SeL4 verfication in Isabelle/HOL and verification of a OpenBSD driver

Adriana Stancu
University of Bucharest
Bucharest, Romania
adriana.stancu@unibuc.ro

prof. dr. Ioana Leustean
University of Bucharest
Bucharest, Romania
ioana@fmi.unibuc.ro

conf. dr. Paul Irofti
University of Bucharest
Bucharest, Romania
paul.irofti@unibuc.ro

***Abstract*—The seL4 microkernel is currently the only kernel that has been fully formally verified. In general, the increased interest in ensuring the security of a kernel's code results from its important role in the entire operating system.**
One of the basic features of an operating system is that it abstracts the handling of devices. This abstraction is represented by device drivers - the software that manages the hardware. A proper verification of the software component could ensure that the device would work properly unless there is a hardware failure.
This is the motivation why we choose to try to model the behavior of a device driver and build the proof that the code implementation matches the expected behavior. The proof was written in Isabelle/HOL, the code translation from C to Isabelle was done automatically by the use of C-to-Isabelle Parser and AutoCorres tools. We choose Isabelle theorem prover because its efficiency was already shown by being used through the verification of seL4 microkernel.

***Index Terms*—Isabelle, C programming language, verification, kernel, driver**

I. Introduction

The kernel is a crucial component of a system, and direct access to hardware resources leads to an increased risk if a malfunction occurs. In our case, seL4 was designed as a microkernel in order to reduce the impact of software problems to the system's functionalities.
The main topic of interest in the analysis of the seL4 microkernel is the way to prove the functional correctness through the Isabelle/HOL theorem prover. The methods applied in system verification are more powerful and accurate than automated verification techniques such as model checking, static analysis, or deploying the entire kernel in a type-safe language.
This method of proving in Isabelle all the critical properties of the systems allows the analysis of specific aspects such as exploring the branches of execution of safe scenarios (safe execution), but also a set of specifications and proofs of kernel behavior reaching the analysis of implementation in C of the kernel for the ARM platform.

II. SeL4 verification structure

Although work to implement its proofs was started in 2009 [5] by the National ICT Australia - NICTA (Information and Communication Technology Research Center) research team, formal proofs of the kernel are still maintained up to date with the source code.
SeL4 is part of the L4 family - along with other implementations that share the same L4 interface: Pistachio [12], Fiasco [13] or Hazelnut [14].
The proofs that underlie the verification of seL4 system are in the form of Hoare structures that have in their center a code component or whole functions. The difficulty of verifying the seL4 micro-kernel lies in formulating pre-conditions and post-conditions that accurately represent security properties that it must meet. At the same time, the formal representation must be as close as possible to the structure and functionalities implemented in the source code.
An interesting aspect related to the design of a microkernel and the properties of the C code in relation to the form of their verification is the separation of calls to kernel functions in two phases [4]: verification and execution. The verification phase can be understood as a stage of validation of the preconditions: the input data and the permissions on the actions to be performed are verified. The execution consists in the actual running of the system function, benefiting from its verification because the preconditions have already been verified in the previous phase.
An important thing to note is that in the verification phase, the status of the system is not changed, otherwise this separation would no longer be relevant. And the idea brings a valuable advantage in the verification process because it simplifies the system call proof: execution will not return an error if the verification phase has been completed successfully.

III. SeL4 memory management

The seL4 kernel uses a memory allocation model that transfers the control over the allocation of memory from kernel space to applications that have this permission. Permission on memory management is represented by having a structure called capability [3].
A consequence of this transfer of rights is that the kernel heap memory can be precisely partitioned between applications: each application has that part of the heap for which it has a capability that gives it that authority. Separating heap memory is especially important for expressing and demonstrating security properties (integrity and confidentiality).
The basic features of the kernel memory allocation model are as follows [4]:

- Memory allocation is explicit and is performed only when assigning a type (retype) to an untyped memory area.
- The memory allocation is strictly delimited by the specified free memory.
- Kernel objects are not shared or reused.

This memory management model leaves the responsibility for verifying security policies outside the kernel. All that is left is to verify the correctness of the memory allocation algorithm in the kernel. The properties of interest being:

- Checking that the allocated objects are contained in the corresponding areas of free memory;
- Memory areas allocated to objects do not overlap.

