überSpark†: Practical, Provable, End-to-End Guarantees on Commodity Heterogenous Interconnected Computing Platforms

Amit Vasudevan
SEI/Carnegie Mellon University
amitvasudevan@acm.org

Petros Maniatis
Google Research
maniatis@google.com

Ruben Martins
CSD/Carnegie Mellon University
rubenm@andrew.cmu.edu

Abstract

Today's computing ecosystem, comprising commodity heterogeneous interconnected computing (CHIC) platforms, is increasingly being employed for critical applications, consequently demanding fairly strong end-to-end assurances. However, the generality and system complexity of today's CHIC stack seem to outpace existing tools and methodologies towards provable end-to-end guarantees. This paper describes our on-going research, and presents **überSpark**†, a system architecture that argues for structuring the CHIC stack around *Universal Object Abstractions* (üobjects), a fundamental system abstraction and building block towards practical and provable end-to-end guarantees.

überSpark is designed to be realizable on heterogeneous hardware platforms with disparate capabilities, and facilitates compositional end-to-end reasoning and efficient implementation. **überSpark** also supports the use of multiple verification techniques towards properties of different flavors, for development compatible, incremental verification, co-existing and meshing with unverified components, at a fine granularity, and wide applicability to all layers of the CHIC stack.

We discuss the CHIC stack challenges, illustrate our design decisions, describe the **überSpark** architecture, present our foundational steps, and outline on-going and future research activities. We anticipate **überSpark** to *retrofit* and unlock a wide range of unprecedented end-to-end provable guarantees on today's continuously evolving CHIC stack.

1 Introduction

Today's commodity heterogeneous interconnected computing (CHIC) platforms encompass laptops, mobile phones, IoT devices, robots, drones, and self-driving cars. Such platforms redefine the way we interact not only for convenience (e.g., regulating home temperature and lighting, ordering groceries) but increasingly for critical applications as well (e.g., autonomous driving, home security, health care). Consequently, the CHIC stack demands fairly strong end-to-end guarantees for security, correctness, and timeliness.

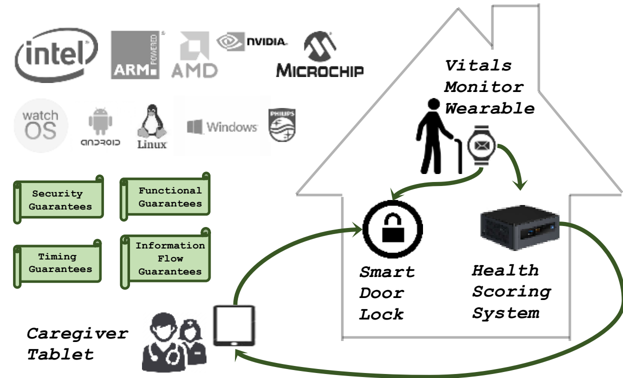

Figure 1. *ElderSafe*, a hypothetical CHIC service for elders who live in an assisted-care facility. *ElderSafe* illustrates an exemplar CHIC stack application comprising heterogeneous hardware, software, and properties, that make achieving *practical and provable* end-to-end guarantees on the CHIC stack very challenging.

Formal verification is a powerful tool for realizing provable guarantees. However, although progress in system verification technology is gaining momentum [26, 39, 42, 53, 60, 73, 109, 110, 124], the system complexity of today's CHIC stack (with a plethora of heterogeneous platforms, configurations, and interactions) is rapidly outpacing the arsenal of current verification tools, most of which focus solely on specific styles of properties and verification methodologies (§8). Meanwhile, competitive markets with low cost of entry, little regulation, and no liability will continue to produce innovative, attractively priced, continuously evolving interconnected computing platforms comprising diverse-origin and disparate hardware and *large* (untrustworthy) software components. This makes achieving *practical and provable* end-to-end guarantees on the CHIC stack very challenging.

1.1 Running Example

Consider *ElderSafe*, a hypothetical CHIC service for elders who live in an assisted-care facility. For the purposes of

†In the fictional Transformers universe, the AllSpark is a powerful object capable of creating a new Transformer by retrofitting ordinary machinery with Sparks – the building blocks of a Transformer. In a similar vein, **überSpark** (**https://uberspark.org**) retrofits the ordinary, commodity, heterogenous, and interconnected computing platform stack with universal, verifiable objects (überobjects or üobjects) towards practical, provable, end-to-end guarantees.

this running example, we limit *ElderSafe* to a simple task: collect readings of vital signs from the subject via a wrist sensor, analyze those readings to classify them as normal or abnormal (requiring assistance), and, in the latter case, authorize staff to unlock the door and assist even if the door is locked (Fig. 1).

We consider an illustrative, concrete ensemble of devices providing the *ElderSafe* service: a wearable vitals monitor (e.g., FDA-approved Caretaker [20] or KardiaBand [2]), an automated early-warning health-scoring system (e.g., Philips IntelliVue Guardian [97]) utilizing activity recognition models [96] (e.g., to predict the likelihood of mortality [100]), an event-management system that runs on the caregiver phone or tablet (e.g., Intellivue App [98]), and an off-the-shelf smart lock (e.g., by Schlage [120]) on the subject's entry door.

ElderSafe is one of the services offered in a typical, smart semi-independent assisted-care facility today (e.g., Jewish Senior Life [92]). To be useful, *ElderSafe* must provide a number of guarantees of different types, some of which are:

Functional guarantees: unlock the door only when an emergency is detected;

Security guarantees: only authorized staff/nurses can unlock the door in a detected emergency; only the subject's wearable can trigger unlocking of that subject's door; only genuine instances of the wearable device operating under a known configuration can trigger unlocking of the door;

Timing guarantees: worst-case response time between detection of emergency and responsive action of unlocking the door is under 5 minutes;

Information-flow guarantees: no information about unrelated readings of the wearable, or the subject's comings and goings detected via the door lock, can leak to the staff or the outside world.

