No Crash, No Exploit: Automated Verification of Embedded Kernels

Olivier Nicole*†, Matthieu Lemerre*, Sébastien Bardin* and Xavier Rival†‡

*Université Paris-Saclay, CEA List, Saclay, France
†Département d'informatique de l'ENS, ENS, CNRS, PSL University, Paris, France
‡Inria, Paris, France

olivier.nicole@cea.fr, matthieu.lemerre@cea.fr, sebastien.bardin@cea.fr, xavier.rival@ens.fr

***Abstract*—The kernel is the most safety- and security-critical component of many computer systems, as the most severe bugs lead to complete system crash or exploit. It is thus desirable to guarantee that a kernel is free from these bugs using formal methods, but the high cost and expertise required to do so are deterrent to wide applicability. We propose a method that can verify both *absence of runtime errors* (i.e. crashes) and *absence of privilege escalation* (i.e. exploits) in embedded kernels from their binary executables. The method can verify the kernel runtime independently from the application, at the expense of *only a few lines* of simple annotations. When given a specific application, the method can verify simple kernels without any human intervention. We demonstrate our method on two different use cases: we use our tool to help the development of a new embedded real-time kernel, and we verify an existing industrial real-time kernel executable with no modification. Results show that the method is fast, simple to use, and can prevent real errors and security vulnerabilities.**

I. Introduction

The safety and security of many computer systems build upon that of its most critical component, the kernel, which provides the core protection mechanisms. The worst possible defects for a kernel are:

- *Runtime errors*, which cause the whole system to crash. In machine code, there is no undefined behaviour, but faulty execution of an instruction in the kernel code that raises an exception (e.g. illegal opcode, division by zero, or memory protection error) is assimilable to a runtime error;
- *Privilege escalation*, where an attacker bypasses the *kernel self-protection* and takes over the whole system. In the case of hypervisors, this also corresponds to virtual machine escape.

The only way to guarantee that a kernel is free from such defects is to verify it entirely using formal methods [1]. Manually proving a kernel using a proof assistant [1]–[6] or deductive verification [7]–[10] can ensure that the kernel complies with a formal specification, thus reaching a high level of safety and security. But this is out of reach of most actors in embedded systems development, because of the huge effort of writing thousands lines of proof (e.g. 200,000 for seL4 [1] or 100,000 for CertiKOS [11]) and of the difficulty to find experts in both systems and formal methods. For such actors, the ideal method would be a fully automated verification where the system developers only provide their code and, with very little configuration or none at all, the tool automatically verifies the properties of interest. In addition, a comprehensive verification should carry to the binary executable, as 1. a large part of embedded kernel code consists in low-level interaction with the hardware, and 2. the compilation toolchain (build options, compiler, assembler, linker) may introduce bugs [12].

Recent so-called "push-button" kernel verification methods [13]–[15] are based on symbolic execution [16]–[18], which has several advantages: it is sufficiently precise to handle binary code without getting lost [19], and it natively works with logical formulas and thus can easily be used to prove manually specified high-level properties. On the other hand, symbolic execution as a verification technique suffers from severe limitations, such as the inability to handle unbounded loops, the need to manually provide all the kernel invariants (for CertiKOS^S [14]: 438 lines of specifications for 1845 instructions, i.e. a 23.7% ratio) and difficulty to scale because of path explosion. These limitations are inherent to symbolic execution and cannot be overcome without a radical change of verification method. It explains, for instance, why such prior approaches considered only round-robin schedulers [13]–[15], while priority-based schedulers (as found in RTOSes) working over an arbitrary number of tasks require an unbounded loop.

Our goal is to push the boundaries of automated kernel verification to make it practical for system developers. We focus on embedded kernels, which are characterized by mostly static memory allocation, and by the fact that they typically exist in many variants (depending on the hardware and application), making the automation of the analysis very important. Our contributions are the following:

- We provide a method (Section III) for *fully automated* verification of *absence of privilege escalation* (APE) and *absence of runtime errors* (ARTE) of small OS kernels, *in-context* (i.e. for a given application layout). We eliminate the need for any manual annotation: first by developing a binary-level *abstract interpreter* [20] analyzing the whole *system loop* (kernel code + a sound abstraction of user code) to automatically *infer* the kernel invariants (rather than merely checking them); second by *proving* (Theorem 3) that APE is an *implicit* property, i.e. which does not require to write a specification (like ARTE [21]);

- We propose an extension of the method (Section IV) for *parameterized* kernel verification (i.e. verification of the kernel independently from the applications). Prior works [13]–[15] model memory using a *flat model* (the memory is a big array of bytes, and addresses are represented numerically) which prevents parameterized verification and limits scalability and precision [22] of the verification. We propose a type-based representation of memory (Section IV-E) that solves these issues. We finally handle differently the boot code (whose goal is to establish the system invariant) and the kernel runtime (whose goal is to preserve the system invariant), yielding further precision enhancement (Section IV-D). If our method requires a tiny number of manual annotations, these also replace the *precondition* on the application that would be otherwise needed;
- We conducted a thorough evaluation on two case studies (Section V), where we demonstrate that 1. it is possible to implement a binary-level abstract interpreter sufficiently precise to verify an existing, industrial operating system kernel without modification and with only very few annotations ($< 1\%$ ratio); 2. abstract interpretation is helpful as a continuous integration tool to find defects during kernel development, especially in embedded kernels that have lots of variants; 3. parameterized analysis can scale to large numbers of tasks, while our in-context analysis cannot.

II. Problem and Motivation

We illustrate the problem that our technique solves on a simple[1] embedded system, for a fictitious 8-bit hardware where the user code executes using three registers *pc′*, *sp′* and *flags′*, distinct (e.g. banked) from those used for the kernel execution.

Kernel code. The execution of *user code* (i.e. applicative, unprivileged code) alternates with that of the *kernel* (which is privileged). For illustration purposes, we consider the simple kernel with the code in Fig 1 ("*kernel*" definition) and the data structures in Fig 2 (the syntax is close to that of C, and the **with** annotations can be ignored for now).

The kernel executes whenever an interrupt occurs. If the interrupt corresponds to a preemption (e.g. user code calls a "yield" system call, or the kernel receives a timer interrupt), it saves the values of the registers of the preempted thread ("*save_context*"), determines the next thread to be executed ("*schedule*", here a simple round-robin scheduler), setups the memory protection to limit the memory accessible by this thread ("*load_mt*"), restores the previous register values of the thread ("*load_context*"), and transitions to user code. In this example, memory protection is done using two Memory Protection Unit (MPU) registers (mpu_1 or mpu_2): user code can only access the memory addresses allowed by one of the *mpu* registers, each giving access to one segment (i.e. interval of addresses). In addition, unsetting the *hardware privilege* flag

[1]While this example is simple to keep it understandable, our method extends to complex embedded kernels featuring e.g. shared memory concurrency, complex boot sequence (including dynamic creation of page tables), real-time schedulers, various hardware drivers and system calls (e.g. for dynamic thread creation).

```
register Int8 sp′, pc′ , flags′, mpu1, mpu2;
Thread *cur; Context *ctx; extern Interface if;

kernel(){
  switch( interrupt_number() ){
    case RESET: init(); load_mt(); load_context();
    case YIELD_SYSCALL | TIMER_INTERRUPT:
      save_context(); schedule(); load_mt(); load_context();
    case ... : ... }
}

save_context(){ ctx→pc = pc′; ctx→sp=sp′; ctx→flags=flags′;}
schedule() { cur = cur→next; ctx = &cur→ctx;}
init() { cur = &if.threads[0]; ctx = &cur→ctx;}
load_mt() { mpu1= &cur→mt→code; mpu2= &cur→mt→data; }
load_context(){ pc′= ctx→pc; sp′= ctx→sp; flags′= ctx→flags; }
```

Fig. 1. A simple embedded OS kernel.

```
type Flags = Int8 with (self & PRIVILEGED) == 0
type Context = struct { Int8 pc; Int8 sp; Flags flags; }

type Segment = struct {
  Int8 base;
  Int8 size_and_rights;
} with self.base ≥ kernel_last_addr

type Memory_Table = struct { Segment code; Segment data; }

type Thread = struct {
  Memory_Table *mt;
  Context ctx;
  Thread *next; }

type Interface = struct {
   Thread[nb_thread]* threads;
   (Int8 with self = nb_threads && self ≥ 1) threads_length; }
```

Fig. 2. Types used to interface the kernel and application, with annotations.

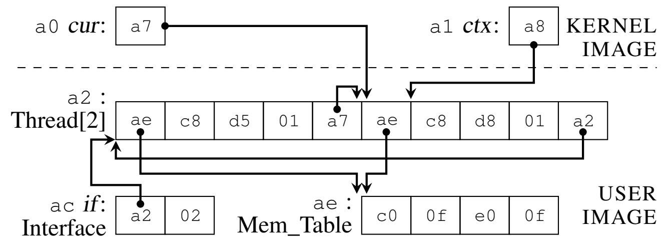

Fig. 3. Example of a memory dump of the system.