As specified, the rights for allocating a certain memory area are represented by holding a capability, these capabilities can be transferred between the kernel components, their transfer can be represented in the form of a tree in which the capabilities are the nodes of the tree.

If for the memory allocation the two properties listed above must be checked, which ensures the separation from the rest of the objects and the integrity of the newly assigned object, for its dislocation two steps are applied to invalidate all references to the memory area to be dislocated:

1) Searching for all the capabilities by which access rights are granted on the memory object;
2) Delete all these capabilities and mark the memory area as free (return it to the unallocated memory block).

For the first stage, the capability transfer tree is used to find and invalidate all capabilities that allow permissions on the memory area. In the second stage, it is verified through the same tree that there are no references in other objects or global references to the area to be released.

IV. Memory access verification

Memory access is an interesting topic in order to model as accurately as possible the behavior of program C. In Isabelle, pointers can be represented as a new type of data in the form: datatype a ptr = Ptr word32, which means that a pointer can be represented only by the word32 address it contains. Using this representation one can reason about the heap memory.

In this sense an important problem is raised by the cases in which it is possible to pass from one type of pointer to another. For example, if we have two different pointers as type (float and int), after we change the value from the address to which one of them points, we cannot be sure that the other has not been changed, if they would have indicated from the beginning to the same address. To ensure that different pointers as a type point to different addresses, the Burstall-Bornat model was used as a solution [9].

The solution developed by Burstall and Bornat is to use a model in which the heap memory is separated into types. Each data type has its own function that maps pointers to their values:

record state =
heap_int word32 ⇒ int
heap_float word32 ⇒ float
heap_intptr word32 ⇒ addr...

This model avoids ambiguities when a pointer of a certain type is modified. In the previous example, if we changed the values contained in the address of an integer pointer, there is no confusion that a pointer to a float value was accidentally changed because the values of the float type are separated from each other. Of course, the fact that this model gives us the certainty that we do not accidentally change values of different types, however, also brings the limitation that it is no longer possible to switch from one data type to another by cast. A memory area, once allocated, remains defined in the corresponding heap memory section until it is released.

V. C to Isabelle conversion

The most interesting component of the formal check in seL4 is the bridge between C language and the proofs in Isabelle. This is also the most complex part of the proofs because the semantics of the C language must be taken into account, memory management including data types, storage of structures in memory, pointer cast between different data structures or pointer arithmetic.

For example, in Isabelle, memory addressing is represented by a function defined on the address space resulting in binary data, without information about the type of data to which the address refers. Apart from this function that models pointers, the way different types of data are stored is treated separately. [6] Abstracting how memory access, data alignment, and how different data types are modeled helped that higher-level proofs not require repetitive checks, such as that a pointer is not null before being accessed, because these checks are already defined as constraints.

The correctness of the C language semantics is not, however, treated as critical to the proofs of the whole system because it adds an additional level to the verification: the validation of the correspondence between the formal model and the result obtained after compiling the source code. However, this validation brings the expected results when the code is compiled with a low level of optimizations. Therefore, the exclusive use of checking the resulting binary file without taking into account the source code was not completely performed.

The proof technique used to ensure the correspondence between the abstract specifications, the formal model of the source code and the model resulting from the analysis of the binary file is the refinement. ”Program C is a refinement of program A, if the set of behaviors of program C is a subset of the behaviors described by program A” [4]. Behavior means a sequence of steps given by the change of system state and the transition between states. The state of the system consists of the state of its components (memory, processes, resources) both those belonging to the user space and those in the kernel.

A. *Isabelle/HOL theorem prover*

Isabelle is an interactive theorem prover that supports several types of formal logic systems. Isabelle/HOL is Isabelle's specialization of Higher Order Logic (HOL). HOL is a type-based logic whose system resembles the one from functional programming languages [8]. Existing types can be classified [5] into:

- basic types, for example the bool(boolean), nat($\mathbb{N}$) or int($\mathbb{Z}$);
- type constructors, for example list and set types. Type constructors are written postfix, that is, after their arguments. For example, nat list is the type of lists whose elements are natural numbers.
- types of functions are denoted by "$\Rightarrow$";
- types of variables are denoted by $'a,' b$, etc.

