Towards End-to-End Verified TEEs via Verified Interface Conformance and Certified Compilers

Farzaneh Derakhshan*, Zichao Zhang†, Amit Vasudevan‡, and Limin Jia§
Carnegie Mellon University, Pittsburgh, USA
Email: *fderakhs@andrew.cmu.edu †zichaoz@andrew.cmu.edu ‡amitvasudevan@acm.org §liminjia@cmu.edu

2023 IEEE 36th Computer Security Foundations Symposium (CSF) | 979-8-3503-2192-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSF57540.2023.00021

***Abstract*—Trusted Execution Environments (TEE) are ubiquitous. They form the highest privileged software component of the platform with full access to the system and associated devices. However, vulnerabilities have been found in deployed TEEs allowing an attacker to gain complete control. Despite the progress made in fully-verified software systems, few deployed TEEs are fully-verified, due to the high cost of verification. Instead of aiming for full-functional correctness, this paper proposes a formal framework and approach that leverages compartmentalization at the source level to bring security-relevant properties verified at the source level down to the binary via existing certified compilers. The benefit of our approach is the relative low cost of verification: developers can use existing automated program verification tools and certified compilers. Our case studies demonstrate how security properties verified on two open-source TEEs at the source level can be pushed down to the compiled code by using an off-the-shelf certified compiler.**

***Index Terms*—Software/Program Verification, Security and Privacy Protection, Specifying and Verifying and Reasoning about Programs**

I. Introduction

Trusted Execution Environments (TEE) form the highest privileged software component. They are used in the vast majority of embedded platforms and encompass BIOSes, firmwares, TEE OSes, and hypervisors. The application domains of TEE range from mobile environments, smartphones, wearables, and low-end IoTs to servers and industrial control systems [1]–[4]. TEEs are a key security mechanism to protect the integrity and confidentiality of applications on a majority of commodity computing platforms [4]–[11] by enabling the execution of privileged and security-sensitive applications inside protected domains isolated from the platform's operating system (OS). On some platforms TEEs leverage certain hardware mechanisms for their functionality (e.g., Intel SGX on x86 and ARM Trustzone [12]). Subversion of a TEE gives the attacker full control of the entire platform since the TEEs are the highest privilege operating software. This is exemplified by the exploits TEEs have faced in recent years [5], [13].

Formal verification of TEEs can remove many of the vulnerabilities. However, verifying safety critical software such as TEEs for their functional correctness has not found practical widespread use, despite the progress made in producing formally verified kernels [14]–[19]. This is due to the prohibitively high cost of verification in terms of money, time, and developer expertise. On the other hand there have been several approaches targeting formally verified TEEs with a focus on practicality such as XMHF [20], uberXMHF [21], Security Microvisor [22], and Contiki [23]. These projects focus on specific security properties in lieu of full-functional correctness, with the goals of being development friendly and using automated verification tools at the source level. However, a significant shortcoming of using source-level verification tools is the lack of guarantees on the compiled code.

One key observation of this work is that we can leverage memory compartmentalization and properties of certified compilers (e.g., CompCert [24], [25]) to prove that the guarantees verified at the source level also hold on the compiled code. We formalize a programming framework based on prior work [21] that advocates programming systems like TEEs as a collection of objects, called überobjects, that access separate memory locations and conform to a public interface. A concrete notion of überobjects targeting C and x86 assembly was first introduced as the building block of überSpark, an architecture for building extensible hypervisors [21] to ensure hypervisor's memory integrity at the source level. This paper takes the abstraction of überobjects one step further and shows formally that (1) if at the source level, überobjects are shown to *respect their interface* (i.e., respect memory separation and the rely-guarantee conditions of all überobjects), then we can verify each überobject separately, and the compositional concurrent multi-core run of them satisfies the same properties; and (2) certain properties verified at the source level also hold on the compiled code if a certified compiler with compositional properties such as CASCompCert [25] is used. These properties not only include standard assertions at function return points but also information flow properties between überobjects (i.e., compartments do not interfere with each other).

To prove the above results, we formalize a general abstraction of überobjects as units for memory compartmentalization and verified interface conformance. Compared to prior work [21], our abstraction is liberated from a specific programming language and architecture (software and hardware), with an abstract semantics that models concurrent execution of überobjects in the context of multiple CPU cores and interrupts. We believe our approach will allow decomposition of verification of compiled code into source-level verification

Copyright 2023 IEEE. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM23-0064

© 2023, Farzaneh Derakhshan. Under license to IEEE.

DOI 10.1109/CSF57540.2023.00021

and using verified compilers that can preserve the verified source-level properties, and thus enabling more practical use of disparate off-the-shelf formal verification tools to obtain end-to-end properties on existing commodity TEEs.

We illustrate our approach in practice via two case-studies. First, we show how our formalism helps translate to binary, the already verified source-level memory separation properties on an existing open-source x86 micro-hypervisor TEE by employing source-level verification [21] and the certified compiler CASCompCert [25]. Then, we demonstrate the language and architecture (software and hardware) independence of our approach by applying it to analyzing a light-weight open-source ARM Trustzone TEE for an embedded platform.

This paper makes the following contributions:

- We provide a general model for überobjects as units of memory compartmentalization and define a locally verifiable predicate to ensure each compartment respects its interface.
- We show that our model enjoys compositional verification at the source level: verifying the local predicate for each compartment carries over to the concurrent multi-core executions with interrupts.
- We show that compartmentalization also allows us to enforce coarse-grained noninterference property easily.
- We prove that certain types of assertions and the noninterference property can be carried over to the binary level by using a suitable compiler.
- Through case studies, we demonstrate that end-to-end security guarantees on *existing* TEEs can be achieved by building a practical, decoupled, tool-chain that leverages existing Certified Compilers (e.g., CASCompcert) and off-the-shelf verification tools (e.g., Frama-C).

The rest of this paper is organized as follows. Section II presents background on TEEs, certified compilers, and überobjects. Section III gives an overview of our framework. Section IV presents details of our formal model including syntax and semantics. Section V discusses high-level theorems and proofs. Section VI presents our case-studies on an existing open-source x86 micro-hypervisor TEE and an open-source light-weight ARM TrustZone TEE. Section VII describes related work and Section VIII presents our conclusions. Detailed proofs and definitions can be found in the extended TR [26].

II. Background

We first review TEE (section II-A). We then present the central element of our system: überobjects, extended from prior work [21] (Section II-B). Next, we review an existing DSL for analyzing inline assembly in überobjects (section II-C). Finally, we review certified compilers (Section II-D).

A. Trusted Execution Environments (TEE) security

TEEs aim at securing the execution of Trusted Applications (TA)s or *trustlets* that run within the protected TEE framework. TEEs are privileged software entities that typically rely on a small portion of dedicated hardware capabilities (e.g., enclaves, memory protection unit, virtualization) in order to boot-strap their execution and then setup required memory protections before running the (rich) guest OS. There have been TEEs for different platform architectures ranging from x86 [5], [20], [21], ARM [6], [8], [9], [11] to RISC-V [27], [28] and low-cost micro-controller units [22], [23].

A majority of existing TEEs and TA's software code-base is written in C and Assembly. The TEE programming language requirement is driven by having to: (a) interface with the OS kernel and device driver components which are mostly written in C for the majority of popular OSes, and (b) have low-level access to system resources including CPU critical registers, memory units, bus bridges, and devices.

TEEs are generally assumed to be more secure than modern OSes due to the memory and privilege separation enforced via a combination of hardware and software mechanisms and their smaller software Trusted Computing Base (TCB), which is several orders of magnitude smaller than standard OS. However, TEEs and TA's have faced many exploits over the past years ranging from privilege escalation, buffer overflows, input validation errors, and integer overflows [5], [13]. This necessitates the formal verification of security properties on TEE and TA code-bases to achieve high assurance on the security posture provided by them.

B. überobject: a framework for compartmentalization

An überobject is a programming compartment (or module) with exclusive access to a memory region and other system resources (e.g., CPU control registers, devices). An überobject's public interface consists of a collection of public API declarations, *pubAPIs*, which can be called by other überobjects to access the guarded memory (and other resources) and can be restricted to a specific calling convention (e.g., based on their integrity labels). A distinguished public API, *init*, sets up the überobject in a known-good initial state (e.g., initializes the überobject when it is loaded on a CPU core for the first time.) Each überobject has a set of internal functions not accessible from other modules. An überobject can also include Assembly functions, discussed further in Section II-C.

Contracts An überobject is accompanied by a behavior contract of its public interface in the form of pre- and post-conditions. The interface guarantees that if the precondition is satisfied upon invoking a public method, then the postcondition is guaranteed to hold upon return of that method.

Sequential vs. concurrent An überobject may be concurrent or sequential. The public methods of a concurrent überobject can be invoked in parallel on multiple CPU cores. In contrast, at most one core can invoke the methods of a sequential überobject at a time. In a concurrent execution with multiple cores, sequential überobjects enforce data race freedom via per-überobject locks. Data race freedom, which forbids two threads from accessing a location simultaneously when at least one of the accesses is a write, is an essential property for preserving the behaviors of source-level programs throughout the compilation process (required by CASCompCert [25]), and cannot be guaranteed in the presence of concurrent überobjects. In this paper, to enforce data race freedom, while

still supporting shared-memory concurrency, we consider sequential überobjects guarding a single resource.

Resources The formalism of this paper models system resources that überobjects have exclusive access to as shared memory locations (heap). This includes the set of special *control registers* (e.g., interrupt control register and interrupt descriptor table register). Only the assembly functions in an überobject with exclusive access to a specified control register may read from/write to it. Our current model does not handle accesses to a device; though extending it to include devices will be straightforward since accessing a device transpires via a memory-mapped IO, a special case of shared memory.

C. CASM: analyzing assembly

For verification purposes, assembly code in überobject is written using CASM, a DSL using C functions to encode assembly instruction semantics (introduced in [21]). We call these functions CASM functions. For example, for the x86 instruction `mov cr3` involving register `eax` there is a corresponding CASM pseudo-function called `ci_movl_eax_cr3`. Each CASM instruction pseudo-function is defined in a hardware model (written in C) and models the corresponding CPU instruction (e.g., access to memory and to registers). During verification, each CASM instruction is replaced by the C source code from the hardware model. The resulting C-only program is verified for required properties (e.g., they respect their specifications and the specification of the other functions they interact with). CASM functions are also verified to respect the C application binary interface (ABI) and stack frames (e.g., not clobber callers registers or stack frames). During compilation, each CASM instruction is replaced by the corresponding Assembly code.

D. Certified compilers

A common goal of certified compilers is to preserve the behaviors of source-level programs throughout the compilation process, ensuring that the behaviors of a target-level execution are a refinement of the source-level execution. Originally, certified compilers, e.g., CompCert [29], only considered whole program compilation in which a closed program written in a single source language is compiled to a target language. To handle more realistic situations, several work has generalized the results of CompCert to more modular settings [24], [25], [30], [31]. Compositional CompCert introduces a linking semantics to allow composition of different modules, each potentially written in a different language. Each source module is compiled separately, with potentially different compilers, to a corresponding target module. The target modules are linked with the same linking semantics. A local structured simulation is introduced based on a rely-guarantee condition that maps source-level memories to target-level memories to prove compiler correctness. CASCompCert uses a similar approach to address concurrency by providing a linking semantics that allows concurrent execution of multiple threads. The correctness proofs of both Compositional CompCert and CASCompCert assume that the source modules satisfy some

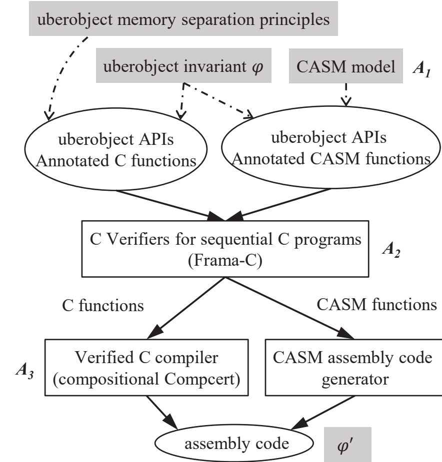

Fig. 1: Framework overview

properties. For example, CASCompCert, among a few other assumptions, requires that individual modules do not leak their stack pointers and that each source execution is data-race free.

III. Formal framework overview

We now present an overview of the formal framework and development flow that we propose to push verified guarantees at the source level to the compiled code. We illustrate the high-level development flow in Fig. 1.

