sets using C macros, and to compose them with set operators (see Section V). Remember that by default a HILARE is not transitive: it does not apply to the callees of a function unless explicitly specified. Built-in operators can be used to refer to the sets of callees or callers of a function (or over-approximate them in case the code base contains indirect calls).

Reasoning with HILARE means reasoning with *conditions* on some *action*: ideally, one should try to express the requirement either as an *invariant* or as a constraint on:

- memory modifications;
- memory (reading) accesses;
- function calls.

In general, properties are easier to express when formulated as a *constraint* on some code operations that must hold under all or most circumstances, except maybe specific ones. This step is detailed in Section V through a number of common specification patterns.

Memory footprint closure. One of the main pitfalls of HILARE specification is forgetting to constrain the modifications of all variables in the footprint of every HILARE. It is important to ensure that such variables cannot be maliciously modified during the execution. Hence, for each location in a HILARE footprint, the practitioner should specify (e.g. as another HILARE) what parts of the code can modify it and under which constraints. This will be illustrated in Section VI.

[6]If not available directly, such a location can be accessed using a binding feature in METACSL that semi-automatically enables such memory to be available at high level through ghost variables. However, static verification of properties involving such bindings might be harder.

Absence of undefined behaviors. Failing to account for potential undefined behaviors and other runtime errors is another trap to avoid. Indeed, even carefully written global specification (or any specification) will be rendered useless by undetected illegal behaviors, because their absence is one of the main assumptions of the verification tools. For example, the following assertion will happily be considered valid by WP though it might be false (since pointer `p` to the character `A` is here used to write an integer).

```
char A, B = 1;
int* p = &A; *p = 0; // Undefined behavior: buffer overflow
//@ assert compiler_dependent: B == 1;
```

Hence it is important to ensure that the `-wp-rte` option is passed to FRAMA-C as mentioned at the end of Section II. Thanks to it, WP will try to prove that every memory access is valid (and report, as expected, a failure in this example).

Assessing the properties

METACSL is a plugin that translates each HILARE into annotations in the target functions, that can then be assessed by existing FRAMA-C plugins, such as EVA, E-ACSL or WP. We will focus here on the usage of WP, that is, through deductive verification.

When running METACSL followed by WP on a HILARE-specified program, most of the verification conditions are usually easily proved valid. Some guidelines about addressing proof failures are laid out in Section VII.

IV. PRESENTATION OF THE USE CASES

The first use case is the bootloader of the WOOKEY [5] project, which aims at prototyping a secure and trusted USB storage device, and is fully open-source. It has two possible banks (i.e. code areas) it can boot from, called *flip* and *flop*. The bootloader performs various security and integrity checks on the bank headers. It then chooses a bank to boot from, locks the other one, checks the integrity of the firmware, and jumps to the chosen bank (either in normal or update mode, depending on the press of a button).

It is a basic yet critical component of the device, as it has full privileges to modify the firmware, and it is the only software component that cannot be patched once the device is built and programmed.

The bootloader uses a global variable called `ctx`, storing some information about the current context of the booting sequence in its fields. In particular, it has two boolean fields `ctx.boot_flip` and `ctx.boot_flop`. Initially, they are both `false`. After the bank choice step in the boot sequence, one (and only one) of them is set to true, depending on which data bank is chosen. The data of the banks themselves is accessible through two pointers `char* flip_data` and `char* flop_data`.

Additionally, the boot sequence is explicitly organized as a finite-state automaton which has an almost linear structure: the control flow is expected to go from the initialization state through six intermediate states until it reaches the final boot state. Thus the control flow goes from a state to the next one, never going back to a previous state. It can only deviate from that linear structure when encountering an error or a security breach, in which case the automaton enters a final error state.

The code is organized as a set of transition functions from one state to another where the actual logic is performed. For example, the final jump occurs in the function that represents the transition to the final boot state. These transition functions are called from a parent function managing the control flow by making calls to a small API designed to ease the manipulation of the automaton. In particular, this API manipulates a global `state` variable, holding the current state of the automaton.

The second use case is a microkernel of an OS dealing with various tasks. Contrary to the first use-case, we intentionally give a very generic simplified description of the system, leaving out the low-level implementation. This simplified description suits a micro-controller target like in WOOKEY, in which the CPU only uses physical addresses.

We assume that the system has a pre-determined[7] number `NUM_TASKS` of applicative tasks, each one being identified by a task number `taskId` > 0 of type `uchar` (see Figure 3). Variable `CurTask` contains the number of the currently executed task. When the (privileged) microkernel services are executed, the variable `Context` is set to `SYSTEM_CTX`, otherwise execution proceeds with ordinary privileges.

The memory is structured in disjoint allocated memory areas that we call regions. The number of regions is given by `NUM_REGIONS`. These memory areas are modeled using two arrays, `RegionStart` and `RegionSize` indicating for each region respectively the pointer to the beginning of the region and its size in bytes. The owner of a region is modeled by the `RegionOwner` array. Thus, region `j` is owned by task `RegionOwner[j]`, starts at address `RegionStart[j]` and contains `RegionSize[j]` bytes. The region is owned by the microkernel if `RegionOwner[j]==SYSTEM_OWNER`, and by an applicative task otherwise. Region `j` is a code region if `RegionKind[j]==CODE_REGION`, and a data region otherwise.

Each task has a priority modeled by `TaskPriority` and a status modeled by `TaskStatus`. A task `i` is ready to be executed if `TaskStatus[i]` is equal to `READY_TSK`, and is waiting or sleeping otherwise. Task `i1` is of a higher priority than task `i2` if `TaskPriority[i1]` $<$ `TaskPriority[i2]` (notice that the highest priority has the smallest value).

Lastly, the activation of a hardware Supervisor Mode Access Prevention (SMAP) feature is symbolized by a boolean `SMAP_enabled` variable. When enabled, the CPU should prevent the microkernel from accessing task memory at all when in privileged mode.

