Proving full-system security properties under multiple attacker models on capability machines

Thomas Van Strydonck  Aïna Linn Georges  Armaël Guéneau  Alix Trieu  Aarhus University  Aarhus University  Aarhus University  Vrije Universiteit Brussel
Anne Marie Timany  Frank Piessens  Lars Birkedal  Dominique Devriese  Aarhus University  Aarhus University  Aarhus University

Abstract—Assembly-level protection mechanisms (virtual memory, trusted execution environments, virtualization) make it possible to guarantee security properties of a full system in the presence of arbitrary attacker provided code. However, they typically only support a single trust boundary: code is either trusted or untrusted, and protection cannot be nested. Capability machines provide protection mechanisms that are more fine-grained and that do support arbitrary nesting of protection. We show in this paper how this enables the formal verification of full-system security properties under multiple attacker models: different security objectives of the full system can be verified under a different choice of trust boundary (i.e. under a different attacker model). The verification approach we propose is modular, and is robust: code outside the trust boundary for a given security objective can be arbitrary, unverified attacker-provided code. It is based on the use of universal contracts for untrusted adversarial code: sound, conservative contracts which can be combined with manual verification of trusted components in a compositional program logic. Compositionality of the program logic also allows us to reuse common parts in the analyses for different attacker models. We instantiate the approach concretely by extending an existing capability machine model with support for memory-mapped I/O and we obtain full system, machine-verified security properties about external effect traces while limiting the manual verification effort to a small trusted computing base relevant for the specific property under study.

I. INTRODUCTION

Assembly-level security primitives are a cornerstone of secure systems, and they come in many forms. CPU-supported security mechanisms like virtual memory, trusted execution environments, virtualization, micro-policies or capability machines all offer a form of encapsulation, which supports the execution of untrusted code with restricted authority. Some architectural security mechanisms, such as virtual memory, virtualization or trusted execution environments, are carefully optimized for specific security abstractions, such as processes, virtual machines or enclaves, and provide poor support for features which fall outside of these abstractions. In this paper, we are interested in such a feature, which is poorly supported by the most-used security mechanisms: nested encapsulation. This refers to scenarios where an encapsulated piece of code (e.g. a user process or virtual machine) further encapsulates a subset of its own code from the rest of its code. When nested encapsulation is poorly supported by the architectural primitives, it can only be supported at the cost of additional effort, complexity and a certain performance loss. For example, library OSs like Graphene require host OS cooperation to implement process isolation [1] and running virtual machines inside virtual machines requires additional context switches with additional overhead [2].

Contrary to many other primitives, micro-policies [3] and capability machines [4], [5], [6] are designed explicitly for generality and flexibility. In particular, capability machines offer good support for nested encapsulation, as we now explain. On a capability machine, capabilities are used to represent authority explicitly. Different forms of capabilities represent authority over memory, the authority to invoke other code, etc. Unprivileged code can easily set up an encapsulation boundary by constructing an object capability: an opaque capability that can be invoked by other code and only makes private state available after invocation. This private state can in turn include other capabilities representing additional authority. An object capability invocation hence represents a context switch between security domains.

Building on related work in high-level languages [7], [8], [9], Skorstengaard et al. and Georges et al. have developed a methodology for robust modular verification of software running on capability machines [10], [11] that supports proving (security) properties in the presence of untrusted code. The idea is to formalize the hardware-provided security guarantees in the form of a universal contract: a separation logic contract that holds for arbitrary, untrusted code on the machine. This universal contract expresses a form of capability safety: the untrusted code’s authority is effectively bounded by the authority of the capabilities it is given access to. In robust modular verification, the program logic allows combining manual verification of a property for certain components with this universal contract for untrusted code to obtain a full-system proof of the property.

For now, this work has remained restricted to proving artificial security properties: for example the fact that an assertion failure flag will never be set [11], [12] or equi-termination of programs [13]. In this paper, we extend this robust modular verification approach to a capability machine with memory-mapped I/O (MMIO). Although this is not technically the most complex feature to add, it does allow us to prove end-to-end system properties that are more interesting and realistic. Contrary to the artificial properties of previous work, our results specify that a security property holds for
the system’s trace of external effects, which we believe is the ultimate goal of verification in many practical settings.

Additionally, we extend the approach to reasoning about nested encapsulation. To formally verify intended security properties in the presence of nested encapsulation, we analyze the system several times using different attacker models, corresponding to different scenarios that the encasulations are (explicitly or implicitly) designed for.

To make our approach concrete, we study the equivalent of BitVisor’s parapass-through virtualization [14], where small wrappers enforce security policies on the interaction with peripherals. Such wrappers rely on a very small TCB and lend themselves well to verification. Particularly, we consider scenarios like the one depicted in Fig. 1 where different parties install wrappers in a nested way. For example, a hardware manufacturer could install a wrapper with exclusive access to certain peripherals and read and write callbacks which other code on the system can use to interact with the peripherals. The wrapper could restrict configuration parameters (perhaps depending on the options purchased by the device manufacturer) simply by applying a bounds check on values written to a particular MMIO address. On top of this, a device manufacturer could install a second wrapper that keeps a counter to enforce a maximum amount of interactions with a particular device or rate-limit the interactions with a device (by consulting an external timer device before allowing an interaction). Another realistic policy could enable a led whenever a camera device is set to capture mode (for privacy reasons). More generally, we believe that many interesting policies could be enforced using nested, unsophisticated low-TCB wrappers. Such policies are directly supported in our model in a way that appears quite realistic for embedded processors (e.g. the SAM D5x/E5x [15]), if they were extended with capability machine security primitives. Fig. 1 shows a possible architecture with a bottom-level wrapper wrapper1 around all peripherals, as well as two nested wrappers (wrapper21 and wrapper22), each consisting of a read and write closure around different peripherals. On top of these wrappers is the remainder of the code base, which can remain untrusted and largely unmodified, except that direct writes to peripheral MMIO addresses need to be replaced with invocations of the wrappers.

A security analysis of nested wrappers should consider different attacker models, for example the four models depicted in Fig. 2. In these diagrams, gray components are treated as untrusted: white components are manually verified for correctness and security (i.e. proper encapsulation towards the gray components). For example, a security analysis of wrapper1 could use the leftmost model: only wrapper1 is trusted and all other code on the system is treated as arbitrary. Because of the machine’s capability safety, it suffices to manually verify the wrapper and its encapsulation to verify the intended property. A second analysis could use the attacker model of Fig. 2 in addition to wrapper1, it relies on the secure wrappers wrapper21 and wrapper22. This second analysis has a larger TCB but can prove the stronger properties that wrapper21 and wrapper22 enforce. The analysis does not inspect the code of wrapper1 for this, but relies on a functional contract for it (perhaps provided by the hardware manufacturer together with wrapper1). Further analyses could use additional attacker models as depicted in Fig. 2 and Fig. 2.

Note that while the system is analyzed several times, we hasten to point out that (1) only wrapper code is manually verified and (2) no wrapper is verified twice: wrapper1 is verified manually, resulting in a functional correctness contract, which is then used when manually verifying wrapper21 and wrapper22. The full system properties are obtained by combining the verification results of each of the wrappers with our general capability safety theorem. Note also that although wrapper21 and wrapper22 are both trusted in the second attacker model, compositionality of our program logic still allows us to verify them separately, relying on a contract for each other’s behavior. This avoids creating unnecessary dependencies when verifying wrappers.

In this paper, we present and explain our approach by considering representative example wrappers on a model capability machine. A first example corresponds roughly to the first two layers of Figure 1, with wrappers enforcing a simple bounds check and a maximum bound on the amount of accesses respectively (i.e. a stateful property). In this first example, we demonstrate that we can modularly reason about independent wrappers and that the verification effort can be shared when they are implemented according to a fixed code structure. Our second example shows that our approach also supports more complex properties that consider interactions with several peripherals. It considers a setup where wrapper22 is replaced with a wrapper wrapper22bis that limits the rate at which a certain peripheral is accessed. Wrapper22bis will only allow accesses to its peripheral after an external timer device indicates that sufficient time has passed. It does not follow the same fixed structure as the first wrappers and is verified separately.