Prior work has shown that the verification results using a sequential verifier on überobjects comprising hypervisor source-code carry over to an execution environment where a sequential hypervisor supports a multi-core unverified guest OS [21]. The aforementioned work postulates that using a certified compiler results could be pushed down to compiled code, but does not include details or proofs. One contribution of this paper is to demonstrate via a formal framework and associated proofs that we can indeed achieve end-to-end security guarantees, in a more general setting with interrupts and multiple CPU cores running verified überobjects, by leveraging results of certified compilers (e.g., CASCompCert [25]).

We illustrate the high-level development flow advocated by our formal framework in Fig. 1. The three main components in development flow are represented in rectangle boxes: a verification tool for C (for example, Frama-C), a certified C compiler, and an assembly code generator. Similar to prior work [21], we envision safety-critical applications that developers aim to analyze using our formal framework and development flow follow a set of programming idioms outlined in Section II-B and Section II-C.

To analyze relevant properties, which we call überobject invariants (e.g., DMA protection is always turned on), the programs are annotated with directives that the analysis tool needs. For example, to use Frama-C's WP plugin, one needs to add specifications and assertions. The programming idioms dictated by our formal framework also are translated to annotations for the C analysis tool to check. These are illustrated as the gray boxes on top of Fig. 1. The dashed lines indicate that these are manually translated into the annotations in the program. Our high-level goal is to show that the überobject

properties φ' (which is the low-level equivalent of the high-level invariant φ) hold on the concurrent execution of the compiled assembly code, even though φ is checked using a sequential C analysis tool on source code. Specifically, we show that (1) our principled shared memory accesses or separation of memory accesses can ease the verification process and that (2) there is a set of assumptions (requirements) we have to place on these tools for the reasoning to be sound.

Compiler requirements A contribution of our work is to provide an abstract criterion for compilers, independent of a specific source and target languages, that ensures compiling each compartment separately preserves key source level properties, i.e., memory separation and überobjects' specifications, in any concurrent execution at the target (binary) level. For instance, a low-level requirement is that compilers preserve exclusive access to the specified control registers by not using them in the compilation process, e.g., via Application Binary Interfaces (ABI) requirements. We show that CASCompCert, in particular, satisfies the required criterion.

Tool compatibility requirements We further identify three categories of tool assumptions, denoted A_i, in Fig. 1. A_1: we assume that the DSL semantics accurately reflect the assembly semantics. A_2: We assume that the C verifier's logic is sound, i.e., it only verifies correct predicates. A_3: We assume that the C semantics used by the C analysis tool and the certified C compiler agree. We also assume the certified C compiler's semantics for assembly is accurate.

Next we discuss how to discharge these assumptions and why they can be satisfied in practice. However, the details on discharging them are out of the scope of this paper. Assumption A_1 can be formally discharged by proving a simulation between CASM and Assembly. Given the small number of instructions used in implementing TEEs, testing-based validation suffices as argued in prior work [21]. Formal C-verifiers satisfy assumption A_2 by either providing a formal proof of their soundness, e.g. Verifiable C [32], or by proving a soundness result for some parts and describing the circumstances that may threaten soundness, e.g., Frama-C. Assumption A_3 can be discharged when working with CompCert C compiler and C verification tools such as Verifiable C and Frama-C. The program logic in Verifiable C uses the same semantics as CompCert, and Frama-C agrees on the semantics of programs written in a subset of C, called Clight.

IV. Model syntax and semantics

We describe the high-level schema of a system of überobjects and their memory state, and introduce syntax and semantics governing their multi-core concurrent execution.

A. Model syntax

überobject syntax The syntactic constructions for defining a überobject are summarized in Fig. 2. A system of überobjects $\mathcal{U}$ maps a unique identifier to a *überobject*, which is a tuple consisting of (1) a language, *lang*, with which *überobject* is implemented, (2) a lock, *ulock*, that ensures *überobject* can only run on one core at any time, (3) an exclusive region of

uberobjects	$\mathcal{U}$	$::=$	$\cdot \mid uid \mapsto \ddot{u}obj, \mathcal{U}$
uber object	*üobj*	$::=$	$(lang, ulock, M, init, \overrightarrow{pubAPI}, \overrightarrow{casmfd}, \overrightarrow{fd})$
public API decl	*pubAPI*	$::=$	$f(\overrightarrow{x{:}\tau}) : \tau' = \mathsf{lockUobj};\ gcmd;\ \mathsf{unlockUobj}$
CASM fun decl	*casmfd*	$::=$	$f(\overrightarrow{x{:}\tau}) : \tau = gcasm$
internal fun decls	*fd*	$::=$	$f(\overrightarrow{x{:}\tau}) : \tau' = gcmd$
language param	*lang*	$::=$	$(\mathsf{syntax}, \longrightarrow, \mathsf{func})$

Fig. 2: Syntax for überobjects

the heap M owned by *überobject*, (4) a public Application Programming Interface, *init*, that sets up the initial state of the überobject, (5) a set of public Application Programming Interfaces, *pubAPI*, (6) a set of CASM functions, *casmfd*, and (7) a set of internal function declarations, *fd*. A public API holds a lock on the memory when it is initialized and releases it only after it returns to avoid a data race.

Instead of modeling the detailed semantics of the source language (e.g., C for most TEE) or the target language (e.g., x86 assembly), we assume each überobject takes in as a parameter, the language (*lang*) that it is implemented in, which we discuss in detail in Section IV-B.

Generalized commands, *gcmd*, consist of commands in agreement with the syntax of the language in which the überobject is implemented (*lang*). CASM commands, *gcasm*, are assembly code for the target hardware architecture (e.g., x86) written in CASM DSL as explained in II-C. For example, in an überobject *uid* with *uid.lang* $=$ C, the functions declared in $uid.\overrightarrow{pubAPI}$, and $uid.\overrightarrow{fd}$ are implemented using commands in the C language, while the functions in $uid.\overrightarrow{casmfd}$ are implemented by CASM commands.

Memory model The layout of the memory for $\mathcal{U}$, with m distinct überobjects is illustrated in Fig. 3. The heap is compartmentalized into m separate memory locations $uid_i.M$ for $i \leq m$. Each heap compartment $uid_i.M$ is a set of addresses defined as $uid_i.M \in \mathcal{P}(Addr)$, such that for all $i \neq j \leq m$, $uid_i.M \cap uid_j.M = \emptyset$.

We summarize the syntax for defining memory below. As illustrated in Fig. 3, our state model also includes the regions preserved for the stack frames, i.e., freelists [33], defined similar to their counterparts in CASCompcert [25].

Exclusive heap	M	$\in$	$\mathcal{P}(Addr)$
Freelist stream	$\mathcal{F}$	$::=$	$F, \mathcal{F}$
Freelist	F	$\in$	$\mathcal{P}^{\omega}(Addr)$
Memory state	σ	$\in$	$Addr \hookrightarrow val$

A freelist can be infinitely large and is used by the thread to allocate its local stack locations as needed. Since termination guarantee is out of scope for this paper, we assume that

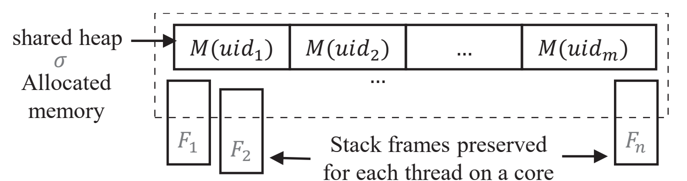

Fig. 3: Memory model

we have a stream of such infinite freelists $\mathcal{F}$ available upon request. The stream of freelists $\mathcal{F}$, is a coinductive definition and consists of freelists of the type $F \in \mathcal{P}^{\omega}(Addr)$, i.e., infinite sets of memory addresses. We assume that all freelists F in the stream $\mathcal{F}$ are mutually disjoint from each other and from the heap locations. We define memory state σ as a mapping from the heap and the allocated parts of the stack to values.

Runtime constructs Our runtime construct consists of multiple CPU cores with the following syntax.

thread pool $T ::= \cdot \mid tid \mapsto \langle uid, mainf, lang, F, \rho, a\rangle, T$
single core state $k ::= \langle cid, T\rangle$
multi core state $K ::= \langle \mathcal{F}; \sigma; \vec{k}; cid\rangle$

Each CPU core k is a tuple of a unique core identifier *cid* and a list of threads T for executing external function calls. A single thread, uniquely identified with *tid*, consists of (1) the main function *mainf* that initializes the thread and can either be a publicAPI or a CASM function, (2) language of the thread *lang*; if *mainf* is a CASM function then *lang* is CASM and otherwise it is the language of *uid*, (3) the freelist F allocated to the thread, (4) the current internal core state ρ of the thread storing key control flow state, (5) the instance, a, that satisfied the precondition *mainf* when the thread was initialized (more details later). Each multicore state K consists of a stream of freelists $\mathcal{F}$, a mapping from addresses to values, σ, a list of CPU cores, and the active (running) core *cid*. The active core *cid* in a multicore state can switch non-deterministically.

The interface specification For each überobject $uid \in \mathsf{dom}(\mathcal{U})$ and every function $f \in uid.\overrightarrow{casmfd} \cup uid.\overrightarrow{pubAPI}$, we fix an interface $\{P(x)\}uid.f\{Q(x)\}$ and collect it in the set $\Delta_{\mathcal{U}}$. $P(x)$ and $Q(x)$ are pre- and post-condition of the function f, respectively. We also refer to them as $uid.f.pre(x)$ and $uid.f.post(x)$. These pre- and postconditions are the behavior contracts of the interface. The memory footprint of the interface for the überobject *uid* is specified by $(uid.M)$.

In line with function specifications in verifiable-C separation logic [34], we define the precondition $uid.f.pre(x)$ to be parametric in x of type A and has a type $A \to \mathtt{Prop}$. A universally quantified version of it, $\forall x : A.uid.f.pre(x)$, is a first-order predicate defined over a global memory state σ, which is a pair of heap and stack as discussed in Section II-B. The heap includes other resources, e.g., control registers. We write $\sigma \vDash uid.f.pre(a)$ to specify that the memory σ satisfies $uid.f.pre(x)$ when x is instantiated with the instance a. We assume that the predicate is only defined over the heap owned by *uid*, i.e. $\sigma\restriction_{(uid.M)}$. The same holds for $uid.f.post(x)$. Moreover, for the sake of simplicity, we assume that A has a simple base type, e.g. a list of integers or a string.

For example, we can enforce the specifications of the public API function foo owned by an überobject *uid*:

$\mathsf{foo} := l_1 = *l_0;$ where $uid.M = \{l_0, l_1\}$ and
$uid.pre.\mathsf{foo}(x) := l_0 \hookrightarrow x$ and $uid.post.\mathsf{foo}(x) := l_1 \hookrightarrow x$

The function foo copies the contents of location l_0 to l_1. The specification states that for any instance a if the pre-condition holds for a when calling foo, i.e. $l_0 \hookrightarrow a$, then the post-condition holds for a when foo returns, i.e. $l_1 \hookrightarrow a$.

B. Local syntax and semantics

The language parameter, *lang*, of an überobject dictates the syntax of its public API and internal functions, and the semantics by which those functions evaluate. The syntax, denoted by `syntax`, defines a grammar for commands, *gcmd*, and describes the internal states, ρ, that manage the local control flow inside a function and the internal call stack, e.g., control continuations.

A pair of a memory state, σ, and an internal state, ρ, describes the program state. The semantics $\longrightarrow$ defines a local transition migrating a program state of the form σ, ρ with respect to a given freelist F: $F \Vdash \sigma, \rho \longrightarrow^{\delta}_{\iota} \sigma', \rho'$. The label δ indicates the footprint (read and write set) of this step; when δ is empty we omit it. The label ι specifies the type of internal step: `abt` stands for a step that results in an abort, `ret` stands for function return, and τ stands for effectless internal steps.

We distinguish between the internal and external calls: a general command defining a function in its public APIs ($\overrightarrow{pubAPI}$), or internal functions ($\overrightarrow{fd}$) can make (a) an internal call to the functions defined in *fd* of the same überobject, (b) an external call to the CASM functions declared in $\overrightarrow{casmfd}$ of the same überobject, or (c) an external call to a public API of another überobject. A CASM command defining a function in $\overrightarrow{casmfd}$ may make an internal call to the functions defined in *casmfd* of the same überobject, or an external call to a public API of another überobject. An überobject makes internal steps with internal calls. When an external call is made, it cannot take internal steps until the external call returns.

The final element of *lang*, the set func includes 4 functions for initializing the internal state (initCore) and governing transitions related to external calls and returns. (`extCall`, `extRet`, and `halt`). The function `extCall` is called when an external call is made, `extRet` is called when an external call returns, and `halt` is called when the current thread is ready to return to its caller. The semantics of a language specifies the behavior of these four functions as follows:

$F \Vdash \mathtt{initCore}(f\,\vec{v}) = \rho$ initializes a core given a public API of an überobject or a CASM function f on arguments $\vec{v}$ consulting the declarations in the überobject that owns them.

