ÜBERSPARK[†]: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor

Amit Vasudevan[*], Sagar Chaki[**], Petros Maniatis[***], Limin Jia[****] and Anupam Datta[****]

[*]amitvasudevan@acm.org – CyLab/Carnegie Mellon University
[**]chaki@sei.cmu.edu – SEI/Carnegie Mellon University
[***]maniatis@gmail.com – Google Inc.
[****]{liminjia,danupam}@cmu.edu – CS/ECE Carnegie Mellon University

***Abstract*—We present überSpark (üSpark), an innovative architecture for compositional verification of security properties of extensible hypervisors written in C and Assembly. üSpark comprises two key ideas: (i) endowing low-level system software with abstractions found in higher-level languages (e.g., objects, interfaces, function-call semantics for implementations of interfaces, access control on interfaces, concurrency and serialization) and enforcing these "verifiable-object" abstractions using a combination of commodity hardware mechanisms and light-weight static analysis; and (ii) interfacing with platform hardware by programming in Assembly using an idiomatic style (called CASM) that is verifiable via tools aimed at C, while retaining its performance and low-level access to hardware. After verification, the C code is compiled using a certified compiler while the CASM code is translated into its corresponding Assembly instructions. Collectively, these innovations enable compositional verification of security invariants without sacrificing performance. We validate üSpark by building and verifying security invariants of an existing open-source commodity x86 micro-hypervisor and several of its extensions, and demonstrating only minor performance overhead, and low verification costs.**

1. INTRODUCTION

The modern hypervisor stack is, by necessity, extensible. Hypervisors not only enable the old hat style of customization, such as modularity for device drivers, but are further extended with convenient functionality for security services such as attestation, debugging, tracing, application-level integrity and confidentiality, trustworthy resource accounting, on-demand I/O isolation, trusted path, and authorization [14], [18], [22], [49], [53], [57], [62], [64], [65], [71], [73], [74], [76], [79], [82]–[85]. Further, the overwhelming majority of deployed hypervisor codebase is written in low-level C and Assembly, due to hardware accesses, developer familiarity, and performance requirements.

[†]In the fictional Transformers universe, the AllSpark is a powerful object capable of creating a new Transformer by bestowing ordinary machinery with sparks – the building blocks of a Transformer. In a similar vein, ÜBERSPARK bestows ordinary hypervisors with verifiable objects (ÜOBJECT) for automated compositional security analysis.

1.1. Problem – The unbridled growth of these extensible hypervisors, while enabling useful functionality, raises significant security concerns. As the size and complexity of these systems increase – not to mention the number of extensions, which may be active in arbitrary combinations – so has the incidence of security-related bugs. Indeed exploitable bugs in extension interfaces have led to compromises in various hypervisors ranging from complex VMMs to micro-hypervisors [2], [3], [26], [27], [44]. Thus, higher assurance in the security properties offered by hypervisors is critically important.

1.2. Solution – We address this challenge by developing überSpark (or üSpark), an architecture for building extensible hypervisors that is: (a) *compatible with commodity systems*; (b) enables *automated compositional verification of security properties*; and (c) produces *performant systems*. Compatibility with commodity systems is crucial to impact developers and deployment ecosystems. üSpark supports development and verification directly at the C and Assembly source-level and enables access to more commodity hardware features. It is thus distinct from prior approaches that sacrifice commodity-compatibility by employing new programming languages or hardware models [33], [36], [80]. Compositionality means that extensible systems can be verified modularly, rapidly, and independently as they are implemented. Specifically, when an extension is added, üSpark does not require complete system re-verification to reestablish properties. While this goal guides much work in high-level languages, achieving it for low-level languages is a significant challenge. Furthermore, it distinguishes us from verification of full functional correctness [31], [33], [43]. We focus only on security invariants – memory separation, control-flow integrity, information flow – and other extension properties that can be formulated as invariants. We verify such properties directly, compositionally, and automatically on the (C and Assembly) implementation. This helps bring to commodity-compatible hypervisors those on-going approaches that offer full functional correctness while enabling precise reasoning on untrusted and unverified system code. Finally, a üSpark hypervisor's performance is close to that of a commodity unverified system.

Key to the power of üSpark is the enforcement of verifiable-object abstractions to hypervisors. The basic building block is a *üobject*, which encapsulates specific system resources and provides an interface – with a well-defined behavioral contract (comprising a use manifest along with formal behavior specifications) – for accessing them. A üobject may represent core components of a hypervisor or an extension and may be concurrent or sequential. Public methods of concurrent üobjects are invoked in parallel by multiple cores whereas sequential üobjects are implemented as monitors, guarding all method invocations via a per-üobject lock. üObjects communicate with each other via function calls.

There are two special üobjects: *prime* sets up a sane initial state, while *sentinel* ensures control-flow semantics even when üobjects with different levels of privilege and trust invoke each other. Together, they enable compositional inductive proofs of security properties expressed as invariants over sequential üobjects via source code analysis and hardware assumptions (see Barnett et al. [8]). A third group of special üAPI üobjects enable access to shared resources. This design enables state-of-the-art tools for automatic verification of sequential C code to be soundly applied to verifying security properties, while still allowing multi-threaded high-performance applications.

In keeping with our first and second design goals, üSpark enforces verifiable-object abstractions using a combination of commodity hardware mechanisms (page-tables and de-privileging) and light-weight static analysis, leveraging off-the-shelf C99 source-code analysis and certified compilation tools. üObjects, including prime and sentinel, are automatically and modularly verified using Frama-C [41], an industrial-strength software analysis and verification framework. We use standard and custom Frama-C plug-ins to perform static verification checks that include: per-üobject behavioral contracts (via a standard weakest-precondition plug-in); abstract variable assertions that enable behavioral asserts as well as üobject control-flow integrity (via a standard abstract-interpretation plug-in on stack frames and other variables); syntactic checks that ensure conformance with a restricted C99 syntax and logical de-privileging of üobjects (via a standard abstract syntax tree analysis plug-in); and, composition checks that enable client üobjects that share a common server üobject to compose soundly (via a custom composition-check plug-in that analyses use manifests).

üSpark also provides an idiomatic use of Assembly, called CASM, to separate it from C code during system construction. During analysis with Frama-C, the CASM code is replaced by a C99 hardware model which models key commodity hardware features. Our custom Frama-C plug-in checks that the syntactic restrictions imposed by CASM are respected by every üobject. The verified üobjects are then compiled into executable binaries. During üobject compilation, all C99 code is processed using the certified CompCert compiler [12] while each CASM instruction is replaced by the corresponding Assembly instruction by our custom Frama-C plugin. The CASM language is designed to ensure that the C and Assembly code operate on disjoint state. Our longer-term goals are to guarantee the semantic equivalence between the hardware model and the corresponding Assembly instructions as well as ensure that verified source code properties carry over to the binary by leveraging the C-Assembly separation to cleanly extend the bisimulation proof of the CompCert compiler to encompass hardware state and Assembly code. Formal proofs of these guarantees, however, are beyond the scope of this paper.

The üSpark object abstraction is distinguished from other systems in that it allows many fine-grained objects in privileged mode. Static analysis enforces logical de-privileging of those objects – e.g., a hypervisor module running in host-mode ring 0 is precluded from accessing page-table structures, thereby being "logically" deprivileged – while control transfer between them does not involve a context switch, thereby significantly helping with system performance, our third design goal.

1.3. Contributions – (a) We present üSpark, an innovative architecture providing verifiable object abstractions for automated compositional verification of hypervisor security properties while targeting commodity compatibility and performance (§4,§5). (b) We use üSpark to incrementally develop and verify security properties of an existing open-source commodity x86 micro-hypervisor with multiple independent security extensions (hypervisor and extensions realized as 11 üobjects with 7001 SLoC; 5544 and 2079 lines of annotations and hardware model; §6,§7). (c) We carry out a comprehensive evaluation showcasing verification metrics, development effort and performance, and report on our experience (1 person yr; üobject verification times from 1–23 minutes with a cumulative time $\approx$ 1hr; 2% avg. runtime overhead over native micro-hypervisor applications with guest performance unaffected; §8,§9).

2. A Motivating Example

We use as a running example, a hypervisor that closely corresponds to our case study, to motivate and explain üSpark. Imagine the hypervisor managing a multi-CPU guest, and supporting optional security extensions that implement various guest-specific and system-wide security properties. The hypervisor manages system devices used by itself, by extensions, and by the guest. System devices execute device firmware in parallel with the

CPUs and perform DMA to/from main memory. The hypervisor and extensions are written in C and assembly.

The hypervisor leverages CPU capabilities, such as memory-mapped I/O (MMIO) and legacy I/O, for system-to-device interaction; it initializes boot CPU (BSP) state; it sets up memory page tables, as well as device allocations and DMA protections (e.g., via an IOMMU); it initializes multi-CPU support via the Local Advanced Programmable Interrupt Controller (LAPIC) and activates processors and sets up their memory page tables and appropriate protections. Constructing a verified hypervisor of this sort, the developers must not only build it and test it well, but also verify its code against a set of general safety properties (e.g., memory integrity) as well as functional invariants on hardware and software state (e.g., IOMMU, LAPIC, CPU states).

Consider now adding two new *verified* extensions to the hypervisor: **`hyperdep`**, which ensures that guest-VM data pages are non-executable; and (b) **`sysclog`**, which ensures that every system call issued by the guest is logged via a dedicated network card to an external trusted entity on the network. In order to preserve the verified status of the system, the developers must prove that: (a) memory integrity is not violated by the extensions; (b) each extension provides its claimed property to guests configured to use it; and (c) the extensions are used in tandem by a guest if and only if they provide a well-defined compositional property (e.g., separability). This is non-trivial, since it requires the construction and verification of inductive invariants that imply the core security properties of the hypervisor, and those of enabled extensions. Also, since extensions are optional, verification must account for all possible configurations – e.g., enabling either **`hyperdep`**, or **`sysclog`**, or both – while avoiding the combinatorial blowup.

Of course, history tells us that two extensions are never enough for any extensible system. What is more, not all extensions come from the same developers or with the same pedigree. Consider, for instance, an *unverified*, strictly optional extension to the hypervisor; this might be an extension that provides essential functionality, but has not been verified, and is taken as an acceptable risk. For our example, let us use **`aprvexec`**, an extension that ensures that guest code pages contain only read-only, whitelisted content. As with **`hyperdep`** and **`sysclog`**, core hypervisor properties, and the properties of other extensions should not be violated by running **`aprvexec`**, and the risk of running **`aprvexec`** should only be suffered by a guest that explicitly enables it and relies on its presumed properties. Note that the guest itself, unless it is verified as rigorously as the rest of the hypervisor, is such an unverified component in the system.

3. Goals and Assumptions

3.1. Goals – Our overarching goal is to enable development of performant extensible hypervisors offering proofs of wide-ranging properties on their code, including low-level memory safety, control-flow guarantees, and information flow, as well as higher-level properties such as trusted network logging (**`sysclog`**) and data execution prevention (**`hyperdep`**), going all the way up to security properties spanning both hardware and software states (IOMMU, LAPIC, network-card and CPU). Also, verification must support properties over shared system states: e.g., both **`hyperdep`** and **`sysclog`** manipulate guest memory protections via the same guest page-tables. Our design goals fall broadly in three categories.

3.1.1. Compositionality: When new components are added, or existing components changed, human re-verification effort should be limited to the changed codebase, yet it should provide guarantees about the entire system under all possible configurations.

3.1.2. Legacy Compatibility & Usability: Our development and verification approach must integrate into the existing hypervisor C and Assembly language programming ecosystem, and cover the entire source code base including commodity hardware and guest OS. We must support extensions that are unverified in order to preserve the legacy ecosystem. However, unverified code (e.g., the guest) must not violate system properties established by verified code. Our development and verification techniques must foster wider adoption by hypervisor developers. We envision that entry-level developers will rely on basic building blocks to provide simple properties while seasoned developers will harness the full verification power to provide stronger guarantees.

3.1.3. Performance: Verification must not preclude aggressive code optimizations for individual components, including extensions, and must not adversely affect runtime performance. Further, commodity guest OS with multi-core hardware must be supported.

3.2. Non-goals – We do not aim for full functional correctness (i.e., verifying that the implementation behaves exactly as specified in a high-level abstraction). This separates the concerns of showing how a complex low-level system achieves low-level formal properties from how those low-level properties refine a high-level abstract model; we focus on the former, since it is a hard and as yet open problem, whereas several on-going work tackles the latter [31], [42].

3.3. Attacker Model and Assumptions – We assume that the attacker does not have physical access to the CPU, memory, chipset or other verified extension-specific system devices (our hardware TCB). Other system devices, the guest OS, and unverified extensions are under the attacker's control. This is reasonable since a majority of today's attacks are mounted by malicious software or untrusted system devices. We assume that our hardware TCB is functionally correct, and we have load-time integrity, i.e., the verified hypervisor is the one securely loaded onto the hardware at boot time. Finally, we assume that the verification tools we use are sound.

4. üSpark Architecture

We next describe our architecture, and how it addresses our goals (§3.1) via verifiable object abstractions (Fig 1)

4.1. üObjects – The basic building block in üSpark– the "üobject" – is used to contain any system component including verified and unverified hypervisor and guest blobs and system devices. Logically, a üobject is a singleton object guarding some otherwise indivisible resources (e.g., registers, memory, devices) and implementing public methods to access them. Public methods are essentially regular function signatures but can be restricted to specific callers (§4.2.1). Every üobject also has a special public method, $init$, to set up the üobject in a safe initial state. A üobject may be concurrent or sequential. The public methods of a concurrent üobject can be invoked in parallel on multiple cores. In contrast, at most one core can invoke the methods of a sequential üobject at a time, as with a traditional monitor. When multiple cores are active, sequential execution is enforced via per-üobject locks.

Each üobject defines its functionality using C and Assembly and is compiled to its binary. Assembly language for a verified üobject is written using CASM, a dialect of C in which Assembly instructions are encoded within regular C functions (CASM functions) via C-like pseudo-function calls (CASM instructions[1]). For example, for the x86 instruction `movcr3` involving register `eax` there is a corresponding CASM pseudo-function called **`ci_movl_eax_cr3`**. Each CASM instruction pseudo-function is defined in the üSpark hardware model (§7.1.2) and bridges the shift between the reference C semantics and the hardware instructions (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. CASM functions are verified to respect the C application binary interface (ABI), which is crucial for the soundness of verification. During compilation, all C functions are processed via a certified compiler while each CASM instruction is replaced by the corresponding Assembly instruction. In contrast to prior code-level verification approaches (§10), CASM supports two-way nested C to Assembly calling with full device modeling. This allows using various verification techniques to prove (higher-level) properties on device states other than just memory and numeric safety (§7.2). CASM also allows aggressive compiler optimizations of the callee C functions including inlining as per compiler specifications, resulting in optimal runtime performance (§8.3). We envision further optimizations including inlining of hand-written CASM code as part of our future work (§11.2).

[1]CASM syntax is similar to existing "asm" keywords supported by traditional C compilers for integrating Assembly language instructions. However, CASM provides a more principled way to integrate Assembly instructions tailored for verification while retaining performance.

Beyond defining its own functionality, a üobject is also accompanied by a behavior *contract*. This consists of a *use manifest* (§4.3) and a formal *behavior specification* of its own public interface, which guarantee that if a certain assumption is satisfied in how a public method is invoked, then a property on the return values is guaranteed to hold upon return of that method, without mention of internal üobject state.

Every üobject is held to a number of invariants, which together guarantee its adherence to the verifiable object abstraction. These invariants include memory and (internal) source-lvel control-flow integrity, so that the code can be reasoned about; and satisfaction of the formal contract, so that the contract alone may overapproximate the üobject, thereby enabling compositional verification; as well as correct initialization. The invariants are discharged via assumptions on the hardware and proofs on the source code of the üobject, and on the contract of üobjects it interacts with (§5, §7.2).

While our use of object encapsulation is similar to existing micro-kernel architectures [42] and prior capability systems [32], [63], üSpark is distinguished by privileged disaggregation, i.e., multiple verified privileged üobjects can be logically deprivileged. This enables us to achieve the sweet spot with both high performance (there is no hardware de-privileging overhead; §8.3.1) and compositional verification (privileged üobjects can be verified seperately; §7.2).

4.1.1. Prime: is the first üobject to execute in a üSpark enabled hypervisor. Prime is verified to satisfy its contract which is: to set up the required system interfaces and associated policies, establish operating stacks, prepare the platform CPU cores, invoke the $init$ methods of other üobjects to initialize their state, and kick-start üobject interactions.

4.2. üObject Interaction – A üobject interacts with another by invoking a public method in its interface with appropriate parameters. All verified üobjects operate on a single stack (one per CPU core) that is set up initially by the prime. Each unverified üobject uses its own, separate stack. Verifiable object abstraction requires üobject-to-üobject control-flow integrity (otherwise returns could land at arbitrary üobject program sites, access controls would be violated, etc.). Therefore, üobjects must also be verified to use their stack correctly (another invariant). For unverified üobjects, that also means that stacks must be switched to/from the unverified üobject stack and a

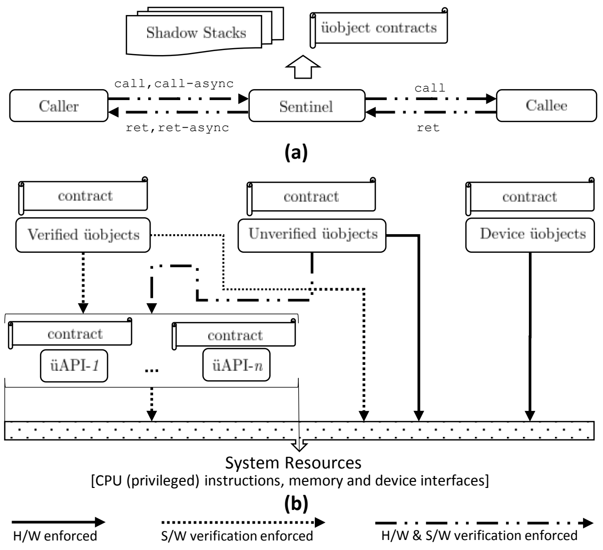


Fig. 1: überSpark: enforces verifiable object abstractions using a combination of commodity hardware and software verification mechanisms to: (a) translate synchronous (call) and asynchronous (e.g., exceptions, intercepts) inter-üobject control transfers, to establish pure function call-return semantics; and (b) establish üobject resource confinement.

separate shadow stack must be maintained for storing return addresses during control transfers. The special *sentinel* üobject performs (verifiably) this functionality.