To summarize, our contributions are the following:

- We extend the capability machine model, program logic and logical relation of Georges et al. [11] with memory-mapped I/O and secure enforcement of I/O properties and reprove...
We consider a machine with finite A machine word w. As such, our presentation here is similar to theirs, though enter (O in Fig. 4. There are six different permissions: opaque (O), in-range (R), read/execute (RX), read/write/execute (RWX). These permissions are standard, except for the opaque permission which provides no privilege and the enter permission, inspired by the M-Machine [5], that can be used to build opaque closures or object capabilities. Enter capabilities are “unsealed” into read/execute capabilities when jumped to. The capability is then available in the pc register and can be copied into another register to restore environment variables in the case of a closure.

Fig. 6 defines the small-step operational semantics of the machine. The machine’s state consists of an execution mode μ and an execution configuration ϕ. The mode μ models the machine’s instruction cycle, which loops infinitely (expressed by Repeat μ) until it reaches a successful done state Done Halted through REPEATHALT or a failed state Done Failed through REPEATFAIL. The REPEATSINGLE rule allows for the execution of single instructions through the EXEC SINGLE rule. If the execution of the instruction is successful, i.e. execution in EXEC SINGLE does not fail or halt and results in a Done SingleStep state, then REPEATSTANDBY allows for another iteration of the processor’s instruction cycle.

An execution step (EXEC SINGLE) requires an executable, in-range capability (p, b, e, a) in the pc register. The word z at address a is then read and decoded into an instruction decode(z) which is executed on the current configuration ϕ to result in a new machine state \( \| decode(z) \| (\varphi) \). Machine instructions i operate over registers r or either integers or registers ρ. The behavior of instruction i in configuration ϕ is specified by \( \| i \| (\varphi) \) defined in Fig. 5. Most instructions use the auxiliary function updPC to increment the pc register at the end of their executions. Since the address space is finite, pointer arithmetic such as a + 1 can fail. For ease of reading, we write \( a + k \) to indicate successful pointer arithmetic.

Instructions fail and halt respectively terminate the execution in a Failed or Halted state. move r ρ copies ρ (its value if it’s an integer, or its contents if it’s a register) into r. Memory can be manipulated using the load and store instructions: load \( r_1 \ r_2 \) reads the value at the address pointed to by the capability in \( r_2 \) assuming it has read permission and is within bounds, and copies the value into \( r_1 \). Similarly, store \( r \ r_2 \) stores \( r \) at the address pointed to by the capability in \( r \) assuming it has write permission and is within bounds. The jmp instruction jumps to a capability, by writing it into the pc register. As explained earlier, the jmp instruction unseals E capabilities into RX capabilities before jumping to them.
a ∈ Addr ≜ [0, AddrMax]

p ∈ Perm ≜ O | E | RO | RX | RW | RWX

e ∈ Event ≜ EventTy × Addr × Z

c ∈ Cap ≜ \{(p, b, e, a) | b, e, a ∈ Addr\}
t ∈ Trace ≜ list Event

w ∈ Word ≜ Z + Cap

ϕ ∈ ExecConf ≜ Reg × Mem × State × Trace

r ∈ RegName ≜ pc | r0 | r1 | \ldots | r31

δ ∈ DoneState ≜ Standby | Halted | Failed

reg ∈ Reg ≜ RegName → Word

μ ∈ ExecMode ≜ SingleStep | Repeat μ | Done δ

m ∈ Mem ≜ Addr → Word

\[ i ::= \text{jmp } r | \text{ jnz } r | \text{ move } r \rho | \text{ load } r | \text{ store } r \rho | \text{ add } r \rho \rho | \text{ sub } r \rho \rho | \text{ eq } r \rho \rho | \text{ lt } r \rho \rho | \text{ lea } r \rho |
\text{ restrict } r \rho | \text{ subseg } r \rho \rho | \text{ isptr } r r | \text{ getp } r r | \text{ getb } r r | \text{ gete } r r | \text{ geta } r r | \text{ fail } | \text{ halt} \]

\[ \text{updPC}(\varphi) = \begin{cases} 
(\text{Done Standby}, \varphi[\text{reg} \mapsto (p, b, e, a + 1)]) & \text{if } \varphi.\text{reg}(\text{pc}) = (p, b, e, a) \\
(\text{Done Failed}, \varphi) & \text{otherwise} 
\end{cases} \]

\[ \text{getWord}(\varphi, \rho) = \begin{cases} 
\rho & \text{if } \rho \in Z \\
\varphi.\text{reg}(\rho) & \text{if } \rho \in \text{RegName} 
\end{cases} \]

\[ \text{updST}(\varphi, e, s) = \varphi[\text{state} \mapsto s][\text{trace} \mapsto \varphi.\text{trace} ++ [e]] \]

\[ i \mapsto \text{fail} \quad \text{(Done Failed, } \varphi) \quad \text{Conditions} \]

\[ i \mapsto \text{halt} \quad \text{(Done Halted, } \varphi) \quad \text{w = getWord}(\varphi, \rho) \]

\[ \text{load } r r_1 \quad \text{updPC}(\varphi[\text{reg}, r_1 \mapsto w]) \quad \varphi.\text{reg}(r_2) = (p, b, e, a) \text{ and } w = \varphi.\text{mem}(a) \text{ and } b \leq a < e \text{ and } p \in \{RO, RX, RW, RWX\} \text{ and } a \notin \text{MMIO} \]

\[ \text{store } r \rho \quad \text{updPC}(\varphi[\text{mem}, a \mapsto w]) \quad \varphi.\text{reg}(r) = (p, b, e, a) \text{ and } b \leq a < e \text{ and } p \in \{RW, RWX\} \text{ and } w = \text{getWord}(\varphi, \rho) \text{ and } a \notin \text{MMIO} \]

\[ \text{jmp } r \quad \text{(Done Standby, } \varphi[\text{reg}, \text{pc} \mapsto \text{newPc}]) \quad \begin{cases} 
\varphi.\text{reg}(r) = (E, b, e, a) & \text{if } \varphi.\text{reg}(r) = (E, b, e, a) \text{, then } \text{newPc} = (RX, b, e, a) \\
\text{otherwise } \text{newPc} = \varphi.\text{reg}(r) & \text{otherwise} 
\end{cases} \]

\[ \text{getr } r_1 \quad \text{updPC}(\varphi[\text{reg}, r_1 \mapsto a]) \quad \varphi.\text{reg}(r_2) = (p, b, e, a) \]

\[ \begin{array}{l}
\text{Repeat SingleStep} \quad (\text{SingleStep, } \varphi) \rightarrow (\text{Done } \delta, \varphi') \\
(\text{Repeat SingleStep, } \varphi) \rightarrow (\text{Repeat SingleStep, } \varphi') \\
\text{Repeat Standby} \quad (\text{Repeat Done Standby), } \varphi) \rightarrow (\text{Repeat SingleStep, } \varphi') \\
(\text{Repeat Done Halted), } \varphi) \rightarrow (\text{Repeat SingleStep, } \varphi') \\
\text{Repeat Fail} \quad (\text{Repeat Done Failed), } \varphi) \rightarrow (\text{Repeat Done Failed), } \varphi') \\
(\text{SingleStep, } \varphi) \rightarrow \begin{cases} 
[\text{decode}(z)](\varphi) & \text{if } \varphi.\text{reg}(\text{pc}) = (p, b, e, a) \land b \leq a < e \land \\
\rho \in \{RO, RX, RWX\} \land \varphi.\text{mem}(a) = z & \text{otherwise} 
\end{cases} \\
\end{array} \]

Figure 3: Machine words, machine state and instructions.

Figure 5: Operational semantics: instruction semantics.
Capabilities can be modified using the `restrict`, `subseg` and `lea` instructions. `restrict` can be used to `decrease` the permission of a capability according to the permission lattice’s partial order `≺`. `subseg` can be used to `decrease` the range of authority of a capability, while `lea` can be used to modify where a capability points to. As `E` capabilities are used to encapsulate code and data, they cannot be modified until they are unsealed, hence the instructions `subseg` and `lea` fail when used with `E` capabilities. Instructions to read capabilities’ fields are also provided: `geta`, `getp`, `getb` and `gete` respectively read the `a`, `p`, `b` and `e` fields of a capability `(p, b, e, a)`. Not shown in Fig. 5 are instructions `jnz` (conditional jump), `add`, `sub` (addition and subtraction), `eq` (equality), `lt` (comparison) and `isptr` to check whether a register contains a capability. Finally, if none of the above cases apply, the execution falls through to a failed state as shown on the last row in Fig. 5.