Terms are represented as in functional programming by applying functions to certain types of arguments. If we have f a function of type $\tau_1 \Rightarrow \tau_2$ and t is a term of type τ_1 then ft is a term of type τ_2. In Isabelle the notation $t :: \tau$ is used to represent that the term t is of type τ. Isabelle's proofs are structured in theories. A theory is a collection of types, functions and theorems, just like a module in a programming language. A theory has the following format:

```
theory T
imports B1 ... Bn
begin
statements , definitions , proofs
end
```

Where B1 ... Bn are the names of the existing theories on which the T theory is based. Each T theory must be in a file called T.thy. HOL contains a Main theory, which contains all predefined basic theories, such as arithmetic, lists, or sets. A theory can include a list of more .thy file, this way we only have to include "AutoCorres.AutoCorres" to have all theories needed for parsing and basic proofs.
The proofs can have the form of theorems or lemmas, both can be used inside other proofs. There are some keywords for applying them in order to reach some goal [11], for example the most common keywords used in our proofs are: *unfolding x* - which applies the definition of *x* on the current goal and *apply x* - which refers to other theorems or set of rules to be used.

B. Parsing C to Isabelle

Approaching the C language from the perspective of obtaining a semantic model on which valid reasoning can then be built is an important contribution of the seL4 system and deserves to be studied in detail. Several steps were taken to translate the C code from seL4 into Isabelle [1]:

1) Each C source file is parsed by an external C preprocessor, which extends "#include" formulas and macro commands as well as other preprocessing directives;
2) The resulting C file is then translated into Simpl by the C-to-Isabelle analyzer [6] developed by M. Norrish, H. Tuch and G. Klein;
3) Each structure used in the program is represented by a record in Isabelle;
4) Local and global variables are analyzed to generate two new types: a record called globals, which contains information about global variables, and *"a myvars"* record for local variables;
5) Each individual function is translated into its equivalent representation in Simple language;
6) Demonstrations are performed on the generated functions, to specify which global variables modify the function.

The post-translation steps in Simpl are embedded in the AutoCorres tool developed by D. Greenaway and specified in detail in his doctoral dissertation [1]. Because this tool uses the result of the C-to-Isabelle parser as input, AutoCorres supports the same subset of the C language. Programs that support loops, function calls, cast between various types, pointer arithmetic, structures, and recursion are supported. On the other hand, as described in the verification of the seL4 system in general, the following are not supported: references to local variables, expressions "goto" or "switch", unions, arithmetic operations in floating point or the use of pointers to functions.
In his paper [1], D. Greenaway describes using the illustrative example shown below, how one can go from the implementation in C of a simple function to the result of the C-to-Isabelle parser - with which it is quite difficult to work, and then to the final form that will be reached after running the AutoCorres tool on the result.
In essence, the purpose of the AutoCorres tool is to abstract the low-level representation given by the C-to-Isabelle parser into a high-level one by the following:

- performing the conversion between the deeply embedded representation (as will be described below) and the shallowly embedded one;
- abstraction of arithmetic operations at machine word level (represented on 32 bits) in operations on the whole set of integers and natural numbers;
- abstracting the heap memory at byte level in separate areas depending on the data type using the Burstall-Bornat model [9];
- simplifying the code and translating the type of variables from the representation into Simple into a form that is easy to reason in Isabelle.

Deeply vs shalowly embedded representations. Before we can begin to formally reason about a program, we must first translate it into the logic used by our theorem demonstrator. To bring the C code into Isabelle, it was necessary to decide which aspects of the code would be translated into the demonstrator logic. If the emphasis is on the structure of program C and its preservation in Isabelle, we say that deeply embedded representation was used, and if the semantics of the program are important in the translation process, then we have a shallowly embedded representation of the source code in Isabelle logic. AutoCorres has as its role the conversion between the structural representation of the C language given by the C-to-Isabelle parser in the semantic representation on which reasoning can then be performed.
An illustrative example used by Greenaway [1] to explain the difference between the two forms of code representation is