4.2.1. Sentinel: is a special üobject that mediates interactions among other üobjects. Thus, an invocation of a public method of a callee üobject by a caller üobject is intercepted by the sentinel and dispatched only after a number of optional runtime checks have succeeded.

These runtime checks logically ensure that the caller may invoke a given public method on the callee according to the üobject manifest (§4.3). For example, an extension can be split between a top half and a bottom half as with traditional device drivers (in our case study, **`sysclog`** could be split into a portion **`sysclognw`** sending log entries via the network, and one that collects and annotates intercepted system calls), ensuring that only the top half may invoke the bottom half at runtime, while still keeping the two isolated from each other and independently verifiable. If caller and callee are both verified, then no runtime check is required, since static analysis enforces the call policy (§4.3). If one is unverified, the sentinel consults the policy dynamically and allows or rejects the call accordingly.

Besides the runtime checks, the sentinel is responsible for transfering control among üobjects. If both are verified, the control transfer is just a function call. But if either is unverified, the sentinel must employ the appropriate control-transfer method for the isolation mechanism imposed on the unverified üobject (e.g., if using ring-based isolation, switch privilege levels, marshal call arguments, etc.). The sentinel may implement control transfers according to a number of concrete ways (hardware virtual machines, software fault isolation, etc.), while still adhering to the high-level invariant for isolation. For example, in our micro-hypervisor implementation, the sentinel traverses both ring-based isolated üobjects, and hardware virtual machines (§6).

The sentinel is an üobject, so it adheres to the same invariants as regular üobjects, but it is also verified to implement its function correctly (perform the checks, properly transfer control, etc.)

4.3. üObject Resource Confinement – üSpark implements *üobject resource confinement* in which distinct system resources are: (a) managed by designated üobjects, (b) protected from access by unauthorized üobjects, and (c) regulated in their use by authorized client üobjects. Such resources include üobject local memory (code, data, stack), system memory (e.g., BIOS data, free memory), CPU state and privileged instructions, system devices and I/O regions. Every üobject includes a *use manifest* in its contract that describes which resources it must access (Appendix B). It is held to the property that it can only use the resources declared in its manifest.

For verified üobjects, üSpark employs a hardware model identifying CPU interfaces to system resources (e.g., I/O and designated memory instructions interface to system devices, instructions that can modify CPU model specific register states etc.) and static analysis to ensure that access to those interfaces respect the üobject's manifest (§7). E.g., **`sysclog`**'s manifest shows that it must access the dedicated network card for its remote logging, and static analysis ensures that the code for **`sysclog`** may access only that network card, nor can any other üobject access **`sysclog`**'s network card.

In contrast, unverified üobjects are held to their use manifests via more direct enforcement mechanisms, such as hardware MMU and privilege protections (virtualization, de-privileging) and software manipulations (e.g., SFI). Unverified üobjects can also be granted direct access to exclusively held system devices so they can perform I/O without any performance overhead (e.g., a guest OS üobject is allocated all the devices except the LAPIC and **`sysclog`**'s network card). Device üobjects use DMA as their interface to other üobjects. üSpark uses hardware IOMMU capabilities to ensure that device üobjects are restricted to perform DMA only to designated üobject DMA memory regions.

4.3.1. üAPI üobjects: are a special set of üobjects that encapsulate shared resources over which system properties are established (§6.4). For example, guest OS üobject memory and CPU state are manipulated by multiple extensions (**`hyperdep`** and **`sysclog`**). üSpark enforces a composition check (§7.2.1), which for a given set of üAPI üobjects checks if a set of "client" üobjects are composable. Note that every üAPI üobject also performs composability checks at runtime for invocations from unverified üobjects. Such composability checks reason about the use-manifest portion of a client üobject's contract, which constrains how that üobject invokes the üAPI's public methods, ensuring some system-specific and üAPI-specific composability guarantee, such as separability. Client üobjects must satisfy the property that whenever they invoke a üAPI call, they obey their own use manifest, and üSpark discharges this property via static analysis on verified üobjects or runtime sentinel checks for unverified üobjects.

4.4. üSpark Blueprint – üSpark also defines a hypervisor *blueprint* (üBP) which a hypervisor implementation is held to. The üBP is a high-level control-flow graph that divides hypervisor execution into three phases: startup, intercept and exception handling which can in turn be customized based on the actual number of system üobjects and their interactions (Figure 2; §6). The üBP along with our high-level proofs (§5) enables us to abstract the hypervisor, running on multi-core platform hardware with system devices and DMA, as a non-deterministic sequential program. This, in turn, allows us to prove invariant properties of üobjects, and the hypervisor as a whole, via sequential source code verification. Further, the üBP also enforces that fragile bits of the hardware state (e.g., CPU and IOMMU) are only touched within a monitor. This, allows us to prove invariant properties encompassing hardware states and keeps our hardware model simple by precluding modeling of concurrent hardware accesses (§7.1.2).

5. üSpark Formalism

This section presents a formalization of üSpark that justifies the soundness of our analysis. For brevity, we first give an overview of the formal reasoning followed by our high-level verification approach and related theorems. Full proof details can be found in Appendix C.

5.1. üSpark Formalism Overview – üSpark reasoning relies foundationally on a set of invariants – properties that must hold throughout the execution of a üSpark hypervisor. The invariants are divided into üSpark system invariants and üSpark general programming invariants (those that pertain specifically to üobject C and CASM functions). Each invariant is proved by reducing it further to a set of *proof-assumptions on hardware* (PAHs) and *proof-obligations on code* (POCs) using the üSpark blueprint (üBP; Fig. 2). POCs are then discharged on all üSpark verified üobjects including the prime and sentinel using specific verification tools and techniques (§7). A hypervisor implementation is compliant with üSpark– and therefore amenable to compositional reasoning – if it satisfies all the üSpark invariants. Full details of invariant-to-PAH/POC mappings, a one-time effort, is described in Appendix D. At a high level, üSpark invariants ensure the hypervisor implementation follows the üBP and that prime is correct, and the first to start in the system, and that it sets up memory protections, stacks, and CPU cores, before starting other execution contexts in a well-defined state. The remaining invariants guarantee that üobjects have memory and control-flow integrity, and the sentinel properly transfers control among them, respecting the concurrent/sequential designation.

5.2. Verification Approach and Theorems – There are two tasks in verifying properties of a üSpark hypervisor: (a) showing that it obeys the üSpark invariants; and (b) showing that it obeys any hypervisor/extension-specific invariant properties. The benefit of (a) is that developers can express system-specific properties in terms of üobjects and their interactions with each other, yet verify those properties separately on each individual üobject in isolation, and on the ensemble of the behavior contracts of all üobjects, without having to perform slow verification of the combined source code for the whole code base.

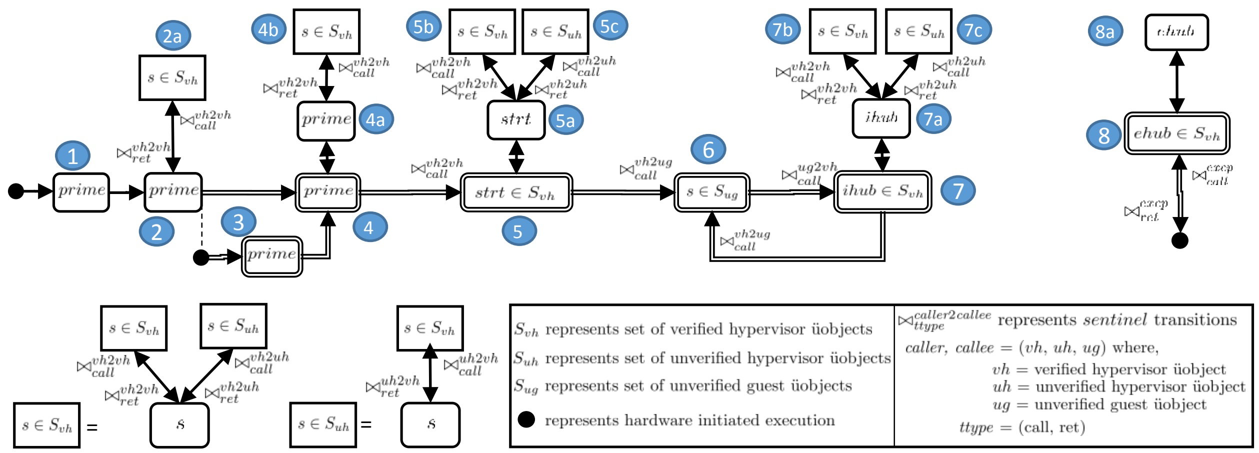
Crucial to the model of üobject are CASM programs, defined below. First, we define a *CASM function* as a CompCert-C99 (CC99) function whose body consists only of a block of Assembly instructions that respect the CC99 ABI. A üobject CASM program is a CC99 program such that: (i) all Assembly code appears only in CASM functions; and (ii) these CASM functions preserve the caller C functions' CPU register state.

Given a üobject CASM program, we are interested in verifying two kinds of properties: (1) invariant properties: whether φ holds at every state (after every instruction), and (2) individual state assertions: whether φ holds at specific program points. We can also specify assumptions (i.e., preconditions), stating that we assume φ holds when a function is called. Verification tools such as Frama-C (§7) take programs annotated with properties to be checked and decide whether the properties hold on all execution traces of the program. We begin with two üSpark theorems essential for the correctness of our approach, which follow directly from the üSpark programming invariants (Appendix C).

Theorem 1 (DisjointCASM)**.** *The union of üobject CASM and C functions preserve the existing semantic preservation property of the certified compiler.*

Theorem 2 (ExitSentinel)**.** *üobject execution can only exit via the sentinel.*

The next theorem states that each üSpark execution is an interleaving of properly nested executions of üobjects, one on each core (a more formal definition can be found in Appendix C). Intuitively, it means that üobject calls and returns are properly nested except that the return of an unverified üobject can be an exception, as an unverified üobject can lie about its return address, but



S_{vh} represents set of verified hypervisor üobjects
S_{uh} represents set of unverified hypervisor üobjects
S_{ug} represents set of unverified guest üobjects
● represents hardware initiated execution
$\bowtie^{caller2callee}_{ttype}$ represents *sentinel* transitions
caller, callee = (*vh, uh, ug*) where,
vh = verified hypervisor üobject
uh = unverified hypervisor üobject
ug = unverified guest üobject
ttype = (call, ret)
prime
$s \in S_{vh}$
$s \in S_{uh}$
$s \in S_{ug}$
strt
$strt \in S_{vh}$
ihub
$ihub \in S_{vh}$
ehub
$ehub \in S_{vh}$
$\bowtie^{vh2vh}_{call}$
$\bowtie^{vh2vh}_{ret}$
$\bowtie^{vh2uh}_{call}$
$\bowtie^{vh2uh}_{ret}$
$\bowtie^{vh2ug}_{call}$
$\bowtie^{ug2vh}_{call}$
$\bowtie^{excp}_{call}$
$\bowtie^{excp}_{ret}$
1
2
2a
3
4
4a
4b
5
5a
5b
5c
6
7
7a
7b
7c
8
8a

Fig. 2: üSpark Hypervisor Blueprint: startup, intercept and exception handling execution phases. *Rounded boxes = üobjects; Square boxes = nested üobject calls; Arrows = intra- and inter-üobject transitions; Single-lines = serialized execution; Double-lines = concurrent execution.*

will be caught by the hardware if it steps out of the üobject memory. This theorem enables us to view üSpark semantically as a concurrent object-oriented program, which is then abstracted as a non-deterministic sequential program for verification.

Theorem 3 (NestedCall). *Consider a legal execution π of üSpark and a sequential üobject s. The projection of π on executions of s consists of a sequence of properly nested executions of s, each on a specific core.*

5.2.1. Hardware Model and Converting Assembly to C: We use C verification tools to verify CASM functions in üobjects by converting Assembly to C. In addition to general-purpose registers (which are preserved to respect the CC99 ABI) these Assembly instructions access special hardware registers (e.g., LAPIC). Let us denote the set of registers accessed by CASM functions in üSpark by $\mathcal{R}_{hw}$. We introduce a set of fresh C variables (denoted $\mathcal{V}_{hw}$), one for each register; replace each Assembly instruction accessing $\mathcal{R}_{hw}$ by one or more CC99 statements that operate in a semantically equivalent way over $\mathcal{V}_{hw}$; replace each $r \in \mathcal{V}_{hw}$ with v_r in assertions used for specifying hardware state during verification. We refer to the mapping between $\mathcal{R}_{hw}$ and $\mathcal{V}_{hw}$, and the induced mapping from Assembly instructions to CC99 statements, as our *hardware model*. We assume that this mapping is correct. We refer to the CC99 function obtained by transforming a CASM function f in this manner as $\widetilde{f}$.

5.2.2. Abstract üSpark: We abstract üobjects as a non-deterministic CC99 (NDCC99) program, i.e., a CC99 program with non-deterministic selection of values from finite sets. In particular, the abstract üSpark üBP consists of a set of abstract üobjects, where each abstract üobject $\widetilde{s}$ is obtained from the corresponding concrete üobject s by converting each function $g \in p(s)$ to an abstract function $\widetilde{g}$; more concretely: by replacing all CASM functions as described above, replacing accesses to data that other cores and devices can modify by non-deterministic values, replacing a call to an unverified üobject by a call to the intercept handler üobject with non-deterministic arguments. The next theorem states that each function g in a sequential üobject refines its abstract version $\widetilde{g}$ in that for each properly nested execution of g, there is a corresponding execution of $\widetilde{g}$. This is crucial to the soundness of our verification.

Theorem 4 (ExecRefine). *If g is a function belonging to a sequential üobject such that all Assembly code in g is in a CASM function satisfying all üSpark programming invariants, and c is any core, then for each properly nested execution τ of g on c there is a corresponding execution $\widetilde{\tau} \in [\![\widetilde{g}]\!]$ such that: $\tau \equiv \widetilde{\tau}$, where $\tau \equiv \widetilde{\tau}$ lifts the per-state equivalence to the trace.*

We use C verification tools to verify POCs directly on üBP (NDCC99 programs) of üSpark. Theorem 4 allows us to lift the verification results to üobject source code, formally stated in the following theorem (we only show the statement for invariant properties; the statement for individual state assertions is similar).

Theorem 5 (InvCompose). *Given any sequential üobject s, let $\widetilde{s}$ be the üBP abstraction of s. If an invariant property φ holds on every execution of $\widetilde{g}(s)$, then φ is an invariant property of every execution of s.*

6. üSpark Hypervisor Implementation

We applied üSpark to XMHF, an open-source micro-hypervisor for the x86 32-bit hardware-virtualized platform [72]. Originally, XMHF consists of a core hypervisor and a single extension (called *hypapp*), that together implement security-specific functionality. The latest version (0.2.2) runs a Ubuntu 12.04 32-bit multi-core guest OS with the core and hypapp at the highest privilege level and has been used to develop a wide variety of security applications [53], [73], [82], [84], [85]. Our goal is üXMHF– an incrementally developed and verified version with deprivileged components, and multiple hypapps. As a first step, we refactor XMHF into: (a) verified hypervisor (vh) üobjects for prime, sentinel, core, üAPIs, and verified hypapps; (b) unverified hypervisor (uh) üobjects for unverified hypapps; and (c) unverified guest (ug) üobjects for the OS (Figure 2); §8 quantifies this refactoring effort.

6.1. Core, Hypapp and Guest üObjects – We instantiate üXMHF core using three vh üobjects: `xcstrt` (startup), `xcihub` (handling ug üobject intercepts), and `xcehub` (runtime harware exception and watchdog handling). We instantiate extensions described in §2 as separate vh and uh üobjects and add support for multiple hypapps within `xcihub`. Finally, we instantiate a ug üobject, `guest` for the guest OS. The `xcstrt` üobject gets control from the prime üobject (§6.2), invokes all registered hypapp üobjects for initialization, and then transfers control to `guest`. The `xcihub` üobject gets control from the sentinel upon any intercept (§6.3) and in turn invokes the hypapp üobjects for `guest` event processing. Upon intercept handling, `xcihub` resumes execution of `guest` ug üobject (Figure 2).

6.2. Prime üObject – The üXMHF boot-loader uses the `GETSEC[SENTER]` instruction to setup a dynamic-root-of-trust and invokes the prime üobject in a hardware protected execution environment with the CPUs in a known good state and interrupts and DMA disabled.

Prime first enumerates devices and uses VT-d IOMMU to restrict their DMA to designated memory regions. It then initializes the vh and uh PAE page tables and the ug 2D EPT page tables for memory protections such that: (i) vh page tables map vh üobject memory regions, including MMIO, with supervisor privileges, and all uh and ug üobject memory regions as user with read-write permissions; (ii) each uh and ug page tables marks only its own region, including MMIO, as user and present; (iii) for uh üobjects, all vh üobject memory regions are marked supervisor; and (iv) for ug üobjects all vh and uh memory regions including MMIO are marked not-present. Prime uses disjoint CPU I/O bitmaps (which are marked supervisor within uh and ug üobject page tables) for uh and ug üobjects' legacy I/O isolation.

Finally, for each CPU in the system, prime: (a) activates protected-mode with paging and hypervisor-mode via control registers `CR0` and `CR4` and the `VMXON` instruction; (b) sets up `SYSENTER` MSRs, interrupt descriptor table and VM control structure (VMCS) to transfer control to the sentinel; and (c) loads vh page tables in `CR3` and transfers control to `xcstrt` core startup üobject.

6.3. Sentinel üObject – For vh to vh üobject control transfers, the sentinel uses an indirect `JMP` instruction. The `SYSEXIT` and `SYSENTER` fast system call instructions are used vh to uh control transfers and vice-versa. In such cases, the sentinel loads the uh page tables into the `CR3` register and transfers control to the uh üobject entry point (or return address via the `SYEXIT` instruction) at the de-privileged level. The sentinel uses the `VMLAUNCH` instruction for a call from a vh to ug üobject. It handles intercepts by transferring control to the vh `xcihub` üobject and upon return from `xcihub` resumes the ug üobject via the `VMRESUME` instruction. In both cases, it loads the ug üobject EPTs prior to the launch. The sentinel handles exceptions by transferring control to the vh `xcehub` üobject. Upon return from `xcehub` execution is resumed via the `IRET` instruction.

6.4. üAPI üObjects – Both the core and hypapp üobjects use üAPI üobjects to influence the ug üobject state. This state includes the ug üobject EPTs and VMCS. We implement üAPI üobjects `ugmpgtbl` and `ugcpust` which present interfaces to the ug üobject EPTs and VMCS respectively. We also implement an additional üAPI üobject `uhcpust` as an interface to shared CPU state between vh and uh üobjects (e.g., MSRs).