B. Adding support for memory-mapped I/O

To allow communication between the CPU and devices, we add support for MMIO. Those additions are indicated in blue in Fig. 3 and 5. For simplicity, we do not yet support interrupts; the CPU and devices must poll for memory updates. We discuss adding interrupts in Section VI.

To model MMIO, we assume a set `MMIO` of addresses reserved for MMIO and augment execution configurations `φ` with a trace `t` of MMIO events `e`, and an environmental state `s` drawn from a set `State` as shown in Fig. 3. An event `e` is a triple of a mode `e.type` (read or write), an address `e.addr`, and an integer `e.value` that is read or written. The operational semantics of our machine in Fig. 5 is parameterized by this set `State`, `MMIO` and two operations `mmioLoad`: `(State × Addr) → (State × Z)` and `mmioStore`: `(State × Addr × Z) → State`, that model how the state of the devices reacts to MMIO loads and stores.

Fig. 5 shows the new behavior of `load` and `store` for MMIO addresses. `load` will use `mmioLoad` to get the value at address `a` from the environment and load it into the register. Similarly, `store` uses `mmioStore` to indicate that a value is written at some address. In both cases, the operations transition the environment’s state and an IORead or IOWrite event is recorded in the trace.

III. Example wrappers

This section details the two examples with nested para-pass-through security wrappers that were described in the introduction. The goal is to illustrate how stakeholders such as the ones in Fig. 1 can set up the capability machine to achieve their security objectives.

A. Three-layer stateful example with orthogonal wrappers

The first system we consider is setup as in Fig. 1 with an additional bottom-most wrapper `w0`. There are hence 3 layers of simple wrappers, where the third layer contains two disjointly operating wrappers. The proof effort for this example can be shared, since the wrappers share a common structure, as will be discussed in section IV-D. The different wrappers enforce the following concrete invariants, modelling the kind of real security properties we discussed in the introduction:

- Wrapper `w0` encapsulates all of MMIO, and creates a read and write closure for MMIO that higher-level wrappers use. It does not enforce its proper predicate, i.e. it enforces the trivially true predicate `P0` on the trace `t` in Fig. 2. We separate this wrapper from the others, since its implementation deviates: it is the only wrapper that does not recursively call another wrapper, but rather accesses MMIO directly.

- Wrapper `w1` ensures that no more than 1000 MMIO-events (read and write combined) occur once the capability machine boots. This invariant is expressed by the predicate `P1` on the trace `t` in Fig. 2.

- To ensure safety of values sent to their respective peripherals, `w21` solely allows positive values, whereas `w22` solely allows negative values. Note that it would be trivial to extend this to an arbitrary bounds check. This is expressed by `P21` and `P22` in Fig. 2, where `P'(e)` is a predicate on traces that holds on a trace `t` iff `P'(e)` holds for all events `e` in `t`.

- We model two different peripherals (e.g. network and display) situated at the memory-mapped addresses `a1` and `a2`, respectively. `w21` only allows events destined for address `a1`, i.e., `w21` enforces predicate `P21` on a filtered view of the general MMIO trace. The filtering is represented by `F21` in Fig. 2. Thus `w21` enforces `P21 ⊩ F21` on the MMIO trace. Analogously, `w22` enforces `P22 ⊩ F22`. The complement filter `F2` will be used to prove that no addresses other than `a1` and `a2` receive MMIO events, by requiring that `F2(t) = [1]`.

The previous description has provided us with sufficient details to define the verification goals of each stakeholder. These goals are formulated as security objectives, and summarized in Fig. 8. Note that `w0` does not have any security objective of its own, since it solely encapsulates MMIO. `OBJ-1` specifies the guarantees that `w1` hopes to achieve from verification. `OBJ-1` states that if the initial memory `m0` and registers `r0` constitute a valid configuration `init_config_1(r0, m0)` (further explained below), then any execution starting from the empty trace `∅` and an arbitrary state `s0` and taking an arbitrary number of steps (denoted by `→*`), will result in a configuration that has a trace `t` satisfying `P1`. In other words, `w1` can be sure that `P1` will hold on any trace of execution, as long as the capability machine boots into a configuration satisfying `init_config_1`.

For the second layer, we get two separate security objectives; `OBJ-21` and `OBJ-22`. This models the situation where the two drivers are developed and verified independently (perhaps by separate developer teams in the same company), relying on a contract for the other wrapper, rather than its exact code. Otherwise, `OBJ-21` and `OBJ-22` are analogous to `OBJ-1`. They enforce that if the initial memory and register configuration is satisfactory, the respective predicates `P21` and `P22` hold over the network and display parts of the final trace `t`, i.e. `P21(F21(t))` and `P22(F22(t))` hold. Additionally, the complement filter `F2` guarantees that no events to addresses other than `a1` or `a2` can ever happen in the system.
The MMIO region represents all of itself, set-up code to initialize the wrapper code, and adversarial of view of wrapper shown at key points in execution.

Fig. 10, which illustrates the control flow of the running instance of wrappers 1 and 21 (22 is to make its security objective provable. Fig. 9 graphically summarizes the assumptions that wrappers 1 and 21 (22 is to an adversary. The contents of important registers are passed to an adversary. Note that these assumptions imply different attacker models for the different security objectives, as detailed earlier in Fig. 2 adversaries are allowed to arbitrarily instantiate the untrusted parts of the system, depicted in red.

We now discuss each subfigure in order, with the aid of Fig. 11 which illustrates the control flow of the running example from machine start-up (at START), until control is passed to an adversary. The contents of important registers are shown at key points in execution.

First, Fig. 9a presents the memory layout from the point of view of wrapper0. As for all wrappers in our system, the memory layout contains 3 major parts: code for the wrapper itself, set-up code to initialize the wrapper code, and adversarial code. The MMIO region represents all of MMIO, and is unique to wrapper0, since no other wrappers access MMIO directly.

The code of wrapper0 itself consists of a read closure from address d_\text{\textsubscript{x0}} to d_\text{\textsubscript{w0}}, a write closure from d_\text{\textsubscript{w0}} to d_\text{\textsubscript{x0}} and local state used by the closures, from d_\text{\textsubscript{x0}} to d_\text{\textsubscript{e0}}. As mentioned before, the read and write closures do not enforce any predicate on the trace; they simply provide read and write functionality to MMIO memory. The local state consists of a single address, which the set-up code will store a capability (RW, MMIO\textsubscript{a}, MMIO\textsubscript{b}, MMIO\textsubscript{c}) for MMIO into. It is hence independent of the current trace, and given by LS\textsubscript{0}(c) in Fig. 7. This capability provides the read and write closures of wrapper0 access to all of MMIO. Fig. 11 demonstrates the code for the write closure. It makes use of the following macros:

- \texttt{reqint} \( r \) \( r_{\text{aux}} \): succeeds iff \( r \) contains an integer
- \texttt{lea_a} \( r \) \( r_{\text{aux}} \): absolute version of \texttt{lea}, that makes the capability in \( r \) point to the address corresponding to \( r \).
- \texttt{rclear} \( r \): clears the set of registers \( r \) by overwriting them with the value 0.

In order to simplify wrapper invocations, we settled on the following common calling convention in our examples:

- \( r_0 \) contains the return address
- \( r_1 \) contains the value to write in case of a write event, and the value that is read in case of a read event
- \( r_2 \) contains the request’s destination MMIO address
- \( r_{25:31} \) are caller-save auxiliary registers, jointly denoted by the set \( R_{\text{aux}} \).

Fig. 11 starts by loading the MMIO capability from \( d_{\text{\textsubscript{x0}}} \) into \( r_{25} \), and making it point to the MMIO address in \( r_2 \). Next, the value in \( r_1 \) is stored through \( r_{25} \), thereby writing it to MMIO memory. Note that if \( r_1 \) is not an integer, or \( r_2 \) is not an MMIO address, this operation will fail. Finally, all auxiliary registers are cleared, and the write closure jumps to its return address in \( r_0 \).

