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 CertiKOSS [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 embedded OS kernel, 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 (mpu1 or mpu2): 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 (PRIVILEGED bit in the flags' register) forbids user code to change the values of mpu1 and mpu2. A special interrupt (RESET) corresponds to the system boot, where the kernel initializes the memory ("init"). Additional interrupts (the other cases) 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

---

1While 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).
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_Tables) 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→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 illustrate 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).

**Whole-system static analysis**

```
<table>
<thead>
<tr>
<th>initial state s₀</th>
<th>boot code</th>
<th>kernel exit E</th>
<th>user code</th>
<th>kernel entry</th>
</tr>
</thead>
</table>
```

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

**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 mpu1 or mpu2, 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 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.

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).

**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).

---

**Fig. 4.** Fully-automated in-context analysis on the system loop.
Abstract interpreters are 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.

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, with more general domains (e.g. intervals, polyhedra) and widening operators to handle loops.

Machine code analysis is challenging. Binary-level static analysis is particularly challenging as 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 ⊤).

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.

C. Sound and precise static analysis for binary code

Abstract interpretation. Abstract interpretation 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.

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, with more general domains (e.g. intervals, polyhedra) and widening operators to handle loops.

Machine code analysis is challenging. Binary-level static analysis is particularly challenging as 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 ⊤).

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.

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.

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 (S, S₀, →), where S is the set of all states, S₀ the set of initial states, and → corresponds to the transition from one instruction to the next.

Definition 1 (Privilege escalation). We define privilege escalation of a transition system (S, S₀, →) 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 (S, S₀, →), where we have replaced the normal transition → (corresponding to the execution of the next instruction) by the empowered transition → (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:
We define a non-trivial state invariant

\[ \forall s \in S, \exists s' : s \rightarrow s' \]

Running an arbitrary sequence of privileged instructions

\[ \text{Assumption 1.} \]

If a transition system satisfies a non-trivial state invariant of the system is \( \top \), true for every state. \[ \text{Theorem 1.} \]

Proof. This follows directly from the fact that for every state \( s \), the only state property different from \( \top \) is vulnerable to privilege escalation, then the only satisfiable state property in the system is the trivial state property \( \top \), true for every state. \[ \text{Theorem 2.} \]

Proof. This is the contrapositive of the fact that if a state exists in the transition system \( S, s_0, \rightarrow \), where privilege escalation happens, this state also exists in \( S, s_0, \rightarrow \).

Thus, verifying APE on a \( (S, s_0, \rightarrow) \) system implies APE on the real \( (S, s_0, \rightarrow) \) 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 \( S \).

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 \( (S, s_0, \rightarrow) \) 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 \( \top \).

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 trivial state properties.

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, \( \{a_2, a_7\} \), was over-approximated by the interval \( [a_2, a_7] \);
- Scalability, because of the need to enumerate large sets. For instance, on a system with 1000 tasks, the values for \( \text{mpu}_1 \) 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).
Step 2: Parameterized static analysis of the kernel. The result of this step is an inductive property $I$, 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 II-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 $i'$ has type Interface *, the analysis computes (using the type annotations) a parameterized invariant $I$ (Fig. 5) implying the following invariant (after boot):

- Address $a0$ (cur variable) has an admissible $Thread*$ value;
- Address $a1$ (ctx variable) has an admissible $Context*$ value;
- Registers mpu1 and mpu2 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 Section I-V-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 \{a2, 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 ($I(s_0)$) holds, i.e. to verify that, given a user image is well-typed (i.e. matches the annotated types), and that $i'$ is the address of an Interface *. The precise description of this process (a kind of type checking) is given in Section I-V-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 $Threads$ 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. 3), 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 $I$ on the kernel runtime which is inductive by construction;
threads

\[ \exists t, u \quad \text{Word} \]
| \quad \text{any word} \\
| \quad \text{Int8} \\
| \quad \text{any non-pointer} \\
| \quad n \\
| \quad \text{type name} \\
| \quad t_i \ast \\
| \quad \text{pointer} \\
| \quad \text{struct} \{ \langle \text{field}\rangle \}^{*} \\
| \quad \text{array with size} \\
| \quad t[e] \\
| \quad \text{record} \\
| \quad t \text{ with } p \\
| \quad \text{refinement type} \\
| \quad t_i \\
| \quad \text{type with offset} \\
\]

\[ \mathbb{T} \times \mathbb{N} \ni t_i \quad \text{=} \quad t_i \]

\[ \mathbb{N} \ni n \quad \text{=} \quad \text{Flags} \mid \text{Context} \mid \ldots \\
| \quad \text{type name} \\
| \quad e \quad \text{=} \quad i \in \mathbb{N} \\
| \quad \text{numeric constant} \\
| \quad \text{nb\_threads} \mid \ldots \\
| \quad \text{symbolic constant} \\
| \quad e + e \mid \ldots \\
| \quad \text{binary op} \\
| \quad p \quad \text{=} \quad \text{self} \leq e \mid \text{self} & e = e \mid \ldots \\
| \quad \text{predicate} \\
\]

- Perform a fully-automated in-context analysis of the kernel boot code with the given user image to get an invariant \( I \) 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 \( \forall s, I(s) \Rightarrow I(s) \).