6.5. üObject Runtime Library – üObjects rely on a set of common functionality implemented in the following libraries: (a) `libuc` with memory and string functions; (b) `libucrypt` with SHA-1 functionality; (c) `libustub` with üobject entry and sentinel CASM stubs; and (d) `libuhw` for platform hardware access.

7. üSPARK HYPERVISOR VERIFICATION

7.1. Verification and Development Tools – We first describe the verification and development tools we use.

7.1.1. Static Analysis with Frama-C: Frama-C [41] is an industrial-strength C99 static analysis and verification toolkit, written in type-safe OCaml. It has a modular architecture and offers different plugins for distinct styles of analysis. We use the following Frama-C plugins: *Deductive verification* via Frama-C's Weakest-Precondition (WP) plugin enables the verification of assume-guarantee behavior specifications on C functions. Those specifications are expressed in the Annotated ANSI C Specification Language (ACSL) [25] in terms of the C source variables and operations. The WP plugin verifies such ACSL specifications statically on the body of the function by discharging verification conditions via an ensemble of external SMT solvers. *Abstract interpretation* via Frama-C's Value plugin analyzes a program using a sound abstraction of its concrete semantics. It is used to prove ACSL assertions placed in the body of the program that express partial specifications about program variables, and can be combined with deductive verification. *Abstract syntax tree (AST) analysis* via Frama-C's AST plugin performs syntactic analysis on control-flow graphs and ASTs to enforce syntactic restrictions, e.g., the absence of primitives like function pointers.

7.1.2. Hardware Model: We have implemented a C99 hardware model for the commodity x86 hardware-virtualized platform, by representing platform features such as CPU registers and system-device states as C variables and describing formally how the hardware (should) behave. The hardware model is a re-usable but trusted component. Our hardware model allows for iterative development, modeling only portions of the device used in proving security invariants. This design principle coupled with serialization enforced by the üSpark architecture blueprint (§4.4), enables us to keep the hardware model simple and amenable to formal validations. Various techniques exist to validate such a hardware model [50], [58] which we plan on exploring as future work (§11).

7.1.3. üSpark Frama-C Plugins: We built üSpark-specific plugins on top of Frama-C as follows: (a) `übp` – enforces üSpark blueprint; (b) `ühwm` – embeds hardware model during verification; (c) `ücasm` – substitutes Assembly mnemonics corresponding to CASM instructions after verification; (d) `ücc` – enforces general üSpark coding rules; (e) `ümf` – parses üobject manifest; and (f) `ücvf` – performs composition check (§7.2.1). These üSpark-specific plugins do not impact the robustness of the Frama-C toolset as we do not modify the kernel or standard plugins. Further, Frama-C's modular architecture helps us keep üSpark-specific Frama-C plugins small, simple, and amenable to manual audits to ensure correctness (§8.1).

7.1.4. Frama-C and CompCert: In keeping with our longer term goal of guaranteeing that the verified source code properties carry over to the binary, we employ the CompCert [11], [12], [46] certified C99 compiler to compile üobjects. CompCert over-specifies C99 implementation-defined and unspecified behaviors and is formally verified to produce semantically equivalent Assembly from a C99 program. Our choice of Frama-C and CompCert is further justified by their semantic compatibility. We empirically tested Frama-C against CompCert's C99 specifications and found that both tools had the same treatment of C99 implementation-defined and unspecified behaviors. Further, both tools employ an identical byte-addressable memory model with base addresses and offsets. Therefore, they combine naturally into a powerful analysis and development workflow towards producing verified system binaries.

7.1.5. Soundness Via Weakening: We weaken our execution model in two cases to enable sound reasoning. First, since current state-of-the-art static analyzers including Frama-C largely assume sequential execution, we treat all reads to DMA memory and all memory reads by a concurrent üobject as non-deterministic, for verification to soundly model interference from devices and other cores. Second, we preclude use of C function pointers

```
 1 void ugmpgtbl_setentry(u32 gsid, u32 addr, u64 v){
 2  /*sysclog*/  {v=v&7; v&=~_X; v|=_R; v|=_W;}
 3  /*hyperdep*/ {v=v&7; v&=~_X; v|=_R; v|=_W;}
 4  /*@assert sysclog:  (!(v&_X) && (v&_R) && (v&_W));*/
 5  /*@assert hyperdep: (!(v&_X) && (v&_R) && (v&_W));*/
 6 }
```

(a)

```
 7 void ugmpgtbl_setentry(u32 gsid, u32 addr, u64 v){
 8  /* sysclog  */ {v=v&7; v&=~_X; v|=_R; v|=_W;}
 9  /* aprvexec */ {v=v&7; v&=~_W; v|=_R; v|=_X;}
10  /*@assert sysclog:  (!(v&_X) && (v&_R) && (v&_W));*/
11  /*@assert aprvexec: (!(v&_W) && (v&_R) && (v&_X));*/
12 }
```

(b)

Fig. 3: Composition check: (a) **`hyperdep`** and **`sysclog`** üobjects both use ugmgtbl üAPI `setentry` interface to set guest memory page protections in a composable manner. (b) **`sysclog`** and **`apprvexec`** both use `setentry` in a non-composable manner.

and CASM indirect jump instructions, which remain challenging for current state-of-the-art static analyzers [21]. In practice (§7.2), this weakining does not stop us from verifying important security properties, since such properties are implemented via sequential üobjects using non-DMA memory.

7.2. üXMHF Verification – Verification of üXMHF consists of: (a) üobject composition check, and (b) verifying üSpark invariants (§5) and üobject local properties. Throughout this section we use vh, uh and ug as acronyms for verified and unverified hypervisor and unverified guest üobjects respectively.

7.2.1. üObject Composition Check: Resources accessed by multiple üobjects are guarded by üAPI üobjects (§4.3.1). Here we check that all üobjects are composable over the set of üAPIs they use. At a high level, this is checked by constructing an assertion that captures the conjunction of the possible values that the two üobjects write to a shared resource, and then verifying that this assertion is not violated. More specifically, for every üAPI üobject, an interface stub function is first created using its manifest. Next, the stub is populated with invariant definitions and assertions (if any) listed in the manifest of every vh and uh non-üAPI üobject that invokes it. Figure 3a shows an example stub for ugmpgtbl üAPI üobject `setentry` interface with **`hyperdep`** and **`sysclog`** hypapps enabled. Lines 2–6 are populated using the corresponding hypapp üobject manifests (Appendix B). Figure 3b shows the same stub with **`sysclog`** and **`aprvexec`** hypapps enabled. Finally, the assertions in the stub are verified under *non-deterministic* inputs. For example, **`hyperdep`** and **`syclog`** both set the read, write and clear the execute bits for the memory protections of the provided guest memory-page (lines 2–3) and are therefore composable; the assertions (lines 4–6) in Figure 3a are valid. However, **`sysclog`** and **`aprvexec`** are not composable (Figure 3b) since **`aprvexec`** sets the execute bit while **`sysclog`** clears the execute bit in the protections for the provided memory-page (lines 9–10). Note, such composition check assertions are also performed at runtime for üAPI invocations from uh

üobjects (§4.3.1). This composition check procedure is üXMHF-specific, and a more general check is an interesting direction for future work.

7.2.2. üObject Compositional Verification: As we discussed in §5, we first verify üSpark invariants via a set of PAHs and specific POCs on all vh üobjects including the prime and sentinel. §7.2.3 describes POC verification in further detail. We then verify each of the üXMHF core, hypapp and üAPI üobjects for their local invariants. For brevity we summarize the **`hyperdep`** üobject verification approach here. Appendix A lists the invariants and verification approach for other üXMHF üobjects. **`hyperdep`** preserves the following invariant over the `ugmpgtbl setentry` üAPI: guest OS provided memory pages are marked read-write and not executable. We use deductive verification to verify the `hyperdep` üobject `activate` method to ensure that the guest page address that is passed is used as the parameter to the `ugmpgtbl` üobject `setentry` method with read, write and no-execute protections. Finally, we verify the üobject runtime library (§6.5) for memory safety including behavior specifications for the memory and string functions within `libuc`. Note, uh üobjects are not verified since their properties follow from üAPI invariants. ensured by our composition check (§7.2.1).

7.2.3. POC Verification: For brevity, we choose a sampling of POCs from a few üSpark invariants ($\mathsf{Inv}^{4}_{\ddot{u}}$, $\mathsf{Inv}^{6}_{\ddot{u}}$, $\mathsf{Inv}^{6}_{\ddot{u}prog}$, $\mathsf{Inv}^{7}_{\ddot{u}prog}$, and $\mathsf{Inv}^{10}_{\ddot{u}}$; see Appendix D) that showcase the importance of all the verification techniques described in §7.1.1. All the üSpark invariant POCs are verified using a combination of these techniques. Note that examples described below are necessary (but not sufficient since they are a sample) for the high-level proofs; for example the NESTEDCALL theorem (§5) cannot be proved if there is no non-overlapping, unity-mapped memory ($\mathsf{Inv}^{4}_{\ddot{u}}$) or DMA protection ($\mathsf{Inv}^{6}_{\ddot{u}}$).

Figure 4 shows a POC code snippet – from the vh üobject page-table setup function within prime – for $\mathsf{Inv}^{4}_{\ddot{u}}$ verified using deductive verification. ACSL *requires-assign-ensure* clause triples (lines 4–11) are used to specify function behavior. In this case they specify that every memory address in the page tables is disjoint with virtual-to-physical unity mapping. ACSL *loop invariant* clause allows specification of loops with data structure invariants (lines 17–25). Finally, ACSL *ghost variables* – C statements and variables only visible in specifications – are most notably used for modular reasoning of nested function calls. For example, line 28 invokes a support function for obtaining the memory protection of the specified memory address. This is aliased into a ghost variable which can then be used within the specification (line 29). In summary, the requires-assigns-ensures clause triplet is sufficient to represent the function behavior, and the loop invariants and ghost

```
 1 //@ ghost u64 gflags[SZ_PDPT*SZ_PDT*SZ_PT];
 2 /*@
 3  ...
 4  requires \valid(vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1]);
 5  ...
 6  assigns vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1];
 7  assigns gflags[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1];
 8  ...
 9  ensures (\forall u32 x; 0<=x< SZ_PDPT*SZ_PDT*SZ_PT ==>
10   ((u64)vhpgtbl1t[x] == (((u64)(x*SZB_4K)
11   & 0x7FFFFFFFFFFFF000ULL) | (u64)(gflags[x]))));
12 @*/
13 void gp_setup_vhmempgtbl(void){
14  u32 i, spatype, slabid=XMHF_SLAB_PRIME;
15  u64 flags;
16  ...
17  /*@
18   loop invariant 0 <= i <= (SZ_PDPT*SZ_PDT*SZ_PT);
19   loop assigns gflags[0..(SZ_PDPT*SZ_PDT*SZ_PT)],spatype,
20    flags,i,vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)];
21   loop invariant \forall integer x; 0 <= x < i ==>
22   ((u64)vhpgtbl1t[x]) == (((u64)(x*SZB_4K)
23   & 0x7FFFFFFFFFFFF000ULL) | (u64)(gflags[x]));
24   loop variant (SZ_PDPT*SZ_PDT*SZ_PT) - i;
25  @*/
26  for(i=0; i < (SZ_PDPT*SZ_PDT*SZ_PT); ++i){
27   spatype=_gp_getspatype(slabid, (u32)(i*SZB_4K));
28   flags=_gp_getptflags(slabid, (u32)(i*SZB_4K),spatype);
29   //@ ghost gflags[i] = flags;
30   vhpgtbl1t[i] = pae_make_pte( (i*SZB_4K),flags);
31  }
32 }
```

Fig. 4: Frama-C ACSL behavior specification and deductive verification: vh üobject memory page-table setup top-level function in prime.

```
   prime.cS:
 1 ...
 2 ci_movl_eax_medi();
 3 ...
   hwm-cpu.c:
 4 void ci_movl_eax_medi(){
 5  ...
 6  if(uhm_cpu_r_edi >= IMMULO && uhm_cpu_r_edi >= IMMUHI)
 7    uhm_immuwr(uhm_cpu_r_edi,uhm_cpu_r_eax);
 8  ...
 9 }
   hwm-iommu.c:
10 void _gxmhfhwm_iommu_wr(u32 addr, u32 val){
11  ...
12  if (addr==IMMUCTRL){ cbuhm_immuctrlwr(val); ... }
13  ...
14 }
   prime-vdrv.c:
15 void cbuhwm_immuctrlwr(u32 val){
16  //@assert !(val & IMMUTE) || (val & IMMUTE) &&
17  // gxmhfhwm_iommu_retaddr == (u32)&gp_ret);
18 }
19 ...
```

Fig. 5: üSpark hardware model and proving IOMMU DMA protection.

variables within the function are used to prove the clause triplet. ACSL is highly expressive with global and type invariants, including first-order, polymorphic, recursive and higher-order specifications [25].

Fig 5 shows a POC code snippet for $\mathsf{Inv}^{6}_{\ddot{u}}$ verified using abstract interpretation and the hardware model. The snippet is part of the DMA protection setup function within prime. Line 2 in Fig 5 shows üobject using a designated CASM instruction to perform device I/O to the IOMMU. The hardware model hooks this CASM instruction to the IOMMU device model if the specified I/O range falls within the IOMMU device space (lines 6–7). The IOMMU modeling then simulates the required logic based on the register accessed and value written (line 12). The hardware model also invokes the appropriate verification driver callbacks whenever such device registers are written to (line 12). This ensures required

Component	Impl (SLoC)	Annot (SLoC)	Verification Time[s]	Verification Mem[GB]
üObject libraries:				
libuc	151	223	101	0.80
libucrypt	88	58	35	0.05
libustub	120	97	5	0.03
libuhw	1706	749	465	0.90
prime	2043	3176	1386	1.10
sentinel	672	501	423	0.75
üXMHF üAPI üObjects:				
ugmpgtbl	128	91	174	0.65
ugcpust	73	46	118	0.70
uhcpust	26	23	99	0.50
üXMHF Core üObjects:				
xcstrt	97	0	53	0.12
xcihub	247	202	147	0.60
xcehub	41	0	48	0.08
üXMHF Hypapp üObjects:				
sysclog	255	213	174	0.75
sysclognw	1193	273	413	0.85
hyperdep	161	31	98	0.70
aprvexec	199	–	–	–
Total/Avg.	7200	5544	3739	0.57
üSpark üAPI composition check			18	0.23
üSpark Hardware model SLoC = 2079				

Fig. 6: üXMHF üobject SLoC and verification time/memory.

device state invariants. For example, assertions in lines 16–17 of the IOMMU control register callback ensure that DMA page-table protections when enabled always point to the populated DMA page tables (which are populated by the prime in a separate function not shown). This ensures that devices can only perform DMA to üobject DMA memory region. Similar techniques are used to: (a) hook designated CASM instructions for üobject access to system memory including ug üobject memory regions; and (b) proving intra-üobject CFI in the presence of both C and CASM functions by ensuring that CASM functions respect the C ABI and preserve callee registers and stack frames (via corresponding hardware model callbacks, assertions, and ACSL annotations).

POCs for $\mathsf{Inv}^{6}_{\ddot{u}prog}$ and $\mathsf{Inv}^{7}_{\ddot{u}prog}$ are verified by analysing the abstract syntax trees (AST) to preclude statements involving function pointers in C functions and to ensure CASM functions always end with a CASM `ret` instruction respectively. The POC for $\mathsf{Inv}^{10}_{\ddot{u}}$ is verified via CFG analysis to enforce üSpark blueprint conformance. Similar AST-based techniques are employed to: (a) embed hardware model statements, (b) substitute Assembly mnemonics, and (c) ensure soundness of the hardware model by precluding C functions from touching hardware model functions and variables and vice-versa.

8. Evaluation

8.1. System size and Verification TCB – üXMHF is implemented in 7001 SLoC *verified* privileged code split into 11 üobjects with 5544 lines of ACSL annotations and 2079 lines of hardware model (Figure 6). We

übp	ücasm	ücc	ümf	ühwm	ücvf	Total
108	296	138	132	199	148	**1021**

Fig. 7: Frama-C üSpark specific plugins are written in OCaml and build atop existing Frama-C kernel and standard plugins.

also implemented an unverified hypervisor extension (**`aprvexec`**; 199 SLoC) to illustrate how unverified and verified hypervisor üobjects interact. Depending on the properties, üobject verification takes 48 seconds to 23 minutes, and up to 1.1 GB of memory. Cumulative verification time is just over an hour, comparing favorably to related verification efforts [34]. Compositional verification enables each üobject to be (re-)verified separately. The prime üobject takes the longest to verify, but typically does not change as often as other üobjects. Decomposing prime into multiple üobjects can further reduce its (re-)verification time significantly.

Our verification TCB comprises the ACSL annotations, the hardware model (§7.1.2), and Frama-C with associated plugins. Modularity of üobject programs helps keep annotations small and feasible for manual review. Various orthogonal techniques exist to validate our hardware model [50], [58] that we plan to explore as future work. Frama-C is an industrial-strength tool used in many critical systems today [41]; we did not encounter any soundness bugs in these tools (§9). Frama-C üSpark specific plugins (totaling 1021 SLoC of OCaml; Figure 7) are modular, simple, and built upon the existing Frama-C kernel and plugins making them amenable to manual audits. Overall, our TCB compares favorably with other prior approaches (Figure 8).

8.2. Developer Effort – üXMHF was developed and verified in a year by a single system developer who was new to Frama-C/ACSL. A fraction of the time was spent adding implementation support for multiple hypapps with a greater part spent on porting to the üSpark hypervisor architecture by creating required üobjects and adding verification related harnesses and annotations. Annotation-to-code ratio (ACR) ranges from 0.2:1 to 1.6:1 (Figure 6). For üobjects whose properties rely solely on üAPI's the ACR is small (e.g., **`hyperdep`**). üObjects with properties requiring functional correctness (e.g., **`sysclog`** and **`xcihub`**) have relatively larger ACR. The prime and sentinel üobjects have the highest ACR since they discharge most of the üSpark invariants.

8.3. Performance Measurements – All performance benchmarks were carried out on a Dell Optiplex 9020 with an Intel Core-i5 4590 quad-core processor with 4GB of memory. All üobjects were compiled with full compiler optimizations turned on.