Such guarantees, strongly enforced, are essential to ensure deployability, regulatory compliance, and to save lives.

Note that, although *ElderSafe* is of limited complexity, it serves to illustrate a typical CHIC stack application comprising heterogeneous hardware, software, and properties.

2 Design Goals for CHIC Guarantees

We strive for the following goals while realizing *ElderSafe*:

▶ *Provable End-to-End Guarantees* – Produce a service where guarantees are formally verifiable, end-to-end, with machine-checkable proofs of those guarantees on the software implementation running on top of the actual CHIC platform hardware.

▶ *Practicality* – The verification overhead of *ElderSafe* should be minimal both from a construction-time and runtime perspective. For example, taking 15 person years only to verify one specific instance of Eldersafe would not be cost-effective. Similarly, having a verified Eldersafe that is too slow to detect emergencies and ensure timely intervention would be impractical. Our solution must be development compatible (evolvable with iterative versions), power-efficient, and performant to be practical.

▶ *Implementation Generality* – Our implementation should use existing components to the extent possible, rather than building new hardware, implementing new software on top of it, and mounting a new verification effort. All of the aforementioned activities are very expensive, and *ElderSafe* can be cost effective only reusing existing products when possible.

3 **überSpark** – Genesis

In this section, we derive the **überSpark** architecture, the main contribution of this paper, via a progressive design exercise for *ElderSafe* (see Table 1 for a summary).

Our objective is to describe the CHIC stack challenges, illustrate the design decisions that satisfy our goals (§2), and motivate the main ideas behind **überSpark**, before presenting it formally in §4.

3.1 Single Monolithic Verified System

We can try to implement *ElderSafe* in a single language and code base (e.g, C), specify properties (e.g., via ACSL Hoare clauses [46]), prove them mechanically (e.g., using a C verification framework such as Frama-C [72]), extract runtime binaries (e.g., via the CompCert certified C compiler [16]) and deploy.

Are we there yet? Unfortunately, this is infeasible for *ElderSafe*. By necessity, the vitals monitor, the health-scoring platform, the smart lock, and the caregiver tablet are distinct physical hardware platforms. We certainly can't physically tether our elderly subject to the door lock, or to the caregivers. So our solution must accommodate distribution, inducing an additional design goal:

▶ *Hardware Distribution* – Ensure that the service can accommodate physically distinct hardware components and be meaningfully distributed across them.

Further, our service comprises disparate hardware architectures: x86 for the health-scoring platform, ARM for the vitals monitor, and a PIC18 microcontroller unit (MCU) for the smart lock, and they each have different functional characteristics and capabilities. This implies an additional goal:

▶ *Hardware Heterogeneity* – The service should be able to accommodate disparate hardware architectures across its components.

3.2 Distributed System of Monolithic Blobs

In order to accommodate physically-distinct hardware components and disparate hardware, we can make *ElderSafe* a distributed system, to have one component per distinct physical hardware platform. We can then prove properties on individual components (e.g., with Frama-C), and then prove

Design Rationale	Design Goals Generated	Design Features
Domain Requirements (§1.1; §2)	***Provable End-to-End Guarantees*** ***Practicality*** ***Implementation Generality***	–
Single Monolithic Verified System (§3.1)	*Hardware Distribution* *Hardware Heterogeneity*	–
Distributed System of Monolithic Blobs (§3.2)	*Modularity and Layering*	üobjects (§4.1)
Components as Collections of Objects (§3.3)	*Software Heterogeneity*	üobject Collections (§4.2)
Hybrid Isolation (§3.4)	*Resource Closure*	üobject Resource Interface Confinement (§4.3);
Resource Interface Confinement (§3.5)	*Verification Bridging*	üobject Instantiation and Execution (§4.4); and üobject Interactions (§4.5)
Verification Bridges (§3.6)	*Running Correct Object Collections* *Allowing Correct Platforms*	üobject Verification Bridge (§4.6)
Attestation and Authentication (§3.7)	–	üobject Reporting (§4.7)

Table 1. `überSpark` architecture derivation via a progressive design exercise for *ElderSafe*. For each design rationale, we present the design goals generated by each iteration of the design excercise, and the corresponding design features that address those goals. The shaded cells represent the candidate design features that together make up the `überSpark` architectural components.

end-to-end guarantees about the distributed system (e.g., via protocol verification on top of TLA+ [76] or Dafny [60]).

Are we there yet? Each monolithic blob is too big to verify in one piece. The size limit for a single verifiable program varies by methodology, but it is anecdotally estimated to be in the tens of thousands of lines of high-level language code (e.g., C) [60, 124].

In contrast, the Linux kernel alone is several millions of lines of code [31], and an analytics application on top of it runs in the hundreds of thousands of lines of code, when math and machine-learning libraries are included.

Even going beyond size, different functionalities are often proven in isolation, and then composed to produce a unified guarantee about a system, for scalability and complexity purposes [52].

For example, the guarantees for the firmware are very different from those for the OS kernel or hypervisor, the health-scoring analytics engine, or the sensing module of the vitals monitor. Especially for security, isolation of components from each other can be of paramount importance.

Finally, especially for the purposes of *ElderSafe*, some functionality touches on different hardware capabilities that may require additional care.

For example, the health-scoring application may have libraries to speed up computations on a GPU; it would be infeasible to verify that the GPU remains in some good state for every one of the millions of lines of code involved in the health-scoring application libraries.

Therefore, focusing on a single software module to provide those additional capabilities may be essential for verification. Thus, an additional goal emerges:

▶ *Modularity and Layering* – The service should be able to decompose its components into separate modules, even within software on the same hardware platform.