The purpose of the set-up code is to create the previously discussed read and write closures, and to pass these to Adversary 0 securely. Fig. 12 lists the concrete set-up code for wrapper0. When the machine boots, the pc register is assumed to point to Set-up Code 0 and contains an omniscient capability granting access to all of memory. For simplicity reasons, all wrapper code is assumed pre-loaded in memory, but initial memory cannot contain any capabilities. The set-up code starts by deriving the previously discussed MMIO capability from the omnipotent pc, and storing it at address \( d_{\text{\textsubscript{x0}}} \). Next, it derives the read and write closures from the pc and stores them in \( r_1 \) and \( r_2 \). Lastly, it restricts the omnipotent pc to the adversary region, clears all auxiliary registers and jumps to the...
adversary, thereby loading r0 into the pc. Fig. 10 demonstrates how execution starts in Set-up Code 0, and how, when jumping to Adversary 0, pc, r1, and r2 are set up as previously described. From the point of view of wrapper0, the concrete code stored
inside Adversary 0 is irrelevant, as capability safety ensures that no adversary will be able to bypass its read and write closures.

Let us now consider the memory layout for wrapper1 in Fig. 9b. The layout is similar to Fig. 9a except for the topmost region. Since Set-up Code 0 gets to execute before passing control to Set-up Code 1, wrapper1 assumes the existence of a region Layout 0, whose instructions satisfy a contract that captures the behavior of Set-up Code 0. Consequently, init_config1 in OBJ-1 requires the machine to boot inside the Layout 0 region and pass control to Set-up Code 1 at the end, with the register state as specified in Fig. 10. Requiring a contract rather than precise code makes the differences of the different layers more independent, and will, e.g., allow the hardware vendor to optimize network packet handling without affecting any proofs in higher-up layers.

The code for wrapper1 itself is similar in layout, but makes use of more extensive local state than wrapper0. Concretely, the read and write closures ensure that the local state always satisfies LS1 in Fig. 7. LS1 is defined in terms of a general local state LS_gen. LS_gen describes the layout of local state for wrapper x (with x ≠ 0). It specifies that the first three addresses contain a RWX capability c_bx, for all of local state, and the read and write closures c_r p(x) and c_w p(x), where pr(x) represents the layer-below wrapper, which x will call. For example, pr(22) = 1. Having c_bx is required because jumping to an E capability in our capability machine results in a pc with RX permission, which disallows updating the local state. Lastly, LS_gen takes a list of custom values c_v as an argument. This allows wrappers to specify additional local state to aid in enforcing their invariants. In this case, Fig. 7 defines LS1 = λt. LS_gen([length(t)]), where x = 1 because wrapper1 requires access to its own local state, and pr(x) = 0 because wrapper1 will call wrapper0 to have it perform MMIO. Additionally, c_v = [length(t)], because wrapper1 requires one extra address to store local state; a counter length(t) that corresponds to the number of MMIO events performed so far, to compare it to 1000.

Fig. 13 demonstrates how the read closure of wrapper1 uses local state. It makes use of a generic read_wrapper template, which we use for any non-bottom-level wrapper in this example. The template uses a list of instructions check_read to check whether the wrapper’s predicate (e.g., P1 in this case) would still hold after the next MMIO event, and update the local state if this is the case. First, the template verifies whether r2 is a valid address, and not just any integer. This is done using the is_addr r raux1 raux2 macro, which succeeds iff r contains an integer that corresponds to a memory address. Then, it calls upon the checking instructions. Lastly, it loads c_r0 from address d ls1 + 1 and jumps to it. For wrapper1 the template is instantiated with check_1, which checks that P1 holds. These instructions first load c_ls1 from d ls1, then load length(t) from d ls1 + 3, check whether length(t) + 1 < 1000 still holds, and fail if this is not the case. If the check passes, length(t) + 1 is stored to d ls0 + 3, ensuring that LS1 holds again after the call to wrapper0.

The set-up code for wrapper1 is very similar to the code we discussed in Fig. 12. The only difference is that, rather than ensuring that LS0([1]) holds, the set-up code needs to satisfy LS1([1]) before jumping to Adversary 1. Fig. 10 again shows the state of key registers at that point.

Finally, Fig. 22 summarizes wrapper21. Wrapper21 again assumes a contract for lower-level wrappers, that is satisfied by the region Layout 0 + 1. This contract captures the behavior of Set-up Code 0 and 1 combined, i.e., the first two steps in Fig. 10. Similarly, a new region Layout 22 satisfies a second contract that captures the behavior of Set-up Code 22. Both contracts appear in the definition of init_config21 in OBJ-21 and enable more modular code development. We do not discuss wrapper22 separately, since its layout is the dual of wrapper21. Both wrappers share the same adversary.

The code for wrapper21 itself is similar to the code for wrapper1. It enforces LS21 in Fig. 7 which is again defined in terms of LS_gen. No custom local state is needed to check P21. The set-up code is also similar, but it jumps to wrapper22 before control is passed to Adversary 2, as shown in Fig. 10. This Figure demonstrates how Set-up Code 21 sets up the closures for wrapper21 in r3 and r4, whereas Set-up Code 22 (or in this case the assumed contract for Layout 22) sets up its closures in r1 and r2. Set-up Code 21 uses r1 and r2 to pass the closures for wrapper1 to Set-up Code 22.

B. Rate limiting

To demonstrate how the previous example generalizes to support a more complex property that requires interaction with multiple devices, we implemented and verified a second example where wrapper22 is replaced by wrapper22_bis; a wrapper that implements rate limiting. The other wrappers and security objectives remain unchanged. Concretely, wrapper22_bis relies on a trusted, memory-mapped timer device, and only allows an IO-event or to or from its peripheral (at the same address a2 that wrapper22 used) to occur when a value of 1 has been read from the timer address #timer beforehand. In other words, wrapper22_bis enforces the predicate P22_bis in Fig. 14 on a version of the MMIO trace that has been filtered through F22_bis. Here, the function last returns the most recent event in t, if any. In summary, security objective OBJ-22 is replaced by
The wrapper’s concrete closures that we hand-verify should enforce the invariant, as Section IV-B further explains. Additionally, the concrete Set-up Code of the wrapper and (specifications for) the set-up code of all layer-below wrappers should respect the invariant. The latter is simple to prove, since set-up code does not perform IO itself.

2) The arbitrary adversarial code in the Adversary region should be safe to execute, i.e., have no way of bypassing the wrapper’s closures and breaking the invariant. Section IV-C discusses how we employ a semantic model to reason about the safety of unknown code in the capability machine.

In Section IV-D we finally discuss an approach to sharing the verification effort for wrappers that have a common structure. This is not required for our verification approach, but it allowed us to reduce the verification effort involved in proving the first example.

A. Invariants to enforce security objectives

In this section, we detail how each wrapper \( x \) employs invariants \( \text{inv}_x(x) \) to prove modularly that its security objective \( \text{Obj}_x \) holds at each step of execution. It is insufficient to have an invariant that simply states that the security objective holds continuously. To be of general use, each wrapper requires three additional types of reasoning to be possible with its invariant.

First, each wrapper has a view of the physical trace, which is the part of the physical trace that it knows about. In our first example, wrappers 0 and 1 view the entire trace, whereas the views of wrappers 21 and 22 consist of all events addressed to \( a_1 \) and \( a_2 \), respectively. A wrapper \( x \) will only ever see a subview of the layer-above wrapper \( \text{pr}(x) \)’s view. By expressing the view of \( x \) in terms of the view of \( \text{pr}(x) \), the invariant ensures that all wrappers’ views are indeed recursively views of the actual physical trace, obtained through repeated filtering.

Secondly, in case multiple wrappers \( x_1, \ldots, x_n \) have a common layer-below wrapper \( \text{pr}(x_1) \), it should be possible to modularly reason about events they admit. For example, we should not have to know anything about the view of \( x_n \) on the trace, to reason about the view that \( x_1 \) has on the trace.

Lastly, the invariant needs to ensure that any local state that the wrapper requires for its correct operation is enforced on the wrapper’s current view of the trace. For example, to prove correct operation of wrapper \( x_1 \) in our first example, the invariant must be able to guarantee that the number of MMIO events that wrapper \( x_1 \) keeps track of, is the number of events in the most up-to-date view of the trace.