(*PRIVILEGED* bit in the *flags′* register) forbids user code to change the values of mpu_1 and mpu_2.

A special interrupt (RESET) corresponds to the system *boot*, where the kernel initializes the memory ("*init*"). Additional interrupts (the other **case**s) would perform additional actions, like other system calls or interfacing with hardware.

Memory layout and parameterization. Let us now look at the memory layout of the kernel (Fig. 3). The kernel is *parameterized*, i.e. independent from the *user tasks* running on it: both the kernel and user tasks can be put in separate executable *images* and linked at either compile-time or boot-time. This separation is necessary for closed-source kernel vendors, and also allows certifying the kernel image independently to reuse this certification in several applications.

For instance, in Fig. 3, addresses `a0..a1` come from the kernel executable (the *kernel image*), while addresses `a2..b2` come from the user tasks executable (the *user image*). While Fig. 3 represents a system composed of two threads sharing the same memory table, the same kernel image can be linked to another user image to run a system of 1000 threads, each with different memory rights.

A consequence of this parameterization is that the addresses of many system objects (e.g. of type `Thread` or `Memory_Table`) vary and are not statically known in the kernel. It makes the code much harder to analyze and explains why in existing automated verification approaches [13]–[15], these objects must be statically allocated in the kernel, hardcoding a fixed limit on the number of (e.g.) `Threads`.

Interface and precondition. The kernel and user images agree on a shared *interface* to work together. In our example, this interface consists in having the variable *if* at an address known by both images; a common alternative is to use the bootloader to share such information [23]. Also, the system is not expected to work for any user task image with which the kernel would link: for instance the system can misbehave, if *if*$\rightarrow$*threads* points to *ctx*, to the kernel code or to the stack. Thus, system correctness depends on some *precondition* on the provided user image.

Verifications.To formally verify that a system satisfies properties such as APE or ARTE, one has to find *invariants*, i.e. properties that are inductive and initially hold. Experience has shown that in OS formal verification, *"invariant reasoning dominates the proof effort"* [2]: e.g. in seL4, 80% of the effort was spent stating and verifying invariants [1]. Contrary to existing manual [1]–[6], semi-automated [7]–[10], or "push-button" [13]–[15] methods, our method automatically infers these invariants. We can perform our verification in two settings:

- *in-context*, i.e. for a given user image layout. The invariant that we compute fully automatically for the example kernel is given in Section III-B;
- *parameterized*, i.e. independently from the user image. In this case, a small number of annotations is needed to describe the interface between the kernel and admissible user images; once this is done, the invariant is computed fully automatically (for our kernel, it is given in Section IV-C).

III. Fully automated in-context analysis

We now present our method for static analysis of the kernel *in-context*, i.e. for a given user image. The goal of this section is to provide a method that can verify APE and ARTE on small, simple kernels with no human intervention.

A. Method overview

Our method builds upon the following key points:

Key 1: Closed-loop kernel verification. Contrary to existing methods, we verify the kernel code *within* the whole *system loop* (Fig 4) comprising both the kernel and user code. Still, we do not need to analyze the user code itself. Instead, we abstract it by an *empowered* transition, corresponding to the execution of any sequence of instructions (Section III-D).

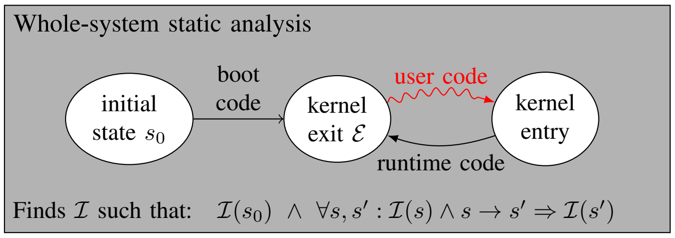
Whole-system static analysis
initial state s_0
boot code
kernel exit $\mathcal{E}$
user code
kernel entry
runtime code
Finds $\mathcal{I}$ such that: $\mathcal{I}(s_0) \wedge \forall s, s' : \mathcal{I}(s) \wedge s \rightarrow s' \Rightarrow \mathcal{I}(s')$

Fig. 4. Fully-automated in-context analysis on the system loop.

Key 2: Binary-level abstract interpretation. We built a sound static analyzer (Section III-C) based on *abstract interpretation* [20] that works directly on the binary executable of the kernel. This static analyzer simultaneously reconstructs the control-flow graph (CFG) of the kernel, and attaches to every instruction an *abstract state*, summarizing the set of all the reachable values for each register and memory address. The result of the analysis is an (inductive) invariant guaranteed to be correct by construction, i.e. to encompass all behaviours of the code.

Key 3: Kernel implicit properties. We use the computed invariant to verify whether some *implicit properties* hold. Implicit properties are properties which do not require writing a manual specification, and thus can apply to any kernel. One such property is the well-known absence of run-time errors (ARTE). More surprisingly, absence of privilege escalation (APE) is also implicit (Theorem 3, Section III-D).

B. Illustration on the example kernel

We illustrate our method on the example kernel of Fig. 1.

User code transition. The real user-code transition (which does not appear in the example kernel code) consists in executing the instruction pointed to by the *pc* register. The empowered transition is equivalent to nondeterministic assignment of arbitrary values to:

- the user registers pc', sp', and $flags'$;
- the memory locations that are not protected by either mpu_1 or mpu_2, if $flags'$ has the *PRIVILEGED* bit unset;
- every memory location, if *PRIVILEGED* is set in $flags'$.

Note that the implementation of this transition is not specific to the kernel that we analyze (it is not "hardcoded"): it is only specific to the hardware on which the kernel is implemented.

Computed invariant. Using our binary-level abstract interpreter on the system composed of the kernel code with the special user code transition, starting with the initial state of Fig. 3, our analysis computes an inductive invariant which implies[2] that, after boot:

- Addresses `a0` (*cur*), `a6` and `ab` hold either `a2` or `a7`;

[2]The invariant that we actually compute is flow-sensitive, and also contain information about the kernel code, control-flow graph, stack variables and registers, which we omit for the sake of brevity.

- Address `a1` (*ctx* variable) can hold values `a3` or `a8`;
- Register mpu_1 always holds `ae` and mpu_2 always holds `b0`;
- Addresses `a2` and `a7` always hold `ae`;
- Addresses `a5` and `aa` and the `flags`$'$ register can hold any value with the *PRIVILEGED* bit unset;
- Addresses `a3`, `a4`, `a8`, `a9`, and all the other registers can have any value;
- Addresses in the range [`c0..cf`] and [`e0..ef`], (made accessible by the *mpu* registers) can have any value;
- The contents of all the other addresses is constant.

This automatically computed invariant must be manually written in prior automated methods [13]–[15].

Verifying implicit properties. To verify APE, the invariants imply that when the user code executes, the system is set up such that execution is not *PRIVILEGED*, and the memory protection tables prevent the user code from modifying kernel data. Actually, the mere presence of a non-trivial invariant implies that APE is impossible (details in Section III-D). Verifying ARTE (i.e. no faulty execution of an instruction) is simple as our invariant incorporates the CFG (i.e. the set of all instructions executable by the kernel) and computes a superset of values for every operand of every instruction.

C. Sound and precise static analysis for binary code

Abstract interpretation. *Abstract interpretation* [20], [24] is a general method for building static analyzers that automatically *infer program invariants* and verify program properties. Abstract interpreters are *sound*, in the sense that the invariants that they infer are correct by construction. Abstract interpreters have been successfully deployed to verify large safety- or mission-critical systems from their source code [21], [25].

In essence, abstract interpretation consists in propagating an abstract state that represents a superset of all the possible values for memory and registers, until a fixpoint is reached. It can be seen as a generalization of data flow analysis [26], with more general domains (e.g. intervals, polyhedra) and widening operators to handle loops [20].

Machine code analysis is challenging. Binary-level static analysis is particularly challenging [22], [27], [28] as 1. the control-flow graph is not known in advance because of computed jumps (e.g. `jmp @sp`) whose resolution requires runtime values, 2. memory is a single large array of bytes with no prior typing nor partitioning, and 3. data manipulations are very low-level (masks, flags, legitimate overflows, low-level comparisons, etc.). In other words, it is prone to imprecisions, and imprecisions can often snowball, up to a point that the analyzer can only return the set of all states (i.e. the trivial invariant, denoted $\top$).