3.3 Components as Collections of Objects

Modularity and layering can be achieved by breaking a monolithic blob into a collection of functional, verifiable objects within a platform.

For example, on the *ElderSafe* health-scoring platform, the health-scoring analytics engine and the caregiver tablet signaling interface are broken into verifiable object collections.

We can then isolate memory of objects from each other within a collection, enforce control-flow integrity, prove properties about individual objects (e.g., health-scoring analytics returning a result within a worst-case execution time), and prove properties about the composition of objects on a component (e.g., health score computed and transmitted to the caregiver tablet from the health scoring platform).

Are we there yet? Modularization and isolation typically involve refactoring code and leveraging hardware capabilities as needed (e.g., de-privileging, virtualization).

While some *ElderSafe* software components such as OS kernels are open-source and use open-source development tool-chains (e.g., Linux), other components such as BIOS and firmware use proprietary tool-chains and are binary-only (e.g., Schlage lock firmware [95]).

Further, these software components are often independently developed by developers with different pedigree, and are either completely opaque (e.g., Schlage lock firmware [95]) or have partial API visibility (e.g., health-scoring applet in Intellivue [44, 45]). This makes achieving modularity and isolation a challenge, and implies an additional goal:

► *Software Heterogeneity* – The service should accommodate diverse-origin software components and make it possible to ensure all other goals in the presence of partial visibility and dubious engineering practices.

To make matters worse, a given hardware platform can have a wide range of capabilities in support of modularity and isolation (e.g., x86 and ARM processors with and without hardware virtualization, Intel x86 with SGX [63, 91] and ARM Trustzone [30] to contain parts of sensitive data processing, or PIC18 MCU with a single privileged address space and built-in memory).

This further motivates the goal of accommodating disparate hardware architectures (hardware diversity), but especially with respect to functionally-divergent capabilities.

3.4 Hybrid Isolation

We can achieve modularity and layering while accommodating diverse-origin software and hardware with different capabilities via *hybrid* isolation.

We can isolate collections via verification when having visible, verifiable objects in both source and binary (e.g., SFI [105]). We can use hardware capabilities (e.g., deprivileging) to isolate collections from other untrusted components or trusted but unverified objects.

For example, the vitals-monitor application is split into a collection of verifiable objects and isolated from the untrusted OS within the wearable.

Are we there yet? Unfortunately, this is still insufficient for *ElderSafe*. Some software objects do more than what we need them for (e.g., the vitals monitor app includes a GUI subsystem [84] when all we really care about is the periodic transmission of vital sensor readings).

Further, a component contributing to an end-to-end guarantee can be characterized by a function (e.g., sensor value read within the vitals sensor driver), collection of functions (e.g., SSL library), thread (e.g., smart door lock control), process (e.g., caregiver application) or a VM (e.g., health-score analytics engine) that in turn interacts with other unverified components for their functionality (e.g., driver relying on OS kernel support functions).

Achieving efficient hybrid isolation in this multi-granular execution environment is challenging. This implies an additional goal:

► *Resource Closure* – Resources contributing towards an end-to-end guarantee should be encapsulated within an object at a given granularity (e.g., function, driver, process) with specified interfaces.

3.5 Resource Interface Confinement

We can define a use policy for a verifiable object that consists of a specific entry point and resources the object is allowed to modify. We can then prove properties on the object and resource closure (e.g., via ACSL and Frama-C as before) while isolating everything else via hybrid isolation (§3.4).

For example, the vital sensor hardware can be encapsulated by an object within the vital sensor application stack and isolated from the OS.

Are we there yet? Unfortunately, there are multiple object collections encapsulating resources, existing on different hardware platforms, which together achieve an end-to-end guarantee (e.g., vitals monitor, health-scoring analytics, caregiver application and smart lock all contribute towards the guarantee of unlocking the door only when an emergency is detected).

Further, these objects can be implemented in different languages (e.g., Java and C for caregiver application, binary assembly for smart lock firmware), and provide different flavors of properties (e.g., functional, timing; cf. §1.1), which in turn requires different verification tools and methodologies (e.g., Frama-C for C [124], Krakatoa for Java [43], Dafny [18] and ProVerif/CryptoVerif [15] for crypto, TLA+ for protocol verification [76]).

Furthermore, using different verification tools often implies different formalizations of hardware environment assumptions such as memory, concurrency and interrupts. For instance, Frama-C only models memory as bytes. Thus, an additional goal emerges:

► *Verification Bridging* – Verification tools and methodologies should be bridged to connect with each other soundly, and help prove and compose properties of different flavors on objects running on different hardware environments.

3.6 Verification Bridges

We can enforce strict execution entry and exit points for an object and extract a sound high-level sequential execution abstraction, that is consistent with hardware environment assumptions such as concurrency, pre-emption and memory ordering, for composing objects.

Different verification tools and methodologies can be bridged via intermediate verification layers (e.g., Why3 [85], Boogie [10]), with hardware environment details (e.g., memory model, instructions, and device interfaces and semantics) tied in, and invariants and properties proven at the intermediate layers.

Are we there yet? Attackers can compromise the unverified portions of the platform software stack preventing verified

objects from executing in the first place (e.g., smart lock firmware hijacks [95]).

Further, an adversary can spoof a device altogether (e.g., unlock the door via a spoofed wearable). This implies two additional goals:

▶ *Running Correct Object Collections* – Ensure that platforms are running the correct (verified) object collections.

▶ *Allowing Correct Platforms* – Ensure that the correct platform is participating in the protocol.

3.7 Attestation and Authentication

We can employ platform static and/or dynamic root of trust [34] in conjunction with code attestation [90, 125] and/or property-based attestation [21, 75] to ensure that platforms are running the required (verified) object collections.