$F \Vdash \mathtt{extCall}(\rho) = \langle f\,\vec{v}, \rho'\rangle$ returns a pair $\langle f\,\vec{v}, \rho'\rangle$ when ρ calls an external function f with arguments $\vec{v}$ and puts ρ to a waiting state ρ'.

$F \Vdash \mathtt{extRet}(\rho, v) = \rho'$ updates ρ, waiting for the return of an external call, to a new state ρ', with a return value v.

$F \Vdash \mathtt{halt}(\sigma, \rho) = \langle v, \rho'\rangle$ returns ρ' and a return value v iff $(F \Vdash \sigma, \rho \longrightarrow_{\mathtt{ret}} \sigma, \rho')$ with the global memory state σ.

C. Operational semantics

The concurrent multicore semantic rules are of the form $\langle \mathcal{F}; \sigma; \vec{k}; cid\rangle \Longrightarrow^{\delta}_{\iota} \langle \mathcal{F}'; \sigma'; \vec{k}'; cid'\rangle$. The δ is inherited from the underlying local internal steps and refers to read/write footprints of the step. The label ι is still used to specify the type of multicore step. For instance, `load` stands for loading a configuration and `call` stands for an external call, and `ret` stands for return. Selected rules are summarized in Fig. 4.

Cmd

$$\frac{\begin{array}{ll} T = \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho, a), T_1 & F \Vdash_{\mathit{sl}} \sigma, \rho \longrightarrow^{\delta}_{\tau} \sigma', \rho' \\ T' = \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho', a), T_1 & \mathit{dom}(\delta) \subseteq \mathit{uid}.M \cup F \end{array}}{\langle \mathcal{F}; \sigma; \langle \mathit{cid}, T \rangle, \vec{k}_1; \mathit{cid} \rangle \Longrightarrow^{\delta}_{\tau} \langle \mathcal{F}; \sigma'; (\mathit{cid}, T'), \vec{k}_1; \mathit{cid} \rangle}$$

Switch

$$\frac{\mathit{cid}' \in \mathit{dom}(\vec{k})}{\langle \mathcal{F}; \sigma; \vec{k}; \mathit{cid} \rangle \Longrightarrow^{\mathtt{emp}}_{\tau} \langle \mathcal{F}; \sigma; \vec{k}; \mathit{cid}' \rangle}$$

Load

$$\frac{\begin{array}{c} \forall i \leq n.\, \mathit{uid}_i \in \mathit{dom}(\mathcal{U}) \qquad \forall i \neq j \leq n. \mathit{uid}_i \neq \mathit{uid}_j \\ F_i \Vdash \mathit{uid}_i.\mathit{lang}.\mathsf{initCore}(\mathit{uid}_i.\mathit{init}) = \rho_i \\ \sigma_0 \vDash \mathit{uid}_i.\mathit{init}.\mathit{pre}(a_i) \qquad \mathcal{F}_0 ::= F_1 \cdots F_n :: \mathcal{F} \\ T_i = \mathit{tid}_1 \mapsto (\mathit{uid}_i, \mathit{init}, \mathit{uid}_i.\mathit{lang}, F_i, \rho_i, a_i) \\ \vec{k} = \langle \mathit{cid}_1, T_1 \rangle, \ldots, \langle \mathit{cid}_n, T_n \rangle \\ \mathit{cid} \in \mathit{dom}(\vec{k}) \qquad \forall \mathit{uid} \in \mathcal{U}.\mathtt{closed}(\mathit{uid}.M, \sigma) \end{array}}{(\mathit{uid}_1.\mathit{init} || \ldots || \mathit{uid}_n.\mathit{init})_{\sigma_0} \Longrightarrow_{\mathtt{load}} \langle \mathcal{F}; \sigma; \vec{k}; \mathit{cid} \rangle}$$

Uber-object-call

$$\frac{\begin{array}{c} \mathcal{F} = F_1 :: \mathcal{F}' \qquad T = \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho, a), T_1 \\ F \Vdash \mathit{sl}.\mathtt{extCall}(\rho) = \langle \mathit{uid}'.f\, \vec{v}, \rho_1 \rangle \qquad \sigma \vDash \mathit{uid}'.f.\mathit{pre}(b) \\ F_1 \Vdash \rho' = \mathit{uid}'.\mathit{lang}.\mathtt{initCore}(\mathit{uid}'.f\, \vec{v}) \qquad \mathit{fresh}(\mathit{tid}') \\ \mathit{uid}' \notin \mathsf{active}(\vec{k}_1) \qquad T' = \mathit{tid}' \mapsto (\mathit{uid}', \mathit{uid}'.f, \mathit{uid}'.\mathit{sl}, F_1, \rho', b), \\ \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho_1, a), T_1 \end{array}}{\langle \mathcal{F}; \sigma; \langle \mathit{cid}, T \rangle, \vec{k}_1; \mathit{cid} \rangle \Longrightarrow^{\mathtt{emp}}_{\mathtt{call}} \langle \mathcal{F}'; \sigma; \langle \mathit{cid}, T' \rangle, \vec{k}_1; \mathit{cid} \rangle}$$

Uber-object-ret

$$\frac{\begin{array}{c} T = \mathit{tid}' \mapsto (\mathit{uid}', \mathit{mainf}', \mathit{sl}', F', \rho', a'), \\ \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho, a), T_1 \\ F' \Vdash \mathit{sl}'.\mathtt{halt}(\sigma, \rho') = \langle v, \rho_1 \rangle \\ \sigma \vDash \mathit{mainf}'.\mathit{post}(a') \qquad F \Vdash \mathit{sl}'.\mathtt{extRet}(\rho, v) = \rho'' \\ T' = \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho'', a), T_1 \end{array}}{\langle \mathcal{F}; \sigma; \langle \mathit{cid}, T \rangle, \vec{k}_1; \mathit{cid} \rangle \Longrightarrow^{\mathtt{emp}}_{\mathtt{ret}} \langle \mathcal{F}; \sigma; \langle \mathit{cid}, T' \rangle, \vec{k}_1; \mathit{cid} \rangle}$$

Wait

$$\frac{\begin{array}{ll} \vec{k} = \langle \mathit{cid}, T \rangle, \vec{k}_1 & T = \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho, a), T_1 \\ F \Vdash \mathit{sl}.\mathtt{extCall}(\rho) = \langle \mathit{uid}'.f\, \vec{v}, \rho_1 \rangle = & \mathit{uid}' \in \mathsf{active}(\vec{k}_1) \end{array}}{\langle \mathcal{F}; \sigma; \vec{k}; \mathit{cid} \rangle \Longrightarrow^{\mathtt{emp}}_{\mathtt{wait}} \langle \mathcal{F}; \sigma; \vec{k}; \mathit{cid} \rangle}$$

Interrupt

$$\frac{\begin{array}{c} \mathcal{F} = F_1 :: \mathcal{F}' \qquad \mathit{uid}' := \mathit{uid}_{\mathit{cidInt}} \\ F_1 \Vdash \mathit{uid}'.\mathit{lang}.\mathtt{initCore}(\mathit{uid}'.\mathit{init}) = \rho' \\ \mathit{fresh}(\mathit{tid}') \qquad T' = \mathit{tid}' \mapsto (\mathit{uid}', \mathit{uid}'.\mathit{init}, \mathit{uid}'.\mathit{sl}, F_1, \rho', _), T \\ \mathit{uid}' \notin \mathsf{active}(\vec{k}_1) \end{array}}{\langle \mathcal{F}; \sigma; \langle \mathit{cid}, T \rangle, \vec{k}_1; \mathit{cid} \rangle \Longrightarrow^{\mathtt{emp}}_{\mathtt{intStart}} \langle \mathcal{F}'; \sigma; \langle \mathit{cid}_{\mathit{int}}, T' \rangle, \vec{k}_1; \mathit{cid} \rangle}$$

Interrupt-ret

$$\frac{\begin{array}{c} \mathit{uid}' := \mathit{uid}_{\mathit{cidInt}} \qquad T = \mathit{tid}' \mapsto (\mathit{uid}', \mathit{mainf}', \mathit{sl}', F', \kappa', _), \\ \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \kappa, a), T_1 \\ F' \Vdash \mathit{sl}'.\mathtt{halt}(\sigma, \rho') = \langle v, \rho_1 \rangle \\ \sigma \vDash \mathit{mainf}'.\mathit{post} \qquad T' = \mathit{tid} \mapsto (\mathit{uid}, \mathit{mainf}, \mathit{sl}, F, \rho, a), T_1 \end{array}}{\langle \mathcal{F}; \sigma; \langle \mathit{cid}_{\mathit{int}}, T \rangle, \vec{k}_1; \mathit{cid} \rangle \Longrightarrow^{\mathtt{emp}}_{\mathtt{intEnd}} \langle \mathcal{F}; \sigma; \langle \mathit{cid}, T' \rangle, \vec{k}_1; \mathit{cid}_{\mathit{int}} \rangle}$$

Fig. 4: Abstract semantics

The execution of a program on a system with n distinct cores and an initial global state memory σ_0, starts with a *configuration*, $\mathcal{C}$, of the form $(\mathit{uid}_1.\mathit{init}||\ldots||\mathit{uid}_n.\mathit{init})_{\sigma_0}$, where σ_0 assigns initial values to the heap locations. A configuration is well-defined iff for all $i \leq n$, $\mathit{uid}_i \in \mathsf{dom}(\mathcal{U})$ and for all $i \neq j \leq n$, $\mathit{uid}_i \neq \mathit{uid}_j$. A well-defined configuration describes a system of überobjects ready to initialize n distinct überobjects on the cores by calling their *init* functions.

The rule Load takes care of this initialization: It (1) initializes the internal core state ρ_i for each uid_i, using the corresponding `initCore` function; (2) starts a new thread on each core; (3) assigns n disjoint freelists from $\mathcal{F}$ to each thread; (4) checks that σ_0 satisfies the precondition of each überobject uid_i's main public API, $\mathit{uid}_i.\mathit{init}.\mathit{pre}(x)$, for an instance a_i instantiating x. We continue running these threads until they return (via one of the rules); at that point, we need to ensure that the post-condition of the thread holds for the exact same instance a_i. Thus, we carry this instance a_i in the thread to use it when checking the post-conditions.

Moreover, the load rule requires each heap compartment to be closed, i.e., $\mathsf{closed}(\mathit{uid}.M, \sigma_0)$, formally defined in Section V-B, states that any pointer stored in an address in the domain of $\mathit{uid}.M$ has to point to another location in $\mathit{uid}.M$. In other words, a heap compartment cannot store a pointer to another heap compartment or a stack frame. This property implies an instance of respecting borders on the heap (see Section IV-A): a function cannot find a pointer on its memory that points to a location not accessible to it.

The Load rule chooses a core identifier $\mathit{cid} \in \mathsf{dom}(\vec{k})$ as the active core. The active core cid can be switched to any other $\mathit{cid}' \in \mathsf{dom}(\vec{k})$ any time using the Switch rule, allowing us to model a preemptive concurrent dynamics.

The rest of the semantic rules, except the last two handling interrupts, are defined based on the internal state of the top thread on the stack of the running core. For example, the Cmd rule is fired when the top thread on the stack of the running core wants to take a local (τ) step. The rule ensures that the multicore state also steps accordingly. Cmd asserts that the read/write footprint of the internal core state only touches the memory locations the thread has exclusive access to.

If the top thread on the stack of the running core calls a public API, the rule Uber-object-call (1) identifies the callee and the arguments passed to it using the function `extCall`, (2) updates the current internal core state to wait for its callee to return, and (3) spawns a new thread on top of the core's stack, initializes its internal core state using the function `initCore`, and assigns a fresh freelist to it. It also checks that the call respects the specifications by ensuring that σ satisfies the pre-condition of the callee for some b.

To avoid complications of implementing locks in our semantics, we simulate them by adding an extra condition on Uber-object-call that allows initialization of a public API of uid' as a callee if uid' does not appear anywhere on the stack of any of the cores in $\vec{k}_1$, i.e. $\mathit{uid}' \notin \mathsf{active}(\vec{k}_1)$. If this criterion is not met, then the überobject has already had the lock on a different core, and thus the caller's thread needs to wait using the Wait rule until the lock is released. Note that the implementation of locks may result in livelock, e.g., two überobjects may mutually wait for each other.