Our precise binary-level static analysis. To have sufficient precision, our analyzer builds upon state-of-the-art techniques for machine code analysis. The full formal description of our static analyzer is out of the scope of the paper, but is available in a technical report [29]. We summarize here the most important techniques that we use:

- *Control flow:* Control and data are strongly interwoven at binary level, since resolving computed jumps requires to precisely track values in registers and memory. Following Kinder [30] or Védrine et al. [27], our analysis computes simultaneously the CFG and value abstractions;
- *Values:* We mainly use efficient non-relational abstract domains (reduced product of the signed and unsigned meaning of bitvectors [28], congruence information [31]), complemented with symbolic relational information [28], [32], [33] for local simplifications of sequences of machine code;
- *Memory:* Our memory model is ultimately byte-level in order to deal with very low-level coding aspects of kernels. Yet, as representing each memory byte separately is inefficient and imprecise, we use a stratified representation of memory caching multi-byte loads and stores, like Miné [34]. Moreover, we do not track memory addresses whose contents is unknown;
- *Precision:* To have sufficient precision, notably to enable strong updates [24] to the stack, our analysis is flow-sensitive and fully context-sensitive (i.e. inlines the analysis of functions), which is made possible by the small size and absence of recursion typical of microkernels. Moreover, we unroll the loops when the analysis finds a bound on their number of iterations;
- *Concurrency:* We handle shared memory zones through a flow-insensitive abstraction making them independent from thread interleaving [25]. Our weak shape abstract domain (Section IV-E) represents one part of the memory in a flow-insensitive way. For the other zones we identify the shared memory zones by intersecting the addresses read and written by each thread [35], and only perform weak updates on them.

D. APE is an implicit property

We will show that the mere existence of an invariant implies absence of privilege escalation.

We model the execution of code as a transition system $\langle \mathbb{S}, \mathbb{S}_0, \rightarrow \rangle$, where $\mathbb{S}$ is the set of all states, $\mathbb{S}_0$ the set of initial states, and $\rightarrow$ corresponds to the transition from one instruction to the next.

Definition 1 (Privilege escalation). *We define* privilege escalation *of a transition system* $\langle \mathbb{S}, \mathbb{S}_0, \rightarrow \rangle$ *as reaching a state which is both* privileged *and* not kernel-controlled*.*

Thus, an attacker can escalate its privilege by either gaining control over privileged kernel code (e.g., by code injection), or by leading the kernel into giving it its privilege (e.g., by corrupting the *flags*$'$ register).

The model that we analyse is $\langle \mathbb{S}, \mathbb{S}_0, \rightsquigarrow \rangle$, where we have replaced the normal transition $\rightarrow$ (corresponding to the execution of the next instruction) by the *empowered* transition $\rightsquigarrow$ (which non-deterministically executes any sequence of instructions permitted by the hardware). We call this transition *empowered* since it gives more power to the user to attack the kernel, as formalized in the following theorems:

Theorem 1. *The set of reachable states for the $\langle \mathbb{S}, \mathbb{S}_0, \rightarrow \rangle$ transition system is included in the set of states reachable for $\langle \mathbb{S}, \mathbb{S}_0, \rightsquigarrow \rangle$.*

Proof. This follows directly from the fact that for every s, $\{s' : s \rightarrow s'\} \subseteq \{s' : s \rightsquigarrow s'\}$ $\square$

Corollary 1. *If there are no privilege escalation in the transition system $\langle \mathbb{S}, \mathbb{S}_0, \rightsquigarrow \rangle$, there are also no privilege escalation in the transition system $\langle \mathbb{S}, \mathbb{S}_0, \rightarrow \rangle$*

Proof. This is the contrapositive of the fact that if a state exists in $\langle \mathbb{S}, \mathbb{S}_0, \rightarrow \rangle$ where privilege escalation happens, this state also exists in $\langle \mathbb{S}, \mathbb{S}_0, \rightsquigarrow \rangle$. $\square$

Thus, verifying APE on a $\langle \mathbb{S}, \mathbb{S}_0, \rightsquigarrow \rangle$ system implies APE on the real $\langle \mathbb{S}, \mathbb{S}_0, \rightarrow \rangle$ system.

Now, to turn APE into an implicit property, we rely on the following assumption:

Assumption 1. *Running an arbitrary sequence of privileged instructions allows to reach any state of $\mathbb{S}$.*

This assumption is very reasonable, as privileged instructions can do anything that can be done by software. Using it, we can prove the following theorems:

Theorem 2. *If a transition system $\langle \mathbb{S}, \mathbb{S}_0, \rightsquigarrow \rangle$ is vulnerable to privilege escalation, then the only satisfiable state property in the system is the* trivial state property $\top$, *true for every state.*

Proof. If a privilege escalation vulnerability exists, then any state can be reached (Assumption 1). Hence, the only state invariant of the system is $\top$. $\square$

We define a *non-trivial state invariant* as any state invariant different from $\top$.

Theorem 3. *If a transition system satisfies a non-trivial state invariant, then it is invulnerable to privilege escalation attacks.*

Proof. By contraposition, and the fact that state invariants are valid state properties. $\square$

Theorems 2 & 3 have two crucial practical implications:

- If privilege escalation is possible, *the only state property that holds in the system is* $\top$, making it impossible to prove definitively any other property. Thus, *proving absence of privilege escalation is a necessary first step* for any formal verification of an OS kernel;
- The proof of *any* state property different from $\top$ implies as a byproduct the existence of a piece of code able to protect itself from the attacker, i.e. a kernel with protected privileges. In particular, we can prove absence of privilege escalation automatically, by successfully inferring *any* non-trivial state invariant with a sound static analyzer.

IV. Parameterized analysis

A. Shortcomings of the flat memory model

The fully automated verification works in practice for small, simple kernels, but has some shortcomings. The root of the problem lies in the *flat memory model* (also used by all the previous binary-level automated methods [13]–[15]), i.e. viewing the memory as a single large array of bytes. This model poses three distinct problems:

- *Robustness*, because imprecisions can cascade so much that the analysis can no longer compute an invariant. This would happen for instance if, in our example kernel, the set of possible values for *cur*, $\{\texttt{a2}, \texttt{a7}\}$, was over-approximated by the interval $[\texttt{a2}..\texttt{a7}]$;
- *Scalability*, because of the need to enumerate large sets. For instance, on a system with 1000 tasks, the values for mpu_1 may point to 1000 different arbitrary addresses, which must be enumerated if we want to be robust;
- *Over-specialization*: it is impossible to analyze programs without knowing the precise memory layout of all the data. In particular, the example kernel cannot be analyzed independently from the user image, even if this kernel is independent from the user image.

We propose an abstraction that lifts those three limitations, at the expense of writing a small number of simple annotations.

B. Method overview

Our method extends that of Section III with the following key points:

Key 4: a type-based shape abstract domain. Instead of representing the memory and addresses numerically, we reason (Section IV-E) about the *shape* of the (user image) memory based on the *types* used to access it. Specifically, we verify that the kernel preserves the *well-typedness* of the user image (according to provided types, i.e. Fig. 2 for the example kernel), and we use the types to compute the invariant over the values handled by the kernel. This solves the robustness and scalability issues of the flat model and allows *parameterized analysis*, i.e. to verify the kernel independently from the user image. The kernel memory is still represented numerically, which allows to verify the kernel using its *raw binary*.

Key 5: differentiated handling of boot and runtime code. We propose to handle kernel runtime and boot differently. On the one hand, the kernel runtime *preserves* existing data structure invariants, while the boot code *establishes* them by initializing the structures. This makes the runtime suitable for verification by our shape domain, which aims at verifying the preservation of memory invariants. On the other hand, when the user image is known, boot code execution is mostly deterministic (only small sources of non-determinism remain: multicore handling, device initialization, etc.); this makes it easy to analyze it robustly using fully automated in-context analysis (Section III). Based on these observations we propose a method (Section IV-D) where boot code is verified using the fully automated in-context analysis, and the runtime code is verified using the parameterized static analysis.

Summary: The 3-step method. We propose a 3-step method that relies on these keys:

- *Step 1*: *Lightweight annotation of the interface types*. These types (Fig 2) are known as they are needed to develop applications using the kernel, yet additional annotations are necessary (Section IV-E).

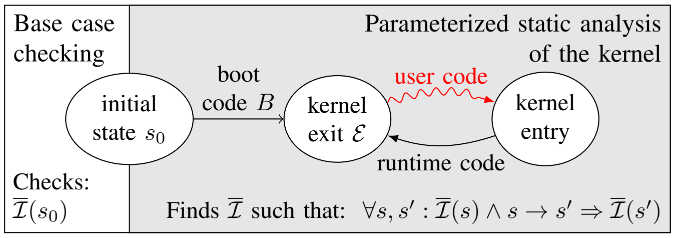
Base case checking
Parameterized static analysis of the kernel
boot code B
user code
initial state s_0
kernel exit $\mathcal{E}$
kernel entry
runtime code
Checks: $\overline{\mathcal{I}}(s_0)$
Finds $\overline{\mathcal{I}}$ such that: $\forall s, s' : \overline{\mathcal{I}}(s) \land s \to s' \Rightarrow \overline{\mathcal{I}}(s')$

Fig. 5. Parameterized verification when user memory starts initialized.