given by the observation on the following equation: 2 + 2 = 4. If we want to prove that the left side is equal to the right one, we can perform the addition (treating the expression as shallowly embedded) and state that the sentence is true. But if we look at the structure of the equation (deeply embedded), on the left we have 3 characters and on the right only one. So we can say that the two parts are not equal because we did not give any semantics to the assembly operation and its terms. Structural treatment is not helpful if we want to prove certain statements about a program. For this reason, the semantic representation of the C code is an important contribution in the verification of the seL4 kernel, and this is done through the Greenaway's AutoCorres tool.
The semantic representation obtained with AutoCorres aims to capture the behavior of C programs, the representation in Isabelle that the program can change the overall state of the system, can contain loops for which there is a possibility that they may not end, may have exceptions or other errors and it can treat them differently or it can contain random choices. Many of these requirements are covered by the extensive use of existing monads in Simpl (Skip, Basic, Cond, Guard, etc.) and the addition of new constructions such as: gets, return, whileLoop. The latter brings the greatest similarity between the source imperative language and the functional one in which it is modeled.

C. C subset limitations

Of course, C-language constraints were required, so we are using only a subset of the C99 standard specifications [7]. One of the most relevant restrictions is that pointers to functions are not supported. Pointer data types are defined as functions that must return data stored at those addresses. In this case, if the pointers refer to the address of a function, there is no guarantee that the reference cannot be circular, and the address of the function must also be resolved.
Also, sequences that could change the flow of the code, such as code jumps using "goto" or "switch", are not supported. Other components that are not supported due to the unpredictability of the GCC compiler when it might try to optimize the positioning of data in memory are unions or bit fields.

D. Limitation case - calling function pointers

The limitation of not being able to call functions that were set via their address to a function pointer was a major drawback in the integration of the driver because we need to address programable tasks to be executed in the future. The tasks may come from the device driver, the timer or other additional file but the main loop should only the the corresponding function that was set via its pointer. A simplified program would be the one below where the C-to-Isabelle parser fails to translate the last function because the call to *foo()* is done via the function pointer *p_fun*.

```
static int counter;

void foo(void) { counter++; }

void (*p_fun)(void);

void set_function(void) { p_fun = foo; }

void call_function(void) {
    if(p_fun) p_fun();
}
```

Currently, there are a few workarounds valid for this situation, but all of them cannot provide the full proof, they only skip certains parts of the program or proofs in order to provide a translation avoinding the part were the function pointer is used:

- skip parsing the *call_function* by adding the */** DONT_TRANSLATE */* adnotation, this variant was used during the proof because the other translations ar enot affected, we only had to avoid proving some properties that involved the function pointer;
- add the following adnotations before parsing the C file, this will assert those theorems as axioms rather than try to prove them:

```
declare [[quick_and_dirty = true]]
declare [[sorry_modifies_proofs = true]]
```

- add some other adnotations before parsing the C file, this will not try to prove the theorem involded in function pointers:

```
[[calculate_modifies_proofs = false]]
```

VI. Driver verification

Drivers are pieces of software part of monolithic kernel (but can also run is userspace), whose purpose is to interact with hardware devices or buses and to provide a interface between the kernel and those components. We choose to verify drivers as a further development of seL4 verification because the drivers are independent enough from the kernel structure, thus the verification process does not need to take into consideration the particularities of the kernel where the driver came from.
The main objective of driver modeling in Isabelle is to generalize the verification of kernel drivers. We started from an OpenBSD driver which was adapted to only use basic C function instead of kernel depended ones. This simulation comes at a cost, we have to assume that the rest of the system works correctly because the verification will cover only the driver functionalities.
The parts for which we applied this assumption are the hardware related components like bus communication and reading/writing form device registers. We suppose that the bus works correctly and the register behavior matches the datasheet specifications. In general cases, this separation between the software driver and the hardware components is helpful to investigate if some strange device behavior is generated by a hardware fault or the software does not work as expected.

A. OpenBSD octrng driver

The driver used for prototyping seL4 verification is a hardware random number generator for Octeon boards [15]. We choose this driver because it has a small configuration sequence and it is pretty isolated from the OpenBSD kernel (there are no major dependencies from other drivers or kernel components). The driver structure is very simple, it has two important functions:

- *octrng_attach* - driver initialization routine, its purpose is to configure the hardware in order to start generating random values. It maps the registers of the device in local address space and sets bits 62 and 63 (OCTRNG_ENABLE_OUTPUT, OCTRNG_ENABLE_ENTROPY) from the register 0x1180040000000 (OCTRNG_CONTROL_ADDR). By setting the output and entropy flags, the device will start generating random values. The output is then read every 10 miliseconds.
- *octrng_rnd* - a periodic call of this routine will provide each time the result from the device. The random value is retrieved from the register 0x1400000000000 (OCTEON_RNG_BASE + OCTRNG_ENTROPY_REG). This value will be added then to the entropy pool each time this function is called.