When the top thread on the stack of the active core halts: If there is at least one other thread on the stack waiting for its result, the rule Uber-object-ret uses the function `extRet`

to update the internal core ρ based on the return value v. It also ensures that the post-condition of *mainf'* holds for the argument a'. By construction we know that a' is the same argument that satisfied the precondition of *mainf'*. If there is no other thread waiting on the core, we terminate the core.

Interrupts Interrupts are modelded as an interrupt handler überobject for each core. For the core *cid*, we appoint the interrupt handler überobject uid_{cidInt}. As a überobject, uid_{cidInt} has its own assigned memory location, which consists of the core's interrupt description table and its interrupt flag, describing whether interrupts are allowed for the core or not. uid_{cidInt} has two distinct public APIs: *init*, and *setCtx*. Other überobjects may call the public API function *setCtx* of uid_{cidInt} interrupt handler to change the interrupt flag of the core.

The function *init* (which takes no arguments) checks whether interrupts are available for the core, and if so it calls a public API of an überobject corresponding to the interrupt service routine. Our preemptive model allows *init* to take control of the core at any point in time, assigning the highest priority to the interrupts. As a result, the *init* public API needs to have an always true pre-condition to ensure that it can always be initialized: $uid_{cidInt}.init.pre = \texttt{true}$.

The INTERRUPT rule models the preemptive semantics. To enforce the locks of überobjects, we require the extra premise $uid' \notin \mathsf{active}(\vec{k_1})$. Thus, in our model, an interrupt cannot get interrupted. When the interrupt handler terminates, INTERRUPT-RET hands back the core to the original interrupted thread asserting the post-condition of the interrupt handler's *init* public API. This rule is similar to other return rules, except that we do not update the internal state of the interrupted thread after the return. As a result we need to distinguish this specific return from others such as UBER-OBJECT-RET. We assign a subscript, *int*, to the core *cid*, i.e., cid_{int}, when the interrupt handler of the core initializes (INTERRUPT), and remove it after it returns (INTERRUPT-RET).

V. METATHEORY

We show how our abstract compartmentalization yields good properties. Before delving into the technical details, we first presenting a high-level road map of our formal results.

A. Overview of formal results

Source-level composition of compartments [Section V-B] We prove a source-level modular verification property: properties verified of each überobject in isolation hold on concurrent executions of the whole system. To do so, we formalize the property of respecting the interface (i.e., respecting memory footprint boundaries and specified pre- and post-conditions) as a verifiable predicate and prove that if each überobject respects the interface in isolation, any concurrent run of the whole system respects the interface. In particular, in any concurrent run, when a public API or CASM function returns, the global memory state satisfies the function's post-condition.

The proof technique is via rely-guarantee reasoning. We define an invariant called the core invariant [Def. 5] on multicore states and prove that if each überobject respects the interface in isolation and the initial multicore (prompted by the LOAD rule) satisfies the invariant, then each concurrent step of the multicore in the abstract semantics preserves the invariant [Thm. 1]. The core invariant ensures that a multicore state can always progress according to the abstract semantics [Thm. 2]. The design of our abstract semantics ensures that the progress property is enough for lifting the local properties of überobjects to any concurrent execution. As a corollary of the progress theorem [Cor. 3], in any concurrent run, the caller satisfies the pre-conditions of the callee (e.g., via the fifth premise of the UBER-OBJECT-CALL rule), and the callee satisfies its post-condition upon return (e.g., via the fourth premise of the UBER-OBJECT-RET rule). Moreover, we prove that any concurrent execution is data race free, i.e., no two threads access a location concurrently when at least one of the accesses is a write [Thm. 4].

Source-level secure flow of information [Section V-C] We then show one application of our modular verification, by defining an interface to ensure a noninterference property for our concurrent system. We prove the noninterference result for our system by assigning an integrity label to each memory compartment and imposing an Information flow calling convention on überobjects [Def. 6]. Our noninterference theorem states that given two configurations that agree on the values stored in their high-integrity memory compartments and the implementation of überobjects with exclusive access to those compartments, any concurrent execution of the first configuration can be simulated by a concurrent execution of the second one such that the values stored in the high integrity parts of the memory continue to be the same [Thm. 5].

From source to target level property preservation [Section V-D] Next, we outline the properties required by a compiler to preserve the properties verified at the source-level to the target level. We show that CASCompCert satisfies these properties and can be used for our case studies, which are implemented in Clight and CASM language.

A compiler abstractly consists of a memory transformation function and a code transformation function. We require the memory transformation function of a compiler to be well-defined in the sense that it preserves the disjointedness of the heap compartments from source to target [Def. 7]. A target-level system of überobjects [Def. 8] is obtained by compiling the source überobjects using a compiler with a well-defined memory transformation function. Next, we define interface-preserving property of a compiler that ensures that if a source-level überobject respects its interface then its target-level counterpart also respects the interface [Def. 9]. We show that if each überobject is compiled in isolation by an interface preserving compiler, then the concurrent execution of the target-level system of überobjects enjoys the same properties, i.e., preservation, progress, and data race freedom, as the source-level system of überobjects [Cor. 6]. Finally, we review the definition of sequential compiler correctness introduced by Jiang et al. [25] [Def. 10], and use it to prove that correct sequential compilers are interface preserving [Thm. 7].

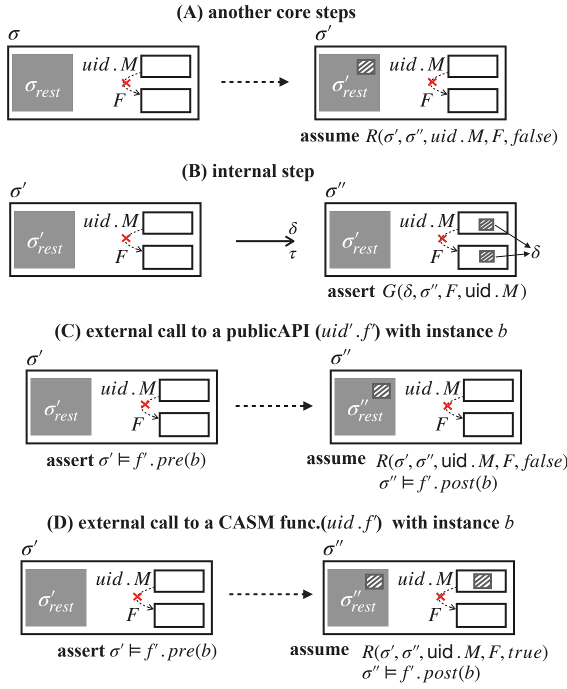

Fig. 5: Respecting the interface

B. Source level composition of compartments

To apply the rely-guarantee reasoning principle of concurrent systems [35], [36], we first formally define the rely and guarantee conditions of each überobject, which we refer to as the local requirements of each überobject.

Local predicates for respecting the interface We begin by defining a few auxiliary predicates on the system configurations. To aid the explanation, we show 4 pairs of memory states in Fig. 5. For each, $uid.M$ and F represent the local heap and freelist of *uid*, respectively. The gray σ_{rest} is the memory of other überobjects. The hashed rectangle illustrates portions of the heap that changes across the pair. The crossed dashed line states that there is no pointer on $uid.M$ pointing to F.

We define the function $\mathtt{isCasm}(f)$ to return true when f is a CASM function and false otherwise. A memory region M is *closed* w.r.t. σ, $\mathbf{closed}(M, \sigma)$, iff it does not store a pointer to any other region in σ than itself: $\forall l, l' {:} Addr.\ if\ l \in M\ and\ l' = \sigma(l),\ then\ l' \in M$.

Next, we define a *rely condition* of a thread stating what portions of two memory states σ and σ' are allowed to store different values. Fig. 5.A illustrates a scenario where the thread *tid* belonging to *uid* temporarily pauses execution when the memory state is σ, allowing other programs to execute, and resumes when σ' is the memory state. The rely condition, written $R(\sigma, \sigma', M, F, \mathit{modified})$, is what *tid* assumes others can modify while it is paused. Here, $M \subseteq dom(\sigma)$ is a portion of the heap (in this scenario, it is *uid*'s local heap $uid.M$) and $F \subseteq dom(\sigma)$ is the stack (freelist). The condition holds iff σ' extends σ but keeps M closed. If the boolean variable *modified* is `false`, then σ and σ' have to agree on the value stored both on the heap with addresses in M and the stack F; otherwise, σ and σ' only need to agree on the values stored in addresses on the stack. If the thread is paused by an external call to another überobject Public API or by a core switch, we set *modified* to be `false` and the rely condition ensures that neither the thread's exclusive heap compartment nor its stack will be touched by the time the thread is resumed (Scenarios A and C of Fig. 5). If the thread is paused by a call to a CASM function of the same object which has access to the same heap compartment, we set *modified* to be `true` and the rely condition only ensures that the thread's stack will be untouched (Fig. 5.D). Formally: $R(\sigma, \sigma', M, F, \mathit{modified})$ iff
$\mathbf{closed}(M, \sigma') \wedge dom(\sigma) \sqsubseteq dom(\sigma') \wedge \forall l \in S, \sigma(l) = \sigma'(l)$
where $S = F$ *when* $\mathit{modified} = \mathtt{true}$; $S = M \cup F$ *otherwise*.

We next define *guarantee condition* of a thread when it evaluates its own code. The guarantee condition $G(\delta, \sigma, F, M)$ holds iff M is closed with respect to σ and the footprint δ only touches the memory regions in F and M, i.e., $\mathbf{closed}(M, \sigma) \wedge dom(\delta) \subseteq F \cup M$. Fig. 5.B illustrates that each internal step of a thread must satisfy the guarantee condition.

We use our rely and guarantee conditions to define what it means for a thread (belonging to überobject *uid*) running on a CPU core to *respect the specifications* of its postcondition Q and its callees' preconditions in the presence of other threads and CPU cores. It is written $(\langle \rho, \sigma \rangle \{Q\})_{F,uid,sl}$, where ρ is the internal state of the current thread, σ is the current memory state, Q is the postcondition of the current thread, F is the set of stack locations of the current thread, and *sl* is the language that the function the current thread is executing is written in.

Definition 1 (Respecting the specs)**.** $(\langle \rho, \sigma \rangle \{Q\})_{F,uid,sl}$ *iff for all* σ' *with* $R(\sigma, \sigma', uid.M, F, \mathtt{false})$*,*

1) *if* $F \Vdash \rho, \sigma' \longrightarrow^{\delta}_{\tau} \rho', \sigma''$ *then* $(\langle \rho', \sigma'' \rangle \{Q\})_{F,uid,sl}$*, and*
2) *if* $F \Vdash sl.\mathtt{halt}(\rho, \sigma') = \langle v, \rho' \rangle$ *then* $\sigma' \vDash Q$*, and*
3) *if* $F \Vdash sl.\mathtt{extCall}(\rho) = \langle f' \vec{v} {:} \tau, \rho_1 \rangle$*, then for all* b *s.t.* $\sigma' \vDash f'.pre(b)$ *and for all* σ'' *s.t.* $R(\sigma', \sigma'', uid.M, F, \mathtt{isCasm}(f'))$ *and all return values* v*, if* $\sigma'' \vDash f'.post(b)$ *then* $(\langle \rho', \sigma'' \rangle \{Q\})_{F,uid,sl}$ *where* $F \Vdash sl.\mathtt{extRet}(\rho_1, v) = \rho'$.

Def. 1 assumes $R(\sigma, \sigma', uid.M, F, \mathtt{false})$, the rely condition, to hold: we can rely on the other cores to keep our version of global state memory σ intact with respect to the locations F and $uid.M$ (Fig. 5.A). Condition 1) states that any step the thread takes results in a new configuration that respects the same specifications. Condition 2) states that right before the thread halts, the global state memory satisfies its postcondition. Condition 3) says, in the case of an external call, the global state memory satisfies the precondition of the callee. Moreover, the callee may alter the global memory state, but the caller can assume that its callee only extends the global state memory according to the agreed-upon boundaries. Scenarios C and D of Fig. 5 illustrate this condition.

Note that Def. 1 does not dictate functions to return and allows them to abort, since aborted programs also satisfy the invariants while it executes. This aligns with the vision of our formalization: to allow coexistence of compartments where some terminate normally and others abort, as long as they preserve each other's specifications (pre- and post-

conditions) in a concurrent run. By verifying the specifications of a function using tools like Frama-C and Verifiable-C, we can ensure the function being respectful with the specs.