In the remainder of this section, we flesh out these three types of reasoning by means of Fig. 16 which demonstrates how different aspects of the wrappers’ invariants help us achieve the desired reasoning. At the end, we showcase some concrete invariants used in our first example. Note that the enforcement of the security objective itself is trivially expressed by adding the condition \( P_x(t_x) \) to the invariant, as Fig. 16 shows.

1) Connecting \( x_1 \) to \( \text{pr}(x_1) \): In general, a wrapper \( x_1 \)’s view of the trace is a filtered view, a subsequence of the layer-below wrapper \( \text{pr}(x_1) \)’s view. We can connect different layers this way: the invariant for wrapper \( x_1 \) in Fig. 16 owns a resource \( \text{filter}_{\text{val}} \gamma_{\text{pr}(x_1)} F_{x_1} t_{x_1} \), which states that \( x_1 \)’s view on the trace is \( t_{x_1} \) and that \( t_{x_1} \) is obtained by applying a filter \( F_{x_1} \) to the view \( t_{\text{pr}(x_1)} \) that \( \text{pr}(x_1) \) has. As shown in this section, wrappers 0 and 1 view the entire trace, whereas the views of wrappers 21 and 22 consist of all events addressed to \( a_1 \) and \( a_2 \), respectively. A wrapper \( x \) will only ever see a subview of the layer-above wrapper \( \text{pr}(x) \)’s view. By expressing the view of \( x \) in terms of the view of \( \text{pr}(x) \), the invariant ensures that all wrappers’ views are indeed recursively views of the actual physical trace, obtained through repeated filtering.

Secondly, in case multiple wrappers \( x_1, \ldots, x_n \) have a common layer-below wrapper \( \text{pr}(x_1) \), it should be possible to modularly reason about events they admit. For example, we should not have to know anything about the view of \( x_n \) on the trace, to reason about the view that \( x_1 \) has on the trace.

Lastly, the invariant needs to ensure that any local state that the wrapper requires for its correct operation is enforced on the wrapper’s current view of the trace. For example, to prove correct operation of wrapper \( x_1 \) in our first example, the invariant must be able to guarantee that the number of MMIO events that wrapper \( x_1 \) keeps track of, is the number of events in the most up-to-date view of the trace.

In the remainder of this section, we flesh out these three types of reasoning by means of Fig. 16 which demonstrates how different aspects of the wrappers’ invariants help us achieve the desired reasoning. At the end, we showcase some concrete invariants used in our first example. Note that the enforcement of the security objective itself is trivially expressed by adding the condition \( P_x(t_x) \) to the invariant, as Fig. 16 shows.

1) Connecting \( x_1 \) to \( \text{pr}(x_1) \): In general, a wrapper \( x_1 \)’s view of the trace is a filtered view, a subsequence of the layer-below wrapper \( \text{pr}(x_1) \)’s view. We can connect different layers this way: the invariant for wrapper \( x_1 \) in Fig. 16 owns a resource \( \text{filter}_{\text{val}} \gamma_{\text{pr}(x_1)} F_{x_1} t_{x_1} \), which states that \( x_1 \)’s view on the trace is \( t_{x_1} \) and that \( t_{x_1} \) is obtained by applying a filter \( F_{x_1} \) to the view \( t_{\text{pr}(x_1)} \) that \( \text{pr}(x_1) \) has. As shown in
valid\( (F) \) \( \triangleq \exists P_e : \text{Event} \to \text{Prop} \left( \forall t. F(t) = \text{filter}(P_e, t) \right) \)

\[
\text{orthogonal}(F_1, F_2) \triangleq \forall t. F_1(F_2(t)) = F_2(F_1(t)) = []
\]

1. \( \text{filter}\_\text{full} \gamma t \ast \text{filter}\_\text{val} F t' \to F(t) = t' \)
2. \( \text{filter}\_\text{val} \gamma F t \ast \text{filter}\_\text{val} \gamma F' t' \to \text{orthogonal}(F, F') \)
3. \( \forall F', \text{valid}(F') \land \text{orthogonal}(F, F') \to F'(t) = F'(t') \to \) \( \text{filter}\_\text{full} \gamma t \ast \text{filter}\_\text{val} \gamma F t \) \( \to \) \( \text{filter}\_\text{full} \gamma t' \ast \text{filter}\_\text{val} \gamma F F'(t) \)

Figure 15: Three main properties of the filter\_full and filter\_val abstractions built on top of the filter resource algebra in Iris, and definition of the auxiliary notions of validity and orthogonality they require.

Given this notion of orthogonality, we defined a novel filter resource algebra (an Iris construct used to define custom separation logic resources) to reason about independent, orthogonal updates to a trace. The previous resources filter\_full \( \gamma t \) and filter\_val \( \gamma F t \) are in fact defined in terms of this resource algebra. Fig. 15 defines the three most important properties that these two resources satisfy. First, we define a filter \( F \) to be valid if it filters the trace by a decidable event predicate \( P_e \). Property 1 states that \( t' \) is indeed a view of \( t \) through the filter \( F \). Property 2 states that any two filter\_val resources are guaranteed to be orthogonal. Property 3 then leverages orthogonality to express how we may update traces modularly: if all orthogonal, valid filters are unaffected by a trace update, then we can update a filter\_full and filter\_val without requiring ownership of any other filters. Fig. 16 illustrates how the filters \( x_1, \ldots, x_n \) are connected to their common layer-below filter, assuming orthogonal filtering predicates.

The reader might wonder about the case where the views of \( x_1, \ldots, x_n \) are not orthogonal. This case does not occur in our current examples, but Iris offers multiple resource algebras that can be used to reason about different forms of shared views on the trace. For example, imagine a scenario where a clock closure (similar to the timer closure in our second example) is shared between wrapper\( _{21} \) and wrapper\( _{22}\)\_bis, such that both wrappers can read timestamps from it and enforce their own predicates (e.g. “at least \( X \) seconds have to pass between any two writes to \( o_2 \)”). The wrappers 21 and 22\_bis would still require an orthogonal view to know about all events at respectively \( a_1 \) and \( a_2 \), but now also a partial view for their reads from \( o_\text{clock} \). The partial view ensures that wrapper\_22\_bis does not need to update its view whenever wrapper\_21 reads a timestamp from the clock, and vice versa. Iris already contains a monotone resource algebra, that could allow implementing such partial views.

2) Reasoning modularity about sibling wrappers: In the case where multiple wrappers \( x_1, \ldots, x_n \) have a common layer-below wrapper \( p(x_1) \), reasoning about each wrapper’s view can happen modularly if the wrappers have orthogonal views on the trace, where orthogonality for two filters \( F \) and \( F' \) is denoted orthogonal\( (F, F') \) and defined in Fig. 15. For example, \( F_{21} \) and \( F_{22} \) are orthogonal in our first example, so we can update the view of the trace \( F_{22}(t) \) that wrapper\_21 has, without requiring any knowledge of wrapper\_22\_bis’s view \( F_{22}(t) \), and vice versa. Similarly, we might have multiple wrappers that only write certain ranges of output values, that only ever read, respectively write values, that always write specific pairs of MMIO values, etc. Note that orthogonality is more flexible than disjointness since it does not presuppose a notion of intersection, but that it degenerates to the latter in case we consider filters that filter on individual events (as is the case in our examples).