B. Sample driver for verification

Before parsing the C driver implementation into Isabelle, some kernel particularities had to be implemented separately because we extracted only the driver file, not the other kernel functionalities included by the headers. We only had four components that had to be mimicked in new functions or files in order to fulfill the octrng driver interactions:

Bus communication. In the original driver, the bus is accessed via *bus_space_x()* functions, where *x* can refer to register mapping, reading or writing on the bus. In our case, we replace the buss access with simple reading or writing to local memory. This way, the bus is supposed to always deliver the read/write commands except the timing (a bus write may need more time than writing to a local variable). But in this case the timing is not relevant because all actions are done sequentially.

Device registers. Because the bus communication is removed, we had to find a way to express the register behaviour using local memory. So the implementation separated the useful registers in a static structure. For octrng driver, we only need the control register, so we have the structure *rng_regs* with only one member *control_addr* which will be the absolute address of the control register.

```
static struct reg {
  unsigned long control_addr;
} rng_regs;
```

Reading or writing the devices register is usually done by mapping the physical registers in memory. Usually this involves a communication with the device via the bus on which it is located. For our model however, the device is just a representation of the actual one, so there are no physical registers and our transfers on the bus are simply reading or writing operations from the device register structure.

There is a important point for our model resemble as much as possible the real one: the internal register behaviour. In our case, only some registers are important and we have to cover these cases: enabling the output bit, the entropy bit and reading the control value. These cases are implemented inside two functions *set_register* and *get_register*. The first function modifies the required register with a given value while the second one reads a register or returns the value of the current timer if both output and entropy flags are set.

Global timer. The timer has two functions thorough the driver implementation. One is inherited from the original driver: scheduling the random function to be repeated every 10 miliseconds. The other has been added for verification purposes while the actual hardware device is missing: mimicking the random value by returning the timer value instead of the random value from the device register. Because we do not have access to an actual timer either, we will simply use a global variable that will be incremented by the *idle()* function each time the main loop chooses a task to be scheduled.

Task scheduling. The initialization call to the *attach* function of the driver is done from a separate file that has the purpose to simulate a very simple scheduling of tasks. Basically, this is a loop guarded by a timeout, and we check for tasks waiting to be scheduled during each iteration. This loop also calls the *idle* function to increase the global timer. The tasks are stored into a static array of tasks, each task having the following components:

```
typedef struct {
    int timeout;
    unsigned long start;
    void (*timeout_fun)(void);
} Task;
```

Scheduling a task to run some *foo()* function after 3 time units in the future means to call the function that sets a new task in the task queue with *timeout* set to 3, *start* set to current time value and *timeout_fun* pointing to the *foo()* function. The task queue is a circular buffer, each task addition increments the index of the newest task added. After finnishing a task, it will be erased from the array.

C. Proving driver function correctness in Isabelle

Our driver representation is translated into Isabelle/HOL by applying successively the C-to-Isabelle parser and then the AutoCorres tool. A limitation of these tools is that we can only parse one .c file at a time and provide one corresponding .thy file. In seL4, some of the .c files have produced isolated theory files and these theories are then included where needed. However, there is a starting point to parse all the other files and this is the *kernel_C* preprocessor output file. We used the same approach by including the *octrng* driver and the timer implementation inside the .c file containing the main loop. This file can be reprocessed by running the command

$gcc - DINCLUDE\_C\_FILES - E - CCrun\_tasks.c >$ $run\_tasks.c\_pp$ and producing the *run_task.c_pp* file which will be used as input for the Isabelle translation.

The *INCLUDE_C_FILES* compilation flag was added in order to easely switch between including the header and .c files into the main loop. The .c files are needed to be included instead of the headers because the otherwise we will not get the translation of the function implemented in the additional .c files.