8.3.1. üSpark Microbenchmarks: The cost of a CASM NULL function call is only 12 clock cycles. Sentinel call overhead for verified-to-verified üobject transitions is 2x w.r.t NULL function call (Figure 9). This is due to

System/TCB	Compiler	HW Model	Annot./Specs.	Verification Tools	Other
Verve	In TCB	NS	NS	Boogie, BoogieASM, TAL checker, Z3	Iso-gen, boot-loader
seL4	In TCB	NS	In TCB	Isabelle/HOL, HOL4, Myreen, Sonolar, Z3	boot-loader
Hyper-V/Vcc	In TCB	In TCB	In TCB	Vcc, Boogie, Z3	boot-loader
Ironclad	Out-of TCB	In TCB	In TCB	Boogie, BoogieASM, Dafnyspec, Symdiff, Z3	None
mCertiKOS	Out-of TCB	NS	In TCB	Coq	None
üSpark	Out-of TCB	In TCB	In TCB	Frama-C, Frama-C üSpark plugins, Z3, CVC3, Alt-Ergo	None

Fig. 8: Development and Verification Tools Trusted Computing Base (TCB) Comparison: All systems in addition employ a preprocessor (either built-in or stand-alone) for macro substitution and file inclusion and an assembler and linker to produce machine code; *NS = Not supported*

Verified–Verified	Verified–Unverified / Unverified–Verified			
	SEG	CR3	TSK	HVM
2x	37x	48x	70x	278x

Fig. 9: üSpark Microbenchmarks: Sentinel üobject call overheads w.r.t regular NULL function call in privileged mode.

CPUID	RDMSR	WRMSR	XSETBV	CRx	VMCALL	SIPI
100	98	98	100	100	99	99

Fig. 10: üXMHF Microbenchmarks: core intercept handling clock-cycle latency as % of native XMHF performance without üSpark.

sysclog	hyperdep	aprvexec	ropdet	iousb	ionet	iodisk	ioser
97	99	91	89	95	96	99	99

Fig. 11: üXMHF Hypapp and I/O Benchmarks as % of native XMHF performance without üSpark.

SPEC	ioz-read	ioz-write	compbench	apache
100	100	100	100	100

Fig. 12: üXMHF Guest CPU and I/O Benchmarks as % of native XMHF Guest performance without üSpark.

control transfers to the sentinel and üobject entry points and return addresses via JMP instructions. For transitions involving unverified üobjects the sentinel overhead is broken up into: (a) software overhead such as register saving, parameter marshalling, and call-policy enforcement; and (b) hardware deprivileging overhead. As seen, segmentation and CR3-based page tables provide the lowest overheads (37x and 48x), but are still an order of magnitude larger than the verified-to-verified sentinel call overhead. Hardware deprivileging adds a significant portion (upward of 60%) to the sentinel call in this case. These overheads are comparable to existing unverified disaggregated systems and micro kernels (§10).

8.3.2. üXMHF Microbenchmarks: For purposes of micro benchmarking we measure the üXMHF `xcihub` üobject, which handles several intercepts required for guest execution. üXMHF delivers near native XMHF performance in all cases (Figure 10). We attribute the small overhead for certain intercepts to the code refactoring using üobjects.

8.3.3. üXMHF Guest Benchmarks: We execute both compute-bound and I/O-bound applications for guest benchmarking purposes. For compute-bound applications, we use the SPEC-INT 2006 suite. For I/O-bound applications, we use the iozone (disk reads and writes with 4K block size and 2GB file size), compilebench (project compilation benchmark), and Apache web server performance (ab tool with 200,000 transactions with 20 concurrent connections). üXMHF does not affect native XMHF's guest performance in all cases (Figure 12).

8.3.4. üXMHF Application Benchmarks: We use the hypapps described in §6.1 along with another unverified hypapp **`ropdet`** (which captures guest branch information for ROP detection) for hypapp performance benchmarking. We wrote a guest üobject that interacts with the hypapps to leverage their services as follows. For **`sysclog`**, activate syscall logging by setting the syscall code page to no-execute and perform sample syscalls. For **`hyperdep`**, set a data page to no-execute and perform data read and write operations on that page. For **`aprvexec`**, setup a code page for approved execution, and invoke the hypapp to approve and lock the page against writes, before executing a sort function on that code page. Finally, for **`ropdet`**, register a test function over which ROP detection is to be performed, and a invoke the test function to collect branch information. Figure 11 shows the performance overhead for these hypapps compared to native XMHF without üSpark. Verified **`sysclog`** and **`hyperdep`** run close to native XMHF speeds (2% avg. overhead). Unverified **`aprvexec`** and **`ropdet`** incur higher overheads (9% and 11% respectively). The overhead is due to üAPI invariant checks (<10%) and the sentinel cost of deprivileging, shadow stack and parameter marshalling (§8.3.1).

For I/O performance benchmarks, we wrote a mix of DMA I/O (usb and net) and programmed I/O operations (disk and serial) within a hypervisor üobject. The I/O performance overhead (Figure 11) is anywhere from 1-5% with the DMA-based I/O incurring more overhead. We attribute the higher DMA-based I/O overhead to the IOMMU page tables for DMA access. Note that üSpark does not actively interpose on any I/O operations, which results in a much lower overhead. These I/O overheads also match up to existing micro hypervisor I/O architecture overheads [67], [72], [85].

9. Experience and Lessons Learned

9.1. Frama-C – The WP plugin's limited casting support helped detect erroneous esoteric casts, e.g., pointer to int/u8. While the Value plugin cannot propagate states to arbitrarily large loops, the semantic unrolling option helped propagate states only for desired functions so memory/time resources can be well spent. WP loop invariants are versatile in supporting unbounded loops with nesting. WP discharges proofs more effectively when operating over single-dimensional array accesses for mutating assignments and invariants and simple statements using shift and bit-wise operators. WP also caused proof failures in certain cases with local variable aliasing of function parameters; using parameter variables directly ameliorated the issue. We did not encounter any soundness bugs in Frama-C and its plugins.

9.2. Verification Theories – Automated verification results vary by theory, e.g., Alt-Ergo and Z3 failed to discharge a few verification conditions (VC) that CVC3 handled. Frama-C's ability to combine provers was very useful; CVC3, Z3 and Alt-Ergo together solved all the VCs generated during verification.

9.3. Annotations – ACSL is versatile in its support for writing partial specifications (e.g., memory safety of SHA-1) and assertions as well as complete specifications (e.g., page-table setup). Futher, ACSL annotations use actual C variables and operations. This expressivity spectrum thus allows system programmers to easily transition into the verification domain by initially using simple assertions and function contracts (partial specifications) and iteratively mastering complete specifications.

9.4. CompCert – The C99 subset handled by CompCert suffices to implement most systems-level software constructs. However, struct bit fields with packing and alignment within struct fields are currently unsupported. We added methods with bitwise operators to pack, unpack, deconstruct, and align such variables in the sources.

10. Related Work

10.1. Unverified monolithic – SELinux [66], AppArmor [1] and FBAC [59] are some examples of OS kernel modifications that add features to an existing (privileged) kernel to enforce various access control policies. Such approaches suffer from the lack of assurance and separation: a bug in an extension or the core can exist, and then affect other parts of the system arbitrarily.

10.2. Unverified disaggregation – Xen/Xoar [17] converts Xen into deprivileged partitions. NOVA [67] deprivileges everything (including VMM modules), except for a small privileged micro kernel. Safe composition of OS kernel extensions include extensible operating systems [10], [15], [20], [23], [39], [61], kernel driver isolation [13], [28], [47], [48], [69], [70], [77], interposition mechanisms [29], [35], [37], [40] and API compatability libraries [5], [7], [9], [30], [56], [78]. Xax [19] confines untrusted application code to an ABI for accessing OS services. SGX [4] protects application code from (buggy) privileged code. Disaggregation brings mere isolation but no formal guarantees on its own.

10.3. Verified sandboxing – SFI [52], [54], [60], [75], [81] is a software-based approach for application-level memory isolation but lacks support for low-level privileged instructions and hardware device access, which are necessary for hypervisor and its extensions. Also, SFI employs unverified binary rewriting which can change the semantics of the program and break invariants necessary for compositional verification. Singularity [36] sacrifices legacy compatibility with a complete redesign of a OS written in type-safe languages (MSIL/TAL) and uses software mechanisms to isolate processes (SIP) and supports only memory and type-safety properties.

10.4. Verified kernels – seL4 [43] verifies full functional correctness of the C implementation (7500 LOC) of the micro kernel by showing that it refines an abstract specification. Their specifications don't support abstractions among the kernel or the different kernel modules. These interdependencies often lead to more complex invariants which are difficult to prove (20 person years). Further, seL4 does not allow adding properties using untrusted services; such additions require direct integration into the kernel and lengthy re-verification. Furthermore, there is no support for Assembly (ASM) or device states, which precludes verification of low-level code interacting with devices; (1200 C and 500 ASM SLoC remain unverified). mCertiKOS [31] follows a similar approach to seL4 but makes the abstract specification layered to reduce the interdependencies among the kernel and various extensions and makes the verification process more tractable for an admittedly stripped down version of the original CertiKOS kernel (single-core, non-preemptible custom guest OS, basic process and syscall handling). There is no hardware model and support for ASM is limited to only general-purpose registers. Adding extra system instruction support and device models does not seem trivial; even the stripped down version of the kernel has 300 C and 170 ASM SLoC unverified. This is attributed to memory model limitations of their methodology [31]. Lastly, both mCertiKOS and seL4 require the developer to write line-for-line specifications for C/ASM code in a different abstract language (Isabelle/HOL or Coq/Ocaml/Lasm) with a very steep learning curve.

The VCC project [16], [45] verifies the functional correctness of a fixed Hyper-V hypervisor codebase running a multi-CPU guest, via automated theorem proving. However, the code annotations do not support abstractions among the core hypervisor or drivers. This leads to complex invariants due to interdependencies; only 20% of the hypervisor code-base has been verified [16]. Further, their ASM verification methodology and lack of a full hardware model only allows proving memory safety and arithmetic properties for ASM functions while precluding compiler optimizations for the corresponding C callee functions [51]. XMHF [72] employs the CBMC model checker with assertions on the C code of a micro-hypervisor to verify memory integrity. However, multiple extensions or composing other properties on top of memory integrity are unsupported. Further, that effort *assumes* interface confinement and leaves out 422 C and 388 ASM SLoC due to limitations of CBMC with large-loops and lack of a hardware model.

10.5. Verified System Stack – In Verve [80], a simplified OS and applications are verified for type and memory safety using a Hoare-style verification condition (VC) generator and automated theorem proving. Ironclad [33] extends Verve with support for higher-level application properties. High-level specifications (written in Dafny) are translated to corresponding code with VCs discharged via an automated theorem prover; the verification took 3 person-years. Verisoft [6] integrates hardware and software, with high-level specifications written in C0 (a tiny subset of C semantics) and refined down to a custom CPU semantics. The verification took 20 person-years on a simple OS with only a disk driver. System stack verification approaches, while powerful, sacrifice compositionality, legacy compatibility and performance. Any changes to kernel code and/or extension configuration requires lengthy re-verification (in person years). Further, the entire system software stack has to be re-implemented in type-safe languages such as C# and TAL (in Verve) or in high-level Dafny specifications (in Ironclad) or on a non-commodity CPU abstraction (in Verisoft). Furthermore, these approaches lack support for co-existence with unverified programs or a guest OS.

11. Limitations and Future Work

We now discuss current limitations of our approach with pointers to future work towards bridging these gaps.

11.1. Hardware Model – Our hardware model is currently a trusted component. However, orthogonal techniques such as path-exploration lifting [50] and mechanized x86-multiprocessor semantics [58] provide a solid foundation on which we plan to build upon and validate our hardware model in the future.

11.2. CASM and Certified Compilation – Our high-level proofs depend on Compcert's specification of the C memory and register semantics and CASM's adherence to those semantics (discharged as invariants on the source-code and our hardware model) to ensure that the C and Assembly code operate on disjoint state. In the future, we plan on leveraging recent developments with Compcert such as the ability to compile and link multi-module source programs [68] to cleanly extend the bi-simulation proof of the CompCert compiler to encompass hardware state and Assembly code. Future work also involves proving (e.g., via bi-simulation) the semantic equivalence between the hardware model and the corresponding Assembly instructions and demonstrating the semantic synergy between CompCert, CASM and the Frama-C kernel more rigorously for proved properties to translate to the binary.

11.3. Functional Verification – Our focus in this paper is on security invariants and trace properties and functional correctness to support such properties. We are optimistic that liveness properties and full-functional correctness are achievable future goals and not any more harder than existing approaches [31], [33], [43].

11.4. Concurrency – We have shown that a practical multi-threaded system with interesting security properties can be built by dealing with a serialized execution model and sequential verification in lieu of complex concurrent verification. However, we do realize the importance of relaxing our serialized execution model especially in high-performance computing environments and plan on leveraging source-level multi-threaded verification (e.g., Frama-C mthread plugin [24]) to address concurrency in the future.

11.5. Soundness of Tools – Similar to existing approaches, we assume that the verification tools such as Frama-C with associated plugins and back-end theorem provers such as Z3, CVC3 and Alt-Ergo are sound (§8.1,§3.3). Discharging this assumption, while a desirable goal, is currently an open and hard problem in the face of formal methods. However, seminal breakthroughs such as certified software model-checking [55] and formal verification of C static analyzers [38] give us hope that proving soundness of our verification tools will indeed be possible in the future.

11.6. System Software Applicability – Our future work involves exploring the applicability of üSpark to more general-purpose hypervisors (e.g., Xen and KVM). The immediate challenges we envision there include unraveling complex data structures, supporting dynamic memory allocations and use of indirect function calls in addition to supporting some form of concurrency. Aside from hypervisors, we are also exploring the applicability of üSpark to other system software subsystems such as the BIOS, device firmware and the operating-system kernel and drivers including vertical integration among these stacked subsystems.

12. Conclusion

We presented überSpark, an innovative architecture enforcing verifiable object abstractions in low-level C and Assembly languages and leveraging them in combination with off-the-shelf C software verifiers and certifying compilers to produce high assurance hypervisors for commodity platforms. We incrementally developed and verified a commodity x86 micro-hypervisor using üSpark, and performed a comprehensive evaluation which shows automated compositional verification with modest development effort and minimal runtime overhead.

Availability: überSpark and üXMHF sources are available at: **`http://uberspark.org`**

Acknowledgements: We thank the anonymous reviewers for their detailed comments. We also thank Úlfar Erlingsson and Martín Abadi for their feedback and insights. This work was partially supported by the AFOSR MURI on Science of Cybersecurity, the NSA/CMU Science of Security Lablet, and the NSF CNS-1018061 grant. Copyright 2016 CyLab and CMU [2].

References

[1] Novell, AppArmor, and SELinux Comparison. http://www.novell.com/linux/security/apparmor/selinux_comparison.html.

[2] CVE-2008-3687: Heap-based buffer overflow in Xen 3.3, when compiled with the XSM:FLASK module, allows unprivileged domain users (domU) to execute arbitrary code via the flaskop hypercall. https://cve.mitre.org/, 2008.

[3] VMSA-2009-0006: VMware patches for ESX and ESXi resolve a critical security vulnerability. http://www.vmware.com/security/advisories/, 2009.

[4] Software Guard Extensions Programming Reference 329298-001. http://software.intel.com, 2013.

[5] http://cygwin.com, 2014.

[6] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. *J. Autom. Reasoning*, 42(2-4):389–454, 2009.

[7] J. Appavoo, M. Auslander, D. Edelsohn, D. D. Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Providing a linux api on the scalable k42 kernel. In *ATC*, 2003.

[8] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004. Workshop on Formal Techniques for Java-like Programs (FTfJP), ECOOP 2003.

[2]This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003658

[9] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proc. of EuroSys*, 2013.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proc. of SOSP*, 1995.

[11] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM*, 2006.

[12] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *In Proc. of IEEE ARITH*, 2013.

[13] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *TEC*, 2010.

[14] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar. Towards verifiable resource accounting for outsourced computation. In *ACM VEE*, 2013.

[15] D. R. Cheriton and K. J. Duda. A caching model of os kernel functionality. In *OSDI*, 1994.

[16] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLS*, 2009.

[17] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proc. of SOSP*, 2011.

[18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. of CCS*, 2008.

[19] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proc. of OSDI*, 2008.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP*, 1995.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proc. of CCS*, 2015.

[22] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proc. of IEEE/ACM ASE 2010*, 2010.

[23] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of OSDI*, 1996.

[24] Frama-C. Mthread plug-in. http://frama-c.com/mthread.html, 2012.

[25] Frama-C Team. ACSL: ANSI/ISO C Specification Language v1.9. http://www.frama-c.com, 2015.

[26] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worrying about Data Structure Size. In *IEEE S&P*, 2010.

[27] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical Report CMU-CyLab-08-008, CMU CyLab, 2008.

[28] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *ASPLOS*, 2008.

[29] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. Slic: An extensibility system for commodity operating systems. In *ATC*, 1998.

[30] D. Given. http://lbw.sf.net/, 2010.

[31] 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.

[32] N. Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, Oct. 1985.

[33] 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 *Proc. of OSDI*, 2014.

[34] 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 *Proc. of OSDI*, 2014.

[35] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *WINSYM*, 1999.

[36] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[37] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. *SIGOPS Oper. Syst. Rev.*, 27(5):80–93, Dec. 1993.

[38] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified c static analyzer. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 247–259, New York, NY, USA, 2015. ACM.

[39] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. of SOSP*, 1997.

[40] Y. A. Khalidi and M. N. Nelson. Extensible file systems in spring. *SIGOPS Oper. Syst. Rev.*, 27(5):1–14, Dec. 1993.

[41] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *FAC*, 27(3):573–609, 2015.

[42] 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*, 32(1):2:1–2:70, Feb. 2014.

[43] 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 *Proc. of SOSP*, 2009.

[44] K. Kortchinsky. Cloudburst: A VMware guest to host escape story. Black Hat, 2009.

[45] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM*, 2009.

[46] X. Leroy. Formal certification of a compiler backend, or: programming a compiler with a proof assistant. In *Proc. of POPL*, 2006.

[47] B. Leslie, P. Chubb, N. Fitzroy-dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. In *Journal of Computer Science and Technology*, 2005.

[48] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, 2004.

[49] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proc. of USENIX Security*, 2008.

[50] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for Lo-fi emulators. *SIGPLAN Not.*, 47(4):337–348, Mar. 2012.

[51] S. Maus, M. Moskal, and W. Schulte. Vx86: x86 assembler simulated in C powered by automated theorem proving. In *Proc. of AMAST*, 2008.

[52] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security*, 2006.

[53] 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*, May 2010.

[54] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: Better, faster, stronger SFI for the x86. *SIGPLAN Not.*, 47(6):395–404, 2012.

[55] K. S. Namjoshi. Certifying model checkers. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22,*

2001, Proceedings, pages 2–13, 2001.