V. COMMON HILARE SPECIFICATION PATTERNS

This section exposes the actual syntax of HILARE specification, along with a set of commonly used patterns to specify

[7]This model realistically assumes that the numbers of tasks and regions are determined at compilation and do not change. However it is not a hard limitation, and dynamic changes can also be supported.

```
typedef unsigned char uchar;
typedef unsigned int uint;
// Number of tasks, defined at compilation
#define NUM_TASKS ...
// Number of regions, defined at compilation
#define NUM_REGIONS ...
#define SYSTEM_CTX 0
#define SYSTEM_OWNER 0
#define READY_TSK 0
#define CODE_REGION 0

char  Context;      // System (SYSTEM_CTX) or Task context
uchar CurTask;      // Current task
uint  SMAP_enabled; // 0 disabled, 1 enabled
uchar TaskPriority[NUM_TASKS]; // Priority of task i
uchar TaskStatus  [NUM_TASKS]; // Ready or waiting

char* RegionStart[NUM_REGIONS]; // Start of region i
uint  RegionSize [NUM_REGIONS]; // Size  of region i
uchar RegionOwner[NUM_REGIONS]; // Owner of region i
uchar RegionKind [NUM_REGIONS]; // Code or Data
```

Figure 3: Modeling tasks and memory regions in a microkernel

usual integrity and confidentiality properties. It is intended to serve as a reference during a specification task and to support Section VI where they are put together to form more complex properties. The process allowing the verification of HILARE specification is described in Section VII, and is useful to have a good understanding of HILARE.

```
meta \prop,
     \name(Name),
     \targets(Targets),
     \context(Context),

Predicate;
```

Figure 4: Base pattern of a HILARE

A HILARE has the form illustrated in Figure 4, named *base pattern*. It specifies that Predicate must be valid in the given Context for all functions in Targets. Name denotes an user-defined name for the HILARE. As will be discussed later, Context can be one of **\strong_invariant**, **\precond**, **\postcond**, **\weak_invariant**, **\writing**, **\reading** or **\calling**. Predicate is an ACSL predicate (the reader can refer to the ACSL specification [2] for the grammar) and can be a very general property, but is usually some form of validity check of different memory operations (hence manipulating memory locations) or a global invariant.

Furthermore, Targets is a set of functions. The set can simply be described *in full*, but it can also be built using several built-in terms, listed below.

a) All functions

The `\ALL` term represents the set of all functions defined in the program under analysis.

b) Set operators

Usual set operators can be used to build the target set, such as union `\union(S1,S2)` and difference `\diff(S1,S2)`. Notably, `\diff(\ALL,S)` specifies a target set of the form "all functions except the ones in `S`".

c) Callees closure

The `\callees(S)` operator returns the transitive closure of `S` by function call, i.e. the set containing `S` and every function (transitively) called by functions in `S`. It is especially useful when dealing with programs with clearly defined entry points.

The following subsections give different possible meanings to the concepts of Predicate and Context in the base pattern by describing how they can work together with the target set to specify interesting properties. Wherever it appears, symbol Location refers to a variable or a range of elements of an array, represented by their addresses.

A. Global weak invariant

Name specifies that predicate Prop holds at the beginning and the end of each function in the Targets set.

```
meta \prop,
     \name(Name),
     \targets(Targets),
     \context(\weak_invariant),

Prop;
```

For example, the following property states that every function needs the variable `logic_state` to be correct (for some unspecified definition of `is_correct`) in the pre-condition and must ensure it is still correct in the post-condition. However, the state may be temporarily incorrect inside the function.

```
meta \prop,
     \name(state_always_valid),
     \targets(\ALL),
     \context(\weak_invariant),

is_correct(logic_state);
```

To disallow even brief violations of the invariant, one should use the **\strong_invariant** context, which ensures the invariant is valid at each *sequence point* of the program, as seen in Section V-C.

B. Global post-condition

Name specifies that the predicate Prop holds at the end of each function in the Targets set.

```
meta \prop,
     \name(Name),
     \targets(Targets),
     \context(\postcond),

Prop;
```

It is similar to V-A but omits the pre-condition on all functions. Both of them are simple ways to automatically add components to a large number of function contracts.

C. Global strong invariant

Name specifies that the predicate Prop holds at every step of each function in the Targets set.

```
meta \prop,
     \name(Name),
     \targets(Targets),
     \context(\strong_invariant),

Prop;
```

D. No memory modification

Name specifies that no function in the Targets set directly writes[8] to Location.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\writing),

\separated(\written, Location);
```

For example, the following HILARE means that the `logic_state` global variable can never be written to.

```
meta \prop,
    \name(state_never_changed),
    \targets(\ALL),
    \context(\writing),

\separated(\written, &logic_state);
```

In practice, when MetACSL processes such a HILARE for further verification, it iterates through all target functions and write instructions, and adds an assertion of Predicate where **`\written`** has been replaced by the particular location being written to by the local instruction. The same process is used for all of the following variations of this pattern.

Note that Location may still be written to by functions that are not in Targets but that are *called by* functions in Targets. To automatically include these functions, the `\callees` operator is very useful.

As mentioned above, `\diff` is also very useful here, to indicate that only a fixed set (e.g. for initialization) of functions is allowed to write to some object. For instance, the following HILARE states that `private_key` can only be set in `enc_init`.

```
meta \prop,
    \name(only_init_allowed),
    \targets(\diff(\ALL, {enc_init})),
    \context(\writing),

\separated(\written, &private_key);
```

E. Conditional memory modification

Name restricts the situations when functions in the Targets set can write to Location. When Guard holds, the write is allowed only if Constraint also holds just before the write operation.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\writing),

Guard ∧
\overlaps(\written, Location)
⇒ Constraint;
```

This is one of the most used patterns. Note that Guard can be omitted to ensure that Constraint holds on *all* write accesses to Location.

For example, the following HILARE is similar to the example in V-D but instead of simply forbidding memory modification, it allows it only if `has_privilege` is true.

```
meta \prop,
    \name(state_change_requires_privilege),
    \targets(\ALL),
    \context(\writing),

\true ∧ // Can be omitted
\overlaps(\written, &logic_state)
⇒ has_privilege ≠ 0;
```

[8]Indirect writes i.e. instructions hidden behind function calls are not considered.