- *Step 2*: *Parameterized static analysis of the kernel*. The result of this step is an inductive property $\overline{\mathcal{I}}$, which is an invariant from any state where this property is true.
- *Step 3*: *Base case checking* consists in verifying that the parameterized inductive invariant initially holds.

C. Illustration on the example kernel

We illustrate how the 3-step method is applied on the example kernel (Fig. 5).

1. Lightweight type annotation. First, we annotate the types accessible from *Interface* with annotations. In larger kernels, this mostly consists in indicating the size of arrays (Fig. 2, line 17), and which pointers may be null. We also specify that the memory zone accessible to a task is disjoint from the kernel address space (Fig. 2, line 7), and that saved user flags are unprivileged (Fig. 2, line 1). Note that these manual annotations are smaller (and simpler) than the full invariant (Section III-B); this is even more true on larger kernels.

Concretely, what we call annotations consists in a set of types like in Fig. 2. These types must be written in a separate file and provided to our tool along with the kernel executable. No binary or source code is annotated; instead, our tool only requires to know the address and the type of the interface entry point (type `Interface` at address `ac` in the example), i.e. the memory location that the kernel should use to access any element of the interface. The set of types is necessarily provided by the kernel (and bootloader) developers as part of the software development kit; the "**with**" predicates required to complete the verification can also be provided by the documentation, by kernel developers, or finding them can be part of the verification work. We give practical details about this in the experimental section (Section V).

2. Parameterized static analysis of the kernel. Assuming that these annotations describe the user image, and that the variable *if* has type Interface $*$, the analysis computes (using the kernel image) a parameterized invariant $\overline{\mathcal{I}}$ (Fig. 5) implying[2] the following invariant (after boot):

- Address `a0` (*cur* variable) has an admissible `Thread`$*$ value;
- Address `a1` (*ctx* variable) has an admissible `Context`$*$ value;
- Registers mpu_1 and mpu_2 have admissible `Segment` value;
- Register *flags'* has an admissible `Flags` value;
- The memory in the user image is well-typed, i.e. matches the types of Fig. 2.

We postpone the exact meaning of the invariant, including definition of "well-typed" and "admissible values" for a type, to

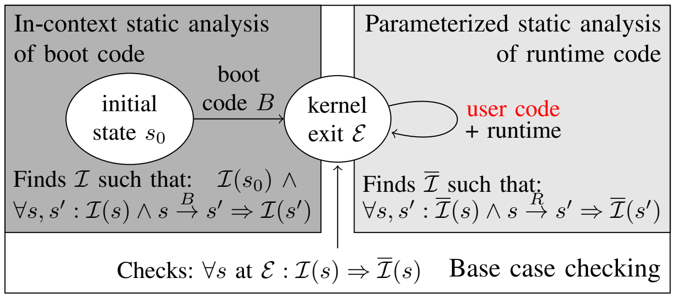
In-context static analysis of boot code
Parameterized static analysis of runtime code
boot code B
initial state s_0
kernel exit $\mathcal{E}$
user code + runtime
Finds $\mathcal{I}$ such that: $\mathcal{I}(s_0) \land \forall s, s' : \mathcal{I}(s) \land s \xrightarrow{B} s' \Rightarrow \mathcal{I}(s')$
Finds $\overline{\mathcal{I}}$ such that: $\forall s, s' : \overline{\mathcal{I}}(s) \land s \xrightarrow{R} s' \Rightarrow \overline{\mathcal{I}}(s')$
Checks: $\forall s$ at $\mathcal{E} : \mathcal{I}(s) \Rightarrow \overline{\mathcal{I}}(s)$
Base case checking

Fig. 6. Parameterized verification when user memory needs initialization.

Section IV-E. For the user image of Fig. 3, the above invariant corresponds to that of Section III-B: for instance, the set of admissible values for the type Thread$*$ in that case is $\{\texttt{a2}, \texttt{a7}\}$. However, note that instead of describing the invariant for one given user image, this new invariant describes what happens for *all* (well-typed) user images, hence is parameterized.

3. Base case checking. It remains to be checked whether our assumption on the initial state ($\overline{\mathcal{I}}(s_0)$) holds, i.e. to verify that, a given user image is well-typed (i.e. matches the annotated types), and that *if* is the address of an Interface $*$. The precise description of this process (a kind of type checking) is given in Section IV-E; it consists in particular in checking that the value contained in memory locations supposed to hold a type t, is indeed admissible for t; e.g. that address `a6`, of type Thread $*$, holds either `a2` or `a7`.

Note that the technique is no longer fully automated, as the user has to provide type annotations (even if most of it comes from the existing types of the interface). But let us observe that the example kernel works correctly only when linked with a well-typed user image (according to Fig. 2), and thus *parameterized verification of this kernel is impossible* without limiting, using a manual *precondition*, the admissible user image. Thus in general, *parameterized kernel verification is impossible without user-supplied annotations*, given here using the type annotations.

D. Differentiating boot and runtime code

In the example system, all the data structures in the user image are already initialized and thus are initially well-typed according to Fig. 2. In other systems this might not be the case: for instance, the values of the *next* field in *Thread*s could be uninitialized, and the boot code would have to create the circular list; or the Memory_Table table could be dynamically allocated and filled during the boot, as in our case studies.

To handle these cases, we propose to perform the base case checking when the kernel has finished booting and enters its main loop (Fig. 6), rather than at the beginning of the execution. For this, we:

- Perform the parameterized analysis of the kernel starting from an initial state where the user image would already be initialized. Even if the true initial state is not initialized, we still get a parameterized invariant $\overline{\mathcal{I}}$ on the kernel runtime which is inductive by construction;

$\mathbb{T} \ni t, u$	$::=$	$\texttt{Word}$	any word
	$\mid$	$\texttt{Int8}$	non-pointer
	$\mid$	n	type name
	$\mid$	t_i*	pointer
	$\mid$	$\textbf{struct} \ \{ \ (t \ \langle \textit{field} \rangle ;)^* \ \}$	record
	$\mid$	$t[e]$	array with size
	$\mid$	$t \ \textbf{with} \ p$	refinement type
$\mathbb{T}{\times}\mathbb{N} \ni t_i$	$::=$	t_i	type with offset
$\mathcal{N} \ni n$	$::=$	$\texttt{Flags} \mid \texttt{Context} \mid \ldots$	type name
e	$::=$	$i \in \mathbb{N}$	numeric constant
	$\mid$	$\texttt{nb\_threads} \mid \ldots$	symbolic constant
	$\mid$	$e + e \mid \ldots$	binary op
p	$::=$	$\textbf{self} \leq e \mid \textbf{self}\&e = e \mid \ldots$	predicate

Fig. 7. (Simplified) grammar of types.

- Perform a fully-automated in-context analysis of the kernel boot code with the given user image to get an invariant $\mathcal{I}$ at the end of the boot code;
- Check that all the "in-context" states at the end of the boot match the parameterized invariant, i.e. check that $\forall s,\ \mathcal{I}(s) \Rightarrow \overline{\mathcal{I}}(s)$.

Once this is done, the full system is verified: the combination of both analyses imply the existence of an invariant, and thus APE; moreover, both analyses allow to check that there is no software exception neither in the boot nor in the runtime.

This method uses two characteristics of the boot code:

- First, one goal of the boot code is to initialize the data structures so that they are well-typed according to Fig. 2. The initial value for these types is unimportant as they will be overwritten, so the code will also work if these data structures are initialized.
- The execution of the boot code is almost deterministic (except for hardware device handling and multicore execution) once the user image is known, thus the fully-automated in-context static analysis on this case is both robust and scalable (as it propagates mostly singleton values).

Note that this method avoids the need to specify (and annotate) the code with the actual precondition on the user image; here, the precondition that we verify is that "the user image should be such that its initialization will make it well-typed", using the fact that it is easier to describe the runtime types than the initial types.

E. The type system of the weak shape domain

Our type-based abstract domain is designed to verify the preservation of the memory layout of the interface, expressed using a particular dependent type system. A full formal presentation of the domain would be out of the scope of the paper, but here we introduce the type system and how well-typedness enforces the preservation of memory layouts. Following our example and for the sake of simplification, all scalar types have size 8 bits, but in practice our system handles multi-byte words.