Next we define a coinductive property to state that the current thread does not leak references as it executes, written $RM(F, uid, (\rho, \sigma), sl)$, where the arguments have their usual meaning. For all σ' with $R(\sigma, \sigma', uid, F, \texttt{false})$

1) if $F \Vdash (\rho, \sigma') \longrightarrow_{\tau}^{\delta} (\rho', \sigma'')$, then $G(\delta, \sigma'', F, uid.M)$ and $RM(F, uid.M, (\rho', \sigma''), sl)$, and
2) if $F \Vdash sl.\texttt{halt}(\rho, \sigma) = \langle v, \rho' \rangle$ then the return value v is not a pointer, and
3) if $F \Vdash \langle f' \vec{v}, \rho_1 \rangle = sl.\texttt{extCall}(\rho)$ then (a) none of the arguments $\vec{v}$ are a pointer, and (b) for all σ'' with $R(\sigma', \sigma'', uid.M, F, \texttt{isCasm}(f'))$ and any return value v' that is not a pointer, we have $RM(F, uid.M, (\rho', \sigma''), sl)$ where $F \Vdash sl.\texttt{extRet}(\rho_1, v') = \rho'$.

Condition 1) says any step the thread takes asserts the guarantee condition: it does not leak any references and only changes its own heap and stack (Fig. 5.B). Moreover, the step results in a new configuration that has the same property. Condition 2) says if the thread halts, its return value is not a pointer. Condition 3) says, in the case of an external call, the thread does not leak any pointers to the callee. Moreover, the caller can assume that its callee only extends the global memory according to the agreed-upon boundaries (Fig. 5.C,D). The state after the return continues to have the same property.

We define *respecting the boundary* to mean that when a function $f\vec{v}$ is initialized, its execution does not leak any references. We write it as $\mathsf{MemClose}(sl, uid, f\vec{v}, \sigma, F)$, where f is a public API or CASM function belonging to *uid*, implemented in *sl*, and assigned the freelist F.

Definition 2 (Respecting the boundary). *We write* $\mathsf{MemClose}(sl, uid, f\vec{v}, \sigma, F)$ *iff for any internal core state* ρ *if* $F \Vdash sl.\texttt{initCore}(f\vec{v}) = \rho$ *and* $\mathbf{closed}(uid.M, \sigma)$ *then* $RM(F, uid.M, (\rho, \sigma), sl)$.

With Def. 1 and Def. 2, we define what it means for a publicAPI or CASM function declared in an überobject $uid \in \mathcal{U}$ to *respect the interface* as follows.

Definition 3 (Respecting the interface). *A function f defined as a public API or CASM function in an überobject $uid \in \mathcal{U}$ and implemented with language sl,* respects the interface*, iff for all global memory states σ, all non-pointer arguments $\vec{v}$, and any fresh freelist F,*

- *it respects the specifications, i.e.,* $\forall a.$ *if* $\sigma \vDash f.pre(a)$, *then* $(\langle \rho, \sigma \rangle \{f.post(a)\})_{\{F, uid, sl\}}$ *for* $F \Vdash sl.\texttt{initCore}(f\vec{v}) = \rho$ *and*
- *it respects the memory boundaries, i.e.* $\mathsf{memClose}(sl, uid.M, f\vec{v}, \sigma, F)$.

An überobject respects the interface iff all its public APIs and CASM functions respect the interface. A set of überobjects $\mathcal{U}$ respects the interface, if all $uid \in \mathcal{U}$ respect the interface.

Next, we define what it means for a system of überobjects $\mathcal{U}$ to be *valid* with respect to a global memory state σ. This property matches the conditions required by the premises of the LOAD rule in Fig. 4. We will use it to form well-defined configurations that can be successfully initialized by LOAD.

Definition 4 (Valid system of überobjects). *We call a system of überobjects $\mathcal{U}$* valid *w.r.t a global memory state σ iff the precondition of every überobject $uid \in \mathcal{U}$ is satisfiable, and its exclusive memory region is closed w.r.t. σ, i.e., $\forall uid \in dom(\mathcal{U})$, and for some a, $\sigma \vDash uid.\texttt{init}.pre(a)$ and $\mathbf{closed}(uid.M, \sigma)$.*

Global properties for respecting the interface Next, we define an invariant on the threads of a multicore state. It states that for each core: (a) the thread on top of the stack respects the specifications and boundaries, and (b) a thread, *tid*, sitting in the stack waiting for an element on top of it to halt will respect the specifications and boundaries, assuming that its top thread, $t(tid)$, asserts the rely condition when it returns.

Definition 5 (The core invariant). *A multicore state $\langle \mathcal{F}, \sigma, \vec{k}, cid \rangle$ satisfies the core invariant $\mathcal{I}$, iff for each $\langle cid, T \rangle \in \vec{k}$, where $T = tid \mapsto (uid, mainf, sl, F, \rho, a), T'$, we have $\mathbf{top}(\sigma, tid \mapsto (uid, mainf, sl, F, \rho, a))$ and for any $tid' \mapsto (uid', mainf', sl', F', \rho', a') \in T'$ we have $\mathbf{waiting}(T, \sigma, tid \mapsto (uid', mainf', sl', F', \rho', a'))$.*

(a) *$\mathbf{top}(\sigma, tid \mapsto (uid, mainf, sl, F, \rho, a))$ if and only if $(\langle \rho, \sigma \rangle \{mainf.post(a)\})_{F, uid, sl}$ and $RM(F, uid.M, (\rho, \sigma), sl)$.*
(b) *$\mathbf{waiting}(T, \sigma, tid \mapsto (uid, mainf, sl, F, \rho, a))$ iff for all σ' with $R(\sigma, \sigma', uid.M, F, \texttt{isCasm}(t(tid).mainf))$ and all return values v (that is not a pointer), we have if $\sigma' \vDash t(tid).mainf.post(b)$, then $\mathbf{top}(\sigma', tid \mapsto (uid, mainf, sl, F, \rho', a))$, where $\rho' = sl.\texttt{extRet}(\rho, v)$, and $t(tid)$ is the thread on top of tid in T, and b is the instance of the thread tid, i.e., $b = t(tid).a$.*

We prove that when all functions respect the interface, the core invariant is preserved in any concurrent execution. Then, we use this result to show that under the conditions ensuring that a configuration initializes successfully, the core can always progress by taking a step other than SWITCH.

Theorem 1 (Preservation of the invariant). *If a system of überobjects $\mathcal{U}$ respects the interface, then every step of the abstract semantics (Fig. 4) preserves the core invariant.*

Proof. The proof is by case analysis on the rules of Fig. 4. See the extended TR for the complete proof. □

Theorem 2 (Progress). *If a system of überobjects $\mathcal{U}$ respects the interface and is valid w.r.t. a global memory state σ, then we can successfully initialize the well-defined configuration $(uid_1.init|| \ldots ||uid_n.init)_\sigma$, and every core in the compositional concurrent run of the configuration enjoys progress, i.e. every core can either take a step other than* SWITCH *or it terminates (with* TERM, DONE, *or* ABORT*).*

Proof. We prove that the core resulting from a load of such configuration satisfies the core invariant, and the core invariant is enough to ensure progress. See the extended TR. □

The progress property is a strong result; using it, we can guarantee that in any concurrent execution, if a public API or CASM function is ready to return, the global memory state satisfies the function's post-condition.

Corollary 3 (Preservation of the specs globally)**.** *Consider a system of überobjects $\mathcal{U}$ that respects the interface and is valid w.r.t. a global memory state σ. In any concurrent execution of a well-defined configuration $(uid_1.init||\ldots||uid_n.init)_\sigma$, if a thread on a core is ready to halt, then we can establish the post condition of the main function running on the thread.*

Finally, we establish data race freedom: in any concurrent execution, no two threads access a location concurrently when at least one of the accesses is a write.

Theorem 4 (Data race freedom)**.** *Every well-defined configuration is data race free.*

This property is a consequence of our memory compartmentalization and holding locks on each compartment.

C. Source-level secure flow of information

We take results from Section V-B one step further and prove a standard noninterference property. We show that by assigning an integrity label to each heap compartment and establishing a calling convention, we can ensure that the data stored in the low-integrity regions do not influence those with higher (or incomparable) integrity labels.

We define a security lattice $\Psi := \langle \mathcal{L}, \sqsubseteq \rangle$, where $\mathcal{L}$ is a set of integrity labels, denoted by ξ, and $\sqsubseteq$ is a partial order on $\mathcal{L}$. $\xi \sqsubseteq \xi'$ if ξ' has lower or equal integrity as ξ. We assume there is a map from *uid* to its integrity label, an element in $\mathcal{L}$. We sometimes use *uid* to denote the integrity label of *uid*, e.g., we write $\xi \sqsubseteq uid$ to mean that *uid* has a lower or equal integrity than ξ. We enforce the following information flow policy on external calls and returns, which can be enforced statically by locally type checking each compartment:

Definition 6 (IF calling conventions)**.** *Function f with integrity level ξ can externally call an f' of lower or equal integrity ξ', i.e., $\xi \sqsubseteq \xi'$, and freely pass arguments to it. f can only get back a return value from f' if $\xi = \xi'$.*

An überobject adheres to the IF calling convention iff all its public APIs and CASM functions adhere to the IF calling convention. A set of überobjects $\mathcal{U}$ adheres to the IF calling convention, iff all $uid \in \mathcal{U}$ adhere to the IF calling convention.

Our noninterference result states that if two configurations agree upon the high-integrity initial überobjects and the high-integrity memory locations, if one configuration can reach a state, the other one can simulate the execution and the resulting memory states still agree on the high integrity memory locations. To simplify the noninterference statement, we assume that all überobjects terminate and do not abort. The termination assumption is necessary for establishing the simulation. In a non-terminating setting, a low-integrity überobject in the second configuration may be trapped in an internal loop and cannot take the step required for completing the simulation. We can eliminate the assumption by proving termination. The majority of the überobject's functions that implement a TEE are short-running services without unbounded loops or recursion. The programmer can verify their termination either manually or using automated tools, e.g., the termination checker provided by Frama-C.

Theorem 5 (Noninterference)**.** *Consider an interface-respecting system of überobjects $\mathcal{U}$ that adheres to the IF calling convention, and is valid w.r.t. initial global memory states σ_0 and σ'_0. For any integrity level $\xi \in \Psi$, consider two well-defined configurations $(uid_1.init||\ldots||uid_n.init)_{\sigma_0}$, and $(uid'_1.init||\ldots||uid'_n.init)_{\sigma'_0}$, s.t. $\forall uid \in \mathcal{U}. \forall l \in uid.M.$ if $uid \sqsubseteq \xi$, then $\sigma_0(l) = \sigma'_0(l)$, and $\forall i \leq n$, with $uid_i \sqsubseteq \xi$ or $uid'_i \sqsubseteq \xi$, we have $uid_i = uid'_i$. If one configuration reaches a state with global memory σ, then the other one can simulate the execution and reach a state with global memory σ' s.t. $\forall uid \in \mathcal{U}. \forall l \in uid.M.$ if $uid \sqsubseteq \xi$, then $\sigma(l) = \sigma'(l)$.*

D. From source to target level property preservation

Our ultimate goal is to bring the source-level properties: preserving the specs and coarse-grained noninterference introduced in Sections V-B and V-C to the target level. The idea is that we may use a different compiler for each überobject, but we identify a set of requirements that each has to satisfy. We show that these requirements are enough to preserve the property of respecting the interface for each compartment, i.e., if a public API or CASM function of an überobject respects the interface at the source level, its compiled version also respects the interface. Then we use the same abstract semantics of Fig. 4 to compose the compiled compartments for concurrent execution and use the same set of theorems we used for the source level to establish the properties, e.g., progress, of the concurrent compositional run at the target level. The requirements that we identify are orthogonal to the functional correctness requirement of compilers; we are only interested in establishing the spec-preservation and coarse-grained noninterference of the compiled components.

An interface-preserving compiler A compiler is a pair of functions, CT and MT, transforming code and memories from the source to the target level, respectively. The function MT:$Addr_s \hookrightarrow Addr_t$ (partially) maps source-level memory addresses ($Addr_s$) to addresses at the target level ($Addr_t$).

Aligned with the prior work on certified compositional compilers, we require MT to map all locations on the source-level heap to a location on the target-level heap. We also require this mapping to be injective on the heap locations. This requirement allows us to map each compartment on the heap to a corresponding compartment $\mathsf{MT}(uid.M)$ and preserve the disjointness of the heap compartments. We call this property *well-definedness* of memory transformations and formalize it below. We require that different compilers used for each compartment agree on their memory mapping MT. We fix this common memory transformation MT for the rest of this section. We refer to source and target level heaps as $\texttt{heap}_s$ and $\texttt{heap}_t$, respectively.