F. Precise conditional memory modification

Name restricts the situations when functions in the Targets set can write to the global variable Var. When Guard holds, the write is allowed only if the relation between its previous and new value is valid according to the predicate Relation.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\writing),

Guard ∧
\overlaps(\written, &Var)
⇒ Relation;
```

Relation is a particular instance of a constraint, that can refer to the value of Var before and after the write using respectively **`\at`**`(`Var`, Before)` and **`\at`**`(`Var`, After)`. For example, the following HILARE only allows assigning increasing values to the global `var`.

```
meta \prop,
    \name(increasing_values),
    \targets(\ALL),
    \context(\writing),

\overlaps(\written, &var)
⇒ (\at(var, Before) ≤ \at(var, After));
```

G. No memory access

Name specifies that no function in the Targets set directly reads from Location.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\reading),

\separated(\read, Location);
```

Similarly to memory modification (Section V-D), target operators can be very useful to build more complex properties.

H. Conditional memory access

Name restricts the situations when functions in the Targets set can directly read from Location. When Guard holds, the read is allowed only if Constraint also holds at the time of reading.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\reading),

Guard ∧
\overlaps(\read, Location)
⇒ Constraint;
```

This is the dual of pattern V-E, and is used for similar purposes.

I. No function call

Name specifies that no function in the Targets set may directly call Function.

`\tguard` is a MetACSL built-in predicate that expands to its argument if it is well-typed, and to `\true` otherwise.

Here, its presence is necessary since `\called` and Function can of course have non-compatible prototypes, making the disequality ill-typed. In that case, `\called` is obviously not Function, so that the HILARE holds. See the METACSL reference paper [4] for more information about `\tguard` and the similar `\fguard` predicate, that evaluates to `\false` if the argument is ill-typed.

```
meta \prop,
     \name(Name),
     \targets(Targets),
     \context(\calling),

\tguard(\called ≠ Function);
```

VI. COMBINING PATTERNS TO EXPRESS COMPLEX PROPERTIES

This section demonstrates how the previous patterns can be articulated into a larger specification methodology on realistic confidentiality or integrity properties. To that end, we come back to the use cases described in Section IV and detail the specification process of some selected properties.

A. *The* WOOKEY *bootloader*

Definitive bank choice. In the code of the bootloader, one single function called `loader_exec_req_selectbank` has the task of choosing which data bank (*flip* or *flop*) to boot from, and set the `ctx.boot_flip` and `ctx.boot_flop` accordingly. It is necessary to ensure that:

- the behavior of that function is correct (at the end, only one of the two fields is true),
- the two fields are modified only by that function.

Whereas the first requirement may be specified with a usual function contract on `loader_exec_req_selectbank`, the second one requires HILARE properties (one for each field) and is a direct application of pattern V-D, as seen in the first property of Figure 5. A similar property can be written for the *flop* bank.

```
meta \prop,
     \name(ctx_flip_only_mod_in_selectbank),
     \targets(\diff(\ALL, loader_exec_req_selectbank)),
     \context(\writing),

\separated(\written, &ctx.boot_flip);

meta \prop,
     \name(selectbank_callees_flipflop_neutral),
     \targets(\diff(\callees(loader_exec_req_selectbank),
                    loader_exec_req_selectbank)),
     \context(\postcond),

ctx.boot_flip == \old(ctx.boot_flip)
∧ ctx.boot_flop == \old(ctx.boot_flop);

meta \prop,
     \name(flip_read_means_flip_chosen),
     \targets(\ALL),
     \context(\reading),

(ctx.boot_flop ∨ ctx.boot_flip)
∧ \overlaps(\read, &ctx.flip_data)
⇒ ctx.boot_flip;
```

Figure 5: Specification of the bank choice requirements

As stated before, the behavior of the function itself can be specified with a simple ACSL contract stating that at the end of the function, either `ctx.boot_flip` or `ctx.boot_flop` is set (but not both), or an error has been raised. To prove this contract however, the solver must know that the callees of `loader_exec_req_selectbank` do not change the value of the two fields. This is an immediate consequence of the previous property, and can be stated as a HILARE to be used by solvers using pattern V-B: in Figure 5, `selectbank_callees_flipflop_neutral` states that all callees of our function (except the function itself, which is included in the `\callees` set) must ensure the values of the *flip* and *flop* fields are the same at the end and at the beginning. This is an instance of a high-level lemma, as described later in Section VII.

Confidential bank data. A confidentiality property that the code should respect is that after the bank choice has been made, the data of the dropped bank should never be accessed again by the bootloader. This is an instance of pattern V-H, as seen in the last property of Figure 5. Namely, `flip_read_means_flip_chosen` stipulates that if a bank has been chosen (if one of the fields is true) and data belonging to *flip* is being read, then it should mean that *flip* is the chosen bank. Of course, a similar HILARE can be specified for *flop*.

Valid boot sequence transitions. Another set of properties concerns the bootloader's automaton. As described previously, the bootloader uses an explicit finite-state automaton structure to guide its control flow, ensuring each step is correctly executed, and in the appropriate order, and that any error or security breach stops the boot sequence. While the design of this automaton and its steps is important, it is also critical to verify that the actual code does not infringe the automaton rules.

Each transition in the automaton is embodied by a function that carries out the specific actions to get to the next state (for example, performing integrity checks on the data bank), and then actually changes the state of the automaton.

Each of these transition functions has a `nextstate` formal parameter that should be used to change the current state at the end of the function. The current state of the automaton is stored in a global `state` variable. To ensure that each transition function correctly changes the state, we can write Figure 6's `transitions_honor_next_state` property, which uses pattern V-B.

There are several things to notice:

- The target set is defined as the set of all transition functions (which must be defined manually using a C macro) *minus* the functions for the boot and error states, which never return.
- The property uses the `\formal` operator, which enables it to refer to formal parameters of the target functions: `\formal(x)` refers to a formal parameter named `x` which must exist in the prototype of each target function.