[56] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. *SIGARCH Comput. Archit. News*, 39(1):291–304, Mar. 2011.

[57] D. Quist, L. Liebrock, and J. Neil. Improving antivirus accuracy with hypervisor assisted analysis. *J. Comput. Virol.*, 7(2), May 2011.

[58] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-cc multiprocessor machine code. *SIGPLAN*, 44(1):379–391, 2009.

[59] Z. C. Schreuders, C. Payne, and T. McGill. Techniques for automating policy specification for application-oriented access controls. In *Proc. of ARES*, 2011.

[60] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proc. of USENIX Security*, 2010.

[61] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of OSDI*, 1996.

[62] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of SOSP*, 2007.

[63] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: A fast capability system. In *SOSP*, SOSP '99, pages 170–185, 1999.

[64] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proc. of CCS*, 2009.

[65] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *EuroSys*, 2006.

[66] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux LSM. *NSA*, 2001.

[67] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proc. of Eurosys*, 2010.

[68] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional compcert. In *POPL*, pages 275–287, 2015.

[69] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. of SOSP*, 2003.

[70] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM TOCS*, 23(1):77–110, Feb. 2005.

[71] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[72] 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 *Proc. of IEEE S&P*, 2013.

[73] 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*, 2012.

[74] A. Vasudevan, N. Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proc. of IEEE HICSS*, 2011.

[75] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS OSR*, 27(5):203–216, Dec. 1993.

[76] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proc. of EuroSys 2012*, 2012.

[77] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.

[78] Wine. http://www.winehq.org/, 2014.

[79] X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *Proc. of NDSS*, 2011.

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

[81] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. of IEEE S&P*, 2009.

[82] M. Yu, V. D. Gligor, and Z. Zhou. Trusted display on untrusted commodity platforms. In *ACM CCS*, pages 989–1003, 2015.

[83] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of SOSP*, 2011.

[84] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE S&P*, 2012.

[85] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O. In *Proc. of IEEE S&P*, 2014.

Appendix A
Verification of üXMHF üobjects local invariant properties

We now describe our verification approach in detail for verifying the invariant properties of the üXMHF üobjects shown in Figure 13. For all the üobjects we verify via deductive verification that the üobject entry point function transfers control to the appropriate method handler for a given public method.

We verify the üAPI üobjects via abstract interpretation. For the `uhcpust` üobject we verify that the `write` method, in case of a write to MSR EFER, always preserves the EFER bits required for üSpark functionality. For the `ugmpgtbl` üobject we verify that the `setentry` method's entry parameter does not fall within hypervisor memory regions. Finally, for the `ugcpust` üobject we verify that the `write` method disallows writes to any host-specific state in the guest VMCS.

For the `xcihub` üobject we employ deductive verification to verify the `main` method such that, for any given intercept a special function `hcbinvoke` is called with the intercept type and associated parameters. `hcbinvoke` is then verified to ensure that it calls all the registered hypapp üobjects for that intercept.

For the `sysclog` üobject we employ deductive verification to first verify that the `init` method invokes the `ugmpgtbl` üobject `setentry` method with the syscall page address with read and no-execute protections. We then verify that the syscall trap handler obtains syscall information via a call to the `ugcpust` üobject `read` method and stores this information to the network log buffer via a call to the `sysclognw` üobject `log` method.

We verify the `sysclognw` üobject via a combination of deductive verification and abstract interpretation. We use deductive verification to verify the `log` method to ensure that: (a) the buffer passed in as parameters is stored in the network buffer data structure, and (b) when the buffer is full, its contents are copied into the üobject $dmadata$ region, buffer is reset, and the network send function is invoked. We then verify the `send` function via abstract interpretation to ensure that it programs the network card hardware to read from the $dmadata$ region, transmit the buffer, and wait for end of transmission signal.

We use deductive verification to verify the `hyperdep` üobject `activate` method to ensure that the guest page address that is passed is used as the parameter to the `ugmpgtbl` üobject `setentry` method with read, write and no-execute protections.

Note, `aprvexec` (unverified) üobject is not verified since its properties follow from the `ugmpgtbl` üAPI invariants ensured by our composition check as described in §7.2.1.

üObject	**Type**	**Invariant Property**
xcihub	vh	On intercept invoke corresponding hypapp handler
ugcpust	vh	Writes to host state only by prime or sentinel
uhcpust	vh	No writes to host MSR EFER
ugmpgtbl	vh	No mapping of hypervisor memory regions
hyperdep	vh	Guest OS provided memory-pages are marked read-write and not executable
sysclog	vh	On system call trap intercept, log syscall information to network log buffer
sysclognw	vh	Log info in network log buffer and transmit buffer when full
aprvexec	uh	Guest OS approved code pages are always marked read-only and executable

Fig. 13: üXMHF Core, üAPI and Hypapp üobject invariants; vh = verified hypervisor üobject, uh = unverified hypervisor üobject

APPENDIX B
üXMHF üOBJECT USE MANIFEST

Figure 14 shows partial üobject manifest listings for the following üXMHF üobjects: **`sysclog`**, **`sysclognw`**, **`hyperdep`**, **`aprvexec`**, **`ugmpgtbl`**, **`ugcpust`** and **`rguest`**. The salient manifest definitions are described below:

- üobject type (TY) definition indicates the type of the üobject which can be verified hypervisor (vh), unverified hypervisor (uh) or unverified guest (ug)
- üobject local memory resource (RML) definitions identify the üobject code, data, stack and dmadata extents
- üobject non-local memory resource (RMG) definitions identify access to memory belonging to other üobjects. Valid access types are `READ` and `WRITE`. This is used for accessing guest OS (**`rguest`**) memory regions
- üobject device resource definition (RD) indicates the devices (if any) allocated to the üobject. The `INCL` entry identifies devices to be included (with PCI device and vendor IDs); a special entry 0xffff:0xffff indicates allocation of all system devices that are not allocated to other hypervisor üobjects
- üobject CPU instruction resource (RC) definition indicates which privileged CASM instruction macros (if any) the üobject is allowed to use
- üobject callee üobject dependency (SD) definitions list üobjects a given üobject can invoke
- üobject üAPI call capabilities and invariant definitions (UI) provide information about the üAPI's the üobject can invoke along with the corresponding invariants preserved
- üobject üAPI interface declaration definitions are only applicable to üAPI üobjects and indicate all the üAPI interface declarations that the üAPI üobject supports

The üSpark `ümf` Frama-C plugin uses üobject manifests to output a C99 file with a üobject information data structure which is used by: (a) prime to setup üobject page tables and resource allocations; (b) sentinel to enforce üobject to üobject call capabilities; and (c) üAPI üobjects to restrict üAPI interface invocations. üObject manifests are also used by the üSpark `ücvf` Frama-C plugin to generate üAPI stubs for composition checks and by the üSpark `ücc` Frama-C plugin to restrict CASM privileged instruction use within a given üobject. Finally, the üSpark `übp` Frama-C plugin uses üobject manifests to enforce üSpark blueprint conformance.

```
sysclog.manifest:
...
TY:vh:::
...
RML:CODE:0x200000::
RML:DATA:0x200000::
RML:STACK:0x600000::
RML:DMADATA:0x200000::
...
RMG:READ:ugrguest::
...
SD:ugmpgtbl:::
...
UI:ugmpgtbl:GETENTRY:(void)0;:(void)0;
UI:ugmpgtbl:SETENTRY:{{v=v&7;
 v&=~_X; v|=_R; v|=_W;}}:
 /*@assert (!(v&_X) && (v&_R) && (v&_W));*/
...
sysclognw.manifest
...
RD:INCL:0x8086:0x10b9::
...
hyperdep.manifest
...
SD:ugmpgtbl:::
...
UI:ugmpgtbl:GENTRY:(void)0;:(void)0;
UI:ugmpgtbl:SENTRY:{v=v&7; v&=~_X;
 v|=_R; v|=_W;}:/*@assert !(v&_X)
 && (v&_R) && (v&_W));*/
...
aprvexec.manifest
...
TY:uh:::
...
SD:ugmpgtbl:::
...
UI:ugmpgtbl:GENTRY:(void)0;:(void)0;
UI:ugmpgtbl:SENTRY:{v=v&7; v&=~_W;
 v|=_R; v|=_X;}:/*@assert (!(v&_W) &&
 (v&_R) && (v&_X));*/
...
ugmpgtbl.manifest
...
UD:GENTRY:u64 ugmpgtbl_getentry(u32 gsid, u32 addr)::
UD:SENTRY:void ugmpgtbl_setentry(u32 gsid,u32 addr,u64 v)::
...
ugcpust.manifest
...
RC:ci_vmread:::
RC:ci_vmwrite:::
...
rguest.manifest
...
TY:ug:::
...
RD:INCL:0xffff:0xffff::
...
```

Fig. 14: Partial üobject manifest listings for üXMHF implementation showing salient manifest element definitions.

Appendix C
üSpark Invariants, Models, Semantics and Proofs

üSpark reasoning relies foundationally on a set of invariants – properties that must hold throughout the execution of a üSpark hypervisor. The invariants are divided into üSpark system invariants (Figure 15) and üSpark general programming invariants (those that pertain specifically to üSpark üobject C and CASM functions; Figure 16). Each invariant is proved by reducing it further to a set of *proof-assumptions on hardware* (PAHs) and *proof-obligations on code* (POCs) using the üSpark blueprint (üBP; §4–Figure 2). POCs are then discharged on all üSpark verified üobjects including the prime and sentinel using specific verification tools and techniques (§7). A hypervisor implementation is compliant with üSpark– and therefore amenable to compositional reasoning – if it satisfies all the üSpark invariants. Full details of invariant-to-PAH/POC mappings, a one-time effort, can be found in Appendix D. We proceed by first describing a formal model of the üSpark architecture followed by detailed semantics, verification approach and associated theorem proofs. Throughout, we make assumptions that follow from the üSpark invariants, and make this relationship explicit by stating invariants in square braces right after the assumptions they imply.

üSpark Architecture Model: Figure 17 shows a formal model of the üSpark architecture. At the highest level the system (Sys) is composed of system entities, resource states and system interfaces. System entities include a set of programs (p) üobjects on system CPUs executing concurrently with devices (d) üobjects. There are verified and unverified hypervisor and guest program üobjects with prime and sentinel being two special verified hypervisor üobjects responsible for system startup and üobject interactions (§4). Every program üobject has the following non-overlapping or disjoint ($\diamond$) memory sections: (a) $code$ for üobject code; (b) $data$ for üobject data; (c) $dmadata$ for performing DMA to/from üobject and devices; (d) $mmio$ for accessing device interfaces via memory-mapped I/O; and (e) $sysmemro$ and $sysmemrw$ for read-only and read-write system memory such as BIOS and free memory regions. In addition, every unverified üobject has its own $stack$ section while all verified üobjects operate on a single $stack$ section contained within the prime üobject.

System resource state comprises: (a) üobject writable states ($data$ and $sysmemrw$) for all üobjects; (b) CPU state (program and system control states) for all CPUs; and (c) device state (device internal $data$ and MMIO interface via $sysmemrw$)

System interfaces dictate interfaces via which üobjects and system devices can access system resources and are enforced via a combination of hardware (hw) and light-

$\mathsf{Inv}_{\ddot{u}}^{1}$	üSpark begins execution with the entry point of a distinguished initial "prime" üobject s_I in single-core mode with just core 1 activated
$\mathsf{Inv}_{\ddot{u}}^{2}$	A special "asynchronous" function $startcores(s)$ activates all cores $i > 1$ and begins executing a designated üobject s immediately thereafter; all cores remain active thereafter for the system lifetime.
$\mathsf{Inv}_{\ddot{u}}^{3}$	Asynchronous control transfers (hardware interrupts, exceptions and intercepts) respect the blueprint state execution threading and transitions
$\mathsf{Inv}_{\ddot{u}}^{4}$	üObject memory regions are unity-mapped and non-overlapping
$\mathsf{Inv}_{\ddot{u}}^{5}$	üObject s accesses only its own memory
$\mathsf{Inv}_{\ddot{u}}^{6}$	üObject code, data and stack regions are DMA protected
$\mathsf{Inv}_{\ddot{u}}^{7}$	üObject code is write-protected
$\mathsf{Inv}_{\ddot{u}}^{8}$	Inter-üobject synchronous control-flow respect blueprint transitions
$\mathsf{Inv}_{\ddot{u}}^{9}$	Each core has its own stack at all times and stays within the stack limits.
$\mathsf{Inv}_{\ddot{u}}^{10}$	Blueprint state has state appropriate execution threading (multi-core or single-core)
$\mathsf{Inv}_{\ddot{u}}^{11}$	Locks behave like "memory fences"; any write preceding a call to unlock is observed by any read following the next call to lock

Fig. 15: üSpark System Invariants

$\mathsf{Inv}_{\ddot{u}prog}^{1}$	CASM functions preserve caller registers
$\mathsf{Inv}_{\ddot{u}prog}^{2}$	CASM functions establish local stack frame non-overlapping with incoming caller stack frame
$\mathsf{Inv}_{\ddot{u}prog}^{3}$	CASM functions have conditional and unconditional branches local to the function
$\mathsf{Inv}_{\ddot{u}prog}^{4}$	CASM functions establish callee incoming stack frame for calls to other C or CASM functions
$\mathsf{Inv}_{\ddot{u}prog}^{5}$	CASM functions tear down local stack frame before returning
$\mathsf{Inv}_{\ddot{u}prog}^{6}$	CASM functions end with return instruction
$\mathsf{Inv}_{\ddot{u}prog}^{7}$	No function pointers in C functions
$\mathsf{Inv}_{\ddot{u}prog}^{8}$	C and CASM functions do not write to caller stack frame params and return-address
$\mathsf{Inv}_{\ddot{u}prog}^{9}$	CASM functions can only encode instructions within the domain of CASM instruction set
$\mathsf{Inv}_{\ddot{u}prog}^{10}$	CASM non-local control transfer instructions can only be to fixed function entry points

Fig. 16: üSpark Programming Invariants

weight static analysis ($sw\text{-}verif$). Verified üobjects can access its own üobject state and allowable CPU state via CPU instructions and can access other üobject states via the sentinel – all enforced via $sw\text{-}verif$ based on the üobject use manifest. Unverified üobjects are confined to their üobject state and directly modifiable CPU state via commodity hardware support for deprivileging. Unverified üobjects can further access verified üobject state only via the sentinel, a capability enforced using a combination of hw and $sw\text{-}verif$. Finally, system devices are confined to accessing only üobject $dmadata$ regions leveraging the DMA controller hw (e.g., IOMMU).

CASM: An "Assembly function" is a CompCert-

System Entities

$System$	Sys	$:=$	$(Programs, Devices, CPUs)$
$Programs$	p	$:=$	$Objects$
$Devices$	d	$:=$	$Objects$
$Objects$		$:=$	$(Verified\text{-}primeobject, Verified\text{-}sentinelobject, Verified\text{-}programobjects, Unverified\text{-}programobjects)$
$Verified\text{-}primeobject$	$prime$	$:=$	$(code \diamond data \diamond stack[mcpus][msize] \diamond mmio \diamond dmadata \diamond sysmemrw \diamond sysmemro)$
$Verified\text{-}sentinelobject$	$sentinel$	$:=$	$(code \diamond data \diamond prime.stack \diamond S_{pu}.stack \diamond sysmemro)$
$Verified\text{-}programobjects$	S_{pv}	$:=$	$(code \diamond data \diamond mmio \diamond dmadata \diamond sysmemrw \diamond sysmemro)$
	S_v	$:=$	$\{prime\} \cup \{sentinel\} \cup S_{pv}$
$Unverified\text{-}programobjects$	S_{pu}	$:=$	$(code \diamond data \diamond stack[mcpus][msize] \diamond mmio \diamond dmadata \diamond sysmemrw \diamond sysmemro)$

System Resource States

$Object\text{-}state$	S^i_{st}	$:=$	$(p_i.data \diamond p_i.sysmemrw)$
$CPU\text{-}state$	C^j_{st}	$:=$	$(CPU\text{-}programstate_j, CPU\text{-}systemcontrolstate_j)$
$Device\text{-}state$	D^k_{st}	$:=$	$(d_k.data \diamond d_k.sysmemrw)$

System Interfaces

$(sw\text{-}verif)$	$p_i \in S_v$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$\rhd$	S^i_{st}		
$(sw\text{-}verif)$	$p_i \in S_v$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$\rhd$	C^j_{st}		
$(sw\text{-}verif)$	$p_i \in S_v$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$\rhd$	$sentinel$	$\rhd$	$S^i_{st} \in (S_v \cup S_{pu})$
$(sw\text{-}verif)$	$p_i \in S_v$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$(\rhd$	$p_i.mmio)$	$\rhd$	D^k_{st}
(hw)	$p_i \in S_{pu}$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$\rhd$	S^i_{st}		
(hw)	$p_i \in S_{pu}$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$\rhd$	$CPU\text{-}programstate_j$		
$(hw, sw\text{-}verif)$	$p_i \in S_{pu}$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$\rhd$	$sentinel$	$\rhd$	$S^i_{st} \in S_v$
(hw)	$p_i \in S_{pu}$	$\rhd$	$(CPU_j, CPU\text{-}instructions)$	$(\rhd$	$ps_i.mmio)$	$\rhd$	D^k_{st}
(hw)	d_k	$\rhd$	$(DMA\text{-}controller)$	$\rhd$	$p_i.dmadata,$		$p_i \in (S_v \cup S_{pu})$

$CPU\text{-}instructions$	$:=$	$\{platform\ specific\}$
$DMA\text{-}controller$	$:=$	$\{platform\ specific\}$

Fig. 17: üSpark Architecture Formal Model

C99 (CC99) function whose body consists only of a block of Assembly instructions that respect the CC99 ABI. A CASM program is a CC99 program such that the following two conditions hold: (i) **(CASM1)** All Assembly code appears only in Assembly functions; and (ii) **(CASM2)** every register that is both accessed by an Assembly function and by code generated from C source is also required to be preserved by the C ABI. In particular, this means that from the point of view of C source code, an Assembly function is "pure" (i.e., side-effect free). These assumptions are captured by invariants shown in Figure 16, and these invariants are verified directly on the üobject source code.

We begin by stating and proving two foundational üSpark theorems essential for the correctness of our approach which follow directly from the üSpark programming invariants (Figure 16).

Theorem 1 (DISJOINTCASM)**.** *The union of üobject CASM and C functions preserve the existing semantic preservation property of the certified compiler.*

Proof. CompCert semantic preservation property: If S has well defined semantics (does not go wrong) then S and C are observationally equivalent; S is source, C is compiled binary.