3) Incorporating local state: While the security objective \( P_e(t_x) \) has to be satisfied at each step of execution (i.e. atomically), the local state can temporarily be out of sync with the trace, e.g. if a wrapper updates its local state before invoking the layer-below wrapper to perform IO, or if updating the local state takes multiple instructions. For this reason, \( \text{inv}(x) \) consists of two parts: an atomic invariant \( \text{inv}\_\text{st}, x \) and \( \text{inv}\_\text{at}, x \) which ensures that the security objective holds continuously, and a so-called non-atomic invariant \( \text{inv}\_\text{at}, x \) which ensures (among others) that the local state is satisfied. The resources \( \text{cur}\_\text{tr} \gamma l_x t_x \)
The first three invariants represent wrappers 0, 1 and 21. Since own security objective. These contracts are expressed in terms of a program logic for our capability machine, which we inherit from Georges et al. and extend with rules for the MMIO cases of the load and store instructions. The program logic contains the following weakest precondition assertion:

\[
\forall \varphi. P \rightarrow (Q \rightarrow \wp \text{Repeat SingleStep} \{ s, \varphi(s) \})
\]

In terms of this abstraction, the functional correctness contract of the write closure of a wrapper \( x \) could look roughly as follows (omitting error cases and technical details):

\[
\begin{align*}
& \wp \text{Repeat SingleStep} \{ s, Q(s) \} \\
& \left\{ \begin{aligned}
& \text{filter}_\text{val} \gamma_x (\lambda t. \text{filter}(P_c, t)) \ (t * P_c (\text{IOWrite}, a, v)) \\
& \rightarrow \text{pc} \rightarrow (p_w, b_w, e_w, a_w) \\
& \rightarrow r_0 \rightarrow w_{\text{ret}} \ast r_1 \rightarrow v \ast r_2 \rightarrow a \\
& \text{Repeat SingleStep} \\
& \text{filter}_\text{val} \gamma_x (\lambda t. \text{filter}(P_c, t)) \ (t + (\text{IOWrite}, a, v)) \\
& \rightarrow \text{pc} \rightarrow \text{updatePcPerm}(w_{\text{ret}}) \\
& \rightarrow r_0 \rightarrow w_{\text{ret}} \ast r_1 \rightarrow v \ast r_2 \rightarrow a
\end{aligned} \right. 
\]

where the updatePcPerm function maps \( E \) capabilities to their RX counterparts and leaves other capabilities untouched, and we used the fact that filters in our current examples are event-based (cfr. the definition of validity in Fig. 15) to rewrite \( F_x \) as \( (\lambda t. \text{filter}(P_c, t)) \). This contract expresses that when the write closure is invoked and the pc contains a capability \((p_w, b_w, e_w, a_w)\), with arguments \( v \) and \( a \) and a return capability \( w_{\text{ret}} \) in appropriate registers, then execution will jump back to \( w_{\text{ret}} \) with unmodified register contents. Additionally, a \( \text{filter}_\text{val} \gamma_x (\lambda t. \text{filter}(P_c, t)) \) resource is required to update the \( \text{filter}_\text{full} \gamma_x t \) resource in \( \text{invP}\_x \) using property 3 in Fig. 15. To prove the orthogonality precondition to this property, \( P_c \) needs to accept the requested external effect (IOWrite, \( a, v \)).

2Two caveats: (1) we do not use this abstraction in our Coq development but use the unfolded definition directly and (2) most of our contracts use a variant of this definition that allows the program to fail at an arbitrary point of execution.
In the postcondition, an updated filter_val resource is returned to express that the effect has been performed.

Unfortunately, the above contract does not quite work. The problem is that higher-level wrappers who wish to invoke a lower-level write closure do not directly own the necessary resource filter_val \( \gamma_x \) (\( \lambda t. \text{filter}(P_x, t) \)) \( t \). As we discussed, this resource is embedded in an invariant inv_p. Because this invariant is atomic, it must be restored after every individual instruction, as was already illustrated in Fig. 17. It is therefore not possible to extract the resource from the invariant for the duration of the wrapper invocation.

To solve this problem, we employ the technique of Higher-Order Concurrent Abstract Predicates (HOCAP) [17, 18]. The client wrapper will not extract the filter_val \( \gamma_{\text{pr}(x)} \) (\( \lambda t. \text{filter}(P_x, t) \)) \( t \) resource from its invariant and restore it after the invocation. Instead, it delegates the work of updating its invariant to the invoked wrapper, and the invoked wrapper will do this at exactly the right execution step, namely the step that executes the MMIO write. This means the contract will look as follows (again omitting error cases and technical details):

\[
E(v) \triangleq \forall v. (R(v) \land \text{pc} \rightarrow v \star \text{pc}(r \neq \text{pc}) \rightarrow w) \\
\rightarrow \text{wp} \text{Repeat SingleStep} \{ \_ \} \}
\]

\[
R(v) \triangleq \forall (r, w) \in R, r \neq \text{pc} \forall v \}
\]

\[
V(w) \triangleq \forall (z, v) \in V, w \triangleq \top \}
\]

\[
V'(w) \triangleq \forall (a, b, e, a) \in V, w \triangleq a \land w' \rightarrow w)
\]

In this contract, the caller provides a so-called view shift to the wrapper invocation: a type of logical callback that expresses how the lower-level wrapper \( \text{pr}(x) \) can update the client wrapper \( x \)'s invariant for them. View shifts are the reason why, in Fig. 17 wrapper\(_0\) was capable of immediately reestablishing the atomic invariants of wrapper\(_1\) and wrapper\(_2\) when it performed MMIO. Note that the client does not need to provide their filter_val \( \gamma_x \) (\( \lambda t. \text{filter}(P_x, t) \)) \( t \) resource beforehand, but instead is allowed to rely on the invoked wrapper’s filter_full \( \gamma_x \) \( t \) resource in the proof of the view shift.

In this proof, the view shift is allowed to consume additional client resources \( P \) that the caller has to provide at the start of the invocation, and produces resources \( Q \). Note that, since top level wrappers contain no filter_full \( \gamma_x \) \( t \) resource that needs to be updated externally, their contracts need not be parameterized by a view shift. In other words, the same boolean \( tP \) we discussed in the previous section will determine whether a wrapper’s contract is parameterized by a view shift.

If \( x \) is neither a top-layer nor a bottom-layer wrapper, then both \( x \) and \( \text{pr}(x) \) are parameterized by a (different) view shift, and a conversion between both needs to happen to prove the contract of \( x \). Using the invariant inv\(_{\text{pr},x}\), we can prove the following generally applicable view shift lowering lemma, to abstract away most reasoning related to view shifts when

\[
E(v) \triangleq \forall \text{reg, } R, \text{pc} \rightarrow v \star \text{pc}(r \neq \text{pc}) \rightarrow w)
\]

\[
\rightarrow \text{wp} \text{Repeat SingleStep} \{ \_ \} \}
\]

\[
R(v) \triangleq \forall (r, w) \in R, r \neq \text{pc} \forall v \}
\]

\[
V(w) \triangleq \forall (z, v) \in V, w \triangleq \top \}
\]

\[
V'(w) \triangleq \forall (a, b, e, a) \in V, w \triangleq a \land w' \rightarrow w)
\]

proving contracts:

**Theorem 1 (View Shift Lowering):**

\[
\text{inv}_{\text{pr}}(\lambda t. \text{filter}(P_x, t)), P_x, tP_x, \gamma_{\text{pr}(x)}, \gamma_{\text{bx}}, \gamma_x) \Rightarrow
\]

\[
(\forall t. \text{filter}_\text{full} \gamma_x t \ast P \Rightarrow \text{filter}_\text{full} \gamma_x (t \ldots) \ast Q) \Rightarrow
\]

\[
(\forall t. \text{filter}_\text{full} \gamma_{\text{pr}(x)} t \ast P \Rightarrow \text{filter}_\text{full} \gamma_{\text{bx}} (t \ldots) \ast Q) \Rightarrow
\]

\[
\text{cur} \_ t \text{r} \_ \gamma_{\text{bx}, x} t \Rightarrow \text{filter}_\text{full} \gamma_{\text{pr}(x)} (t \ldots) \ast Q \ast
\]

\[
\text{cur} \_ t \text{r} \_ \gamma_{\text{bx}, x} (t \ldots)
\]

This theorem states that the view shift that \( x \) is parameterized by can be lowered to one that satisfies the contract for \( \text{pr}(x) \), if \( x \) can provide the additional guarantee that \( P_x(t) \) and \( P_x(t \ldots) \) hold (which is precisely what \( x \) checks before admitting the event anyway), and if \( x \) makes \( \text{pr}(x) \) update the trace in the local state invariant inv\(_{\text{bx}}\). Recursively applying this theorem allows efficiently deriving higher-level driver specifications from lower-level ones. Note that view shift lowering accumulates resources in \( P \) and \( Q \) until the bottom-level driver is reached, at which point all higher-level invariants inv\(_{\text{pr}}\) are updated at once.