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.

Annotated types. Fig. 2 consists in a sequence of type definitions, which maps type names (\( \in \mathbb{N} \)) to types (\( \in \mathbb{T} \)). The grammar of types (Fig. 7) is similar to C, with the addition of refinement types (using predicates prefixed by with).

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

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

Subtyping and overlapping regions. Sometimes the regions overlap. For instance the content at offset 3 in a \( \text{Thread}[2] \) is the content at offset 3 in a \( \text{Thread} \), which is the content at offset 2 in a Context, which is a Flags. Thus, we want the address \( a6 \) to be labeled by all four of \( \text{Thread}[2]_3 \), \( \text{Thread}_3 \), Context2, and Flags. We express this property as a subtyping relationship: \( \text{Thread}_3 \) is a subtype of Context2 (written \( \text{Thread}_3 \subseteq \text{Context}_2 \)), meaning that addresses labeled by \( \text{Thread}_3 \) are also labeled by Context2. We found out that this subtyping relationship is very important for programs written in machine languages: while in C we can distinguish \( t \) from \( &t \rightarrow mt \) (when \( t \) is a 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 \( \text{Thread}[2] \) 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 \( u_{i+k} \subseteq t_k \) 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 \( t \text{ with } p \) 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):

\[ \forall t_1, u_j \in T \times N : t_i \not\in u_j \land u_j \not\in t_i \Rightarrow \{ a \in \mathbb{A} : \mathcal{L}(a) \subseteq t_i \} \cap \{ b \in \mathbb{A} : \mathcal{L}(b) \subseteq u_j \} = \emptyset \]

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\(_a\) is all the bitvectors with their PRIVILEGED bit unset; and the set of values corresponding to type Context\(_0\) is \{a3, a8\}, i.e. the set of addresses that have Context\(_0\) as label. Note that the definition of this latter set relies on the labeling \( \mathcal{L} \).

Formally, we provide an interpretation \( \langle \cdot \rangle_{\mathcal{L}} : T \times N \rightarrow \mathcal{P}(\mathbb{V}) \), mapping a (type, offset) pair to a set of values as follows.

\[
\begin{align*}
\langle \text{Int8} \rangle_{\mathcal{L}} &= \langle \text{Word} \rangle_{\mathcal{L}} = \mathbb{V} \\
\langle t_i \rangle_{\mathcal{L}} &= \{ a \in \mathbb{A} : \mathcal{L}(a) \subseteq t_i \} \\
\langle \{ x \in \langle \{ \cdot \rangle_{\mathcal{L}} : p(x) \} \rangle \rangle_{\mathcal{L}} &= \bigcap_{x \in \bigcap_{t_i} \langle t_i \rangle_{\mathcal{L}}} \langle t_i \rangle_{\mathcal{L}} \text{ for other types}
\end{align*}
\]

**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 \( \mathcal{L} \), defined as:

\[ \forall a \in \mathbb{A}, \ m[a] \in \langle \mathcal{L}(a) \rangle_{\mathcal{L}} \]

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 Thread[2], which is a subtype of Thread[4]; the only addresses tagged with a subtype of Thread[4] are a2 and a7, so the memory at a6 can contain only a2 or a7; this is correct in Fig. 3. Conversely, if \( m[a6] \) were equal to a6, the couple \( (m, \mathcal{L}) \) 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 \( \mathcal{L} \) is a labeling, results in a new memory \( m' \), for which \( \mathcal{L} \) is still a labeling. This is a useful property: it means that after a store, a register with type Task0* 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 \( (m, \mathcal{L}) \) exists such that \( (m, \mathcal{L}) \) is well-typed; that the memory at address a0 contains values in \( \langle \text{Thread} \rangle_{\mathcal{L}} \); the memory at address a1 contains values in \( \langle \text{Context} \rangle_{\mathcal{L}} \); 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—a SME whose engineers are not formal method experts—using standard compiler toolchains.

We consider a port of the kernel to a 4-core 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 interprocess 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 BIN-
SEC’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 verifica-
cation 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.
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;

<table>
<thead>
<tr>
<th># shape annotations</th>
<th>converted</th>
<th>manual</th>
<th>Generic</th>
<th>Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>57 (5.11%)</td>
<td>58 (5.20%)</td>
</tr>
<tr>
<td>Kernel version</td>
<td>BETA</td>
<td>v1</td>
<td>BETA</td>
<td>v1</td>
</tr>
<tr>
<td>invariant</td>
<td>status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>computation</td>
<td>time (s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>647</td>
<td>417</td>
<td>599</td>
<td>406</td>
</tr>
<tr>
<td># alarms in runtime</td>
<td>1 true error</td>
<td>2 false alarms</td>
<td>1 false alarm</td>
<td>1 true error</td>
</tr>
<tr>
<td>user tasks</td>
<td>checking</td>
<td>status</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>time (s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>29</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>Proves APE and ARTE?</td>
<td>N/A</td>
<td>✓</td>
<td>N/A</td>
<td>✓</td>
</tr>
</tbody>
</table>
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 CPU0 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.

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).

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 1.6s and 73s. 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 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** \([7]–[10], [13]–[15], [39]\): 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.
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.

References