Similarly, physical authentication can be leveraged to determine the authenticity of participating platforms [80, 117]. ***Are we there yet?*** **Yes!** We have generated no new obligations towards achieving our goals (§2).

3.8 The Road to *ElderSafe*

Table 1 reviews which design goals were foundational requirements, and which were generated by each iteration of the design exercise.

At this point, we seem to have satisfied our original three goals of provable end-to-end guarantees, practicality and implementation generality (§2), as well as the secondary goals we generated along the way.

Specifically, we adapt to heterogeneous hardware, by splitting the system into per-hardware-device collections, and the verification bridge enables composition of verified collections across hardware differences. We support heterogeneous white-box and black-box software via hybrid isolation (through inspection and verification or via hardware confinement); resource interface confinement enables us to limit the scope of big blobs of supplied software only to the functionality we care to incorporate (and prove guarantees about).

The verification bridge, besides helping with heterogeneous hardware and their disparate capabilities, also enables the bridging across verification disciplines that tackle different types of properties.

Generality and practicality are delivered by a combination of our handling of existing heterogeneous software and hardware, attestation and authentication for trustworthy reporting of verified collections, as well as by our use of language-based and verification-based isolation, rather than solely using the hammer of hardware de-privileging.

Intuitively, the combination of the aforementioned characteristics and challenges that made *ElderSafe* tricky and entailed the different steps in our design genesis naturally generalize to a broad spectrum of today's burgeoning CHIC stack applications: smart-homes, healthcare, smart-grid, autonomous drone deliveries, self-driving cars.

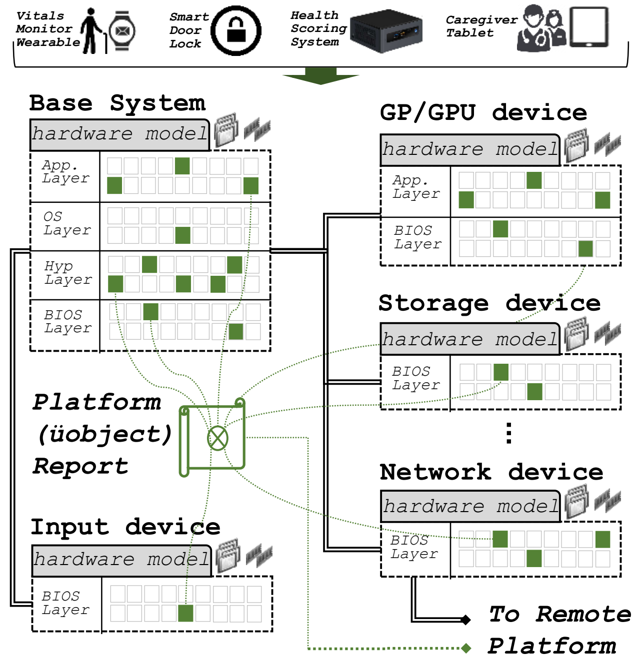

Figure 2. The **überSpark** architecture embraces a novel *micro-multi-kernel* design paradigm, towards practical and provable end-to-end guarantees on CHIC platforms. Every node of the CHIC platform stack is decomposed into (legacy) unverified components and a collection of protected, verifiable, and reportable üobjects (shaded blocks), that retrofit with the unverified components incrementally, at a fine granularity, and wide applicability to all layers of the CHIC stack.

4 überSpark – System Architecture

We now describe the **überSpark** system architecture (see Fig. 2) distilling from our design goals (§2) and constructive design effort (§3).

The **überSpark** architecture embraces a novel *micro-multi-kernel* design paradigm, towards practical and provable end-to-end guarantees on CHIC platforms. We draw from both micro-kernel support for component cohesiveness [73, 82, 115] and multi-kernel support for heterogeneous hardware [12, 38, 119], but with an emphasis on incremental and compositional end-to-end reasoning and efficient implementation.

At a high level, **überSpark** divides every node of the CHIC platform stack into legacy, unverified components and a collection of protected, verifiable üobjects (see Fig. 2 and § 4.1). The **überSpark** architecture is designed to support the use of multiple verification techniques towards properties of different flavors, for development compatible, incremental verification, co-existing and meshing with unverified components, at a fine granularity, and wide applicability to all layers of the CHIC stack.

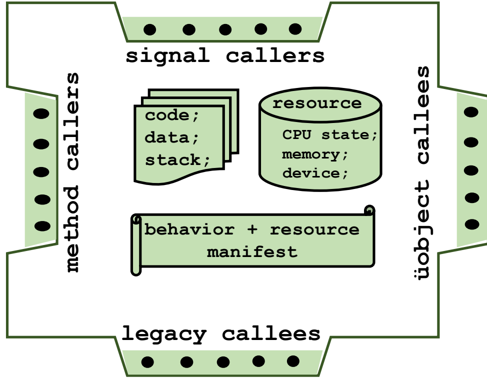

Figure 3. The logical building block of **überSpark**, the üobject provides a design/development-time singleton object abstraction guarding exclusive indivisible system resources and supporting shared-memory concurrency and linearizability. üobjects provide principled `call-return` interfacing for entry, interruption, legacy code invocations and other üobject invocations, enabling fine-granularity meshing with various `CHIC` stack programming idioms, while at the same time facilitating assume-guarantee style reasoning and composition in the presence of multi-threaded executions.

4.1 üobjects

The logical building block of **überSpark**, an üobject, is a singleton object guarding some exclusive, indivisible resources such as CPU state and registers, memory, and hardware conduits (hardware signaling and data transfer such as device endpoints, DMA, Mailboxes, etc.).