**C. A semantic model for capability safety**

In this section, we describe how we reason about the fact that unknown adversarial code satisfies the security objectives. We use a logical relations model to capture the notion of capability safety, i.e. the universal contract, that the hardware capabilities provide. Concretely, we will verify that our own wrapper closures are safe to execute, and that the adversary’s code is safe to execute, starting from safe register states, in particular states containing the wrappers’ closures.

Since our capability machine builds upon the bare-bones capability machine of Georges et al. [11], we reuse their logical relation without revocation and its formalization in Iris, and repeat a simplified version of the separation logic definition in Fig. 18. Additionally, we reprove their fundamental theorem (Theorem 2 below) in the presence of MMIO.

We restrict our explanation of Fig. 18 to the essentials required to understand the key concepts behind our proofs. The model consists of three different, mutually recursive relations:

- The value relation \( V : \text{Word} \rightarrow \text{IProp} \) (with IProp the type of propositions in Iris) defines when a word is safe.
- The expression relation \( E : \text{Word} \rightarrow \text{IProp} \) defines when a capability can safely be executed in the pc register.
• The register relation $R : \text{Reg} \to \text{iProp}$ lifts the value relation to an entire register bank (bar the pc). A register bank is safe if each general purpose register is safe. Technically, these relations are well-defined by so-called guarded recursion, see [11], [19].

We first discuss the definition of the value relation. $V(w)$ specifies an upper bound on the authority over memory that the word $w$ carries. Since integers and $O$-capabilities do not represent any authority over memory, they are always safe, as expressed by $T$. Enter capabilities are safe if it is safe to execute them in the pc after jumping to them (hence the RX permission). The presence of the persistent modality $\square$ and the later modality $\triangleright$ can be ignored by readers unfamiliar with them. All remaining capability types have read permission and these capabilities are safe if, for each $a'$ in their memory range $[b, e)$, an Iris invariant (denoted by a boxed assertion) exists that asserts that memory location $a'$ will always contain a safe value. The assertion $a' \triangleright p \triangleright \text{wp} \land w$ in the invariant expresses unique ownership of the memory location $a'$ with permission $p'$ and stored word $w$. Additionally, this assertion implies that $a'$ cannot be an $\text{MMIO}$ location. This ensures that adversaries cannot gain direct access to $\text{MMIO}$ memory and bypass our wrappers. Note that the invariant permits using a $p'$ that is at least as strong as $p$, i.e. $p \leq p'$. This allows for downgrading of capability permissions and aliasing of different permissions. Notice that no additional assertions have to be added to the definition of $V(p, b, e, a)$ for capabilities that carry additional write or execute authority. The intuitive reason is that the invariant already enforces that any written value needs to be valid, and that having read authority over a part of memory suffices for an adversary to copy code over to a region that it has write-execute permission over, and execute it there.

Next, we discuss the execution relation $\mathcal{E}$. It states that, given ownership of any initial register bank $\text{reg}$ that satisfies the register relation $\mathcal{R}$, we can safely run the machine with $v$ in the pc register. The weakest precondition assertion $\text{wp}$ uses a trivial postcondition $T$, which at first sight might seem odd. This suffices because Iris’ weakest precondition implicitly enforces that all invariants hold at each step of execution. Thus, any invariants related to $\text{OBJ}$ that a wrapper might define will also be enforced through the expression relation by the $\text{wp}$ assertion. For example, if wrapper1 were to set up an invariant that ensures that $P_1$ holds over the current trace $t$, i.e. $P_1(t)$, then this invariant is upheld at each step of execution when jumping to Adversary 1, under two conditions. First, Adversary 1 should be safe to execute, i.e. $\mathcal{E}(\text{RW}_X, \text{adv}_{b_1}, \text{adv}_{e_1}, \text{adv}_{b_1})$ holds. Second, to meet the precondition of the expression relation, Set-up Code 1 has to pass the adversary a safe register block, which notably requires proving that the read and write closures for wrapper1 are safe, since they are shown to be located in $r_1$ and $r_2$ when jumping to Adversary 1 in Fig. 10. This latter condition can be proven relatively straightforwardly from the contracts we described in Section IV-B. We hence focus on proving the different adversaries in our examples safe, by leveraging the FTIR (fundamental theorem of the logical relation).

The fundamental theorem has the following simple statement:

Theorem 2 (FTIR): \( \forall w, V(w) \rightarrow \mathcal{E}(w) \).

In other words: if a word is safe, it can safely be executed as well. The statement is identical to Georges et al.’s FTIR, but the proof is slightly different due to the presence of $\text{MMIO}$.

Through this theorem, it is easy to prove that for each region Adversary $X$ in Fig. 9, $\mathcal{E}(\text{RW}_X, \text{adv}_{b_X}, \text{adv}_{e_X}, \text{adv}_{b_X})$ holds. Since each init_config $X$ predicate requires initial memory to not contain any capabilities, since integers are always safe, and since the different set-up codes do not change the adversary’s memory, $V(\text{RW}_X, \text{adv}_{b_X}, \text{adv}_{e_X}, \text{adv}_{b_X})$ holds. The fundamental theorem then proves the result.

D. Sharing verification effort for fixed-structure wrappers

For wrappers $x$ that satisfy some conditions, we have developed a method in Coq that requires minimal manual verification effort in proving the functional contract and safety of the closures for these drivers. The conditions are as follows:

• Respect the calling convention described in Section III
• Enforce their security objective using an atomic invariant of the form $\text{inv}_p(F_x, F_x, t_p)$,
• Consist of a single read and write closure, and only jump to the read and write closures of their layer below driver.
• The state invariant $\text{inv}_{st}$ expresses both ownership of the code for $x$, and describes the closures for $\text{pr}(x)$ in the local state, and will hence be parameterized by the relevant addresses of both $x$ and $\text{pr}(x)$. It is then defined as follows:
  $\text{inv}_{st}(\text{d}_r\text{pr}(x), \text{d}_w\text{pr}(x), \text{d}_e\text{pr}(x), t_r, t_w, t_e)$.
• Have an implementation that satisfies a specific template, which is parameterized by instructions check_read and check_write to check $F_x \odot F_x$ in respectively the read and write closures. Wrapper 1 in Fig. 12 is structured like this, for example.

If these conditions are satisfied, the only manual effort involved is a proof that check_read and check_write indeed correctly enforce $P_x \odot F_x$ on $x$’s view of the trace, denoted by check_spec_read($F_x, F_x, \text{check_read}$) and check_spec_write($F_x, F_x, \text{check_write}$), respectively. These conditions are enforced in the local state invariant $\text{inv}_{st}$, adding the parameters $P_x$ and $F_x$ to it.

Under the above conditions, we can define simple contracts in the style of Section IV-B for the driver’s read and write closures, that only depend on $d_r\text{pr}(x)$, $d_e\text{pr}(x)$, $d_r\text{pr}(x)/d_w\text{pr}(x)$, and $t_p$ (to determine whether or not a view shift is required in the contract), and are denoted read_spec($d_r\text{pr}(x)$, $d_e\text{pr}(x)$, $d_r\text{pr}(x)$, $t_p$) and write_spec($d_r\text{pr}(x)$, $d_e\text{pr}(x)$, $d_r\text{pr}(x)$, $t_p$).

We can then state the following lifting theorem:

Theorem 3 (Lifting-theorem-read):

$$\text{inv}_v(F_x, F_x, t_p) \Rightarrow \text{read_spec}(d_r\text{pr}(x), d_e\text{pr}(x), d_r\text{pr}(x), \text{False})$$

$$\text{inv}_{st}(d_r\text{pr}(x), d_w\text{pr}(x), d_e\text{pr}(x), t_r, t_w, t_e, F_x, F_x) \Rightarrow \text{read_spec}(d_r\text{pr}(x), d_e\text{pr}(x), d_r\text{pr}(x), t_p)$$

A similar result holds for the write spec. Note that this theorem is generic in $t_p$. It states that if all necessary conditions on the state of a wrapper are met ($\text{inv}_{st}$), and if an invariant
enforces \( F \) and \( P \) on the trace (inv\(_P\)), then we can lift a spec for the pointed-to wrapper in the previous layer, to the current layer. The bottom-most wrapper, i.e. \( \text{wrapper}_P \), in the examples, constitutes the base case in this theorem, and still has to be manually verified to satisfy \( \text{read\_spec} \) and \( \text{write\_spec} \), since its code and local state do not satisfy the template (cfr. Fig. [11] and LS\(_0\) in Fig. [4]).