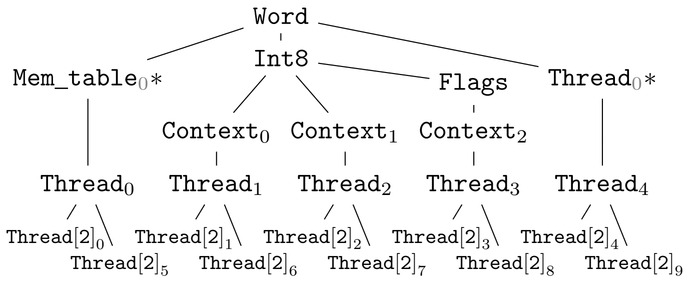

Fig. 8. Subtyping relations inside a $\texttt{Thread}[2]$ region.

Annotated types. Fig. 2 consists in a sequence of *type definitions*, which maps *type names* ($\in \mathcal{N}$) to *types* ($\in \mathbb{T}$). The grammar of types (Fig. 7) is similar to C, with the addition of *refinement types* [36] (using predicates prefixed by **with**). Type annotations apply to objects of the interface. They allow to specify arithmetic constraints (integer values) non-nullity (pointers) and length constraints (for arrays whose length is not known statically).

Types as labels for regions. The types are used to label regions in memory. For instance, in Fig. 3, the region $\texttt{a2} - \texttt{ab}$ has type $\texttt{Thread}[2]$, while $\texttt{ae} - \texttt{b1}$ has type $\texttt{Mem\_Table}$. When an address is statically allocated, the corresponding declaration provides the labeling (e.g. the addresses $\texttt{a2} - \texttt{ab}$ have been allocated by the global C declaration Thread thread_array[2]).

Formally, labels are represented using a *labeling* $\mathscr{L} \in \mathbb{A} \to \mathbb{T} \times \mathbb{N}$, mapping addresses to (type, offset) pairs. For instance in Fig. 3, the label of $\texttt{a2}$, $\mathscr{L}(\texttt{a2})$, is $\texttt{Thread}[2]_0$ (byte 0 of a $\texttt{Thread}[2]$ region); $\mathscr{L}(\texttt{a3}) = \texttt{Thread}[2]_1$ (byte 1 of a $\texttt{Thread}[2]$ region); etc.

Subtyping and overlapping regions. Sometimes the regions overlap. For instance the content at offset 3 in a $\texttt{Thread}[2]$ is the content at offset 3 in a $\texttt{Thread}$, which is is the content at offset 2 in a $\texttt{Context}$, which is a $\texttt{Flags}$. Thus, we want the address $\texttt{a6}$ to be labeled by all four of $\texttt{Thread}[2]_3$, $\texttt{Thread}_3$, $\texttt{Context}_2$, and $\texttt{Flags}$. We express this property as a *subtyping* relationship: $\texttt{Thread}_3$ is a *subtype* of $\texttt{Context}_2$ (written $\texttt{Thread}_3 \sqsubseteq \texttt{Context}_2$), meaning that addresses labeled by $\texttt{Thread}_3$ are also labeled by $\texttt{Context}_2$. We found out that this subtyping relationship is very important for programs written in machine languages: while in C we can distinguish t from &t→mt (when t is a $\texttt{Thread}*$), there is no distinction between them in machine code (because mt has offset 0). Thus in machine code, a pointer to a structure must simultaneously point to its first field.

Fig. 8 provides the subtyping relations inside a $\texttt{Thread}[2]$ region, derived from the definitions of Fig. 2. To unclutter notations, we allow ourselves to write t instead of t_0. We derive the subtyping relationships as follows: if u contains a t at offset i, then $u_{i+k} \sqsubseteq t_k$ for all k such that $0 \leq k < \text{sizeof}(t)$ (with $t_0 = t$ in the case of scalar types). In addition, for any t, the refined type "t **with** p" is always a subtype of t.

Separation. In our type system, two addresses whose types are not in a subtype relationship are separated (i.e. do not alias):

$$\forall t_i, u_j \in \mathbb{T} \times \mathbb{N} : \; t_i \not\sqsubseteq u_j \wedge u_j \not\sqsubseteq t_i \Rightarrow$$
$$\{a \in \mathbb{A} : \mathscr{L}(a) \sqsubseteq t_i\} \cap \{b \in \mathbb{A} : \mathscr{L}(b) \sqsubseteq u_j\} = \emptyset$$

This property ensures, for instance, that writing to an address in a `Thread`[2] region does not modify the content of any `Mem_table` region.

Types as sets of values. Types are used not only to label addresses, but also to represent sets of values. For instance, the set of values corresponding to `Int8` is $\mathbb{V}$, the set of all the bitvectors of length 8; the set of values corresponding to `Flags` is all the bitvectors with their *PRIVILEGED* bit unset; and the set of values corresponding to type $\texttt{Context}_0*$ is $\{\texttt{a3}, \texttt{a8}\}$, i.e. the set of addresses that have $\texttt{Context}_0$ as label. Note that the definition of this latter set relies on the labeling $\mathscr{L}$.

Formally, we provide an *interpretation* $(\!|\cdot|\!)_{\mathscr{L}} \in \mathbb{T} \times \mathbb{N} \to \mathcal{P}(\mathbb{V})$, mapping a (type, offset) pair to a set of values as follows.

$$(\!|\texttt{Int8}|\!)_{\mathscr{L}} = (\!|\texttt{Word}|\!)_{\mathscr{L}} = \mathbb{V}$$
$$(\!|t_i*|\!)_{\mathscr{L}} = \{a \in \mathbb{A} : \mathscr{L}(a) \sqsubseteq t_i\}$$
$$(\!|t \textbf{ with } p|\!)_{\mathscr{L}} = \{x \in (\!|t|\!)_{\mathscr{L}} : p(x)\}$$
$$(\!|s_k|\!)_{\mathscr{L}} = \bigcap_{\substack{t \in \mathbb{T} \\ s_k \sqsubseteq t_i}} (\!|t_i|\!)_{\mathscr{L}} \text{ for other types}$$

Well-typedness. The most important property of this type system is, given a set of types like in Fig. 2, the *well-typedness* of a memory m with regard to a labeling $\mathscr{L}$, defined as:

$$\forall a \in \mathbb{A}, \quad m[a] \in (\!|\mathscr{L}(a)|\!)_{\mathscr{L}}$$

which means that the contents of memory m at address a should be one of the admissible values for the type label of a. For instance in Fig. 3, the address `a6` is labelled as $\texttt{Thread}[2]_4$, which is a subtype of $\texttt{Thread}_0*$; the only addresses tagged with a subtype of $\texttt{Thread}_0$ are `a2` and `a7`, so the memory at `a6` can contain only `a2` or `a7`; this is correct in Fig. 3. Conversely, if $m[\texttt{a6}]$ were equal to `ae`, the couple $(m, \mathscr{L})$ would not be well-typed.

The type-based shape abstract domain. Our type-based shape abstract domain does not track the contents of the rest of user image memory at all (which makes it very efficient). Instead, it guarantees that each operation *preserves well-typedness*: i.e. that modifying a memory m for which $\mathscr{L}$ is a labeling, results in a new memory m', for which $\mathscr{L}$ is still a labeling. This is a useful property: it means that after a store, a register with type $\texttt{Task}_0*$ will still point to a `Task` structure, and not, e.g., to a page table. To do so, it tracks the types that are contained in the registers and kernel memory (i.e. stack, global variables, etc.) at each program point.

Let us return to the invariant computed by the parameterized analysis (Section IV-C). The exact meaning of the invariant is that $(m, \mathscr{L})$ exists such that $(m, \mathscr{L})$ is well-typed; that the memory at address `a0` contains values in $(\!|\texttt{Thread*}|\!)_{\mathscr{L}}$; the memory at address `a1` contains values in $(\!|\texttt{Context*}|\!)_{\mathscr{L}}$; etc. The precise user memory and labeling at Fig. 3 is indeed a special case of this invariant.

V. Case study & Experimental Evaluation

We seek to answer the following Research Questions:

RQ0: **Soundness check** Does our analyzer fail to verify APE and ARTE when the kernel is vulnerable or buggy?

RQ1: **Real-life Effectiveness** Can our method verify real (unmodified) embedded kernels?

RQ2: **Internal evaluation** What are the respective impacts of the different elements of our method?

RQ3: **Genericity** Can our method apply on different kernels, hardware architectures and toolchains?

RQ4: **Automation** Is it possible to prove APE and ARTE in OS kernels fully automatically?

RQ5: **Scalability** Can our method scale to large numbers of tasks?

A. Experimental setup

ASTERIOS. We consider for RQ1 and RQ2 the ASTERIOS kernel, an *industrial* solution for implementing security- and safety-critical hard real-time applications, used in industrial automation, automotive, aerospace and defense. The kernel is developed by KRONO-SAFE — an SME whose engineers are not formal method experts — using standard compiler toolchains.