```
meta \prop,
    \name(transitions_honor_next_state),
    \targets(\diff(TRANSITION_FUNCTIONS,
                \union(loader_exec_req_boot,
                       loader_exec_error,
                       loader_exec_secbreach))),
    \context(\postcond),

\fguard(
    logic_state == \formal(nextstate)
    ∨ logic_state == LOADER_ERROR
);

meta \prop,
    \name(state_wrapper_only_called_in_transitions),
    \targets(\diff(\ALL,
        \union(TRANSITION_FUNCTIONS, INIT_FUNCTIONS)))
    \context(\calling),
\tguard(\called ≠ loader_set_state);
```

Figure 6: Partial specification of the bootloader's automaton

- The automaton state at the end can either be the specified next state or the error state.
- As the target set *may* contain functions without a `nextstate` formal parameter (which would make `\formal` raise a type error), the whole predicate must be surrounded by `\fguard`, thus defaulting to `\false`: should that case occur, the corresponding verification conditions will fail, so that users will be warned and have the ability to either amend the HILARE or fix the function to follow appropriate naming conventions.

Enforced automaton control flow. Here, we check that transition functions correctly manipulate the state. However, other functions may still do anything with the `state`, rendering the previous property useless if expressed alone. To strengthen the overall verification, we can specify that the state is only modified by these transition functions (and various initialization functions).

In the previous section (V-D), we showed how to specify that the actual variable is only modified through the `set_state` setter. If we keep that property, it is then enough to check that this setter is only called by transition functions, with an instance of pattern V-I. It is materialized in the last property of Figure 6, where the target is defined as "all functions minus transition functions and initialization functions" (using set difference and union), and we check that these target functions do not call the setter.

Lastly, a quick analysis of the code shows that the transition functions are called by a single function `loader_exec_automaton_transition` managing the control flow, itself called by `loader_exec_automaton` and then by `main`.

Though it is necessary to specify that these management functions operate correctly (in a way not detailed in this paper), we also want to ensure that these functions are only called in that order, i.e. that there is no inner code that spuriously calls the automaton management functions recursively. We can use that same pattern V-I three times to specify that these functions are only called by their expected parents.

Overall, the specified properties ensure that:

```
meta \prop,
    \name(region_integrity_task),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

∀ i∈ℤ; 0 ≤ i < NUM_REGIONS
∧ Context ≠ SYSTEM_CTX
∧ \overlaps(\written,
              RegionStart[i] + (0 .. RegionSize[i] - 1))
⇒ RegionOwner[i] == CurTask ∧ RegionKind[i] ≠ CODE_REGION;

meta \prop,
    \name(region_integrity_system),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

∀ i∈ℤ; 0 ≤ i < NUM_REGIONS
∧ Context == SYSTEM_CTX
∧ \overlaps(\written,
              RegionStart[i] + (0 .. RegionSize[i] - 1))
⇒
(RegionOwner[i] == CurTask ∨ RegionOwner[i] == SYSTEM_OWNER)
∧ RegionKind[i] ≠ CODE_REGION;

meta \prop,
    \name(region_confidentiality_task),
    \targets( \diff( \ALL, init ) ),
    \context(\reading),

∀ i∈ℤ; 0 ≤ i < NUM_REGIONS
∧ Context ≠ SYSTEM_CTX
∧ \overlaps(\read,
              RegionStart[i] + (0 .. RegionSize[i] - 1))
⇒ RegionOwner[i] == CurTask ∧ RegionKind[i] ≠ CODE_REGION
```

Figure 7: Spec. of region integrity and confidentiality

- The call graph has a correct structure. The functions with the actual bootloading logic are only called by the automaton management functions, which are not called back.
- The automaton management functions are correct: they call transition functions according to the current state and specify a correct next state.
- That next state is correctly used by transition functions.
- The automaton state is not spuriously modified by inner functions.

This demonstrates that an initial, high-level and fuzzy requirement such as "the code is correct w.r.t. the automaton structure" must often result in a set of tightly coupled HILARE, where some enforce the requirement directly while the others ensure that no "side-channel" exists to modify the anchor points of the property outside of the intended flow of execution. It took approximately 3 person-months to specify and verify every requirement on WOOKEY.

B. A simple microkernel

The simplified microkernel described in Section IV is by nature a critical component of its host system, and as such should uphold several confidentiality and integrity properties pertaining to different aspects of its behavior.

Task memory isolation. One such aspect is the strict compartmentalization of tasks: a task should only read from and write to the memory regions it owns. Moreover, it should not access regions containing code in any way. This can be specified with similar HILARE using patterns V-E and V-H.

The first property in Figure 7 iterates on all regions and states that if a memory region is modified (that is, if the

```
#define FORALL_REGION(name, pred) \
    (∀ name∈ℤ; 0 ≤ name < NUM_REGIONS ⇒ pred)
#define CONTEXT_IS_SYSTEM (Context == SYSTEM_CTX)
#define REGION_OWNED_BY_SYSTEM(r) \
    (RegionOwner[r] == SYSTEM_OWNER)
#define REGION_RANGE(r) \
    (RegionStart[r] + (0 .. RegionSize[r] - 1))
#define SMAP_ENABLED (SMAP_enabled ≠ 0)

meta \prop,
    \name(micro_kernel_confidentiality),
    \targets( \diff( \ALL, init ) ),
    \context(\reading),

FORALL_REGION(r,
  CONTEXT_IS_SYSTEM
  ∧ ¬ REGION_OWNED_BY_SYSTEM(r)
  ∧ \overlaps(\read, REGION_RANGE(r))
  ⇒ ¬ SMAP_ENABLED
);

meta \prop,
    \name(micro_kernel_integrity),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