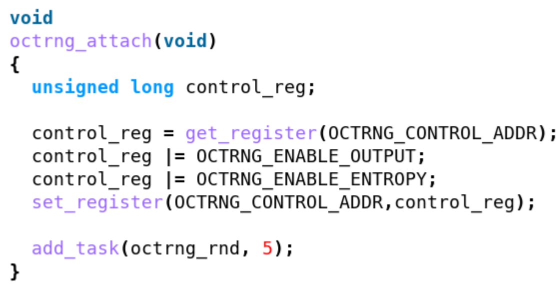
The structure of the theory file contains the import statements that include AutoCorres theories and all the helper theories. The C-to-Isabelle parser is applied by simply declaring the input preprocessed file and calling the *install_C_file*. After this step we have all the C function translated into SIMPL theorems. In order to obtain the final representation of those theorems, the AutoCorres tool is applied on the target file. Inside the main context of this theory we can start defining new terms, functions or proving new lemmas about the translated C functions.

```
theory Run_Tasks
  imports
    "AutoCorres.AutoCorres"
begin

external_file "run_tasks.c_pp"
install_C_file "run_tasks.c_pp"
autocorres "run_tasks.c_pp"

context run_tasks begin
definitions, lemmas, theorems ...

end
```

After the translation into Isabelle is finished, we can access the functions from C as theorems in Isabelle. For example, a C function *foo* is represented as a theorem named *foo_def*. The definition of a function can be viewed in JEdit by typing $thm foo'\_def$. All the additional functions implemented in all the included files will be translated. We will analyze only the two functions related to the *octrng* driver: *octrng_attach* and *octrng_rnd*. Also the constants defined in .c file needs to be redefined if we want to use the same names through the new theorems or lemmas. The C constants have been translated directly into their values, but we can give a name to the same values as Isabelle definitions (for example the enable output flag will be defined in Isabelle as *definition* “OCTRNG_ENABLE_OUTPUT $\equiv (1 << 1)$ :: word32“).

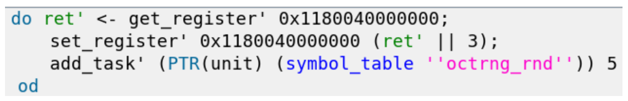
The attach function. This is where the device configuration takes place and also the task of periodically checking the value is programmed. The Isabelle translation of the function in Fig. 2 matches the C code: the flags of the control register are set accordingly.

What we would like to verify from this function is that, after the execution of *octrng_attach* the device state is ready for generating random values, i. e. the control register is set corectly. We can model this inside a lemma in the form of a Hoare triple $\{P\}C\{Q\}$, where P and Q are the precondition and respectively the postcondition, C is the executed program.



```
void
octrng_attach(void)
{
  unsigned long control_reg;

  control_reg = get_register(OCTRNG_CONTROL_ADDR);
  control_reg |= OCTRNG_ENABLE_OUTPUT;
  control_reg |= OCTRNG_ENABLE_ENTROPY;
  set_register(OCTRNG_CONTROL_ADDR,control_reg);

  add_task(octrng_rnd, 5);
}
```

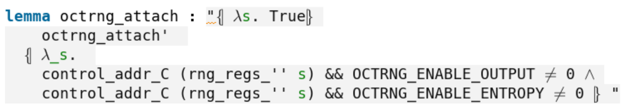
Fig. 1. *octrng_attach* C implementation



```
do ret' <- get_register' 0x1180040000000;
    set_register' 0x1180040000000 (ret' || 3);
    add_task' (PTR(unit) (symbol_table ''octrng_rnd'')) 5
 od
```

Fig. 2. *octrng_attach* Isabelle definition

In our case, we want to verify that running the $octrng\_attach$ program function in any program state, the control register will have set to 1 the flags for enabling output and entropy. So the precondition is always $True$ because there are no requirements and int the postcondition we check the bits of the flags. This proof is straightforward, we only need to use



```
lemma octrng_attach : "⦃ λs. True⦄
    octrng_attach'
  ⦃ λ_s.
    control_addr_C (rng_regs_'' s) && OCTRNG_ENABLE_OUTPUT ≠ 0 ∧
    control_addr_C (rng_regs_'' s) && OCTRNG_ENABLE_ENTROPY ≠ 0 ⦄ "
```

Fig. 3. *octrng_attach* verification lemma

unfolding to apply all the functions and definitions needed. The weakest precondition tool (*wp* command) will compute the necessary precondition that we have to prove further. All the provided goals can be derived automatically from the function definition. Except for the bit operations for which we need to explicitly apply the *word_bitwise* theorems.