We consider a port of the kernel to a *4-cores* ARM Cortex-A9 processor with *ARMv7* instruction set. It relies on ARM MMU for memory protection (*pagination*). The kernel features a hard real-time scheduler that dispatches the tasks between the cores (migrations are allowed), and monitors timing budgets and deadlines. The kernel adopts a "static microkernel" architecture, with unprivileged services used to monitor inter-process communication. The configuration for the user tasks is generated by the ASTERIOS toolchain, and all the memory is statically allocated. The system is parameterized: the kernel and the user images are compiled separately and both are loaded at runtime by the bootloader. We have analyzed two versions:

- **BETA**, a preliminary version where we found a vulnerability;
- **V1**, a more polished version with the vulnerability fixed and debug code was removed.

The code segment of each kernel executable contains 329 functions (`objdump` reports around 10,000 instructions), shared between the kernel and the core unprivileged services. Finally, KRONO-SAFE provided us with a sample user image.

EducRTOS. For research questions where we need some flexibility (RQ0, RQ3–5), we developed a new embedded kernel called EducRTOS. It contains a variety of features (e.g. different schedulers, dynamic thread creation) complementary to those of ASTERIOS (e.g. x86 instead of ARM, segmentation instead of pagination). EducRTOS is also being used to teach operating systems to master students. Depending on the included features, the kernel size ranges between 2,346 and 2,866 instructions.

Implementation. Our static analysis method is implemented in a prototype named BINSEC/CODEX: a plugin (41k lines of OCaml) on top of the open source BINSEC [37] framework for binary-level semantic analysis. We reuse the ELF parsing, instruction decoding, and intermediate representation lifting of the platform [38] and have reimplemented a whole static analysis on top of it. Since the analysis is performed on BINSEC's intermediate representation, the analysis implementation is entirely independent from the hardware architecture, apart from the empowered transition. To implement this transition, we use a simple sound approximation consisting in setting to an arbitrary value any unprotected register or memory location. Its implementation takes 23 lines of OCaml for ARM and 62 lines for x86.

Availability. BINSEC/CODEX and EducRTOS are open-source and part of the artifact accompanying this paper, but ASTERIOS is a commercial product.

Experimental conditions. We performed our formal verification completely independently from KRONO-SAFE activities. In particular, we never saw the source code of their kernel, and our interactions with KRONO-SAFE engineers were limited to a general presentation of ASTERIOS features. We ran all our analyses on a standard laptop with an Intel Xeon E3-1505M 3 GHz CPU with 32 GB RAM. All measurements of execution time or memory usage have been observed to vary by no more than 2 % across 10 runs. We took the mean value of the runs.

B. Soundness check (RQ0)

Our method is sound *in principle*, in that it proves by construction APE and ARTE only on kernels where these properties are true. Yet, this ultimately depends on the correct implementation of the static analysis; in this preliminary experiment, we want to check that our tool indeed does not prove APE and ARTE on buggy kernels.

Protocol. We deliberately introduce 4 backdoors in EducRTOS by creating 4 new system calls that jump to an arbitrary code address with kernel privilege, grant kernel privilege to user code segments, write to an arbitrary address, or modify the memory protection tables to cover parts of the kernel address space. These backdoors can easily be exploited to gain control over the kernel. Additionally, we add 3 bugs possibly leading to crashes in existing system calls: a read at an arbitrary address, an illegal opcode error and a possible division by zero.

Results. For each of the added vulnerabilities, our analysis does not prove APE (it either failed or reported an alarm at the instruction where the error occurs); for each bug, it does not prove ARTE.

Conclusions. BINSEC/CODEX was able to detect all the privilege escalation vulnerabilities and runtime errors that we explicitly introduced.

Additional notes. This corresponds to the experience we had while developing EducRTOS, where several times we launched BINSEC/CODEX and discovered unintentional bugs (wrong check on the number of syscalls, write to null pointers) that were not detected by testing. Our method even discovered a bitflip in a kernel executable that occurred when the file was copied.

C. Real-Life Effectiveness (RQ1)

Protocol. Our goal here is to evaluate the *effectiveness* of our parameterized verification method on an unmodified industrial kernel, measured by: (1) the fact that the *method indeed succeeds* in computing a non-trivial invariant for the whole system, i.e., computes an invariant under precondition for the kernel runtime and checks that the user tasks establish the precondition; (2) the *precision* of the analysis, measured by the number of *alarms* (i.e. properties that the analyzer cannot prove); (3) the *effort* necessary to setup the analysis, measured by the number of lines of manual annotations; and (4) the *performance* of the analysis, measured in CPU time and memory utilization.

TABLE I
MAIN VERIFICATION RESULTS ON ASTERIOS

		Generic		*Specific*	
# shape annotations	converted	1057			
	manual	57 (5.11%)		58 (5.20%)	
Kernel version		BETA	V1	BETA	V1
invariant computation	status	✓	✓	✓	✓
	time (s)	647	417	599	406
# alarms in runtime		1 **true error** 2 false alarms	1 false alarm	1 **true error** 1 false alarm	**0** ✓
user tasks checking	status	✓	✓	✓	✓
	time (s)	32	29	31	30
Proves APE and ARTE?		N/A	∼	N/A	✓

The bulk of the annotations (1057 lines) consists in type definitions, and was automatically extracted from the sample user image, using debug information in the user executable. Debug information is otherwise not used by the analysis (and the kernel executable does not contain any). Manual annotations consist in changing the definitions by adding "**with**" predicates, or information about the length of arrays. These annotations were reverse-engineered as the ones required for the kernel analysis to succeed with no remaining alarm (and also correspond to necessary requirements for the code to run without errors), but they could be obtained from the documentation or directly provided by the OS developers.

We consider both ASTERIOS kernel versions and two configurations (i.e., sets of type annotations):

- ***Generic*** contains types and parameter invariants which must hold for all legitimate user images;
- ***Specific*** further assumes that the stacks of all user tasks in the image have the same size. This is the default for ASTERIOS applications, and it holds on our case study.

Results. The main results are given in Table I. The ***generic*** annotations consist in only 57 lines of manual annotations, in addition to 1057 lines that were automatically generated (i.e. 5% of manual annotations, and a manual annotations per instructions ratio of 0.58%). The ***specific*** annotations adds one more line. When analyzing the **BETA version** with these annotations, only *3 alarms* are raised in the runtime:

- One is a **true vulnerability**: in the supervisor call entry routine (written in manual assembly), the kernel extracts the system call number from the opcode that triggered the call, sanitizes it (ignoring numbers larger than 7), and uses it as an index in a table to jump to the target system call function;

but this table has only 6 elements, and is followed by a string in memory. This off-by-one error allows an `svc 6` system call to jump to an unplanned (constant) location, which can be attacker-controlled in some user images. The error is detected as the target of the jump goes to a memory address whose content is not known precisely, and that we thus cannot decode;

- One is a false alarm caused by *debugging code* temporarily violating the shape constraints: the code writes the constant `0xdeadbeef` in a memory location that should hold a pointer to a user stack (yielding an alarm as we cannot prove that this constant is a valid address for this type), and that memory location is always overwritten with a correct value further in the execution;
- The last one is a false alarm caused by an imprecision in our analyzer when user stacks can have different sizes.

When analyzing the **v1 version**, the first two alarms disappear (no new alarm is added). Analyzing the kernel with the ***specific*** annotations makes the last alarm disappear. In all cases user tasks checking succeeds.

*Analyzing the **v1** kernel with the **specific** annotations allows to reach 0 alarms, meaning that we have formally verified APE and ARTE.*

Computation time is *low*: less than 11 minutes for the parameterized analysis, and 35 seconds for base case checking.

Conclusions. This experiment shows that *it is feasible to verify absence of privilege escalation of an industrial microkernel using only fully-automated methods* (albeit with a very slight amount of manual annotations), and *without any change to the original kernel*. Especially:

- The analysis is *effective*, in that it identifies real errors in the code, and verifies their absence once removed;
- The analysis is *very precise*, as we manage to reach 0 false alarms on the correct code, and we had no more than 2 false alarms on each configuration of the analysis;
- The *annotations burden is very small* (58 simple lines), as the kernel invariant is computed automatically and most of the shape annotations are automatically converted from the interface types;
- Finally, the *analysis time*, for a kernel whose size is typical for embedded microkernels, *is small* (between 406 and 647 seconds). Analysis of CPU_0 alone takes only 86 seconds.

D. Evaluation of the method (RQ2, sketch only)

We performed a detailed evaluation of the method, described in full in the supplementary data[3].

This experiment consists in evaluating, on the **v1** kernel version, our 3-step method (Section IV-B), using different configurations for the weak shape domain. Results show that the *shape domain is necessary for a parameterized verification of the kernel* (otherwise the analysis aborts due to imprecision). While 1057 lines of type annotations are automatically converted from the types for the user image interface, only *10 manual additional lines are necessary* for the analysis to *succeed in computing an invariant* (and 58 to eliminate all the alarms in the runtime).