Definition 7 (Well-defined memory transformation)**.** *A memory transformation* $\mathsf{MT} : Addr_s \hookrightarrow Addr_t$ *is well-defined iff*

1) *it is total on* $\mathtt{heap}_s$*, i.e., its domain includes the source-level heap locations,* $\mathtt{heap}_s \subseteq dom(\mathsf{MT})$*,*
2) *restriction of* MT *to* $\mathtt{heap}_s$*, i.e.* $\mathsf{MT}\lceil_{\mathtt{heap}_s}$*, is injective. Where* $\mathsf{MT}\lceil_{\mathtt{heap}_s}$:$\mathtt{heap}_s \hookrightarrow Addr_t$ *is the same as* MT *but with a restricted domain.*
3) *each source-level heap location is mapped to a target-level heap location, i.e.,* $\mathsf{MT}(\mathtt{heap}_s) = \mathtt{heap}_t$*. where* $\mathsf{MT}(\mathtt{heap}_s)$ *is the set of all locations* $l \in Addr_t$ *with* $l' = \mathsf{MT}(l)$ *for some* $l \in Addr_s$*.*

For each überobject $uid \in \mathcal{U}$ in the source level, we define a corresponding überobject uid_t in the target-level. uid_t owns the exclusive memory location $\mathsf{MT}(uid.M)$: the mapping of *uid*'s exclusive memory to the target level. It has the same set of function declarations with their code being translated to the target language via the code transformation mapping CT. For example, for a public API $f(\overrightarrow{x{:}\tau}) : \tau' = gcmd$ declared in *uid*, we add $f(\overrightarrow{x{:}\tau}) : \tau' = \mathsf{CT}(gcmd)$ to the public API declarations of uid_t. The CASM functions are translated to the assembly language by an identity code transformation on their original assembly implementation.

The definition of the interface for public APIs and CASM functions remains intact in the compilation process. At the target level, a function's specifications (i.e., its pre- and post-conditions) are defined over the target heap (rather than the source heap). To build the target level specifications, we substitute any occurrence of source heap locations in the source level specifications with its corresponding target heap location. For example, the source-level interface $\{P(x)\}uid.f\{Q(x)\}$ maps to the target-level interface $\{P_t(x)\}uid_t.f_t\{Q_t(x)\}$, where $P_t = [\mathsf{MT}(uid.M)/uid.M]P$. We use Δ_t for the set of target-level specifications derived from Δ.

Definition 8 (Target-level system of überobjects)**.** *Given a source-level system of überobjects* $\mathcal{U}$*, we build a target-level system* $\mathcal{U}_t$*, such that* $uid \mapsto \ddot{u}obj \in \mathcal{U}$ *iff* $uid_t \mapsto \ddot{u}obj' \in \mathcal{U}_t$*. Where for üobj as* $(sl, ulock, M, init, \overrightarrow{pubAPI}, \overrightarrow{casmfd}, \overrightarrow{fd})$*, üobj′ is* $(tl, ulock, \mathsf{MT}(M), init_t, \overrightarrow{pubAPI}_t, \overrightarrow{casmfd}_t, \overrightarrow{fd}_t)$*. sl is the source-level language of uid and tl is the target language.*

Each target-level überobject inherits its integrity level and lock from its source-level counterpart. Recall that the lock is a conceptual lock, modeled by the WAIT rule in the semantics. The rest of the target-level überobject definitions are carried over from the source level similarly. We use a subscript t on the target-level entities to distinguish them from their source-level counterparts. A configuration in the target-level runs concurrently using the same semantic rules in Fig. 4. With the minor exception that in the last premise of the CASM-CALL, we put the language of the new thread to `ASM` instead of `CASM`.

We now define the notion: an *interface-preserving* compiler.

Definition 9 (Interface-preserving compiler)**.** *A compiler is interface-preserving iff if* $\mathcal{U}$ *respects the interface* Δ*, then* $\mathcal{U}_t$ *respects the interface* Δ_t *and moreover every control-flow transfer due to external calls and returns across different überobjects at the target level has a corresponding external call or return at the source level.*

$$\begin{array}{cccc} F \Vdash_{sl} \sigma_s, \rho_s & \xrightarrow[\tau]{\delta}{}^{*} & \sigma_s', \rho_s' & \begin{array}{c} G(\delta, \sigma_s', F, uid.M) \\ \text{and } \sigma_s' \vDash f_s.post_s(a) \end{array} \\ \wedge\!| & & \wedge\!| & \text{iff} \\ F \Vdash_{tl} \sigma_t, \rho_t & \xrightarrow[\tau]{\delta_t}{}^{*} & \sigma_t', \rho_t' & \begin{array}{c} G(\mathsf{MT}(\delta), \sigma_t', F_t, uid_t.\mathsf{MT}(M)) \\ \text{and } \sigma_t' \vDash f_t.post_t(a) \end{array} \end{array}$$

Fig. 6: Module local simulation (the halt case)

The second condition ensures that if each $uid \in \mathcal{U}$ satisfies the Information flow calling convention, then $uid_t \in \mathcal{U}$ satisfies the information flow calling convention too. It is straightforward that the system of überobjects each compiled with an interface-preserving compiler preserves the global spec-preservation and noninterference properties. We can apply the same theorems proved in Section V-B to the target-level system of überobjects as they also preserve the interface.

Corollary 6. *Assume that each uid* $\in \mathcal{U}$ *is compiled with an interface-preserving compiler. If each uid* $\in \mathcal{U}$ *respects the interface, then every target-level configuration enjoys the global spec-preservation and the coarse-grained noninterference properties.*

Next, we show that the sequential compiler correctness from CASCompCert [25] implies that the compiler is interface-preserving as long as the target language is deterministic.

Review of CASCompCert's [25] definitions We provide a high-level description of CASCompCert's definitions (Definitions 2 and 3 [25]) and adapt them to our syntax. Now we fix the target language of all compartments to be x86-assembly to match that of CASCompcert's.

Jiang et al. [25] define their sequential compiler correctness to prove that compositional and concurrent program compilation preserves the semantics of whole programs. The definition is based on a module-local simulation that is compositional and preserves read and write footprints, and uses an invariant $\mathtt{Inv}(\sigma, \sigma_t)$ on global memory states σ and σ_t. The invariant $\mathtt{Inv}(\sigma, \sigma_t)$ states that the source and target-level memory states agree on the values stored in their corresponding addresses: $\mathtt{Inv}(\sigma, \sigma_t)$ iff for all addresses l, l_t if $l \in dom(\sigma)$ and $\mathsf{MT}(l) = l_t$ then $l_t \in dom(\sigma_t)$. Moreover, if $\sigma(l) = v$ and v is not a memory address, then $\sigma_t(l_t) = v$, and if $\sigma(l) = l'$, with l' being an address, then $\mathsf{MT}(l') = \sigma_t(l_t)$. As long as this invariant holds and no function stores a stack pointer on a heap at both source and target levels, well-definedness of MT implies that for a first-order specification P defined over heap locations and its target-level counterpart P_t, we have $\sigma \vDash P$ iff $\sigma_t \vDash P_t$. An adaptation of CASCompCert's sequential compiler correctness in our notation is as follows:

Definition 10 (Sequential compiler correctness)**.** *A sequential compiler* $\langle \mathsf{CT}, \mathsf{MT} \rangle$ *from the source language sl to target language tl is correct, if successful initialization of each function* $f\,\vec{v}$ *to core* ρ *implies the successful initialization of its target level counterpart* $f_t\,\vec{v}_t$ *to core* ρ_t*. Moreover, for any global memory states* $\sigma : \mathtt{heap}_s \hookrightarrow val$ *and* $\sigma_t : \mathtt{heap}_t \hookrightarrow val$ *that are closed w.r.t. the heap locations, and satisfy the invariant*

#1	Respecting the specs (Def. 1) a function satisfies its pre- and post-conditions
#2	Respecting the boundary (Def. 2) [a.] each function has its own local stack. [b.] no pointer to a stack location is stored on the heap. [c.] no pointer is sent via arguments/return values of an external call. [d.] a pubAPI or CASM function writes/reads from its stack and the assigned heap. [e.] no pointer to an unallocated location on the heap.

Fig. 7: Requirements for überobject interface respecting

$\texttt{Inv}(\sigma, \sigma_t)$ *on global memory states, there is a module-local simulation from* $\langle \rho, \sigma \rangle$ *to* $\langle \rho_t, \sigma_t \rangle$*, i.e.* $\langle \rho, \sigma \rangle \leq \langle \rho_t, \sigma_t \rangle$*.*

The module-local simulation $\langle \rho, \sigma \rangle \leq \langle \rho_t, \sigma_t \rangle$ (illustrated in Fig. 6) is defined inductively based on the structure of the source-level internal core state ρ and an index on the number of local computation steps. The main idea of its definition is as follows: if the internal core state at the source level (ρ) calls an external function or halts, satisfying its guarantees while relying on others to handle its heap properly, then the target-level core (ρ_t) eventually, after none or some τ-steps, can take a corresponding action, i.e., calling the same external function or halt. The target internal core state assumes a similar rely condition stating that others will handle its memory state safely and asserts similar guarantees. The formulation of the module-local simulation ensures that at the point where both source and target internal core states halt or call an external function, the invariant on their global memory state holds and their footprints on the heap match.

CASCompCert: an interface-preserving compiler Our goal is to prove that if a function declared in the source überobject respects the interface and the sequential compiler is correct, then the compiled target überobject also respects the interface. The main step in the proof is to form a backward module-local simulation from the target to the source-level states. We need to show that if the target level takes a step and shall rely on some properties to assert the required guarantees, then the source level core (has already done or) will eventually take a similar step with an equivalent rely-guarantee conditions. In our setting, where we assume that the target language is deterministic, the backward simulation can be deduced from a forward simulation from the source to the target-level states [37]. We also show that the rely-guarantee conditions on the source and target are equivalent since the simulation establishes the invariant $\texttt{Inv}(\sigma, \sigma_t)$ on global memory states.

Theorem 7. *Correct sequential compilers as defined in [25], are interface-preserving.*

A corollary of Thm. 7 is that if the überobjects are written in C language, we can compile our public APIs with the sequentially correct compiler implemented in CAScompCert, and the CASM functions with the identity compiler, to preserve the coarse and fine-grained noninterference.

VI. Case Studies

In this section we present two case studies that show instances of verified TEEs on two different hardware architectures/platforms (x86 and ARM), where properties proved on the source can carry over to the binary. These case studies demonstrate the generality of our formal framework.

A. *uberXMHF TEE*

Our first case-study is uberXMHF (üXMHF), an open-source, microhypervisor TEE for the x86 32-bit hardware virtualized platform verified for its guest memory separation properties at the source level [21]. We first give an overview of üXMHF; then we discuss an example source-level property of guest memory separation; finally, we discuss how the assumptions required by our framework are satisfied by the verification conditions used in verifying the memory separation properties of üXMHF, which allows us to bring source-level properties to the compiled binary code.

1) Overview: üXMHF uses hardware virtualization and runs a Ubuntu 12.04 32-bit multicore guest OS with the micro-hypervisor executing at the highest privilege level. It has been used to develop a wide variety of security applications [38]–[45]. üXMHF is built using the überobject abstraction (see Section II-B) which consists of: (a) a set of verified micro-hypervisor core logic üobjects and (b) a set of verified micro-hypervisor extensions. Together, these verified üobjects set up an execution environment for an untrusted OS that is separated from the hypervisor via hardware virtualization. üXMHF has been verified for the security property of guest memory separation which means that the guest OS cannot directly access hypervisor memory regions [21]. During verification, üXMHF Assembly language code is replaced by a C99 hardware model and together with all the verified üobjects are analyzed via Frama-C to enforce the aforementioned guest memory separation property (not full functional correctness).

2) Page-table setup and memory separation: Fig. 8 shows a code snippet of a verified üobj that sets up the unverified guest OS memory separation page tables. Text in green are Frama-C ACSL *requires-assign-ensure* clauses, asserting which variables can be written to (assign) and what constraints these variables must satisfy. Together, lines 4-10 specifies that micro-hypervisor memory regions are inaccessible to the guest OS by asserting page table entries' permissions bit are set correctly. Then *loop invariant* (lines 17-21) defines data structure invariants: the page table is always populated and the memory protection flags are always set (support function in line 30 obtains the memory protection of the memory address which is aliased into a ghost variable `g_flags` in line 28 for verification) such that the untrusted guest cannot access hypervisor protected memory regions.