An üobject implements *method callers* to access the resources it guards. Method callers are essentially regular function signatures, along with an access-control list (ACL) on allowed callers (§4.5.1). See Fig. 3 for an illustration. In addition, an üobject also implements *signal callers* to handle signals (interrupts, exceptions, traps, etc.; §4.4), and *legacy callees* and *üobject callees* for principled invocation of legacy, unverified components and other üobjects respectively (§4.5).

An üobject is accompanied by a *use manifest*. This consists of a *resource specification* (§4.3), and an additional formal *behavior specification* [58] of its own method and signal callers, which guarantee that if some assumptions are satisfied in how a method or signal caller is invoked, then a property on the return values is guaranteed to hold upon return of that method or signal, without mention of internal üobject state.

Logical isolation of üobjects may be enforced via typical OS and micro-kernel containers. Such external enforcement might be necessary for üobjects running in different address spaces. However, formally-verified üobjects running in the same address space need no such external enforcement; they enjoy the same isolation, enforced via machine-checked proofs. This helps us achieve the sweet spot with both high performance (there is no hardware de-privileging or message-passing overheads) and compositional verification (üobjects can be verified separately), even in the presence of other unverifiable (and unavoidable) legacy components.

4.2 üobject Collections

An üobject collection is a set of üobjects that share a common memory address space. Collections are bridged via hardware conduits (hardware pathways for signaling and data transfer, e.g., DMA, memory-mapped I/O, etc) or sentinels (§4.5.1).

A special set of üobjects, called *primes*, are responsible for instantiating üobject collections on a given platform and/or a memory address space (§4.4). See Fig 4 for an illustration.

In principle, üobject collections can also be nested, modulo the hardware providing necessary conduits, e.g., guest OS üobject collection inside a hardware VM within the base system collection in Fig 2.

4.2.1 Hardware Model

Every üobject collection also has an associated hardware model formalizing the CPU and modeling the memory and associated hardware conduit end-points. The hardware model is crucial for verifying properties over collections of hardware state (e.g., state of CPU registers and memory) and assertions that are part of the üobject contract within and across üobject collections,

We envision a modular and layered hardware model where only the required subset of the hardware is modeled and used during verification, e.g., the hardware model for an Intel SGX-backed [91] üobject is simpler than an üobject executing with hypervisor privileges. This greatly aids verification automation and facilitates validation of the hardware model against real hardware [87, 103].

We further advocate for the hardware model to be specified in an abstract specification language (e.g., UML-B [114]) which can then be automatically synthesized down to desired target languages such as C, Java and Coq [9, 79, 121]. This allows the hardware model to be more readily integrated into existing verification toolchains and methodologies that could be employed to verify an üobject (§5.2).

üobject collections thus abstract heterogeneous hardware platforms, allowing each collection (along with its üobjects) to be verified separately down to their hardware states while allowing composition of such verified properties across collections.

4.3 üobject Resource Interface Confinement

Every üobject includes a *resource specification* within its manifest that describes possibly sensitive resources that it may access (e.g., code, data, stack, global system data, CPU state and collection hardware conduit end-points).

üobjects are held to their resource specification via a combination of hardware and/or software mechanisms.

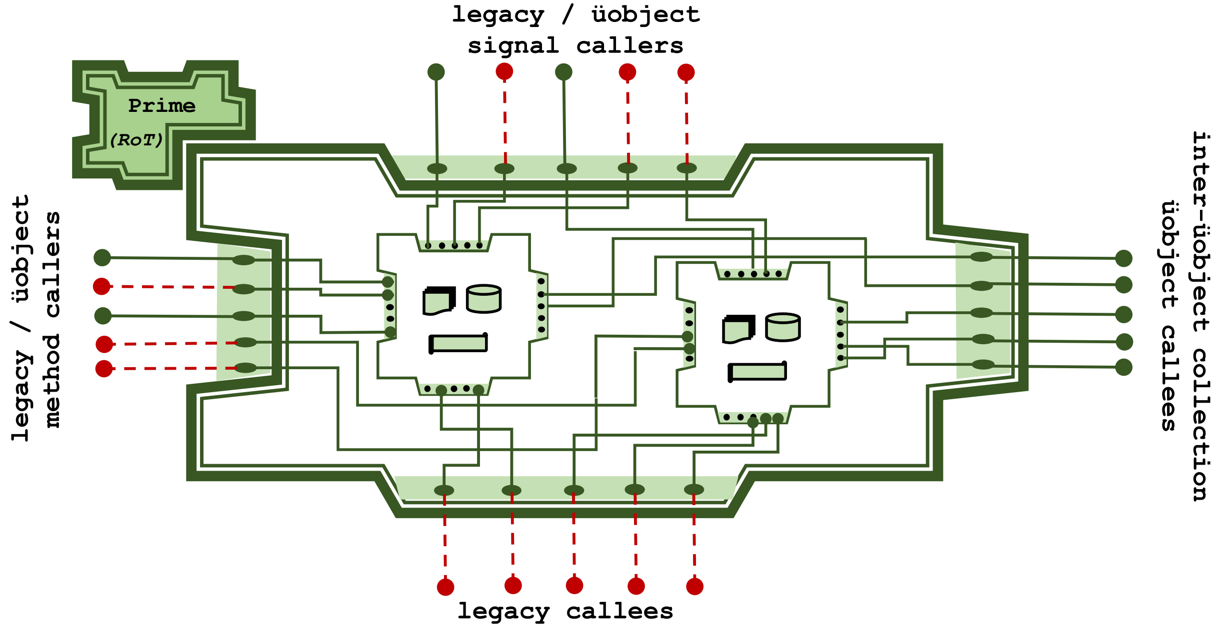

Figure 4. **überSpark** üobject collections are runtime abstractions that comprise a set of üobjects sharing a common memory address space within a given CHIC stack layer. Collections are boot-strapped by a special set of üobjects called *primes*, that form the CHIC platform root-of-trust entities. üobject collections are bridged via *sentinels* (solid line segments), abstractions that enforce call routings, enable logical privilege separation, and üobject caller/callee mediation, both within and across üobject collections, and legacy component invocations (dotted line segments), while permitting flexible implementations.