The periodic rng function. This function should constantly retrieve the random value and add it to the pool. Because we only have the driver part and not the rest of the OpenBSD kernel, this value will be the timer value and the randomness poll is just a global variable which will be updated by calling this function. The C implementation just reads the value of the output register and saves it into *rand_value* global variable, then schedule another function execution after 10 time units. The Isabelle representation matches the same behavior, the only thing different here is that all the global variables from the C program are now represented as Isabelle terms, for example the integer *rand_value* is translated in Isabelle as *rand_value_”* a term of type sword32 (signed word on 32 bits). The verification lemma for *octrng_rnd* has a few more preconditions than the initialization function because we have to firstly make sure that the function can be executed (the task queue is not full) and then that the driver is configured properly (the output and entropy flags are set). The additional clause $\lambda s.timer\_s = a$ represents that is any given state s the global timer variable may have a label a for its value.

```
void
octrng_rnd(void)
{
  unsigned int value;

  rand_value = get_register(OCTRNG_ENTROPY_REG);
  add_task(octrng_rnd, 10);
}
```

Fig. 4. *octrng_rng* C implementation

```
do ret' <- get_register' 0;
    modify (rand_value_''_update (λa. ret'));
    add_task' (PTR(unit) (symbol_table ''octrng_rnd'')) 10
od
```

Fig. 5. *octrng_rng* Isabelle definition

What we want to prove it that the same value will be set to the global *rand_value* and this is the precondition $\lambda\_s.rand\_value\_s = a$. The verification will be done using the same proofs as for the previous lemma: first we apply the definition of all functions used and the apply the weakest precondition tool. The goals obtained this way are easily to prove by applying the auto method.

This lemma could be improved by adding some other specifications like checking that the same function will be called after 10 time units or that the function will be always called in time. The proofs that involve task scheduling were avoided because the translation of the funciton that runs the actual task is not parsed as described in the C subset limitation section.

Main loop and other lemmas. The driver functions are bound toghether inside a small program that simulate a simple scheduler of tasks. The main loop does the initialization of the environment including the call of the *octrng_attach* function, then a main loop will check for each time unit if there are tasks whose timeout expired so their function has to be run. We can add lemmas for those additional functions mainly because some of they might be useful in proving other properties. For example, a simple function *idle* increases the global timer after each iteration of the main loop. The lemma for this function could verify that the timer is modified exactly by 1 after running this function in any program state. Its proof

```
lemma octrng_rnd:
 "{|
   λs. timer_'' s = a ∧
   running_tasks_'' s < MAX_QUEUE ∧
   current_task_'' s < MAX_QUEUE ∧
   control_addr_C (rng_regs_'' s) && OCTRNG_ENABLE_OUTPUT ≠ 0 ∧
   control_addr_C (rng_regs_'' s) && OCTRNG_ENABLE_ENTROPY ≠ 0
  |}
  octrng_rnd'
  {| λ_s. rand_value_'' s = a|}!"
```

Fig. 6. *octrng_rng* verification lemma

```
lemma idle_increases [simp]:
 "{| λs. timer_'' s = a |}
  idle'
 {|λ_s. timer_'' s = a + 1|} "
```

Fig. 7. A lemma for *idle* function

is evident, we only have to apply weakest precondition tool and then the *auto* method for applying the simplifications. A proof that is more interesting is the one that states the main loop runs until a timeout occurs. This is done by limiting the timer with a maximum value, if this value is reached no other task will be called. The difference between this lemma's proof

```
lemma main_function:
  "{| λs. timer_'' s = 0 ∧ running_tasks_'' s = 0 |}
   main'
   {|λ_s. timer_'' s = TIMEOUT|}!"
```

Fig. 8. A lemma for *main* function

and the other is that here we have loops so we have to first provide a proof that those loops ends. Because the function that actually runs the task is not parsed, we will only prove the main loop, the one that increases the timer via the *idle* function and continuously run until timeout. This aspect is specified in the *main_function* lemma: if we call the main function from a state where the timer is not started and there are no running tasks, then at the end the timer will have reached the timeout value. In order to prove this loop we have to specify and invariant and a measure:

- the invariant is a property that has to be true before, during and after the main loop ends - because we want to prove something about the timer value, the invarant specifies that at any state of the loop, the timer will have a value between 0 and the timeout limit;
- the measure is a value that has to decrease each loop iteration - following the same model, the measure in our case is the distance between the timer and the timeout limit;

```
lemma main_function:
  "{| λs. timer_'' s = 0 ∧ running_tasks_'' s = 0 |}
   main'
   {|λ_s. timer_'' s = TIMEOUT|}!"
```

Fig. 9. Definition of invariant and measure for main loop

We can apply these two definitions via the *whileLoop_add_inv* monad and obtain a proof goal that can be further broken into smaller goals using the weakest precondition tool.

VII. Conclusions

The seL4 project has made outstanding contributions to the verification of C programs, offering a comprehensive set of proofs ranging from common security issues (integrity, confidentiality, availability) to detailed modeling of code behavior after compilation. These proofs were performed in order to obtain a kernel that could be used in real life scenarios. The importance of the project is augumented by the solutions implemented on the way to achieving this goal:

- functional correctness proof of the seL4 microkernel;
- implementing proofs on the binary code;
- creation of a system through which the correspondence between the binary file and the source code from which it was compiled is verified;

- analysis of the C code and its automatic translation into a functional language on which reasoning can be built.

An encouraging conclusion of this project is that the automatic abstraction of the source code using the AutoCorres tool reduces the complexity of the effort to demonstrate [2] the properties of the system.
This direction could facilitate the inclusion of verification as an important step in the development of software projects. With this idea in mind, we tried to include a kernel driver, but some other components of a kernel or general C programs could be verified if they meet the language subset limitations. These constraints seem to be the major barrier for expanding the C-to-Isabelle and AutoCorres tools into a generalized framework for safely translating C code into theorems. If the step of this translation could allow all programming language features to be used, some other C critical programs could be brought into Isabelle and reason about the correctness of their specifications.

References

[1] D. Greenaway, *Automated proof-producing abstraction of C code*, Ph.D. Dissertation, School of Computer Science and Engineering, University of NSW, Sydney, Australia, August 2014, web source: https://ts.data61.csiro.au/publications/nicta_full_text/8758.pdf.

[2] D. Greenaway, J. Lim, J. Andronick, G. Klein, *Dont Sweat the Small Stuff - Formal Verification of C Code Without the Pain*, NICTA and UNSW, Sydney, Australia, web source: https://ts.data61.csiro.au/publications/nicta_full_text/7629.pdf.

[3] G. Heiser, *The seL4 Microkernel - An Introduction*, White paper, The seL4 Foundation, Rev. 1.2, 2020.

[4] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewel, R. Kolanski, G. Heiser, NICTA and UNSW, *Comprehensive Formal Verification of an OS Microkernel*, Sydney, Australia, ACM Transactions on Computer Systems, Vol. 32, No. 1, Art. 2, February 2014.

[5] 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*, ACM Symposium on Operating Systems Principles, MT, USA, 2009.

[6] H. Tuch, G. Klein, M. Norrish, *Types, Bytes, and Separation Logic*, Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Nice, France, 97-108, 2007.

[7] International standard ISO/IECISO/IEC 9899, Committee Draft, September 7, 2007, web source: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[8] R. Bird and P. Wadler, *Introduction to Functional Programming*, Prentice Hall, Englewood Cliffs, 1988.

[9] R. Burstall, *Some techniques for proving correctness of programs which alter data structures*, Machine Intelligence 7, 1972.

[10] T. Nipkov, *Programming and Proving in Isabelle/HOL*, web source: https://isabelle.in.tum.de/doc/prog-prove.pdf, checked on July 2021.

[11] Web source Isabelle homepage: https://isabelle.in.tum.de/.

[12] Web source L4Ka Pistachio: https://www.l4ka.org/65.php.

[13] Web source L4Re Fiasco: https://l4re.org/fiasco/build.html.

[14] Web source L4Ka Hazelnut: https://github.com/l4ka/hazelnut.

[15] Web source: https://github.com/openbsd/src/blob/master/sys/arch/octeon/dev/octrng.c.