[3]https://binsec.github.io/assets/publications/papers/2021-rtas-supplementary-data.pdf

E. Genericity (RQ3)

Protocol. We applied our method both to ASTERIOS and EducRTOS, two kernels running on different architectures (resp. ARM and x86) and memory protection mechanisms (resp. segmentation and pagination). We also performed a parameterized verification of 96 EducRTOS variants. Each variant was compiled with a different valuation of the following parameters:

- compiler: GCC 9.2.0, Clang 7.1.0.
- optimization flags: `-O1`, `-O2`, `-O3` or `-Os`.
- scheduling algorithms: round-robin, fixed-priority, or earliest-deadline-first scheduling.
- dynamic thread creation: enabled or disabled.
- debug printing: enabled or disabled.

Results. We could *parametrically* verify all 96 EducRTOS variants. The verification requires 98 lines of type definitions (which we extracted automatically, as we did for ASTERIOS, from debug information) and 12 to 14 lines of manual annotations (depending on the scheduler), with 12 lines being common to all variants; corresponding to an annotation per instruction ratio of less than 0.59%. The verification of each variant takes between 1.6 s and 73 s. The detailed measurements for each variant are provided in the supplementary data.

Conclusions. Our method is applicable to different kernel features and hardware architectures. It is robust: non-trivial changes in the kernel executable, like introduction of new features, or change in code patterns because of compiler or changes in optimisation levels, do not not require to change the configuration. This robustness makes it an interesting tool for verifying kernels coming in many variants.

F. Automation (RQ4) and Scalability (RQ5)

Protocol. In this experiment, we perform a fully automated in-context verification (with *no annotation*) of a simple variant of the EducRTOS kernel with round-robin scheduling and no dynamic task creation. We then use this kernel to evaluate the scalability of this approach by modifying the number of tasks that run on the system, and compare it against our parameterized method.

Figure 9 provides the execution time and memory used when verifying a system composed of N tasks for both approaches. For the parameterized analysis, the invariant computation takes less than a second, and total analysis time is almost linear in the number of tasks, while requiring only 12 lines of annotations. In this case, the only parts that depends on the number of tasks are the base case checking and the in-context static analysis of the almost deterministic boot code.

By contrast, for the in-context verification, the number of steps to reach a fixpoint, the number of modified memory locations, and the number of target locations for the `Thread*` pointers grow with the number of tasks, resulting in a quadratic complexity. This scalability issue is inherent to the use of in-context verifications [15].

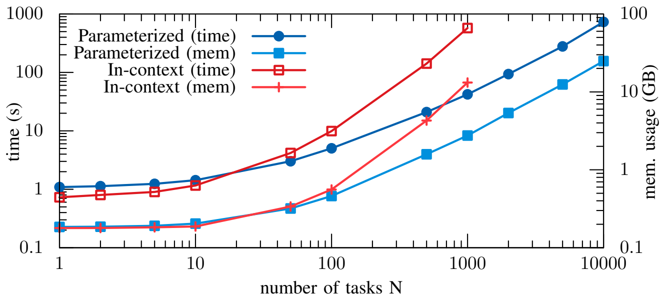

Fig. 9. Performance when verifying a system with N tasks

Conclusions. Fully-automated in-context verification with no annotation is achievable on very simple kernels, but is not robust enough for more complex kernels. Moreover, it does not scale to very large number of tasks. On the contrary, parameterized verification (with very few annotations) is robust and scales almost linearly with large numbers of tasks.

VI. Related Work

There is a long history of using formal methods to verify operating system kernels.

Degree of automation. We can distinguish three classes of verification methods:

- *manual* [1], [3]–[6]: the user has to *provide for every program point a candidate invariant*, then *prove* via a proof assistant that every instruction preserves these candidate invariants;
- *semi-automated*[4] [7]–[10], [13]–[15], [39]: the user has to *provide* the candidate invariants *at some key program points* (kernel entry and exit, loops, and optionally other program points like function entry and exit) and then use automated provers to verify that all paths between these points preserve the candidate invariants;
- *fully automated*: a sound static analyzer [20] *automatically infers* correct invariants for every program point. The user only *provides invariant templates* by selecting or configuring the required abstract domains. This approach has been used to verify source code of critical embedded software [21], but never to verify a kernel.

We demonstrate that abstract interpretation can automatically verify embedded kernels from their executable with a very low annotation burden, sometimes with no manual intervention.

In the parameterized case, we verify kernels with a ratio of annotations per instruction which is at worst of 13 lines for 2346 instructions (0.59%). By comparison, the automated verification of CertiKOS^S [14] required 438 lines for 1845 instructions (23.7%).

[4]Some authors call these techniques automated; we use the word semi-automated to emphasize the difference with fully-automated methods.

Target property. Prior kernel verification methods generally target four kinds of program properties:

- *Functional correctness* [1]–[3], [5]–[7], [10], [40], [41] i.e. compliance to a (manually written) formal specification of the kernel;
- *Task separation*, [14], [42] i.e. verifying the absence of undesirable information flow between tasks;
- *Absence of privilege escalation* [8], [9], [15] (a.k.a. kernel integrity), i.e. proving that the kernel protects itself and that no attacker can gain control over it;
- *Absence of runtime errors* [39] like buffer overflows, null-pointer dereferences, or format string vulnerabilities.

To achieve maximal automation, we focus on *implicit* properties. Especially, we are the first to prove that absence of privilege escalation is implicit.

Note that the invariants we infer could significantly reduce the number of annotations required to verify functional correctness [14], and can generally help any other analysis; for instance worst-case execution time (WCET) estimation [43], [44] requires knowledge about the CFG and memory accesses. Stronger invariants can be inferred by combining our analysis with other domains [45].

Trusted computing base (TCB) and verification comprehensiveness. We only trust that the bootloader correctly loads the ELF image in memory, that the hardware complies with its specification, and that our abstract interpreter (which has no dependencies) is sound. We do not trust the build toolchain (compiler, build scripts, assembler and linker), we analyze all of the code, and we do not assume any unverified hypothesis.

While push-button methods can verify all of the code [13]–[15], more manual methods often [2], [6], [11], [41] leave parts of the code unverified when the verification is hard or overly tedious. While source-level verifications methods sometimes carry to the assembly level [46] or include support for assembly instructions [9], [40], they usually trust the compilation, assembly and linking phases.

To further reduce our TCB, we would have to verify our static analyzer in a proof language with a small kernel (like Isabelle [47] or Coq [48]), a huge effort that has been shown to be feasible [49].

Features of verified kernels. We focus on embedded systems kernels, where the kernel memory is mostly statically allocated (either in the kernel or in the user image) (e.g. [4], [13]–[15], [50]–[52]). We do not consider real-time executives where memory protection is absent or optional, as verifying APE on them would require unchecked assumptions on applicative code. Still, the kernels that we have analyzed feature complex code, including dynamic thread creation and dynamic memory allocation using object pools; different memory protection tables (using x86 segments or ARM page tables) modified at boot and runtime; different real-time schedulers working on arbitrary numbers of tasks; boot-time parametrization by a user image; and usage of multiple cores with shared memory.

Being based on abstract interpretation, our method can address kernels with features out of reach of prior attempts based on symbolic execution (e.g., real-time schedulers). While we can still design kernels to get around the limitations of the tool [14], *we are the first to verify an existing, unmodified real kernel*.

However, like any sound static analyzer, BINSEC/CODEX may still be too imprecise on some code patterns, emitting *false alarms* or failing to compute a non-trivial invariant. For instance, our tool would not handle well self-modification in the kernel, complex lock-free code, or using a general-purpose memory allocator outside of the boot code. Still, we can directly reuse progress in the automated analysis of these patterns. An important pattern for real-time systems is full preemptibility in the kernel. It should be possible to extend our analysis to these systems by leveraging our capacity to handle concurrent executions, notably by adding a stack abstraction that would be independent from the concrete address (as in [22]); this is an important topic for future work.

Verifying systems with unbounded memory. Prior works targeting machine code [3], [13]–[15] use a flat representation of the memory (where the abstract state enumerates all the memory cells in the kernel), causing scalability issues in the presence of a large number of tasks [15] and preventing *parameterized* verification. To handle systems where the amount of memory is not statically known, we need more complex representations that *summarize* memory. Points-to and alias analyses are fast and easy to setup but are too imprecise for formal verification, and generally assume that the code behaves nicely, e.g., type-based alias analyses [53] assume that programs comply with the strict aliasing rule – while kernel codes often do not conform to C standard [41]. On the other hand, shape analyses [54], [55] can fully prove memory invariants, but require heavy configuration and generally cannot scale to a full embedded kernel.