We need to show every CASM function call has CompCert Mach (last step before Assembly code generation) semantics. This in turn would imply preserving CompCert's semantic preservation proofs.

What we need to show about every CASM function:

- Caller registers preserved [$\mathsf{Inv}^1_{\textit{üprog}}$]
- Establish local stack frame non-overlapping with incoming stack frame [$\mathsf{Inv}^2_{\textit{üprog}}$]
- Conditional and unconditional branches local to the function [$\mathsf{Inv}^3_{\textit{üprog}}$]
- Call to function establishes callee incoming stack frame [$\mathsf{Inv}^4_{\textit{üprog}}$]
- Tear down local stack frame [$\mathsf{Inv}^5_{\textit{üprog}}$]
- End with return [$\mathsf{Inv}^6_{\textit{üprog}}$]

■

Theorem 2 (EXITSENTINEL)**.** *üobject execution can only exit via the sentinel.*

Proof. CompCert, by default does not allow inline Assembly within C functions and only generates the following types of synchronous control transfer instructions for üobject C functions: (a) call; (b) ret; and (c) indirect branch. Precluding function pointers in C functions removes (c) [$\mathsf{Inv}^{7}_{\textit{üprog}}$]. Ensuring C and CASM functions do not write to caller stack frame params and return address ensures (a) and (b) are strongly paired [$\mathsf{Inv}^{8}_{\textit{üprog}}$]. Finally, every CASM function can only perform indirect control transfers to a fixed üobject entry point (the sentinel) [$\mathsf{Inv}^{9}_{\textit{üprog}}$,$\mathsf{Inv}^{10}_{\textit{üprog}}$]. By all the above arguments, a üobject execution can can only exit via the sentinel.

■

Next we formally define üSpark semantics.

Runtime The üSpark runtime system consists of non-overlapping unity-mapped üobject memory regions [$\mathsf{Inv}^{4}_{\ddot{u}}$], read-only üobject code regions [$\mathsf{Inv}^{7}_{\ddot{u}}$], and devices confined to perform DMA only to designated üobject DMA memory region [$\mathsf{Inv}^{6}_{\ddot{u}}$]. We write $X = x_1 \diamond \ldots \diamond x_n$ to mean that X is partitioned into $x_1, \ldots, x_n$.

üObject: A üobject is a triple (p, f, e) where p is a CASM program, f is a function in p denoting the entry point of the üobject, and e is a set of calls to functions not in p denoting exit points of the üobject. Given a üobject $s = (p, f, e)$, we write $p(s)$, $f(s)$, and $e(s)$ to mean p, f and e, respectively. üSpark code is partitioned into a finite set of üobjects *üobjects* $= \{s_1, \ldots, s_k\}$. This means that every function in üSpark belongs to one, and only one, üobject. This partitioning induces a transition relation δ over üobjects in a natural way: $(s, s') \in \delta \iff f(s') \in e(s)$ i.e., $\delta(s, s')$ means that s invokes s'. The reflexive and transitive closure of δ is denoted δ^*. Thus, $(s, s') \in \delta^*$ if either $s' = s$, or s invokes s' directly or indirectly.

Core Bringup: üSpark begins execution with the entry point of a distinguished initial "prime" üobject s_I with just core 1 activated [$\mathsf{Inv}^{1}_{\ddot{u}}$]. Subsequently, core 1 executes instructions to activate other cores. A special "asynchronous" function $startcores(s)$ activates all cores $i > 1$ that begin executing üobject s immediately and remain active thereafter for the system lifetime [$\mathsf{Inv}^{2}_{\ddot{u}}$].

Locking: üSpark uses a set of locks $\mathcal{L}$, and functions $lck(l)$ and $ulk(l)$ to acquire and release a lock $l \in \mathcal{L}$. Locks also behave like "memory fences", i.e., any write preceding a call to $ulk(l)$ is observed by any read following the next call to $lck(l)$ [$\mathsf{Inv}^{11}_{\ddot{u}}$].

Sequential and Concurrent üObjects: Our goal is to allow üSpark to use the maximum amount of concurrency possible, while enabling tractable analysis. To this end, the set of üobjects is partitioned into two subsets – sequential and concurrent: *üobjects* $=$ *üobjects*$_s \diamond$ *üobjects*$_c$. When a sequential üobject $s \in$ *üobjects*$_s$ executes on core i, no other core can execute s. In essence, s is a monitor. Each sequential üobject is either executed only in single-core mode, other üobjects are executed only after acquiring a lock [$\mathsf{Inv}^{10}_{\ddot{u}}$]. In contrast, multiple cores can execute a concurrent üobject $s \in$ *üobjects*$_c$ simultaneously.

Core Signal Handling: Core asynchronous signals (e.g., exceptions, intercepts) respect function call semantics [$\mathsf{Inv}^{3}_{\ddot{u}}$]. In other words, when triggered, signals either transfer control back to the point where the original üobject was interrupted or to the entry point of a new üobject (one-way function call).

Memory: System memory is partitioned into one stack per core, and a non-stack area. The non-stack area is further partitioned into one non-stack area per üobject. Thus: $M = stk_1 \diamond \ldots \diamond stk_n \diamond \; data(s_1) \diamond \ldots \diamond data(s_k)$, where stk_i is the stack area for core i, $data(s)$ is the non-stack area for üobject s. Note that this non-stack area is further partitioned into a data area, a code area, a DMA area, and a system area. The memory addresses of each partition are fixed once instantiated and remain unchanged throughout the execution of üSpark [$\mathsf{Inv}^{4}_{\ddot{u}}$]. In general, üobject s executing on core i accesses only $mem(i, s) = stk_i \cup data(s)$ [$\mathsf{Inv}^{5}_{\ddot{u}}$]. We assume that the DMA area is modifiable arbitrarily by devices and therefore all reads to this memory return non-deterministic values.

Core State: A state of core i on üobject s is a pair (m, r) where m is the state of $mem(i, s)$ and r is the state of the core's registers such that the value of pc lies in the üobject's code area. We use the terms "core state" and "state" interchangeably. Given a state $\sigma = (m, r)$ we write $m(\sigma)$ and $r(\sigma)$ to denote m and r, respectively. We also write $stk(\sigma)$ and $nstk(\sigma)$ to denote the stack and non-stack components of σ.

Core State Equivalence: Two states σ and σ' of core c (on same or different üobjects) are core-equivalent, denoted $\sigma \approx_c \sigma'$, if they agree on stack and registers, i.e., $\sigma \approx_c \sigma' \iff stk(\sigma) = stk(\sigma') \wedge r(\sigma) = r(\sigma')$. Two states σ and σ' (of identical or different cores) on üobject s are üobject-equivalent, denoted $\sigma \approx_s \sigma'$, if they agree on the non-stack memory, i.e., $\sigma \approx_s \sigma' \iff nstk(\sigma) = nstk(\sigma')$, if we ignore the DMA area of s.

Core Transition Relation: Given a state σ of core c on üobject s, we write $core(\sigma)$ and $object(\sigma)$ to denote c and s. Given two states σ, σ' of core c (on identical or different üobjects), we write $\sigma \xrightarrow{c} \sigma'$ to mean that executing instruction at $pc(\sigma)$ on c from state σ, and allowing the DMA area of $object(\sigma)$ to be changed arbitrarily, results in the state being updated to σ'. Given function g, we write $entry(g)$, $exit(g)$, $call(g)$ and $ret(g)$ to denote, respectively the first instruction of g, an instruction that returns from g, an instruction that calls g, and an instruction immediately following a call to g.

üObject Semantics: An execution of a üobject $s = (p, f, e)$ on core c is a sequence of states $(\sigma_1, \ldots, \sigma_k)$ of c on s that: (i) ends with a call to an exit point function

$g \in e$ or the return statement of f; and (ii) for $i \in [1, k)$, state σ_{i+1} is obtained by executing instruction at $pc(\sigma_i)$ after allowing possible changes to the memory by other cores and devices. More formally, the following hold:

1) $pc(\sigma_k) = exit(f) \vee \exists g \in e \,\centerdot\, pc(\sigma_k) = call(g)$.
2) $\forall i \in [1, k) \centerdot$
   - $object(\sigma_i) \in \textit{üobjects}_s \implies \sigma_i \xrightarrow{c} \sigma_{i+1}$
   - $object(\sigma_i) \in \textit{üobjects}_c \implies \exists \sigma \centerdot \sigma \approx_c \sigma_i \wedge \sigma \xrightarrow{c} \sigma_{i+1}$

The semantics of üobject s on core c, denoted $[\![s, c]\!]$, is the set of all its executions. Note that for concurrent üobjects, we allow the non-stack area to be modified arbitrarily (by instructions running on other cores). Also, the DMA area can be modified arbitrarily in each execution step. The stack and registers remain unchanged since each core has its own disjoint stack region in memory. Given a üobject execution $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_k) \in [\![s, i]\!]$ we write $fst(\Sigma)$, $lst(\Sigma)$, $object(\Sigma)$ and $core(\Sigma)$ to mean σ_1, σ_k, s and C_i, respectively. Note that the stack-pointer register of core i always lies within the limits of its stack stk_i [$\mathsf{Inv}^9_{ü}$].

üSpark Semantics: An execution of üSpark is obtained by interleaving executions of üobjects on cores while taking into account appropriate changes to the memory and registers. In particular, consider a sequence of üobject executions $\pi = \Sigma_1, \Sigma_2, \ldots, \Sigma_m$. For each index $j \in [1, m]$, the most recent üobject at j, denoted $mrs(\pi, j)$ is the largest index $k \in [1, j)$ such that $object(\Sigma_k) = object(\Sigma_j)$ and $\perp$ if no such index exists. Similarly, the most recent core at j, denoted $mrc(\pi, j)$ is the largest index $k \in [1, j)$ such that $core(\Sigma_k) = core(\Sigma_j)$ and $\perp$ if no such index exists. Formally, an execution of üSpark is a sequence of üobject executions $\pi = \Sigma_1, \Sigma_2, \ldots, \Sigma_m$ such that if $\forall j \in [1, m]$,

$$\Sigma_j \in [\![s_j, C_j]\!] \wedge \sigma_j = fst(\Sigma_j) \wedge \sigma'_j = lst(\Sigma_j)$$

then the following hold:

1) The non-stack memory at the beginning on each execution is identical to the non-stack memory at the end of the last execution of the same üobject:
$$\forall j \in [1, m] \centerdot mrs(\pi, j) = k \implies \sigma'_k \approx_{s_j} \sigma_j$$
2) The stack and registers at the beginning of each execution result from executing the last instruction of the last execution of the same core:
$$\forall j \in [1, m] \centerdot mrc(\pi, j) = k \implies \\ \exists \sigma \centerdot \sigma'_k \xrightarrow{c_k} \sigma \wedge \sigma \approx_{s_j} \sigma_j$$

Properly Nested üObject Execution: Recall the relation δ^* and define $\delta^*(s) = \{s' \mid (s, s') \in \delta^*\}$. A properly nested execution of üobject s on core c is a sequence of üobject executions $\pi = \Sigma_1, \ldots, \Sigma_m$ on c such that:

1) The sequence begins with an execution of $f(s)$, i.e., $\sigma_1 = entry(f(s))$.
2) Each Σ_j is an execution of s or some üobject s' invoked directly or indirectly by s on core c, i.e.,
$$\forall j \in [1, m] \centerdot core(\Sigma_j) = c \wedge object(\Sigma_j) \in \delta^*(s)$$
3) If $\sigma'_j = exit(g)$, and the most recent execution entering a üobject is Σ_k such that $\sigma_k = entry(g')$ then $g' = g$. This means "calls to" and "returns from" üobjects are properly nested.
4) If $\sigma'_j = call(g)$ then $\sigma_{j+1} = entry(g)$, and if $\sigma'_j = exit(g)$ then $\sigma_{j+1} = ret(g)$. This means "calls to" and "returns from" üobjects behave like procedures.

Theorem 3 (NESTEDCALL)**.** *Consider any legal execution $\pi = \Sigma_1, \Sigma_2, \ldots$ of üSpark and any sequential üobject s. Then the projection of π on executions of $\delta^*(s)$ consists of a sequence of properly nested executions of s, each on a specific core.*

Proof. The proof follows from the fact that sequential üobjects are always invoked either in single-core mode or while holding a lock [$\mathsf{Inv}^{10}_{ü}$];, and by showing that a üobject execution can never modify the return address stored on the stack.

We prove that the return address is never modified as described below. We partition sequential üobjects into two groups – "verified" and "unverified".

For verified üobjects, we have compile-time control flow integrity by Theorem 2. Therefore, the return-address is never modified by any intra-üobject control flow. Further, by Theorem 2 inter-üobject control flow from a verified üobject can only occur via the üSpark sentinel. The üSpark sentinel preserves the return-address on the stack-frame to ensure there is inter-üobject control flow integrity [$\mathsf{Inv}^8_{ü}$].

Upon a control transfer to an unverified üobject, the üSpark sentinel ensures that prior to the control transfer the verified üobject stack frame is saved and control is transfered to the unverified üobject in hardware-enforced deprivileged mode on the unverified üobject stack [$\mathsf{Inv}^8_{ü}$]. Subsequently, an unverified üobject can perform a hardware-enforced inter-üobject control flow transition only via the üSpark sentinel. The üSpark sentinel maintains a shadow-stack for all verified to unverified transitions and ensures that the appropriate verified stack frame and verified üobject return-addresses are restored upon a return [$\mathsf{Inv}^8_{ü}$].

■

Hardware Model and Converting Assembly to C: Recall that Assembly instructions in üSpark appear only in bodies of Assembly functions. In addition to general purpose registers (which are preserved to respect the CC99 ABI) these Assembly instructions access a special set of hardware registers which are necessary to interact with devices (e.g., LAPIC). Let us denote the set of register accessed by Assembly functions in üSpark by $\mathcal{R}_{hw}$. In order to verify üSpark using source code analysis tools, we:

1) introduce a set of fresh C variables $\mathcal{V}_{hw} = \{v_r \mid r \in \mathcal{R}_{hw}\}$
2) replace each Assembly instruction accessing $\mathcal{R}_{hw}$ by one or more CC99 statements that operate in a semantically equivalent way over $\mathcal{V}_{hw}$.
3) replace each $r \in \mathcal{V}_{hw}$ with v_r in assertions used for specifying pre-and-post conditions during verification.

We refer to the mapping between $\mathcal{R}_{hw}$ and $\mathcal{V}_{hw}$, and the induced mapping from Assembly instructions to CC99 statements, as our "hardware model". We assume that this mapping is correct. We refer to the CC99 function obtained by transforming an Assembly function f in this manner as $\widetilde{f}$.

üSpark Blueprint: To analyze üSpark, we abstract it further as an non-deterministic CC99 (NDCC99) program, which is a CC99 program that also allows non-deterministic selection of values from finite sets. In particular, our abstract üSpark üBP consists of a set of abstract üobjects, where each abstract üobject $\widetilde{s}$ is obtained from the corresponding concrete üobject s by converting each function $g \in p(s)$ to an abstract function $\widetilde{g}$ as follows:

1) Each read to DMA memory is replaced by a read that returns a non-deterministic value;
2) If g is the entry function of untrusted guest üobject, then $\widetilde{g}$ sets all global variables of s to non-deterministic values and then calls the entry function of the intercept handler üobject; otherwise
3) If g belongs to a concurrent üobject, then $\widetilde{g}$ is obtained from g by replacing each read of a global variable (i.e., data area access) with the read of a non-deterministic value; otherwise
4) If g is an Assembly function then $\widetilde{g}$ is constructed using the hardware model as described earlier; otherwise
5) $\widetilde{g} = g$.

Note that Steps 1 and 2 above cause üBP to be non-deterministic. In particular, Step 1 over-approximates the behavior of a guest by a completely non-deterministic program that interacts with the remaining üobjects by causing intercepts. Step 2 models interference between cores when a concurrent üobject is executing as required by our üobject semantics. Finally, note that üBP is a complete program with no external dependencies.

Equivalence between States of üSpark and üBP: While a state of üSpark consists of the contents of memory and registers, a state of üBP is an assignment of variables to values. However there is a natural mapping between the two – a variable from the source code maps to the memory location it is allocated by the compiler, while a variable v_r introduced by the hardware model maps to the corresponding register r. We say a s state of üSpark is equivalent to a state $\widetilde{s}$ of üBP, denoted $s \equiv \widetilde{s}$, if they are identical modulo this mapping. It can be shown that each üSpark state s is equivalent to a unique üBP state $\widetilde{s}$, and an assertion φ holds on s iff it holds on $\widetilde{s}$.

Semantics of üBP: The semantics of üBP follows directly from that of NDCC99. Specifically, an execution of any function $\widetilde{g} \in$ üBP is a sequence of states $\widetilde{\tau}$ such that if we begin executing $\widetilde{g}$ from state $fst(\widetilde{\tau})$ then $\widetilde{g}$ returns with state $lst(\widetilde{\tau})$ following the CC99 semantics, and allowing for non-deterministic choice. The semantics of $\widetilde{g}$, denoted $[\![\widetilde{g}]\!]$, is the set of all its executions.

We now show that üBP abstracts üSpark in a sound way. At a high-level, this is only correct if the CASM functions are effect free for the C functions. Imagine an inline Assembly code in a C functions that either changes the general register, or control register that points to the correct page table, then after executing the Assembly code, then the state of the C function will be altered after executing the Assembly code. To achieve effect free, first we place all Assembly code in CASM funtion, so no general registers are clobbered, then verify the Assembly code to make sure that other important control registers are not modified and that all control transfers satisfy the invariant.

The following theorem shows that each function g in a sequential üobject refines its abstract version $\widetilde{g}$ in that for each properly nested execution of g, there is a corresponding execution of $\widetilde{g}$.

Theorem 4 (EXECREFINE)**.** *If g is a function belonging to a sequential üobject s.t. all Assembly code in g is in a CASM function satisfying all the $\mathsf{Inv}_{\ddot{u}prog}$ properties, and c is any core, then for each properly nested execution τ of g on c there is a corresponding execution $\widetilde{\tau} \in [\![\widetilde{g}]\!]$ such that: $\tau \equiv \widetilde{\tau}$, where $\tau \equiv \widetilde{\tau}$ lifts the per-state equivalence to the trace.*

Proof. The proof follows from:

1) The definition of properly nested üobject executions where entry into and exit from üobjects are equivalent semantically to function calls and returns, and
2) Our construction of $\widetilde{g}$ from g: it was either left unchanged, replaced by a more non-deterministic version, or (in the case of Assembly functions) replaced by version that operates equivalently over $\mathcal{V}_{hw}$ instead of $\mathcal{R}_{hw}$.