FORALL_REGION( r,
  CONTEXT_IS_SYSTEM
  ∧ ¬ REGION_OWNED_BY_SYSTEM(r)
  ∧ \overlaps(\written, REGION_RANGE(r))
  ⇒ ¬ SMAP_ENABLED
);
```

Figure 8: Specification of the SMAP feature

location targeted by a write overlaps with a region) while in user land, then (i) the owner of that region should be the task itself, and (ii) it should not be a code region. Here, as the Location in pattern V-E, we use `RegionStart[i] + (0 .. RegionSize[i] - 1)`, which represents the *range* of addresses corresponding to region number `i`.

`region_integrity_system`, the second property on Figure 7, follows the same principles. Its Guard captures the modification of regions while in system context (when the privileged microkernel services are invoked from a task) and the Constraint forces the owner of the modified regions to be either the original task or the microkernel. Again, modifications of code regions are forbidden.

These two *integrity* properties are then mirrored using pattern V-H to specify their *confidentiality* counterparts, that is restraining memory accesses instead of modifications. Due to space constraints, we only show in Figure 7 `region_confidentiality_task` which is `region_integrity_task`'s confidentiality counterpart.

Controlled privileged operations. The SMAP feature mentioned in Section IV should also be enforced: when enabled, code in privileged mode should not have any access whatsoever to task regions. Again, this can be specified with instances of patterns V-E and V-H with a structure similar to the previous properties. Figure 8 illustrates such restriction of memory accesses, both for reading and writing. The Guard filters accesses in privileged mode to regions not owned by the microkernel, which can only happen if SMAP is disabled.

Furthermore, Figure 8 demonstrates how the usage of C macros can improve the readability and portability of specifications by abstracting implementation details to simple predicates. In particular, the `FORALL_REGION` macro hides the underlying array, bounds and indices, which are just noise, requirement-wise.

Write XOR execute. Our microkernel discriminates memory regions by their kind: either code or data. While the previous properties ensure that code regions cannot be modified or accessed, executing instructions contained in data regions must also be prevented. Furthermore, a task should not try to jump to another task's code. Finally, when in privileged mode, only microkernel code should be run. These three requirements can be materialized by the HILARE depicted in Figure 9.

```
meta \prop,
    \name(code_execution),
    \targets( \ALL ),
    \context(\calling),

∃ i∈ℤ; 0 ≤ i < NUM_REGIONS
∧ RegionStart[i] ≤ (char*)\called
                   < RegionStart[i] + RegionSize[i]
∧ RegionKind[i] == CODE_REGION
∧
((Context ≠ SYSTEM_CTX ∧ RegionOwner[i] == CurTask)
 ∨
 (Context == SYSTEM_CTX ∧ RegionOwner[i] == SYSTEM_OWNER)
);
```

Figure 9: Code/data exclusion and isolation

Rather than using a specific pattern, `code_execution` is a direct instantiation of Section V's *base pattern*: it is a general validity check of the function call operation, where **`\called`** is the location of any function that may be called. It states that for every call during the execution, the call should land in a region that: (i) is a code region, (ii) is owned by the current task if executing in user mode, and (iii) is owned by the kernel if we are in privileged mode.

```
meta \prop,
    \name(schedule_priority),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

\overlaps(\written, &CurTask) ⇒
TaskStatus[\at(CurTask,After)] == READY_TSK ∧
( ∀ j∈ℤ; 0 ≤ j < NUM_TASKS ∧ TaskStatus[j] == READY_TSK ⇒
TaskPriority[j] ≥ TaskPriority[\at(CurTask,After)] );

meta \prop,
    \name(current_always_ready),
    \targets( \diff( \ALL, init) ),
    \context(\strong_invariant),

TaskStatus[CurTask] == READY_TSK;
```

Figure 10: Specification of scheduling requirements

Valid task scheduling. Specification of the scheduling behavior calls for different specification patterns: when switching contexts, the microkernel should not jump to any task that is ready to run, but to a task with *highest priority*. Hence, whenever the current task changes, the first HILARE in Figure 10 puts a constraint on the *new task* instead of the old one. This is a use-case for pattern V-F. `schedule_priority` has no guard (so all modifications of `CurTask` are captured), but states that any new value of `CurTask` must represent a task that is both ready and of highest priority (compared to other

ready tasks). Notice the use of `\at(CurTask,After)` to refer to the new value of `CurTask`.

While this captures the requirements related to context switch, there are others that our system should honor. One of them, `current_always_ready` of Figure 10, is that while the scheduler can only switch to ready tasks, the current task shall remain in a ready state for the entirety of its execution. This is an *invariant* that should never be broken, hence pattern V-C is used to enforce that status.

Tying up loose ends. Although all the previous HILARE taken together form a consistent set that accurately formalizes some of the requirements for the microkernel, it is of the utmost importance to ensure the absence of loose ends, i.e. to check whether the memory footprint is properly constrained, as mentioned in Section III. For example, properties in Figure 7 expect the data in `RegionOwner` to be correct. However, there is no safeguard preventing malicious code from spuriously changing region owners.

To resolve this issue, it is necessary to carefully track any variable in the footprint of every HILARE, and specify when, if ever, it is allowed to change, with the help of pattern V-D. This is done for some of the variables in Figure 11: for example, `context_modification` ensures that only a set $\mathcal{K}$ of functions[9] should be able to enable or disable privileged mode, and `region_owners_final` states that region owners should never change.

This should be done for all other relevant variables, such as `SMAP_enabled`, `TaskPriority`, `RegionKind`, etc.