We propose a weak type-based shape abstract domain that hits a middle ground: it is fast, precise, handles low-level behaviors (outside of the C standard) and requires little configuration. This is also the first time a shape analysis is performed on machine code.

Marron [56] also describes a weak shape domain, but on a type-safe language with no implicit type casts, pointer arithmetic, nor nested data structures. Outside of fully-automated analyses, Walker et al. [2] already observed in the 1980s that reasoning on type invariants is well suited to OS kernel verification. Several systems build around this idea [8], [10], leveraging a dedicated typed language. Cohen et al. [57] describe a typed semantics for C with additional checks for memory typing preservation, similar to our own checks on memory accesses. While they use it in a deductive verification tool for C (to verify a hypervisor [7]), we build an abstract interpreter for machine code.

VII. Conclusion and future work

We have presented BINSEC/CODEX, an automated method for formally verifying embedded kernels (absence of runtime errors and absence of privilege escalation) directly from their executable, with only a very low annotation burden. We address important limitations of existing automated methods: we allow *parameterized* verification, i.e. verifying a kernel independently from the applications running on it; we handle unbounded loops that are necessary for implementing real-time schedulers; and we *infer* the kernel invariants, instead of merely checking them. As in OS formal verification, *"invariant reasoning dominates the proof effort"* [2] (in seL4, 80% of the effort was spent stating and verifying invariants [1]), this work is a key enabler for more automated verification of more complex systems.

Acknowledgements

The authors warmly thank Gilles Muller, Amit Vasudevan, Pierre-Yves Piriou and Guerric Chupin, as well as the anonymous reviewers, for their very helpful remarks and discussions. Xavier Rival received funding from the French ANR, as part of the Veriamos grant. Matthieu Lemerre and Sébastien Bardin also received funding from the ANR as part of the TAVA grant.

References

[1] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, ACM, 2009.

[2] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA Unix security kernel," *Commun. ACM*, vol. 23, pp. 118–131, feb 1980.

[3] W. Bevier, "Kit: A study in operating system verification," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1382–1396, 11 1989.

[4] R. J. Richards, *Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel*, pp. 301–322. Boston, MA: Springer US, 2010.

[5] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, 2015.

[6] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li, "A practical verification framework for preemptive OS kernels," in *International Conference on Computer Aided Verification*, Springer, 2016.

[7] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova, "Automated verification of a small hypervisor," in *Verified Software: Theories, Tools, Experiments* (G. T. Leavens, P. O'Hearn, and S. K. Rajamani, eds.), (Berlin, Heidelberg), pp. 40–54, Springer Berlin Heidelberg, 2010.

[8] J. Yang and C. Hawblitzel, "Safe to the last instruction: automated verification of a type-safe operating system," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 99–110, 2010.

[9] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überSpark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, 2016.

[10] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, ACM, 2017.

[11] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent OS kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, 2016.

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," *SIGPLAN Not.*, vol. 46, p. 283–294, June 2011.

[13] M. Dam, R. Guanciale, and H. Nemati, “Machine code verification of a tiny ARM hypervisor,” in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED ’13, ACM, 2013.

[14] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, “Scaling symbolic evaluation for automated verification of systems code with Serval,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, (New York, NY, USA), p. 225–242, Association for Computing Machinery, 2019.

[15] J. Nordholz, “Design of a symbolically executable embedded hypervisor,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys ’20, (New York, NY, USA), Association for Computing Machinery, 2020.

[16] L. A. Clarke, “A system to generate test data and symbolically execute programs,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, 1976.

[17] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, p. 385–394, July 1976.

[18] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, p. 82–90, Feb. 2013.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, The Internet Society, 2008.

[20] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1977.

[21] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “A static analyzer for large safety-critical software,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI’03)*, ACM, 2003.

[22] T. Reps, J. Lim, A. Thakur, G. Balakrishnan, and A. Lal, “There’s plenty of room at the bottom: Analyzing and verifying machine code,” in *International Conference on Computer Aided Verification*, Springer, 2010.

[23] B. Ford and E. S. Boleyn, “Multiboot specification.” https://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[24] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.

[25] A. Venet and G. P. Brat, “Precise and efficient static array bound checking for large embedded C programs,” in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[26] G. A. Kildall, “A unified approach to global program optimization,” in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1973.

[27] S. Bardin, P. Herrmann, and F. Védrine, “Refinement-based CFG reconstruction from unstructured programs,” in *Verification, Model Checking, and Abstract Interpretation, VMCAI 2011*, Springer, 2011.

[28] A. Djoudi, S. Bardin, and É. Goubault, “Recovering high-level conditions from binary programs,” in *FM 2016: 21st International Symposium on Formal Methods*, 2016.

[29] O. Nicole, M. Lemerre, and X. Rival, “Binsec/Codex, an abstract interpreter to verify safety and security properties of systems code,” tech. rep., CEA List, ENS, 2021. https://binsec.github.io/assets/publications/papers/2021-rtas-technical-report-analysis.pdf.

[30] J. Kinder, F. Zuleger, and H. Veith, “An abstract interpretation-based framework for control flow reconstruction from binaries,” in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2009.

[31] P. Granger, “Static analysis of arithmetical congruences,” *International Journal of Computer Mathematics*, vol. 30, no. 3-4, pp. 165–190, 1989.

[32] A. Miné, “Symbolic methods to enhance the precision of numerical abstract domains,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, 2006.

[33] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, “An abstract domain of uninterpreted functions,” in *Verification, Model Checking, and Abstract Interpretation*, Springer, 2016.

[34] A. Miné, “Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics,” in *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’06)*, ACM, 2006.

[35] A. Miné, “Static analysis of run-time errors in embedded critical parallel C programs,” in *European Symposium on Programming*, Springer, 2011.

[36] T. Freeman and F. Pfenning, “Refinement types for ML,” in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI ’91, (New York, NY, USA), p. 268–277, Association for Computing Machinery, 1991.

[37] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion, “BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis,” in *SANER*, IEEE Computer Society, 2016.

[38] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, “The BINCOA framework for binary code analysis,” in *International Conference on Computer Aided Verification*, Springer, 2011.

[39] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, “Design, implementation and verification of an extensible and modular hypervisor framework,” in *2013 IEEE Symposium on Security and Privacy*, May 2013.

[40] W. Paul, S. Schmaltz, and A. Shadrin, “Completing the automated verification of a small hypervisor – assembler code verification,” in *Software Engineering and Formal Methods*, Springer, 2012.

[41] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, “Comprehensive formal verification of an OS microkernel,” *ACM Trans. Comput. Syst.*, vol. 32, pp. 2:1–2:70, feb 2014.

[42] J. Rushby, “The design and verification of secure systems,” in *Eighth ACM Symposium on Operating System Principles (SOSP)*, (Asilomar, CA), pp. 12–21, Dec. 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[43] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, “Proving real-time capability of generic operating systems by system-aware timing analysis,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 318–330, 2019.

[44] A. Colin and I. Puaut, “Worst-case execution time analysis of the RTEMS real-time operating system,” in *Proceedings 13th Euromicro Conference on Real-Time Systems*, pp. 191–198, 2001.

[45] P. Cousot and R. Cousot, “Systematic design of program analysis frameworks,” in *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1979.

[46] T. A. L. Sewell, M. O. Myreen, and G. Klein, “Translation validation for a verified OS kernel,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’13, ACM, 2013.

[47] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[48] T. Coquand and G. P. Huet, “The calculus of constructions,” *Inf. Comput.*, vol. 76, no. 2/3, pp. 95–120, 1988.

[49] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, “A formally-verified C static analyzer,” *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 247–259, 2015.

[50] K. Ramamritham and J. A. Stankovic, “Scheduling algorithms and operating systems support for real-time systems,” *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[51] J. T. Mühlberg and F. Leo, “Verifying FreeRTOS: from requirements to binary code,” in *11th International Workshop on Automated Verification of Critical Systems (AVoCS)*, Newcastle University, 2011.

[52] A. A. E. E. Commitee, “Avionics application software standard interface,” ARINC 653, Aeronautical Radio, Incorporated (ARINC), 1996.

[53] A. Diwan, K. S. McKinley, and J. E. B. Moss, “Type-based alias analysis,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’98, ACM, 1998.

[54] M. Sagiv, T. Reps, and R. Wilhelm, “Parametric shape analysis via 3-valued logic,” in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’99, ACM, 1999.

[55] B.-Y. E. Chang and X. Rival, “Relational inductive shape analysis,” in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, 2008.

[56] M. Marron, “Structural analysis: Shape information via points-to computation,” *arXiv e-prints*, p. arXiv:1201.1277, Jan 2012.

[57] E. Cohen, M. Moskal, S. Tobies, and W. Schulte, “A precise yet efficient memory model for C,” *Electronic Notes in Theoretical Computer Science*, vol. 254, pp. 85 – 103, 2009. Proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009).