3) Discharging interface respecting requirements: Fig. 7 summarizes the requirements of respecting the interface required by our überobject architecture on the source code (Def. 3). Requirement #1 is satisfied by üXMHF because pre- and post- conditions of every function (such as the one shown in Fig. 8) are checked by *deductive verification* via Frama-C's

```
1  //@ghost uint64_t gflags[PAE_PTRS_PER_PDPT *
       PAE_PTRS_PER_PDT * PAE_PTRS_PER_PT];
2  /*@
3  ...
4  assigns gp_vhslabmempgtbl_lvl1t[0..(PAE_PTRS_PER_PDPT *
       PAE_PTRS_PER_PDT * PAE_PTRS_PER_PT)-1];
5  assigns gflags[0..(PAE_PTRS_PER_PDPT * PAE_PTRS_PER_PDT
       * PAE_PTRS_PER_PT)-1];
6  ...
7  ensures (\forall uint32_t x;
8    0 <=x< (PAE_PTRS_PER_PDPT * PAE_PTRS_PER_PDT *
         PAE_PTRS_PER_PT) ==>
9    gp_vhslabmempgtbl_lvl1t[x] ==
10   (((x*PAGE_SIZE_4K) & 0x7FFFFFFFFFFFF000ULL) | gflags[x]))
11 @*/
12 void gp_s2_setupmpgtblv(void){
13  uint32_t i, spatype=0, slabid =
        XMHFGEEC_SLAB_GEEC_PRIME;
14  uint64_t flags=0;
15
16  //pt setup
17  /*@ loop invariant a5: 0 <= i <= (PAE_PTRS_PER_PDPT *
       PAE_PTRS_PER_PDT * PAE_PTRS_PER_PT);
18  loop assigns gflags[0..(PAE_PTRS_PER_PDPT *
       PAE_PTRS_PER_PDT * PAE_PTRS_PER_PT)], spatype,
       flags, i, gp_vhslabmempgtbl_lvl1t[0..(
       PAE_PTRS_PER_PDPT * PAE_PTRS_PER_PDT *
       PAE_PTRS_PER_PT)];
19  loop invariant a6: \forall integer x; 0 <= x < i ==> ( (
       uint64_t)gp_vhslabmempgtbl_lvl1t[x]) == ( ((
       uint64_t)(x * PAGE_SIZE_4K) & 0x7FFFFFFFFFFFF000ULL
        ) | (uint64_t)(gflags[x]) );
20  loop variant (PAE_PTRS_PER_PDPT * PAE_PTRS_PER_PDT *
       PAE_PTRS_PER_PT) - i;
21  @*/
22  for(i=0; i < (PAE_PTRS_PER_PDPT * PAE_PTRS_PER_PDT *
       PAE_PTRS_PER_PT); ++i){
23   spatype =  gp_s2_setupmpgtbl_getspatype(slabid,
24          (uint32_t)(i * PAGE_SIZE_4K));
25
26   flags = gp_s2_setupmpgtblv_getflags(slabid,
27          (uint32_t)(i * PAGE_SIZE_4K), spatype);
28   //@ghost gflags[i] = flags;
29
30   gp_vhslabmempgtbl_lvl1t[i] =
31        pae_make_pte( (i*PAGE_SIZE_4K), flags);
32  }
33 }
```

Fig. 8: Verified üobj page-table setup.

WP plugin and value analysis. #2[a.] is satisfied via Frama-C's value analysis checks for memory separation of üobjects and function local stack in combination with the property described previously in §VI-A2 that guarantees page tables are correctly initialized. Further, assertions on the source ensures memory reads and writes are within üobj's local memory and that no pointer to a stack location is stored on the heap (#2[b.]). When a verified üobj writes to the stack (e.g., via Assembly language), a corresponding hardware model callback function is invoked during verification which contains an assertion in the body of that function to check whether the stack-pointer register has a valid address (within the prescribed stack frame; #2[d.]). We will show example assertions for #2[d.] in the next case study. Frama-C's AST analysis is used to ensure that function formal arguments cannot be pointer types (#2[c.]). Finally, #2[e.] is satisfied by üXMHF since neither dynamic memory allocation nor function pointers exists in the source. This is verified via Frama-C's Abstract Syntax Tree (AST) analysis. Moreover, Frama-C's value analysis on üXMHF proves the absence of invalid pointer dereferencing.

```
1      void main(void){
2       ...
3       g_sframe_index ++
4       hwm_cpu_gprs_r13 = (unsigned int)(&
             g_sframe[g_sframe_index].return_addr);
5       casm_init_secure_monitor();
6       g_sframe_index --
7      //@assert (hwm_cpu_gprs_CPSR & __MON_MOD) ==
           __MON_MOD;
8       ...
9
10     }
11
12     void casm_init_secure_monitor(){
13         _impl_hwm_mem_read_write_check =
               casm_read_write_check
14         ...
15         __casm_stmfd_r0_r4();
16         //@assert (hwm_cpu_gprs_r13 >=
               &g_sframe[g_sframe_index].local_params[0])
               && hwm_cpu_gprs_r13 <=
               &g_sframe[g_sframe_index].return_addr
17         ...
18     }
```

Fig. 9: Secure-world hardware memory separation setup.

B. Trustzone TEE

Our second case-study is a light-weight open-source Trustzone TEE [46] (hereon referred to as TZSMC) running on the ARM 32-bit platform on the Freescale iMX53 embedded board [47]. Verifying the guest memory separation property of TZSMC follows the same process as that of üXMHF. The main difference is that instead of x86, we work with ARM and TrustZone. Thus, we show an example of verifying requirement #2.[d] in the context of ARM architecture.

1) Overview of TZSMC: TZSMC employs ARM Trustzone secure-world partitioning to run a simple 32-bit guest OS with the TZSMC components executing at the higest privilege level [46]. TZSMC runs on the Freescale iMX53 embedded board which houses an ARM Cortex-A7 processor with Trustzone hardware secure memory compartmentalization. TZSMC can run secure world service call routines corresponding to secure functionality (e.g., key management, password management etc.) and provides these as APIs to the untrusted guest OS via a Secure Monitor Call (SMC) CPU instruction. TZSMC is not formally verified in its original incarnation. However, for our case study we developed a verified version of TZSMC using an ARM C99 hardware model that we wrote and the Frama-C verification framework.

2) Secure-world memory separation: TZSMC initially boots up in ARM secure world and does the required setup before establishing Trustzone hardware memory separation for the guest OS. This setup involves switching to the Trustzone *monitor* mode and then switching to the non-secure world using a control register, prior to transferring control to the guest OS. Fig. 9 shows a code snippet of our verified TZSMC implementation `main` function that invokes a supporting function to initialize the monitor mode and switches to non-secure world before executing the guest OS. As seen from the figure our verification consists of hardware modeling statements and ACSL *assertions* (line 7) that allow us to ensure that the monitor mode is set after the function call and before transferring control to the guest.

3) Discharging interface respecting requirements: Fig. 7 summarizes the assumptions required by our formalism on the source code (Def. 3). Requirement #1 is satisfied by TZSMC because all assertions (such as the one shown in Fig. 9) are checked by *abstract interpretation* via Frama-C's value analysis. Requirement #2 is satisfied via a combination of strategies. Assumptions #2[c.] and #2[e.] are satisfied via Frama-C's AST analysis and value analysis in a similar fashion to that described previously in §VI-A3. To satisfy assumptions #2[a.], #2[b.], and #2[d.], we model a function call stack and automatically generate code (highlighted in Fig. 9) for verification. Our hardware modeling also aids in ensuring correct stack frame preservation (via verification variables `g_sframe` and `g_sframe_index`) during execution and across function calls (lines 3, 4, 6, and 16). Before a function call we have a function stack prologue (lines 3-4) that prescribes space for the current function call stack and set the address for the stack-pointer register in our hardware model. For every CASM instruction inside the CASM function, we check that the memory reads and writes are restricted to the current call stack and global variables and that no stack location is stored on the heap; this is achieved via the assertions inside the callback function in line 13 (more in the TR). Furthermore, after each CASM instruction, an additional assertion is placed to check if the stack-pointer register has a valid address within the prescribed stack frame (assertion in line 16).

VII. Related work

We discuss closely related work in (1) verified OSes, kernels, and TEEs, which share the goal of producing high assurance mission critical software; (2) assembly analysis and verification tools, which share the goal of analyzing assembly code; and (3) certified compilers, which help us preserve source semantic to bring the verified guarantees to assembly.

Verified OSes, Kernels, TEEs SeL4 [16] is one of the first fully-verified functionally correct kernels. Initially, the guarantees only hold on the C implementation and the correctness of inline assembly and the compiler are assumed. A later paper showed that the guarantees can be proven for the compiled SeL4 via compiler validation [48]. Similar to ours, the bi-simulation relation generated for compiler validation is used to show property preservation. We assume the existence of a certified compiler and formally show that specific kind of security properties can be shown to hold on the compiled code by leveraging properties of the certified compiler. We could swap out the certified compiler in our tool chain by a compiler validation step [49], [50], as long as the bi-simulation relation automatically extracted during the compiler validation process could also bring the properties that we care about down to the compiled code. Many fully-verified kernels directly model and reason about assembly [15], [51]–[55], cutting out the need for a trusted compiler, which we discuss later in this section.

While most of the above mentioned projects aim for proving functional correctness, we only aim for memory separation and a number of security-related assertions. Beyond functional correctness, information flow properties have also been proven for SeL4, CertiKOS, and a separation kernel for ARMv7 [52], [54], [56]. Our goal is similar to theirs: proving noninterference between different domains/partitions/compartments and low-level timing leaks are out of scope.

Our überobjects architecture extends überSpark [21], where the notion of überobjects is first introduced as the building blocks of the überSpark framework with the goal of verifying memory integrity of the überSpark hypervisor [20]. They design several system and programming invariants specifically for überSpark architecture and überobject's C and CASM functions. The invariants are defined to ensure that each überobject can only access its own memory and are verified using Frama-C for each überobject in the überSpark implementation. They further prove that the invariants, with their compositional nature, hold throughout the sequential execution of a überSpark hypervisor. While this prior work on überSpark speculated that the properties can be shown to hold on compiled code via certified compilers, no formal treatment was given there. This work has the following specific formal contributions with regard to this prior work [21].

First, we provide a more general model for überobjects as units of memory compartmentalization and formalize respecting the interface by each überobject as a verifiable local predicate. Our model liberates the überobject definition from a specific programming language and architecture (software and hardware) and allows it to serve as a generic abstraction for memory separation in a *concurrent* setting.

Second, we introduce a detailed abstract semantics in the style of the linking semantics of compositional CompCert [31] and CASCompCert [25]. Our semantics supports concurrency and is particularly designed to model and reason about interrupt handling, allow the execution in the context of multiple CPU cores, and can be connected to certified compilers. In contrast, the semantics from prior work do not provide instruction level semantics; it only describes steps concerning the concurrent operations required to run a multi-core unverified guest OS. It supports neither interrupt-handler nor concurrent execution on multiple CPU cores.

Third, we additionally ensure information flow security between different compartments by assigning an integrity level to each überobject and enforcing a calling convention on them.

Finally, we establish a condition of individual compilers, i.e., the interface-preserving property, which ensures the source-level properties, such as respecting the interface and noninterference, also hold on the target level. Our model permits each source compartment to be compiled separately by its own compiler. Prior work [21] does not reason about linking multi-module source programs and composing compiled compartments at the target level.

Assembly analysis and verification One common approach for projects that verify properties of mixtures of C and assembly or prove the assembly code correct is to rely on a formal model of the assembly. The assembly model is either encoded in a theorem prover [52], [54], or in a hoare-style verification framework like Bedrock [57], BoogiePL [51] or Vale [58], [59], where verification conditions are discharged

automatically by an SMT solver like Z3 [60]. Our tool chain allows developers to directly interact with the C-level analysis tools, not assembly-level or another high-level language like F*, C#, or Dafny [61]. Lifting assembly model to a DSL enables the use of the same C verification tool for analyzing inline assembly, which is similar to the approach that TINA [62] took. Our model is minimal, compared to a realistic model for x86 [63], as our design is driven by the case studies and our needs to mainly check for memory separation.

TINA [62] and RUSTINA [64] lift assembly to an intermediary representation for analysis. RUSTINA specializes in looking for inconsistencies of interfaces of inline assembly (e.g., register clobbering). We could leverage the TINA tool chain for checking the properties that we care about, instead of using our own DSL. However, our CASM DSL allows reasoning about hardware and machine specific registers such as control registers, widely used in low-level system software such as TEEs. Further, our DSL is much lighter-weight than TINA and integrates with existing C verification tools to enable proving properties over hardware and device states.