überSpark can employ the üobject collection hardware model (§4.2.1) identifying CPU interfaces to üobject resources (e.g., designated instructions) and software verification to ensure that access to those interfaces respects the üobject's manifest.

Alternatively, hardware mechanisms (e.g., MMU, privilege protections) and/or binary manipulations (e.g., SFI [105]), can be leveraged to hold üobjects to their resource specification.

überSpark resource interface confinement thus supports shared memory concurrency and linearizability by allowing distinct system resources to be: (a) managed by designated üobjects, (b) protected from access by unauthorized üobjects or legacy components, and (c) regulated in their invocation via method callers by authorized client üobjects or legacy components (Fig. 3 and Fig. 4).

The aforementioned capabilities enabled by üobject resource interface confinement, in conjunction with üobject execution and interaction mechanisms (§4.4 and §4.5) facilitate assume-guarantee style reasoning and composition of verified properties on the CHIC stack, while allowing efficient multi-threaded executions.

4.4 üobject Instantiation and Execution

An üobject can be statically or dynamically instantiated. A special collection of üobjects, called *prime*, is responsible for boot-strapping üobject execution within a given üobject collection (see Figure 4; cf. §4.7).

Primes[1] can employ different isolation mechanisms such as software fault isolation [105] and hardware-assisted containerization [7, 32, 33] to instantiate üobject collections in a protected manner. Primes also initialize the üobject collection CPUs, operating stacks, and policies before kick-starting üobject interactions.

An üobject may be concurrent or sequential. **überSpark** decouples execution threads from execution domains, i.e., an execution trace can span multiple üobjects and across multiple collections.

üobjects can also incur hardware signals such as traps, exceptions, or interrupts. In such cases, hardware capabilities are employed to save the current üobject state before handling the signal, either within the source üobject (via signal callers; § 4.1) or another üobject by employing sentinels (§4.5.1). Once the signal is processed, the source üobject is resumed once again via sentinels.

These design choices enable abstracting concurrent and asynchronous üobject executions as sequential interleavings facilitating verification (e.g., contextual refinement [53]), while supporting the use of commodity signal and threading mechanisms (e.g., deferred procedure calls, user-mode

[1] **überSpark** primes are akin to primes from the fictional Transformers universe which are the highest ranking Transformers that can create other Transformers, and form singular entities that exist (and must exist) threaded through all continuities.

and kernel-mode preemptive threading, light-weight non-preemptive threading etc.).

4.5 üobject Interactions

üobject interactions can be divided into intra-collection, inter-collection and legacy component invocations (Fig. 4). Intra-üobject collection and inter-üobject collection interactions occur via üobject callees while legacy invocations occur via legacy callees (§4.1).

Such interactions model function call-return semantics using a combination of hardware capabilities and software verification. This enables compositional reasoning of the üobject properties [50, 52, 67, 124], i.e, allow properties of üobjects to be specified in terms of their interactions with other üobjects and collections, yet being able to verify those properties separately on each üobject in isolation, while meshing with (legacy) unverified components at the desired granularity.

üobject interactions can happen via software and/or hardware conduits and are facilitated by the *sentinel* abstraction as described below.

4.5.1 Sentinels

Sentinels[2] mediate üobjects interactions and ensure that the caller may invoke a given üobject method on the callee according to the üobject manifest (§4.3; Fig. 4.2). If caller and callee are both verified, then no runtime check is required because üobject verification enforces the call policy. This results in efficient runtime performance (e.g., no hardware de-privileging overhead). If either the caller or the callee is unverified, the sentinel consults the policy dynamically and allows or rejects the call accordingly.

In addition to the runtime checks, sentinels are responsible for transferring control among üobjects, switching stacks, and handling hardware signals by employing the appropriate control-transfer method for the isolation mechanism imposed on the üobject.

For example, if two üobjects are both verified and have the same isolation mechanism (e.g., SFI [105]), then the control transfer is just a function call. But if one has a different isolation mechanism (e.g., hardware segmentation), then the sentinel implements the control transfer leveraging the appropriate hardware capabilities, e.g., for segmentation, switches privilege levels, stacks, and marshals arguments.

Similarly for hardware signals, the sentinel employs the appropriate hardware capabilities (e.g., trap state areas) to handle the signal either within the source üobject or by passing control to another üobject.

[2] **überSpark** sentinels are aptly named after the sentinel Transformers in the fictional Transformers universe, which are guard Transformers designed to combat renegades.

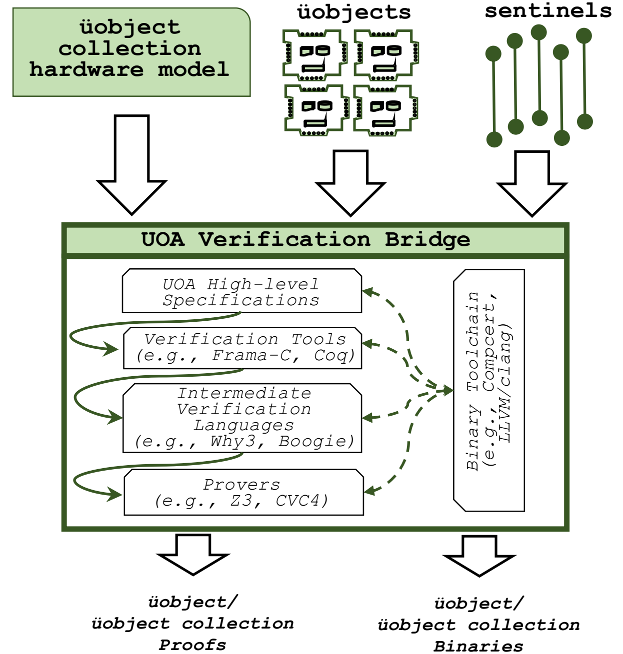

Figure 5. The **überSpark** verification bridge facilitates assume-guarantee style reasoning on the CHIC stack and makes it possible for compositional üobject verification and binary generation while allowing the use of multiple verification tools and techniques.