```
meta \prop,
    \name(context_modification),
    \targets( \diff( \ALL, K ) ),
    \context(\writing),

\separated(\written, &Context);

meta \prop,
    \name(task_status_modification),
    \targets( \diff( \ALL, {scheduler, init} ) ),
    \context(\writing),

\separated(\written, TaskStatus + (0 .. NUM_TASKS - 1));

meta \prop,
    \name(region_owners_final),
    \targets( \ALL ),
    \context(\writing),

\separated(\written,
           RegionOwner + (0 .. NUM_REGIONS - 1));
```

Figure 11: Selected footprint modification constraints

VII. Verification discussion

A. The verification mechanism

As explained in previous sections, the usual verification process of HILARE is to run MetAcsl, which transforms a HILARE into a set of code annotations, and then run Wp, the deductive verification plugin of Frama-C, which tries to prove the various annotations generated by MetAcsl.

[9]Where $\mathcal{K}$ is defined as all kernel entry points such as system calls, exceptions and interrupts.

The transformation from HILARE to code annotations is best explained by example. Consider the example from Figure 1 and assume we want to check that in this program, `A` can only be modified if the current value of `C` is above zero. This can be easily specified with an instance of pattern V-E:

```
meta \prop,
    \name(constant_once_negative),
    \targets(\ALL),
    \context(\writing),

\overlaps(\written, &A) ⇒ C ≥ 0;
```

Now since it is a write constraint, MetAcsl will *weave* this HILARE into the program by instantiating the predicate `\overlaps(\written, &A)⇒C ≥0` before every writing statement, replacing `\written` by affected location. This results in the program listed below (without the earlier ACSL annotations). We also add a `bad` function that deliberately violates our property to further illustrate the transformation.

```
int A, B, C;
void copy(int* p, const int*q) {
    /*@ assert P1: \overlaps(p, &A) ⇒ C ≥ 0; */
    *p = *q;
}
void foo() { ... /* Unchanged */ }
void bad() {
    /*@ assert P2: \overlaps(&C, &A) ⇒ C ≥ 0; */
    C = -1;
    /*@ assert P3: \overlaps(&A, &A) ⇒ C ≥ 0; */
    A = 42;
}
```

First, note that an annotation is inserted before each write. Hence, `foo`, containing only calls and conditionals, is unchanged. Also note how each time `\written` gets replaced by the location of the write (i.e. its address), such as `p` in P_1.

Proving a HILARE is then a matter of using other Frama-C plugins to validate the annotations. For example, the command:

```
frama-c example.c -meta \
        -then-last -wp -wp-rte
```

would instruct MetAcsl to weave the annotated HILARE into the program and to pass the result to Wp, to try and prove the annotations, as well as the absence of runtime errors.

In P_2, `\overlaps(&C, &A)` trivially simplifies to *false*, hence the whole implication is *true*. As expected P_3 cannot be proven valid since it simplifies to `\true ⇒ C ≥ 0` and `C` is provably negative. To prove P_1, we would need to add it as a pre-condition to `copy` since, when using deductive verification, pre-conditions are the only way to have assumptions about the state at call site.

While some technical issues and features of the transformation for more complicated patterns (such as V-F) are omitted here, this gives a good idea on how MetAcsl generates a set of annotations equivalent to a HILARE and how to prove them.

B. Useful tips and pitfalls

Several guidelines and pitfalls were already presented in Section III. We discuss here some additional ones.

Proof failure analysis. If some of the annotations are not proved, this can be due to several reasons:

(i) The program is incorrect with respect to the HILARE: the code (or the HILARE) needs to be fixed. Given the name of the failing annotation, it is easy to trace back to the guilty HILARE.
(ii) There are insufficient pre-conditions on the function: the given context is not sufficient for the property to be valid. One should add pre-conditions needed for the proof.
(iii) The prover is not powerful enough: one should manually subdivide the proof into a few intermediate steps by writing annotations that are easier to prove and can help to deduce the required one.
(iv) Some functions that are called are not specified enough. It is common that, while a function clearly does not modify the state related to a property, this is not reflected in its specification and the prover cannot deduce that, after the call, the memory footprint of the property is unchanged. A first step consists thus in manually specifying the memory footprint of the callees. If this is not sufficient, one should specify the conditions needed as a conditional invariant on all potentially called functions using a HILARE.

Avoid overloading the proof context. When dealing with large functions, it may be useful to use the `-meta-checks` option which instructs METACSL to instantiate a HILARE as a set of *checks* instead of assertions. The difference is subtle: a check simply tries to prove a predicate at a given point, while an assertion additionally adds it to the context for further proofs. Hence at the end of a long function, provers might struggle with an overloaded proof context containing all previous assertions of the function.

High-level lemmas. When faced with problem (iii), a classic solution is to provide lemmas used to cut the goal. While it is possible to write lemmas in ACSL, this solution may not work for a HILARE as it can rely on specific local properties in the target functions or on variables non-available globally (e.g. a function parameter). An efficient solution is to write a HILARE used as a *lemma* to prove another HILARE or an ACSL annotation: it is just as efficient as using a regular lemma in a normal context. `selectbank_callees_flipflop_neutral` in Section VI-A is an example of such an auxiliary HILARE.

VIII. RELATED WORK

Defining and verifying security properties of OS kernels down to the concrete source code was done in several projects, including notably [7, 8, 9, 10, 11, 12], with different strategies. Compared to these projects, which target micro-processors with MMU-based memory isolation, the security properties shown in this paper are just simple examples for the sake of illustration and better match micro-controller setups, which usually have no MMU and thus do not implement virtual memory. For instance, a model of the hardware including the MMU, the processor's privilege levels and the program counter is required to show that the MMU configuration matches the kernel's internal control structures and that the control flow only enters the kernel privilege level at the defined entry points for syscall, exception, and interrupt handling. Moreover, some projects prove higher-level security policy abstractions like information flow enforcement [8, 10, 9], from which confidentiality and integrity properties similar to the ones introduced in this paper can be derived, such as memory isolation in PIP [11], as defined for a separation kernel [13].

Compared to these projects, the contribution illustrated in this paper is the ability to both define and mechanically verify global security properties directly at the level of the C source code. Indeed, all previous approaches first define and prove the security properties on an abstract model of the OS kernel and the underlying hardware and second show that the concrete, low-level code executed refines the abstract model and still verifies the security properties defined on the abstract model. It is argued that for verification purposes, it is easier to reason on an abstract model than on low-level concrete source code [8]. However we believe that using macros appropriately in METACSL, as shown in Section VI, helps defining properties in an abstract way, by pushing implementation details to the macro definitions. Moreover proving that the concrete code refines the abstract model requires significant efforts. In particular, in INTEGRITY [7], the proof that the C source code refines the abstract model is manual and thus cannot be considered as maintainable. Similarly, in SEL4, an abstract model in Isabelle/HOL and the C source code are written separately and all mechanized proofs of security properties rely on a first mechanized proof of correctness that the C source code refines the abstract model [8]. This proof of correctness initially costed 25 person.years [14] and the team is still working on how to improve its maintainability [15].

To make this approach more scalable, in CERTIKOS the OS kernel and the hardware are modeled as a set of small stackable layers (also called *deep specifications*), whose interfaces and observable behaviors are defined in Coq [16]. In each layer, the C or assembly implementation is verified to be a refinement of the layer's upper interface (called *overlay*) assuming that the layer's lower interface (called *underlay*) is correctly implemented by lower layers. The verification of each layer implementation is done using COQ tactics [17]. Global properties can then be proved using only the COQ model of the layers' interfaces. The drawback of such a modularized approach is that it makes it difficult to obtain an efficient implementation of a microkernel, because in microkernels targeting performance the implementations of the required features are entangled. Indeed by principle such software must only implement the bare minimal features required at the kernel privilege level and the performance of the microkernel is critical for the performance of the whole software system.

In PROSPER [10], refinement steps are bypassed using a HOL4 model of the ARMv7 instruction set architecture [18]. The information flow properties are defined and verified on an abstract model of idealized ARMv7 machines representing the user-level execution contexts and the kernel execution context. In the verification process, HOL4 is used to generate pre- and post-conditions for atomic executions of the kernel at

boot-time and from entry to exit to/from the kernel execution context (that is, for hypercall, exception and interrupt handling). The verification of the concrete kernel code thus consists in verifying these pre- and post-conditions and is done at the ARMv7 binary code level using the BAP toolset [19]. Although this approach has not been tested on a unified full-featured OS microkernel, it is likely to be more scalable than in SEL4 and more performance-compatible than in CERTIKOS. However, it forces to reason on a model of the target machine architecture, making the source code an implementation detail, while METACSL allows us to reason on the C source code and its adaptations to the target machine architecture.

In PIP and PROVENCORE, the C source code is automatically generated from an executable specification, in COQ for PIP [11] and in a proprietary language called Smart for PROVENCORE [12], on which all properties and mechanized proofs are written. The extension of the proof to the C code relies on the code generation process, whose proof of correctness is ongoing for PIP [11] and is not public for PROVENCORE. While generating C code from an abstract executable specification makes it easier to prove the actual C code, this adds the open challenge of being able to generate efficient code. Ongoing efforts study COQ tactics [20] that implement heuristics to generate efficient code.

A few security properties of WOOKEY's bootloader were defined in [21]. Some were defined with help from the FRAMA-C GUI and the EVA plugin [22] and verified by inserting assertions and checking that they were not violated using dynamic symbolic execution with KLEE [23].

Bootloader verification was attempted for SABLE [24] and a simplified variant of a RISC-V first-stage bootloader (FSBL) [25]. In both cases, the bootloaders ensure integrity and authenticity of the loaded software and the platform state, although with different strategies. In SABLE, the dynamic root of trust measurement facility of Intel and AMD processors as well as a hardware TPM are used to prevent system image decryption in case of failure and otherwise allow an external user to do remote attestation of the successful boot process. More traditionally and like in WOOKEY, the RISC-V FSBL studied in [25] is the ROM-located first piece of software executed at boot and checks the cryptographic signature of the loaded software.