Certified compilers To establish properties on compiled code, we assume bi-simulation is set up between the source and the target by a certified compiler. As we aim to deal with assembly and forms of concurrency, CASCompCert [25] is a convenient target to show for property preservation. Similar to CASCompcert we introduce an abstract semantics to compose individual überobjects; we also use the semantics for linking both source-level and target-level überobjects. Our semantics, similar to CASCompCert, is concurrent but also supports interrupts and captures pre- and post-conditions of each function in the concurrent execution. We require a few assumptions on our source-level modules too. In contrast to [25], data race freedom of the source execution is not an assumption in our setting but a corollary of our compartmentalized memory model and the assumptions on each individual uberobject. Moreover, as discussed in Section VI, our source-level assumptions are discarded in our case study via off-the-shelf verification tools.

Other certified compilers such as full-abstraction, or those that preserve classes of security properties [65]–[67], could be used to preserve our properties. Our approach targets TEEs using hardware architectures such as x86 and ARM hardware isolation primitives [12], [68], that ensure runtime preservation of the separation in the presence of attackers.

Compartmentalization As pointed out by prior work, compartmentalized systems have the benefit of being able to preserve security properties in the presence of attackers [69]–[71]. We additionally show that the compartmentalization makes reasoning about concurrency much easier, which is a challenging feature to support and is still lacking in many of the existing verified systems. We postulate that if we only compile some compartments to the target level and allow a foreign target-level implementation for the other compartments, we can still prove the spec-preservation and noninterference for the system, as long as the foreign compartments respect the interface.

VIII. Conclusion

We define a formal model of compartmentalization for implementing TEEs. We show that compartmentalization allows us to achieve compositional verification results at the source level and enables us to leverage certified compilers to preserve the guarantees at the target level. We demonstrate via two case studies that security properties verified using our compartmentalization model at the source level on two existing open-source TEEs running on x86 and ARM platforms, hold at the binary level if the code is compiled by CASCompCert.

Acknowledgement. Many thanks to the anonymous reviewers and shepherd for their constructive feedback.

References

[1] A. Fitzek, F. Achleitner, J. Winter, and D. Hein, “The andix research os-arm trustzone meets industrial control systems security,” in *Proc. of INDIN*, 2015, pp. 88–93.
[2] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, “vTZ: Virtualizing ARM TrustZone,” in *USENIX Security*, 2017, pp. 541–556.
[3] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, “Assured: Architecture for secure software update of realistic embedded devices,” *IEEE TCAD*, vol. 37, no. 11, pp. 2290–2300, 2018.
[4] P. Sparks, “The route to a trillion devices,” https://community.arm.com/arm-community-blogs/b/internet-of-things-blog/posts/white-paper-the-route-to-a-trillion-devices, 2017.
[5] S. Fei, Z. Yan, W. Ding, and H. Xie, “Security vulnerabilities of SGX and countermeasures: A survey,” *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021.
[6] Qualcomm Technologies, Inc., “Snapdragon mobile platform - snapdragon security,” 2019. [Online]. Available: https://www.qualcomm.com/snapdragon/security
[7] Nvidia, “Trusted Little Kernel (TLK) for Tegra: FOSS Edition,” 2015. [Online]. Available: http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=blob_plain;f=documentation/Tegra_BSP_for_Android_TLK_FOSS_Reference.pdf
[8] Linaro Limited, “OP-TEE,” 2019. [Online]. Available: https://github.com/OP-TEE/
[9] Huawei Technologies CO., LTD., “EMUI 8.0 Security Technical White Paper,” 2017. [Online]. Available: https://consumer-img.huawei.com/content/dam/huawei-cbg-site/en/mkt/legal/privacy-policy/EMUI%208.0%20Security%20Technology%20White%20Paper.pdf
[10] Trustonic, “Mobile device security is hard - Trustonic makes it easy,” 2019. [Online]. Available: https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp/
[11] Android, “Trusty TEE,” 2019. [Online]. Available: https://source.android.com/security/trusty/
[12] L. ARM, “Arm security technology-building a secure system using trustzone technology,” PRD-GENC-C. ARM Ltd. Apr.(cit. on p.), Tech. Rep, Tech. Rep., 2009.
[13] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, “SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems,” in *IEEE S&P*, May 2020.
[14] C. DeLozier, R. Eisenberg, S. Nagarakatte, P.-M. Osera, M. M. Martin, and S. Zdancewic, “Ironclad C++: A library-augmented type-safe subset of C++,” in *Proc. of OOPSLA*, 2013.
[15] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, “Ironclad apps: End-to-End security via automated Full-System verification,” in *OSDI-USENIX*, 2014.
[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “sel4: Formal verification of an os kernel,” in *ACM SOSP*, 2009.
[17] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, “Comprehensive formal verification of an OS microkernel,” *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.
[18] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo, “Deep specifications and certified abstraction layers,” in *Proc. of POPL*, 2015.

[19] G. C. Hunt and J. R. Larus, “Singularity: Rethinking the software stack,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, Apr. 2007.

[20] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, “Design, implementation and verification of an eXtensible and Modular Hypervisor Framework,” in *IEEE S&P*, 2013.

[21] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, “überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor,” in *USENIX Security*, 2016.

[22] M. Ammar, B. Crispo, B. Jacobs, D. Hughes, and W. Daniels, “Suv—the security microvisor: A formally-verified software-based security architecture for the internet of things,” *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 5, pp. 885–901, 2019.

[23] T. Vörtler, B. Höckner, P. Hofstedt, and T. Klotz, “Formal verification of software for the contiki operating system considering interrupts,” in *IEEE DDECS*, 2015, pp. 295–298.

[24] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C.-K. Hur, “Compcertm: Compcert with c-assembly linking and lightweight modular verification,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, December 2019.

[25] H. Jiang, H. Liang, S. Xiao, J. Zha, and X. Feng, “Towards certified separate compilation for concurrent programs,” in *Proc of PLDI*, 2019.

[26] F. Derakhshan, Z. Zhang, A. Vasudevan, and L. Jia, “Technical report: Towards end-to-end verified TEEs via interface conformance and certified compilers,” Carnegie Mellon University, Tech. Rep., Jan. 2023.

[27] “Keystone – an open framework for architecting tees,” 2022. [Online]. Available: https://keystone-enclave.org/

[28] RISC-V, “RISC-V Open Source Supervisor Binary Interface (OpenSBI),” 2022. [Online]. Available: https://github.com/riscv-software-src/opensbi

[29] X. Leroy, “Formal certification of a compiler back-end or: programming a compiler with a proof assistant,” in *Proc. of POPL*, 2006, pp. 42–54.

[30] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis, “Lightweight verification of separate compilation,” in *Proc. of POPL*, 2016.

[31] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel, “Compositional compcert,” in *Proc. of POPL*, 2015, pp. 275–287.

[32] L. Beringer and A. W. Appel, “Abstraction and subsumption in modular verification of C programs,” in *Proc. of FM*, 2019.

[33] “The memory management glossary.” [Online]. Available: https://www.memorymanagement.org/glossary/f.html#free.list

[34] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, “Vst-floyd: A separation logic tool to verify correctness of C programs,” *Journal of Automated Reasoning*, vol. 61, no. 1, pp. 367–422, 2018.

[35] C. B. Jones., “Specification and design of (parallel) programst,” in *IFIP Congress*, 1983, pp. 321–332.

[36] V. Vafeiadis, “Modular fine-grained concurrency verification,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-726, Jul. 2008. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf

[37] X. Leroy, “A formally verified compiler back-end,” *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[38] S. Echeverría, G. Lewis, C. Mazzotta, C. Grabowski, K. O’Meara, A. Vasudevan, M. Novakouski, M. McCormack, and V. Sekar, “KalKi: a software-defined IoT security platform,” in *Proc. of WF-IoT*, 2020.

[39] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, “Lockdown: Towards a safe and practical architecture for security applications on commodity platforms,” in *Proc of TRUST*. Springer, 2012, pp. 34–54.

[40] A. Vasudevan, N. Qu, and A. Perrig, “XTRec: Secure real-time execution trace recording on commodity platforms,” in *Proc. of HICSS-44*. IEEE, 2011, pp. 1–10.

[41] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, “TrustVisor: Efficient TCB reduction and attestation,” in *IEEE S&P*, 2010, pp. 143–158.

[42] Z. Zhou, M. Yu, and V. D. Gligor, “Dancing with giants: Wimpy kernels for on-demand isolated I/O,” in *IEEE S&P*, 2014, pp. 308–323.

[43] Z. Zhou, J. Han, Y.-H. Lin, A. Perrig, and V. Gligor, “KISS:“key it simple and secure” corporate key management,” in *Proc. of TRUST*. Springer, 2013, pp. 1–18.

[44] M. McCormack, A. Vasudevan, G. Liu, S. Echeverría, K. O’Meara, G. A. Lewis, and V. Sekar, “Towards an architecture for trusted edge iot security gateways,” in *USENIX, HotEdge*, 2020.

[45] S. Echeverría, G. A. Lewis, C. Mazzotta, K. O’Meara, K. Williams, M. Novakouski, A. Vasudevan, M. McCormack, and V. Sekar, “KalKi++: A scalable and extensible iot security platform,” in *Proc. of WF-IoT*. IEEE, 2021, pp. 368–373.

[46] Dongli Zhang, “Trustzone secure and normal world transition tee,” 2022. [Online]. Available: https://github.com/finallyjustice/imx53qsb-code/tree/master/trustzone-smc

[47] “i.mx53 quick start board,” 2022. [Online]. Available: https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/i-mx53-quick-start-board:IMX53QSB

[48] T. A. L. Sewell, M. O. Myreen, and G. Klein, “Translation validation for a verified os kernel,” in *Proc. of PLDI*, 2013.

[49] G. C. Necula, “Translation validation for an optimizing compiler,” in *Proc. of PLDI*, 2000.

[50] J.-B. Tristan, P. Govereau, and G. Morrisett, “Evaluating value-graph translation validation for LLVM,” in *Proc. of PLDI*, 2011.

[51] J. Yang and C. Hawblitzel, “Safe to the last instruction: Automated verification of a type-safe operating system,” in *Proc. of PLDI*, 2010.

[52] D. Costanzo, Z. Shao, and R. Gu, “End-to-end verification of information-flow security for C and assembly programs,” in *Proc. of PLDI*, 2016.

[53] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, “CertiKOS: An extensible architecture for building certified concurrent OS kernels,” in *Proc. of USENIX OSDI*, Nov. 2016.

[54] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, “Formal verification of information flow security for a simple arm-based separation kernel,” in *Proc. of ACM CCS*, 2013, pp. 223–234.

[55] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, “Verifying security invariants in ExpressOS,” in *Proc. of ASPLOS*, 2013.

[56] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, “Sel4: From general purpose to a proof of information flow enforcement,” in *IEEE S&P*, 2013, pp. 415–429.

[57] A. Chlipala, “The bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier,” in *Proc. of ICFP*, 2013.

[58] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, “A verified, efficient embedding of a verifiable assembly language,” *Proc. of POPL*, vol. 3, Jan. 2019.

[59] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, “Vale: Verifying high-performance cryptographic assembly code,” in *USENIX Security*, 2017.

[60] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Proc. of TACAS*, 2008, pp. 337–340.

[61] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Proc. of LPAR*, 2010, pp. 348–370.

[62] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M.-L. Potet, “Get rid of inline assembly through verification-oriented lifting,” in *Proc. of ASE*, 2019.

[63] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, “A complete formal semantics of x86-64 user-level instruction set architecture,” in *Proc. of PLDI*, 2019, pp. 1133–1148.

[64] F. Recoules, S. Bardin, R. Bonichon, M. Lemerre, L. Mounier, and M.-L. Potet, “Interface compliance of inline assembly: Automatically check, patch and refine,” in *Proc. of ICSE*, 2021.

[65] M. Patrignani and D. Garg, “Robustly safe compilation, an efficient form of secure compilation,” *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 1, feb 2021.

[66] C. Abate, R. Blanco, c. Ciobâcă, A. Durier, D. Garg, C. Hriţcu, M. Patrignani, E. Tanter, and J. Thibault, “An extended account of trace-relating compiler correctness and secure compilation,” *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 4, nov 2021.

[67] M. Patrignani and D. Garg, “Secure compilation and hyperproperty preservation,” in *Proc. of CSF*, 2017, pp. 392–404.

[68] VMware, “Software and hardware techniques for x86 virtualization,” Tech. Rep., 2022. [Online]. Available: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/software_hardware_tech_x86_virt.pdf

[69] A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens, “Capableptrs: Securely compiling partial programs using the pointers-as-capabilities principle,” in *Proc. of CSF*, 2021.

[70] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hritcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach, “When good components go bad: Formally secure compilation despite dynamic compromise,” in *Proc. of CCS*, 2018.

[71] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, “A design and verification methodology for secure isolated regions,” in *Proc. of PLDI*, 2016.