In particular, our transformation of Assembly function g to $\widetilde{g}$ is sound because: (i) by definition g respects the CC99 ABI so its operations over general purpose registers are not visible outside the scope of g; and (ii) non-general purpose registers in $\mathcal{R}_{hw}$ are only accessed by Assembly functions and never by code compiled from regular C statements in üSpark (we ensure this via static analysis). ■

Theorem 5 (INVCOMPOSE). *Given any sequential üobject s, let $\widetilde{s}$ be the üBP abstraction of s. If an invariant property φ holds on every execution of $g(s)$, then φ is an invariant property of every execution of s.*

Proof. First, by Theorem 3 we know that any üSpark execution when projected on $\delta^*(s)$ consists of a sequence of properly nested executions of s. Next the proof follows by induction on the length of this sequence. The base case is establish by that the precondition φ holds on the first state. The inductive step is proved via φ is an invariant and Theorem 4 since they imply that every transition step of $g(s)$ on any core preserves φ. ■

Appendix D
Discharging üSpark Invariants as Proof Assumptions on Hardware and Proof Obligations on Code

Figure 18 shows the x86 hardware-virtualized architecture specialization of the üSpark architecture model previously described in Appendix C–Figure 17. Figure 19 shows the corresponding Proof Assumptions on Hardware (PAH) we developed for üSpark.

We now describe how we discharge the üSpark general programming (Figure 16) and system invariants (Figure 15) as a combination of PAHs and Proof Obligations on Code (POC) on the prime, sentinel and verified üobjects in general.

üSpark general programming invariants (Figure 16) $\mathsf{Inv}^{1}_{\ddot{u}prog}$, $\mathsf{Inv}^{2}_{\ddot{u}prog}$, $\mathsf{Inv}^{3}_{\ddot{u}prog}$, $\mathsf{Inv}^{4}_{\ddot{u}prog}$, $\mathsf{Inv}^{5}_{\ddot{u}prog}$, $\mathsf{Inv}^{6}_{\ddot{u}prog}$, $\mathsf{Inv}^{7}_{\ddot{u}prog}$, $\mathsf{Inv}^{8}_{\ddot{u}prog}$, $\mathsf{Inv}^{9}_{\ddot{u}prog}$, and $\mathsf{Inv}^{10}_{\ddot{u}prog}$ are directly discharged via static analysis on each verified üobject source-code.

üSpark system invariants $\mathsf{Inv}^{1}_{\ddot{u}}$ is discharged by `PAH-1(rt)`; $\mathsf{Inv}^{2}_{\ddot{u}}$ is discharged via `POC-36` on the verified prime üobject source-code; and $\mathsf{Inv}^{11}_{\ddot{u}}$ is discharged by `PAH-7(gp)`.

üSpark system invariants $\mathsf{Inv}^{3}_{\ddot{u}}$, $\mathsf{Inv}^{4}_{\ddot{u}}$, $\mathsf{Inv}^{5}_{\ddot{u}}$, $\mathsf{Inv}^{6}_{\ddot{u}}$, $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, and $\mathsf{Inv}^{10}_{\ddot{u}}$ are discharged via a set of PAHs and POCs on the prime, sentinel and all verified üobject source-code. We first list the cumulative PAHs and POCs for each of the aforementioned üSpark invariants. We describe the methodology we use to extract the cumulative PAHs and POCs right after.

$\mathsf{Inv}^{3}_{\ddot{u}}$: hardware (`PAH-1(rt)`; `PAH-4(gp)`, `PAH-5(gp)` `PAH-17(ex)` `PAH-2(smp)`, `PAH-18(ex)`, `PAH-15(dpg)`, `PAH-16(dpg)`, `PAH-12(dpg)`, `PAH-14(dpg)`, `PAH-15(dpg)`, `PAH-13(dpg)` `PAH-19(ex)`, `PAH-18(ex)`,); prime (`POC-11`, `POC-23`, `POC-24`, `POC-34`,); sentinel (`POC-37`, `POC-38`, `POC-41` `POC-39`,); verified üobjects (`POC-7`, `POC-9`, `POC-10`, `POC-44`,);

$\mathsf{Inv}^{4}_{\ddot{u}}$: hardware (`PAH-1(rt)`, `PAH-6(gp)` `PAH-8(dp)`, `PAH-9(dp)`, `PAH-10(dp)`, `PAH-11(dp)` `PAH-12(dpg)`, `PAH-14(dpg)`, `PAH-16(dpg)` `PAH-15(dpg)`, `PAH-13(dpg)`); prime (`POC-6` `POC-22`, `POC-27`, `POC-28`, `POC-29`, `POC-30`,); sentinel (`POC-39`, `POC-40`, `POC-45` `POC-39`, `POC-41`, `POC-42`); verified üobjects (`POC-15`, `POC-16`, `POC-17`, `POC-13`, `POC-14`);

$\mathsf{Inv}^{5}_{\ddot{u}}$: hardware (`PAH-1(rt)`, `PAH-6(gp)` `PAH-6(gp)`, `PAH-12(dpg)`,); prime (`POC-6`, `POC-22`, `POC-27`, `POC-28`, `POC-29`, `POC-21`); sentinel(`POC-47`); verified üobjects (`POC-5`, `POC-15`, `POC-16`, `POC-17`, `POC-13`, `POC-14` `POC-5`,);

$\mathsf{Inv}^{6}_{\ddot{u}}$: hardware (`PAH-1(rt)` `PAH-21(dma)`; `PAH-12(dpg)`, `PAH-16(dpg)`,); prime (`POC-35`,); sentinel (); verified üobjects (`POC-19`);

$\mathsf{Inv}^{7}_{\ddot{u}}$: hardware (`PAH-1(rt)`, `PAH-6(gp)` `PAH-2(smp)`, `PAH-12(dpg)`, `PAH-8(dp)`, `PAH-9(dp)`, `PAH-10(dp)`, `PAH-11(dp)` `PAH-12(dpg)`, `PAH-14(dpg)`, `PAH-16(dpg)` `PAH-15(dpg)`, `PAH-13(dpg)`); prime (`POC-22`, `POC-28`, `POC-29`, `POC-27`, `POC-30`,); sentinel (`POC-39`, `POC-40`, `POC-45` `POC-41`, `POC-42`); verified üobjects (`POC-15`, `POC-16`, `POC-17`, `POC-13`, `POC-14`);

$\mathsf{Inv}^{8}_{\ddot{u}}$: hardware (`PAH-1(rt)`, `PAH-17(ex)` `PAH-18(ex)`, `PAH-15(dpg)`, `PAH-13(dpg)`,); prime (); sentinel (`POC-46`); verified üobjects ($\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, `POC-9`, `POC-10`, `POC-18`);

$\mathsf{Inv}^{9}_{\ddot{u}}$: hardware (`PAH-2(smp)`, `PAH-6(gp)`, `PAH-8(dp)`, `PAH-9(dp)`, `PAH-10(dp)`, `PAH-11(dp)` `PAH-12(dpg)`, `PAH-14(dpg)`, `PAH-16(dpg)` `PAH-15(dpg)`, `PAH-13(dpg)`); prime (`POC-22`, `POC-28`, `POC-29`, `POC-27`, `POC-30`,); sentinel (`POC-46`, `POC-39`, `POC-40`, `POC-45`, `POC-47`, `POC-43`); verified üobjects (`POC-1`, `POC-2`, `POC-11`, `POC-12` `POC-15`, `POC-16`, `POC-17`, `POC-13`, `POC-14`);

$\mathsf{Inv}^{10}_{\ddot{u}}$: hardware (`PAH-1(rt)`, `PAH-2(smp)`, `PAH-12(dpg)`, `PAH-16(dpg)`, `PAH-15(dpg)`,); prime (`POC-36`); sentinel (); verified üobjects (`POC-20`, `POC-21`);

üSpark invariants $\mathsf{Inv}^{3}_{\ddot{u}}$, $\mathsf{Inv}^{4}_{\ddot{u}}$, $\mathsf{Inv}^{5}_{\ddot{u}}$, $\mathsf{Inv}^{6}_{\ddot{u}}$, $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, and $\mathsf{Inv}^{10}_{\ddot{u}}$ described previously are discharged on each state in the üSpark blueprint (§4–Figure 2) using a combination of PAHs (Figure 19) and POCs (Figure 20, Figure 21, and Figure 22). For each state in the blueprint, we first enumerate all categories of üobjects involved in that state (prime, sentinel, verified and unverified hypervisor program üobjects). Finally, for each üobject category we enumerate the POCs and PAHs that need to be satisfied for each of the invariants to hold. Finally, for each invariant we accumulate the PAHs and POCs for each state. The following are the list of states and the corresponding POCs and PAHs for the aforementioned üSpark invariants.

- State-1:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via `PAH-1(rt)`; `POC-7`, `POC-44`, `PAH-4(gp)`, `PAH-5(gp)` `PAH-17(ex)` ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via `PAH-1(rt)`, `POC-6` `POC-22`, `PAH-6(gp)`; ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via `PAH-1(rt)`, `POC-6`, `POC-22`, `POC-5`, `PAH-6(gp)` ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via `PAH-1(rt)` ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via `PAH-1(rt)`, `POC-22`, `PAH-6(gp)` ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, `POC-18`, `PAH-1(rt)`, `PAH-17(ex)` ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via `POC-22`, `PAH-6(gp)` ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via `PAH-1(rt)`, `POC-20`, `POC-21`
- State-2:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via `POC-7`, `POC-11`, `POC-44`, `POC-23`, `POC-24`, `POC-34`, `PAH-18(ex)`, `PAH-4(gp)`,

$CPU\text{-}programstate$	:=	(hwcore-varstackptr, others)
$CPU\text{-}systemcontrolstate$	:=	(hwcore-varinterrupts$(true, false)$, hwcore-varexcptblptr, hwcore-vardpexcpstackptr, hwcore-varpaging$(true, false)$, hwcore-varmempgtblptr, hwcore-varsmpactivate$(true, false)$, hwcore-varmode$(host, guest)$, hwcore-varprivilege$(supervisor, user)$, hwcore-varsyscallesp, hwcore-varsyscallep, hwcore-varsysexitesp, hwcore-varsysexitep, hwcore-varguestmempgtblptr, hwcore-varguesthostep, hwcore-varguesthostmempgtblptr, hwcore-varguesthostesp, hwcore-varguesthostpl, hwcore-varid, others)
$DMA\text{-}controller$	:=	(hwdma-varpgtblbase, hwdma-varenableprot, others)
$CPU\text{-}instructions$	:=	(hwcore-insnrt, hwcore-insnswitchguest, hwcore-insnsyscall, hwcore-insnsysret, hwcore-insniret, others)

Fig. 18: üSpark Architecture Model: x86 hardware-virtualization specialization

PAH-5(gp);
$\mathsf{Inv}_{\ddot{u}}^{4}$ via POC-27, POC-28, POC-29, PAH-6(gp);
$\mathsf{Inv}_{\ddot{u}}^{5}$ via POC-6, PAH-6(gp), POC-27, POC-28, POC-29, POC-5 ;
$\mathsf{Inv}_{\ddot{u}}^{6}$ via POC-35, PAH-21(dma);
$\mathsf{Inv}_{\ddot{u}}^{7}$ via POC-28, POC-29, POC-27, PAH-6(gp);
$\mathsf{Inv}_{\ddot{u}}^{8}$ via $\mathsf{Inv}_{\ddot{u}}^{7}$, $\mathsf{Inv}_{\ddot{u}}^{8}$, $\mathsf{Inv}_{\ddot{u}}^{9}$, $\mathsf{Inv}_{\ddot{u}}^{10}$, POC-18, PAH-18(ex), ;
$\mathsf{Inv}_{\ddot{u}}^{9}$ via POC-28, POC-29, POC-27, PAH-6(gp) ;
$\mathsf{Inv}_{\ddot{u}}^{10}$ via POC-21, POC-36

- State-2a: Verified hypervisor program üobjects invariants POCs/PAHs; and Unverified (trusted) hypervisor program üobjects invariants POCs/PAHs
- State-3:
  $\mathsf{Inv}_{\ddot{u}}^{3}$ via PAH-2(smp), PAH-4(gp), PAH-5(gp), PAH-17(ex), PAH-18(ex), POC-7, POC-44, POC-24, POC-9, ;
  $\mathsf{Inv}_{\ddot{u}}^{4}$ via POC-6, PAH-6(gp), POC-30, POC-15, POC-16, POC-17 ;
  $\mathsf{Inv}_{\ddot{u}}^{5}$ via POC-6, PAH-6(gp), POC-15, POC-16, POC-17 POC-5 ;
  $\mathsf{Inv}_{\ddot{u}}^{6}$ via PAH-21(dma), POC-19 ;
  $\mathsf{Inv}_{\ddot{u}}^{7}$ via PAH-2(smp), PAH-6(gp), POC-30, POC-15, POC-16, POC-17 ;
  $\mathsf{Inv}_{\ddot{u}}^{8}$ via $\mathsf{Inv}_{\ddot{u}}^{7}$, $\mathsf{Inv}_{\ddot{u}}^{8}$, $\mathsf{Inv}_{\ddot{u}}^{9}$, $\mathsf{Inv}_{\ddot{u}}^{10}$, POC-18, PAH-18(ex), ;
  $\mathsf{Inv}_{\ddot{u}}^{9}$ via PAH-2(smp), PAH-6(gp), POC-1, POC-2, POC-30, POC-15, POC-16, POC-17 ;
  $\mathsf{Inv}_{\ddot{u}}^{10}$ via PAH-2(smp), POC-20, POC-21
- State-4: Verified hypervisor program üobjects (multi-core) invariants POCs/PAHs and $\mathsf{Inv}_{\ddot{u}}^{10}$ additionally via POC-21
- State-4a: Verified hypervisor program üobjects invariants POCs/PAHs
- State-4b: Verified hypervisor program üobjects invariants POCs/PAHs; and Unverified (trusted) hypervisor program üobjects invariants POCs/PAHs
- State-5: Verified hypervisor program üobjects (multi-core) invariants POCs/PAHs and $\mathsf{Inv}_{\ddot{u}}^{10}$ additionally via POC-21
- State-5a: Verified hypervisor program üobjects invariants POCs/PAHs
- State-5b: Verified hypervisor program üobjects invariants POCs/PAHs; and Unverified (trusted) hypervisor program üobjects invariants POCs/PAHs
- State-5c: Verified hypervisor program üobjects invariants POCs/PAHs; and Unverified (trusted) hypervisor program üobjects invariants POCs/PAHs
- State-6:
  $\mathsf{Inv}_{\ddot{u}}^{3}$ via PAH-15(dpg), PAH-16(dpg), ;

`PAH-1(rt)`	:=	if `hwcore-insnrt`, all AP cores are halted; `hwcore-varsmpactivate` = false; BSP core `hwcore-varmode` != guest; BSP core `hwcore-varprivilege` = supervisor; BSP core `hwcore-varexcptblptr` = clear; BSP core `hwcore-varinterrupts` = false; system setup to restart on any hardware exception triggered by BSP core; `hwdma-varenableprot` = true; all verified and unverified üobject memory regions are DMA protected
`PAH-2(smp)`	:=	AP awakening results in AP core `hwcore-varmode` != guest; AP core `hwcore-varexcptblptr` = clear; AP core `hwcore-varinterrupts` = false; system setup to restart on any hardware exception generated by AP core
`PAH-3(smp)`	:=	h/w provides unique core id for every core in the system
`PAH-4(gp)`	:=	if `hwcore-varmode` != guest no intercepts are triggered
`PAH-5(gp)`	:=	if `hwcore-varinterrupts` == false, no interrupts are triggered
`PAH-6(gp)`	:=	if `hwcore-varpaging` == true, enforce memory protections as per memory page tables pointed to be `hwcore-varmempgtblptr`
`PAH-7(gp)`	:=	`LOCK` instruction prefix function as memory-fences.
`PAH-8(dp)`	:=	if `hwcore-varprivilege` == user, hardware prevents access to $CPU\text{-}systemcontrolstate$
`PAH-9(dp)`	:=	if `hwcore-varprivilege` == user, hardware prevents access to memory regions marked supervisor in memory page tables pointed to by `hwcore-varmempgtblptr`
`PAH-10(dp)`	:=	if `hwcore-varprivilege` == user and `hwcore-insnsysexit` hardware transfers control to the location contained in `hwcore-varsysexitep` with `hwcore-varprivilege` = user; set `hwcore-varstackptr` = `hwcore-varsysexitesp`
`PAH-11(dp)`	:=	if `hwcore-insnsyscall`, hardware transfers control the location contained in `hwcore-varsyscallep` with `hwcore-varprivilege` = supervisor; it sets `hwcore-varstackptr` = `hwcore-varsyscallesp`
`PAH-12(dpg)`	:=	if `hwcore-varmode` == guest, enforce memory protections as per memory page tables pointed to by `hwcore-varguestmempgtblptr`
`PAH-13(dpg)`	:=	if `hwcore-varmode` == guest, on intercept `hwcore-varmode` = host; `hwcore-varprivilege` = `hwcore-varguesthostpl`; `hwcore-varmempgtblptr` = `hwcore-varguesthostmempgtblptr`; and transfer control to `hwcore-varguesthostep` with `hwcore-varstackptr` = `hwcore-varguesthostesp`
`PAH-14(dpg)`	:=	if `hwcore-insnswitchguest`, hardware switches to guest mode and sets `hwcore-varmode` = guest
`PAH-15(dpg)`	:=	Hardware leaves guest mode only on an intercept
`PAH-16(dpg)`	:=	if `hwcore-varmode` == guest, prevents access to $CPU\text{-}systemcontrolstate$
`PAH-17(ex)`	:=	If `hwcore-varexcptblptr` is clear restart system on any core exceptions
`PAH-18(ex)`	:=	On exception, if `hwcore-varmode` != guest and `hwcore-varexcptblptr` is not clear and `hwcore-varprivilege` == supervisor, perform control transfer to location within the exception table pointed to by `hwcore-varexcptblptr`; setup stack frame with return-address and flags prior to control transfer.
`PAH-19(ex)`	:=	On exception, if `hwcore-varmode` != guest and `hwcore-varexcptblptr` is not clear and `hwcore-varprivilege` == user, `hwcore-varstackptr` = `hwcore-vardpexcpstackptr` and transfer control to location within exception table pointed to by `hwcore-varexcptblptr`; ;setup stack frame with return-address and flags prior to control transfer.
`PAH-20(ex)`	:=	if `hwcore-insniret` and `hwcore-varmode` != guest and `hwcore-varexcptblptr` is not clear, restore flags from stack frame and transfer control to location within stack frame; if stack frame code selector is deprivileged, then `hwcore-varprivilege` == user
`PAH-21(dma)`	:=	if `hwdma-varenableprot` == true and `hwdma-varpgtblbase` is not clear, enforce DMA protection for devices as per device page tables pointed to by `hwdma-varpgtblbase`

Fig. 19: üSpark Proof Assumptions on Hardware (PAH): `rt` = root of trust, `smp` = multi-core, `gp` = general-purpose, `dp` = deprivileged-mode, `dpg` = deprivileged guest-mode, and `ex` = exception handling.