Sentinels can also be realized using hardware conduits such as legacy I/O, memory-mapped I/O, DMA, and mailboxes [8, 32]. In such cases, interactions are enforced via üobject resource interface confinement (§4.3).

4.6 üobject Verification Bridge

überSpark reasoning relies foundationally on the following set of properties that must hold throughout the execution of an üobject: (a) üobject base invariants, and (b) üobject-specific properties.

üobject base invariants are properties that need to hold regardless of what the üobject implements and include memory safety, memory integrity and (internal) control flow integrity. These invariants include ensuring correct stack frame setup and teardown, ensuring the absence of buffer overflows, (otherwise returns could land at arbitrary üobject program sites), parameter marshaling, routing of external calls via sentinels, privilege-level enforcement, etc.

üobject base invariants make assume-guarantee reasoning on the CHIC stack tractable, and make it possible for üobject code to be reasoned about in a compositional manner. The base invariants are also designed to be verified automatically, without developer assistance (e.g., using abstract interpretation techniques [68, 72] or binary-level enforcement [105]),

to allow retrofitting üobjects into an existing legacy unverified codebase with minimal effort.

üobject-specific properties on the other hand, depend on the desired end-to-end guarantees, the resources that the üobject encapsulates, and the üobject implementation.

The üobject verification bridge (see Fig. 5) is based on a key observation that a vast majority of today's state-of-the-art formal analysis tools integrate with (inter-convertible) common verification languages (e.g., Why3, Boogie) [6].

However, existing intermediate languages do not capture both software and hardware requirements expressively. Therefore, **überSpark** defines a high-level abstract specification language for the üobject invariants and üobject execution semantics including sentinels, resource confinement, and the collection hardware model.

The verification bridge translates the üobject invariants and execution semantics to an existing intermediate verification language and/or specification, which can then be used by a specific verification tool and/or methodology in order to prove various classes of üobject-specific properties, including properties over hardware states.

4.7 üobject Reporting

überSpark enables collecting and reporting measurements (e.g., SHA-1, property based attestation [21]) of üobject instantiations within and across platforms. This ensures that platforms are running the correct stack of (verified) üobject collections.

The special set of üobjects, primes, which instantiate üobject collections (§4.4), are also responsible for collecting and reporting üobject measurements.

There can be multiple primes across multiple collections within a given platform chaining together collection measurements (Fig. 2); a *root-prime* forms the root-of-trust for measurements in such cases[3].

Root-of-trust within a prime can be implemented entirely in software (e.g., via static root of trust and software TPM [99]), entirely in hardware (e.g., via dynamic root of trust and hardware TPM [51]), or a combination of hardware and software (e.g., static root of trust and hardware TPM).

überSpark primes can also be extended to allow üobject instantiation via white-listing [126] and to provide physical platform authentication using an external verifier in the form of software-based attestation [106].

4.8 *ElderSafe* in überSpark

Finally, we revisit how the described **überSpark** architecture, as instantiated on *ElderSafe*, satisfies our design goals for CHIC stack guarantees (§2; cf. §3.8).

[3]This is in a similar vein to the doctrines of the fictional Transformers universe where the prime Transformer that is designated as the leader of all the other Transformers holds the Matrix of Leadership that allows harnessing the wisdom of all the other primes.

Since there are four distinct CHIC hardware platforms involved in *ElderSafe* (§1.1), there are four corresponding prime üobject collections, one for each of the smart lock, the wearable vitals monitor, the health scoring workstation, and the caregiver's tablet.

Each prime üobject collection is responsible for establishing a static root of trust for each platform, as well as a dynamic root of trust for the health scoring system, which may be co-tenant with other software on the workstation.

An additional prime üobject collection deals with the static root of trust on the health scoring workstation GPU.

Each prime üobject collection manages corresponding üobject collections within the underlying hardware platform. For example, zooming in on the wearable platform, the prime manages one collection each for the vital-sensing application, the sensing-hardware driver within the OS, as well as the platform collections supporting the boot-up firmware, while reporting on other software and the hardware state; these are distinct collections because they handle different high-level functionalities and resources.

The prime also takes care to create these collections (and their constituent üobjects) with the right isolation container: the vitals-monitoring application is vast and proprietary, so it must be contained via a combination of software verification and hardware de-privileging, whereas the boot-up firmware can be verified and need not be virtualized or otherwise isolated via hardware mechanisms.

Although the wearable platform's prime creates these collections of üobjects, a number of sentinels on the platform handle on-going object interactions: object-to-object calls within each collection, measurement calls for attestation, inter-process or multi-processor communication calls across address spaces and cores, respectively, etc.

Importantly, the vitals-monitoring application üobject collection needs its sentinel in particular, because it must filter out the vast functionality of the vitals-monitoring application, including a GUI, and only keep the few sensing APIs needed by *ElderSafe*.

The sentinels enforce Resource Interface Confinement on this application, only allowing information about the needed sensor readings, and therefore achieving resource closure.

Providing guarantees (security and functional) on the wearable üobject collections requires bridging hardware abstractions on the ARM platform and the memory abstractions of the C/Java runtime executing the vitals-monitoring application.