Finally, we can prove top-level wrapper closures secure using the following theorem:

**Theorem 4 (Wrapper-safety-read):**

\[
\text{read\_spec}(d_{r_x}, d_{e_x}, d_{r_x}, \text{True}) \Rightarrow V(E, \text{Global}, d_{r_x}, d_{e_x}, d_{r_x})
\]

Again, a similar result holds for the write spec.

Since we have abstracted most reasoning related to wrappers into Theorems 3 and 4, the check\_read and check\_write instructions are indeed the only code we need to hand-verify to ensure wrapper safety. We used this approach to verify the first example. The shape of the local state and the code of the second example is slightly different, due to the presence of the timer closure, so it did not fit this approach. However, Theorem 4 still applied once the \( \text{read\_spec} \) and \( \text{write\_spec} \) were proven.

V. RELATED WORK

Hardware-supported security mechanisms as well as software verification have been important ingredients of secure system development for a long time [20]. We discuss the most important lines of related work and how they differ from our results.

One distinguishing feature of our work is that we support **robust** modular verification. Robust verification requires underlying programming language or hardware support to protect verified code from untrusted code. Earlier work has demonstrated how to robustly verify safety properties in settings where that protection is not nested. For instance, Sammler et al. [21] and Jia et al. [22] have proposed approaches to robustly verify safety properties in the presence of untrusted code that is confined using some sandboxing mechanism. Alternatively, Agen et al. [23] have used trusted execution environments (TEEs) like Intel SGX [24] or Sancus [25] to protect a verified module from an unverified context, but verification is at the level of C code and their focus is on proving assertions about the protected module rather than full system properties. In our approach, protection of verified code is provided by the capability-based instruction set architecture, and this enables handling of nested protection.

Protecting verified code from unverified code is of course closely related to protecting trusted system software from untrusted user code. Operating systems, microkernels, and hypervisors use hardware privilege levels to protect themselves, and hence the rich line of work on verifying properties of such system-level software can be seen as an instance of robust modular verification and hence related to our work. Some important milestones include the verification of the seL4 microkernel [26], [27], and the verification of Microsoft’s Hyper-V hypervisor [28]. The focus is however on proving properties (such as functional correctness, or selected safety properties) of the system software under a single attacker model where all non-privileged code is untrusted. Like our work, the verification of CertiKOS [29] supports modular (compositional) and layered verification of device drivers. But an important difference is that verification in CertiKOS is compositional but not robustly modular: only user-level code is isolated at run time from kernel-level code, and any unverified code at kernel-level becomes part of the trusted computing base. The journal version of the CertiKOS driver verification paper [30] also has an extensive overview of other related work on operating system verification.

The usefulness of being able to nest protection systems has been recognized in the system security research community, and several systems have been proposed that support, for instance, nested virtualization [31], [32]. However for none of these systems, any code-level formal guarantees are provided.

It is the reliance on capabilities as the underlying protection mechanism that enables arbitrary nesting for our approach. Capability-based architectures have a rich history [33], [12], [8] and assembly languages [5]. The fact that capabilities support nesting has been observed before, e.g. in Mark Miller’s PhD thesis [34]. It has also been known for a long time that they provide a great foundation for nestable security architectures in high-level languages [35]. Over the past decade, capabilities at the instruction set architecture level have seen renewed interest, largely thanks to the CHERI project [6]. CHERI is a hybrid architecture that supports capability based protection for user-level code and classical memory protection for isolating kernel and processes. Hence, CHERI does not use the nested encapsulation for wrappers that we study in this paper. However, CheriRTOS [36], a CHERI-aware real-time operating system, supports capability-based fine-grained isolation for device drivers and would be a candidate implementation platform for our verified wrapper stacks.

Capability-based systems support the enforcement of security properties in the presence of arbitrary untrusted code in the system through what are typically called **object capability patterns**, like the membrane or caretaker patterns [34]. It is only relatively recently that sufficiently powerful formal reasoning approaches have been developed that can prove such properties. Devriese et al. [7] proposed a reasoning approach based on logical relations, (what we now call) universal contracts and the concept of effect parametricity. Swasey et al. [8] developed the first program logic, OCPL, that can compositionally specify and verify the properties enforced by object capability patterns. Building on these ideas, program logics have been developed to reason about software in low-level capability-based instruction set architectures [10], [11]. These logics, as ours, are built in Iris [12], a separation logic framework for building program logics. Iris integrates, unifies, and simplifies a wide variety of mechanisms for reasoning about programs that can be higher-order, concurrent, or use mutable state. The results in this paper can be seen as an application and extension of these logics to prove security properties for multiple stakeholders in a system with nested encapsulation.
One of the motivations for our approach is the minimization of the Trusted Computing Base (TCB). The various stakeholders in the system want to ensure their security objectives while trusting as little other software as possible. The use of small software modules isolated by some hardware protection mechanism to enforce full-system security properties has been proposed in multiple guises in the system security field. DriverGuard [37] uses virtualization techniques to implement fine-grained protection on IO through specific devices with a small TCB. Para-pass-through virtualization [14] specifically aims to provide full-system guarantees while relying only on a small piece of software, albeit for only a single attacker model. One could argue that micro-kernels or hypervisor-based systems are similar. We show that it is possible to apply this principle at multiple levels in the same system and verify security.

Related to minimization of the TCB is the idea of compartmentalization, breaking a large program in smaller mutually distrusting parts and relying on some underlying security mechanism to protect the parts from one another. Juglaret et al. [38] have studied the formal guarantees provided by a compartmentalizing compiler. They consider multiple compromise scenarios, where an attacker can compromise different subsets of program parts, somewhat similar to our consideration of multiple attacker models. However, they only consider the benefits provided by a compartmentalizing compiler, and do not consider verification.

If the full system security objectives to be verified include statements about IO through a given device, then necessarily the device driver(s) for that device will need to be verified. Hence, the approach proposed in this paper verifies a subset of the driver stack for a device depending on the attacker model. Device drivers have been a target of verification in a wide body of related work, using techniques ranging from model checking (e.g. [39]) to deductive verification (e.g. [40]). The objective of these verification efforts is to improve kernel reliability by showing that drivers correctly use specific kernel APIs, or do not have memory safety or concurrency bugs. That is very different from our objective of verifying that a thin wrapper around a device enforces a specified security property.

VI. CONCLUSION AND FUTURE WORK

The fine-grainedness of hardware capabilities, in combination with object capabilities as a primitive enabling encapsulation, allows for efficient and safe nesting of wrappers enforcing security properties without costly context switches. This would be hard to achieve through conventional, more coarse-grained security primitives. By capitalizing on capabilities and extending an existing formal model for a capability machine, we managed to generalize classical robust modular verification to a nested setting. With nested robust modular verification, a system that consists of multiple layers is verified robustly several times, each time verifying an increasing number of layers and enforcing more security objectives, and considering the rest of the code base as well as the environment untrusted. Crucially, our approach retains the compositionality of the programming logic and does not require verifying the same code multiple times, even when proving security objectives for different layers. Our Iris development provides ease of use when modularly verifying nested wrappers, only requiring specific checking instructions to be verified and lower layer wrappers proven safe, in order to obtain proofs of safety for an entire stack of wrappers. These proofs of wrapper safety are essential in proving the different full-system security properties.

In future work, we wish to extend our basic model of IO, to achieve nested full-system guarantees in more involved settings. Concretely, we believe it to be possible to achieve similar guarantees when adding interrupts to the basic capability machine. This would require formalizing interrupts dynamics, redefining the notion of weakest precondition correspondingly, as well as proving specifications for registered interrupt handlers themselves. Another interesting extension is allowing peripherals to perform some form of capability-restricted DMA access, as hinted at by Markettos et al. [41]. Additionally, with the advent of modern capability hardware in the form of Arm’s Morello prototype [42], it will soon become possible to perform practical experiments on capability-enabled hardware, thereby obtaining a fair comparison between our work and similar approaches using different security primitives.

REFERENCES