`POC-1`	:=	CASM functions always have a constant upper bound for local stack frame size consistent with underflow/overflow guards
`POC-2`	:=	C functions always have a constant upper bound for local stack frame size consistent with underflow/overflow guards
`POC-3`	:=	C and CASM functions can only access caller stack parameters within bounds
`POC-4`	:=	can perform I/O only via dedicated CASM I/O functions
`POC-5`	:=	can access shared system memory and guest üobject memory regions only via dedicated CASM sysmemaccess functions
`POC-6`	:=	verified and unverified üobject binaries are linked into a single binary such that they are disjoint
`POC-7`	:=	`hwcore-varinterrupts` = false at all points
`POC-8`	:=	`hwcore-varmode` != guest
`POC-9`	:=	if not prime üobject, no writes to *var-excptbl*
`POC-10`	:=	if not prime üobject, no writes to `hwcore-varexcptblptr`
`POC-11`	:=	if not prime, no writes to *var-dpexcpstacks*
`POC-12`	:=	if not prime, no writes to `hwcore-vardpexcpstackptr`
`POC-13`	:=	if not prime or sentinel üobjects, no writes to `hwcore-varmempgtblptr`
`POC-14`	:=	if not prime üobject, no writes to `hwcore-varpaging`
`POC-15`	:=	if not prime üobject, no writes to verified üobject memory page-tables
`POC-16`	:=	if not prime üobject, no writes to unverified üobject memory page-tables
`POC-17`	:=	changes to unverified guest üobject page tables should be such that only üobject memory regions and MMIO of üobject allocated devices are mapped at all times
`POC-18`	:=	üobject can only invoke another üobject via the sentinel as per the blueprint.
`POC-19`	:=	if not prime üobject, no direct writes to *var-dmatable*, no writes to `hwdma-varenableprot` and `hwdma-varpgtblbase`
`POC-20`	:=	if not prime üobject, no writes to *hwcore-varsmpactivate*
`POC-21`	:=	if üobject is concurrent and not sentinel and invokes other üobjects that is sequential then: uses lock mechanism, invokes üobject and releases lock mechanism. Otherwise no locking mechanisms are used.

Fig. 20: üSpark general verified hypervisor üobjects Proof Obligations on Code (POC).

$\mathsf{Inv}_{\ddot{u}}^{4}$ via `PAH-12(dpg)`, ;
$\mathsf{Inv}_{\ddot{u}}^{5}$ via `PAH-12(dpg)`, ;
$\mathsf{Inv}_{\ddot{u}}^{6}$ via `PAH-21(dma)` `PAH-12(dpg)`, `PAH-16(dpg)`, ;
$\mathsf{Inv}_{\ddot{u}}^{7}$ via `PAH-12(dpg)`, ;
$\mathsf{Inv}_{\ddot{u}}^{8}$ via `PAH-15(dpg)`, `PAH-13(dpg)`, ;
$\mathsf{Inv}_{\ddot{u}}^{9}$ via `PAH-12(dpg)`, ;
$\mathsf{Inv}_{\ddot{u}}^{10}$ via `PAH-12(dpg)`, `PAH-16(dpg)`, `PAH-15(dpg)`,

- State-7: Verified hypervisor program üobjects (multi-core) invariants POCs/PAHs and $\mathsf{Inv}_{\ddot{u}}^{10}$ additionally via `POC-21`
- State-7a: Verified hypervisor program üobjects invariants POCs/PAHs
- State-7b: Verified hypervisor program üobjects invariants POCs/PAHs; and Unverified (trusted) hypervisor program üobjects invariants POCs/PAHs
- State-7c: Verified hypervisor program üobjects invariants POCs/PAHs; and Unverified (trusted) hypervisor program üobjects invariants POCs/PAHs
- State-8: Verified hypervisor program üobjects (multi-core) invariants POCs/PAHs and $\mathsf{Inv}_{\ddot{u}}^{10}$ additionally via `POC-21`
- State-8a: Verified hypervisor program üobjects invariants POCs/PAHs
- $\bowtie_{call}^{vh2vh}$ and $\bowtie_{ret}^{vh2vh}$:
  $\mathsf{Inv}_{\ddot{u}}^{3}$ via `POC-7`, `POC-37`, `POC-38`, `POC-44`, `POC-9`, `POC-10`; `PAH-18(ex)`, `PAH-4(gp)`, `PAH-5(gp)` ;
  $\mathsf{Inv}_{\ddot{u}}^{4}$ via `POC-15`, `POC-16`, `POC-17`, `POC-14` `PAH-6(gp)` ;
  $\mathsf{Inv}_{\ddot{u}}^{5}$ via `POC-15`, `POC-16`, `POC-17`, `POC-14` `POC-5` `POC-47` `PAH-6(gp)` ;
  $\mathsf{Inv}_{\ddot{u}}^{6}$ via `POC-19` `PAH-21(dma)` ;
  $\mathsf{Inv}_{\ddot{u}}^{7}$ via `POC-15`, `POC-16`, `POC-17`, `POC-14` `PAH-6(gp)` ;
  $\mathsf{Inv}_{\ddot{u}}^{8}$ via $\mathsf{Inv}_{\ddot{u}}^{7}$, $\mathsf{Inv}_{\ddot{u}}^{8}$, $\mathsf{Inv}_{\ddot{u}}^{9}$, $\mathsf{Inv}_{\ddot{u}}^{10}$, `POC-18`, `POC-46` ;
  $\mathsf{Inv}_{\ddot{u}}^{9}$ via `POC-11`, `POC-12`, `POC-46`, `PAH-6(gp)` ;
  $\mathsf{Inv}_{\ddot{u}}^{10}$ via `POC-20`, `POC-21`,
- $\bowtie_{call}^{vh2uh}$ and $\bowtie_{ret}^{uh2vh}$:
  $\mathsf{Inv}_{\ddot{u}}^{3}$ via `POC-7`, `POC-37`, `POC-38`, `POC-44`, `POC-9`, `POC-10`; `PAH-18(ex)`, `PAH-19(ex)`, `PAH-4(gp)`, `PAH-5(gp)` `POC-39`,
  ;
  $\mathsf{Inv}_{\ddot{u}}^{4}$ via `POC-15`, `POC-16`, `POC-17`, `POC-14`

`POC-22`	:=	setup `hwcore-varstackptr` with initial stack; load `hwcore-varmempgtblptr` with unity mapped pagetables that are setup with stack underflow/overflow guards and prime üobject code write-protect and set `hwcore-varpaging` = true; do not write to `hwcore-varpaging` thereafter; `hwcore-varmempgtblptr` is not written to until verified üobject page-tables are setup and loaded. this part of code stays within existing stack limits and does not reload any segment registers
`POC-23`	:=	populates *var-excptbl* table to have all exceptions transfer control to the sentinel; once *var-excptbl* is populated it is not written to thereafter
`POC-24`	:=	`hwcore-varexcptblptr` = *var-excptbl*; `hwcore-varexcptblptr` is not written to thereafter
`POC-25`	:=	populate `hwcore-varsyscallep` to point to sentinel
`POC-26`	:=	set up *var-uvlegiomaps* so that they only map legacy I/O ports for devices allocated to the üobject; no writes to *var-uvlegiomaps* thereafter
`POC-27`	:=	sets up page tables for unverified guest üobjects such that the üobject memory regions are marked deprivileged; verified hypervisor üobjects including prime and sentinel memory regions and unverified hypervisor üobjects are marked not-present; there is one-to-one mapping between virtual and physical memory; unverified guest üobject allocated device MMIO regions are mapped present and read-write; *var-uvlegiomaps* for üobject is mapped present and read-write; no writes to unverified guest üobject page tables thereafter
`POC-28`	:=	sets up page tables for each unverified üobject such that the üobject memory regions are marked deprivileged; verified üobject, sentinel memory regions are marked supervisor; other unverified üobjects are marked not-present; there is one-to-one mapping between virtual and physical memory; üobject code regions are marked read-only; unverified guest üobject, verified üobject and unverified üobject memory regions are disjoint; unverified guest üobject and unverified üobject allocated device MMIO regions are mapped; *var-uvlegiomaps* for üobject is mapped; sets up stack underflow/overflow guards for disjoint BSP and AP *var-dpexcpstacks* and mark them supervisor; no writes to unverified üobject page tables thereafter
`POC-29`	:=	load `hwcore-varmempgtblptr` with verified üobject page tables such that verified üobject memory regions and sentinel are marked supervisor; unverified üobject memory regions are marked deprivileged; there is one-to-one mapping between virtual and physical memory; üobject code regions are marked read-only; unverified guest üobject, verified üobject and unverified üobject memory regions are disjoint; sets up stack underflow/overflow guards for disjoint BSP and AP *var-stacks* and *var-dpexcpstacks* and mark them supervisor; no writes to verified üobject page tables thereafter
`POC-30`	:=	sets up `hwcore-varstackptr` with *var-stacks*[`hwcore-varid`]; load `hwcore-varmempgtblptr` with verified üobject memory page tables and set `hwcore-varpaging` = true; do not write to `hwcore-varpaging` thereafter;
`POC-31`	:=	loads `hwcore-vardpexcpstackptr` for APs with *var-dpapexcpstacks*[`hwcore-varid`]
`POC-32`	:=	sets up `hwcore-varguesthostep` to point to sentinel
`POC-33`	:=	sets up `hwcore-varguesthostmempgtblptr` to verified üobject page table
`POC-34`	:=	loads `hwcore-dpexcpstackptr` for BSP with *var-dpapexcpstacks*[`hwcore-varid` for BSP]
`POC-35`	:=	establish *var-dmatable* such that only üobject allocated devices can do DMA only to üobject dmadata regions and set `hwdma-varpgtblbase` to *var-dmatable* before activating DMA protection by setting `hwdma-varenableprot` to true
`POC-36`	:=	awaken application processors by setting `hwcore-varsmpactivate` = true; don't touch `hwcore-varsmpactivate` thereafter

Fig. 21: üSpark prime verified hypervisor üobject specific Proof Obligations on Code (POC).

POC-37	:=	transfer control to *exception* üobject on any exception
POC-38	:=	ensures no exceptions due to its own code
POC-39	:=	saves current hwcore-varstackptr into hwcore-varsyscallesp register and switches to deprivileged mode via hwcore-insnsysret to transfer control to unverified hypervisor üobject
POC-40	:=	loads unverified hypervisor üobject page tables into hwcore-varmempgtblptr before handing control to unverified hypervisor üobject
POC-41	:=	uses hwcore-insnswitchguest to switch to guest mode only before starting an unverified guest üobject
POC-42	:=	loads guest page tables into hwcore-varguestmempgtblptr before handing control to guest üobject
POC-43	:=	saves hwcore-varstackptr into hwcore-varguesthostesp before transfering control to guest üobject
POC-44	:=	hwcore-varmode != guest except when handing control to guest üobject
POC-45	:=	only write to hwcore-varmempgtblptr to load correct unverified üobject page tables prior to executing corresponding unverified üobject
POC-46	:=	preserve üobject-to-üobject call semantics: register preservation, callee stack frame preservation and paired call and returns
POC-47	:=	do bounded parameter marshalling for dp-call and dp-ret

Fig. 22: üSpark sentinel verified hypervisor üobject specific Proof Obligations on Code (POC).

PAH-6(gp) POC-39, POC-40, POC-45 PAH-8(dp), PAH-9(dp), PAH-10(dp), PAH-11(dp) ;
$\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 POC-5 POC-47 PAH-6(gp) ;
$\mathsf{Inv}^{6}_{\ddot{u}}$ via POC-19 PAH-21(dma) ;
$\mathsf{Inv}^{7}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) POC-39, POC-40, POC-45 PAH-8(dp), PAH-9(dp), PAH-10(dp), PAH-11(dp) ;
$\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, POC-46, POC-18 ;
$\mathsf{Inv}^{9}_{\ddot{u}}$ via POC-11, POC-12, POC-46, PAH-6(gp) POC-39, POC-40, POC-45, POC-47, PAH-8(dp), PAH-9(dp), PAH-10(dp), PAH-11(dp) ;
$\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-20, POC-21,

- $\bowtie^{uh2vh}_{call}$ and $\bowtie^{vh2uh}_{ret}$:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via POC-7, POC-37, POC-38, POC-44, POC-9, POC-10; PAH-18(ex), PAH-19(ex), PAH-4(gp), PAH-5(gp) ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) POC-39, POC-40, POC-45 PAH-8(dp), PAH-9(dp), PAH-10(dp), PAH-11(dp) ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 POC-5 POC-47 PAH-6(gp) ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via POC-19 PAH-21(dma) ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) POC-39, POC-40, POC-45 PAH-8(dp), PAH-9(dp), PAH-10(dp), PAH-11(dp) ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, POC-46, POC-18 ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via POC-11, POC-12, POC-46, PAH-6(gp) POC-39, POC-40, POC-45, POC-47, PAH-8(dp), PAH-9(dp), PAH-10(dp), PAH-11(dp) ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-20, POC-21,
- $\bowtie^{vh2ug}_{call}$:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via POC-7, POC-37, POC-38, POC-44, POC-9, POC-10; PAH-18(ex), PAH-4(gp), PAH-5(gp) POC-41 PAH-12(dpg), PAH-14(dpg), PAH-16(dpg) ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) POC-41, POC-42 PAH-12(dpg), PAH-14(dpg), PAH-16(dpg) ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 POC-5 POC-47 PAH-6(gp) ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via POC-19 PAH-21(dma) ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) POC-41, POC-42 PAH-12(dpg), PAH-14(dpg), PAH-16(dpg) ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, POC-46, POC-18, ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via POC-11, POC-12, POC-46, PAH-6(gp) POC-43 PAH-12(dpg), PAH-14(dpg), PAH-16(dpg) ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-20, POC-21,
- $\bowtie^{ug2vh}_{call}$:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via POC-7, POC-37, POC-38, POC-44, POC-9, POC-10; PAH-18(ex), PAH-4(gp), PAH-5(gp) PAH-15(dpg), PAH-13(dpg) ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) PAH-15(dpg), PAH-13(dpg) ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 POC-5 POC-47 PAH-6(gp) ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via POC-19 PAH-21(dma) ;

$\mathsf{Inv}^{7}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) PAH-15(dpg), PAH-13(dpg) ;
$\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, POC-46, POC-18 ;
$\mathsf{Inv}^{9}_{\ddot{u}}$ via POC-11, POC-12, POC-46, PAH-6(gp) PAH-15(dpg), PAH-13(dpg) ;
$\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-20, POC-21,

- $\bowtie^{excp}_{call}$:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via POC-7, POC-37, POC-38, POC-44, POC-9, POC-10; PAH-18(ex), PAH-19(ex), PAH-4(gp), PAH-5(gp) ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 POC-5 POC-47 PAH-6(gp) ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via POC-19 PAH-21(dma) ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, POC-46, POC-18 ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via POC-11, POC-12, POC-46, PAH-6(gp) ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-20, POC-21,
- $\bowtie^{excp}_{ret}$:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via POC-7, POC-37, POC-38, POC-44, POC-9, POC-10; PAH-18(ex), PAH-19(ex), PAH-4(gp), PAH-5(gp) ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 POC-5 POC-47 PAH-6(gp) ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via POC-19 PAH-21(dma) ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via POC-15, POC-16, POC-17, POC-14 PAH-6(gp) ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, POC-46, POC-18, PAH-20(ex) ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via POC-11, POC-12, POC-46, PAH-6(gp) ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-20, POC-21,
- Verified hypervisor program üobjects:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via PAH-4(gp), PAH-5(gp), PAH-18(ex), POC-7, POC-9, POC-10, POC-44, ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via PAH-6(gp), POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-5, PAH-6(gp), POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via PAH-21(dma), POC-19 ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via PAH-6(gp), POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, PAH-18(ex), POC-9, POC-10, POC-18 ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via PAH-6(gp), POC-1, POC-2, POC-11, POC-12 POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-21, POC-20,
- Verified hypervisor program üobjects (multi-core):
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via PAH-4(gp), PAH-5(gp), PAH-18(ex), POC-7, POC-9, POC-10, POC-44, ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via PAH-6(gp), POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via POC-5, PAH-6(gp), POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via PAH-21(dma), POC-19 ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via PAH-6(gp), POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via $\mathsf{Inv}^{7}_{\ddot{u}}$, $\mathsf{Inv}^{8}_{\ddot{u}}$, $\mathsf{Inv}^{9}_{\ddot{u}}$, $\mathsf{Inv}^{10}_{\ddot{u}}$, PAH-18(ex), POC-9, POC-10, POC-18 ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via PAH-6(gp), POC-1, POC-2, POC-11, POC-12 POC-15, POC-16, POC-17, POC-13, POC-14 ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via POC-20, POC-21,
- Unverified (trusted) hypervisor program üobjects:
  $\mathsf{Inv}^{3}_{\ddot{u}}$ via PAH-4(gp), PAH-5(gp), PAH-8(dp), PAH-19(ex) ;
  $\mathsf{Inv}^{4}_{\ddot{u}}$ via PAH-6(gp), PAH-8(dp), PAH-9(dp), PAH-6(gp) ;
  $\mathsf{Inv}^{5}_{\ddot{u}}$ via PAH-6(gp), PAH-8(dp), PAH-9(dp) ;
  $\mathsf{Inv}^{6}_{\ddot{u}}$ via PAH-21(dma) PAH-8(dp), PAH-9(dp), PAH-6(gp) ;
  $\mathsf{Inv}^{7}_{\ddot{u}}$ via PAH-6(gp), PAH-8(dp), PAH-9(dp), PAH-6(gp) ;
  $\mathsf{Inv}^{8}_{\ddot{u}}$ via PAH-11(dp), PAH-19(ex), PAH-8(dp), PAH-9(dp), PAH-6(gp) ;
  $\mathsf{Inv}^{9}_{\ddot{u}}$ via PAH-11(dp), PAH-8(dp), PAH-9(dp), PAH-6(gp) ;
  $\mathsf{Inv}^{10}_{\ddot{u}}$ via PAH-8(dp), PAH-9(dp), PAH-6(gp)