In both projects the verification follows techniques already used for microkernels but is reported as partially achieved. SABLE follows the strategy of SEL4 [14] while the RISC-V FSBL follows the code generation strategy down to RISC-V binary code using Bedrock2 [26], an imperative language and compiler written in COQ, and the riscv-coq implementation of the RISC-V specification in COQ. Interestingly the RISC-V FSBL verification covers the use of the hardware DMA to load the software image.

The HILARE specification language is related to previous high-level extensions of a contract-based specification language. For example, JML [27] has been extended by Cheon and Perumandla [28] to specify protocols (properties pertaining to the order of call sequences), and by Trentelman and Huisman [29] to express temporal properties. While protocols may be expressible in HILARE (with variations of pattern V-I and using ideas similar to WOOKEY's automaton) as well as a subset of temporal properties (using pattern V-F), it may not be as simple as the syntax provided in their work, since such properties are not the main focus of HILARE.

The general idea of defining a high-level concept in the global scope and then *weaving it* into the implementation is analogous to the Aspect-Oriented Programming (AOP) [30] paradigm: HILARE can be seen as cross-cutting concerns at specification rather than code level. Contexts can then be related to *pointcuts*, which in AOP are a set of control flow points where the code needed by the concern should be added.

IX. CONCLUSION

Convenient solutions for specification of high-level program requirements are highly needed for (formal) verification and certification of real-life software products. In this respect, this paper proposes a pragmatic methodology for large C programs. We describe a set of common property patterns and show how to apply them to specify security-relevant properties on two use cases. One of them—the bootloader of the secure storage device WOOKEY—was fully verified using this approach. The second one—sketching in a generic way selected features of a microkernel—illustrates the ability of our approach to cover many relevant security properties of an OS.

The proposed technique is based on three key ingredients. The first one is the idea to specify a HILARE as a predicate to be automatically instantiated at the relevant program points into normal ACSL annotations that can then be automatically proved by classic deductive verification. In this way, a HILARE can be seen at the same time as a high-level property and a set of the resulting low-level properties. Second, the definition of several contexts (global or weak invariants, reading, writing, calling operations) along with a target set of functions in which the resulting low-level properties should be inserted allows a wide range of possibilities of specification. Third, combined with the expressiveness of ACSL (including predicates and primitives to express memory-related properties such as separation), the overall solution becomes suitable to express a large range of relevant security properties such as memory isolation, integrity and confidentiality.

Future work includes applications of the HILARE-based approach to other real-life software products and an identification of other property patterns possibly necessary to express other kinds of high-level properties.

Acknowledgement. We warmly thank Ryad Benadjila, Arnauld Michelizza, Patricia Mouy, Mathieu Renard, Philippe Thierry and Philippe Trebuchet from the ANSSI for our fruitful discussions about WOOKEY, and the FRAMA-C team for their continuous support. The work of the first author was partially funded by a Ph.D. grant of the French Ministry of the Armed Forces – Defense Innovation Agency. Many thanks to the anonymous referees for their helpful comments.

References

[1] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Asp. Comput.* 27.3 (2015). DOI: 10.1007/s00165-014-0326-7.

[2] P. Baudin, P. Cuoq, J.-C. Filiâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. https://github.frama-c.com/acsl-language/acsl. 2020.

[3] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall. "MetAcsl: Specification and Verification of High-Level Properties". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 11427. LNCS. 2019. DOI: 10.1007/978-3-030-17462-0_22.

[4] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall. "Tame Your Annotations with MetAcsl: Specifying, Testing and Proving High-Level Properties". In: *Tests and Proofs (TAP)*. Vol. 11823. LNCS. Springer, 2019. DOI: 10.1007/978-3-030-31157-5_11.

[5] R. Benadjila, A. Michelizza, M. Renard, P. Thierry, and P. Trebuchet. "WooKey: Designing a Trusted and Efficient USB Device". In: *Annual Computer Security Applications Conference (ACSAC)*. 2019. DOI: 10.1145/3359789.3359802.

[6] N. Kosmatov and J. Signoles. "A Lesson on Runtime Assertion Checking with Frama-C". In: *4th International Conference on Runtime Verification (RV 2013)*. Vol. 8174. LNCS. Springer, 2013, pp. 386–399. DOI: 10.1007/978-3-642-40787-1_29.

[7] R. J. Richards. "Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by D. S. Hardin. 2010. DOI: 10.1007/978-1-4419-1539-9_10.

[8] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. "seL4: from General Purpose to a Proof of Information Flow Enforcement". In: *IEEE Symposium on Security and Privacy*. 2013. DOI: 10.1109/SP.2013.35.

[9] D. Costanzo, Z. Shao, and R. Gu. "End-to-End Verification of Information-Flow Security for C and Assembly Programs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2016. DOI: 10.1145/2908080.2908100.

[10] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. "Formal Verification of Information Flow Security for a Simple Arm-Based Separation Kernel". In: *ACM Conference on Computer & Communications Security (CCS)*. 2013. DOI: 10.1145/2508859.2516702.

[11] N. Jomaa, P. Torrini, D. Nowak, G. Grimaud, and S. Hym. "Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base". In: *International Workshop on Automated Verification of Critical Systems (AVOCS)*. 2018. DOI: 10.14279/tuj.eceasst.76.1080.

[12] S. Lescuyer. "ProvenCore: Towards a Verified Isolation Micro-Kernel". In: *International Workshop on MILS: Architecture and Assurance for Secure Systems, MILS@HiPEAC*. 2015. DOI: 10.5281/zenodo.47990.

[13] J. M. Rushby. "Design and Verification of Secure Systems". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 1981. DOI: 10.1145/800216.806586.

[14] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. "seL4: Formal Verification of an OS Kernel". In: *ACM Symposium on Operating Systems Principles*. ACM, 2009. DOI: 10.1145/1629575.1629596.

[15] J. Andronick. "A Million Lines of Proof About a Moving Target (Invited Talk)". In: *International Conference on Interactive Theorem Proving (ITP)*. Vol. 141. LIPIcs. 2019. DOI: 10.4230/LIPIcs.ITP.2019.1.

[16] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[17] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. "Deep Specifications and Certified Abstraction Layers". In: *ACM Symposium on Principles of Programming Languages (PoPL)*. 2015. DOI: 10.1145/2676726.2676975.

[18] A. Fox and M. O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *International Conference on Interactive Theorem Proving (ITP)*. 2010. DOI: 10.1007/978-3-642-14052-5_18.

[19] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. "BAP: A Binary Analysis Platform". In: *Computer Aided Verification (CAV)*. 2011. DOI: 10.1007/978-3-642-22110-1_37.

[20] C. Pit-Claudel, P. Wang, B. Delaware, J. Gross, and A. Chlipala. "Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs". In: *Automated Reasoning*. LNCS. 2020. DOI: 10.1007/978-3-030-51054-1_7.

[21] ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, and T. Labs. "InterCESTI: Methodological and Technical Feedbacks on Hardware Devices Evaluations". In: *SSTIC 2020*.

[22] S. Blazy, D. Bühler, and B. Yakobowski. "Structuring Abstract Interpreters Through State and Value Abstractions". In: *Verification, Model Checking, and Abstract Interpretation*. 2017. DOI: 10.1007/978-3-319-52234-0_7.

[23] *KLEE*. URL: https://klee.github.io/.

[24] S. D. Constable, R. Sutton, A. Sahebolamri, and S. Chapin. *Formal Verification of a Modern Boot Loader*. Electrical Engineering and Computer Science - Techni-

cal Reports 183. Syracuse University, 2018. URL: https://surface.syr.edu/eecs_techreports/183.

[25] Z. Straznickas. “Towards a Verified First-Stage Bootloader in Coq”. MA thesis. Massachusetts Institute of Technology, 2020.

[26] A. Erbsen and S. Gruetter. *Language and compiler for verified low-level programming*. URL: https://github.com/mit-plv/bedrock2.

[27] G. T. Leavens, A. L. Baker, and C. Ruby. “JML: A Notation for Detailed Design”. In: *Behavioral Specifications of Businesses and Systems*. 1999. DOI: 10.1007/978-1-4615-5229-1_12.

[28] Y. Cheon and A. Perumandla. “Specifying and Checking Method Call Sequences in JML”. In: *International Conference on Software Engineering Research and Practice*. 2005.

[29] K. Trentelman and M. Huisman. “Extending JML Specifications with Temporal Logic”. In: *International Conference on Algebraic Methodology and Software Technology*. AMAST. 2002. DOI: 10.1007/3-540-45719-4_23.

[30] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. “Aspect-Oriented Programming”. In: *European Conference on Object-Oriented Programming*. 1997. DOI: 10.1007/BFb0053381.