For example, to provide real-time guarantees about the delivery of a sensed signal to collections on a different platform (e.g., the health-scoring application on the workstation), the verification bridge must expose the concurrency model to a worst-case execution-time framework.

Although this is a brief slice of the **überSpark** primitives in the context of *ElderSafe*, they demonstrate how the architecture makes it tractable to reason about a complex CHIC

system design, yet adhere to the design goals we set out to achieve.

5 Research Directions and Opportunities

We now discuss interesting research directions that stem from the **überSpark** architecture (§4) and existing tools and methodologies that **überSpark** can benefit from.

5.1 Creation of üobjects

üobjects can be created from the existing CHIC stack by identifying the resources being isolated towards a specific property, and then paring away code that closely operates on such resources. **überSpark** can readily benefit from program slicing [55, 72, 102, 128], data dependency analysis [72, 86], and program synthesis [40, 41, 57], to automatically identify such code fragments for common languages such as C, C++, and Java. For binary-only components, **überSpark** can leverage binary analysis platforms [19, 61] to locate, slice, and stitch together such code fragments. In the long term, machine learning techniques for optimizing existing binary code for a purpose [104] or forms of summarization and question answering [4, 22, 54] may help in this task.

5.2 Verification of üobjects

For open-source üobjects that are written in common languages, **überSpark** can employ refinement proofs [53, 59, 73], source-code level verification [60, 77, 124], and push-button style verification [94, 109, 110, 133] to prove üobject properties. Additionally, for some languages (e.g., C, Assembly, ML), **überSpark** can leverage certified compilers [16, 17, 25, 78, 116], certified parsers and code generation frameworks [11, 24, 25, 69], in association with proven-correct assemblers [71], to translate verified properties into proven-correct binaries. For languages unable to benefit from such schemes or for binary-only üobjects, **überSpark** allows instrumenting the resulting binary code with assertions satisfying required properties [13, 73, 112, 113]. Similarly, for concurrent üobjects we can employ contextual refinement [53], while sequential üobjects can be reasoned with Hoare logic [60, 77, 124].

5.3 üobject Resource Interface Confinement (RIC)

überSpark can employ hardware capabilities such as IOMMU [32] and MMU for RIC of memory and devices; hardware deprivileging can be used for RIC of CPU instructions. Where performance is key or where hardware has limited capabilities **überSpark** can benefit from model-checking [27], abstract program interpretation [72], and software fault isolation [105] to achieve RIC at both source and binary levels. Finally, **überSpark** can leverage (and inform) hardware breakpoints as well as hardware-assisted instruction level guards (e.g., Intel MPX [32]) to enforce efficient fine-grained RIC. In the long term, we envision **überSpark** to inform next generation of hardware capabilities towards efficient RIC (e.g., MPX guards for privileged instructions).

6 Foundational Steps

As with any vision, we first must learn to walk towards our goals, before we can run. We now describe our foundational, walking steps towards realizing the **überSpark** vision.

As a first step, we set out to refactor an *existing* commodity open-source micro-hypervisor, the eXtensible Micro-Hypervisor Framework (XMHF)[4], for the x86 platforms into a üobject collection. XMHF is written in C and Assembly and allows for the micro-hypervisor framework to be extended with extensions to support required functionality [123]. Our goal was to leverage this existing micro-hypervisor functionality and create a prime üobject collection that would enable instantiating üobjects on desired layers of the CHIC stack on x86 hardware platforms and also serve as the foundation for üobject reporting.

We were able to refactor XMHF into 14 üobjects within 6 months, in an incremental fashion, co-existing with regular development [124].

üobject invariants were automatically proven and composed using the Frama-C [72] verification framework, along with a x86 hardware model and assembly language dialect (called CASM) that we developed to work in conjunction with Frama-C. We have so far been able to verify and bridge properties such as memory safety, control-flow integrity, and information flow as trace properties automatically and directly on the source code. We have also been able to automatically prove supporting üobject-specific functional correctness properties.

The verification was bridged with the Compcert [16] certified compiler for binary generation. We were able to support verified üobjects co-existing and meshing with unverified components at multiple granularities, with üobject Resource Interface Confinement (RIC) enforced using IOMMU, MMU (regular and nested page-tables) and pure software verification. The runtime performance overhead for a collection of verified üobjects was less than 2% [124].

Open-source development now continues as the über eXtensible Micro-Hypervisor Framework (überXMHF)[5], which is XMHF in the **überSpark** architecture.

Encouraged by our results on x86 – a particularly rich hardware platform – we turned to low-cost ARM platforms. To this end, we have implemented the first ARMv8 micro-hypervisor based on überXMHF, that currently supports the ubiquitous low-cost Raspberry Pi 3 computing platform [122].

Our ARMv8 based micro-hypervisor implementation uses a novel lightweight trap-inspect-forward (TIF) mechanism to selectively trap and inspect critical peripheral register

[4] http://xmhf.org

[5] https://uberxmhf.org

accesses, before forwarding the access directly to the physical system peripherals. The TIF building block allows us to efficiently implement resource interface confinrment of memory, devices, and DMA, including interrupts, without the requirement of hardware support. This is essential, because such hardware support, e.g., an IO Memory Management Unit (IOMMU) or a Generic Interrupt Controller (GIC), is absent on Raspberry Pi3 and similar low-cost platforms. Further, we achieve all this with low complexity, since we do not resort to complex peripheral emulation and state maintenance. We are able to run Raspbian and realtime linux distributions with runtime overheads of 2-6% [122].

Lastly, we have explored üobject threading and schedulability in the presence of legacy unverified code. We have developed a real-time mixed-trust scheduling framework that is able to offer precise timing guarantees for protected üobjects, while co-existing with a regular (untrusted) legacy OS schedulers [36].

To verify the timing correctness of potentially safety-critical üobjects in our mixed-trust scheduling framework, we propose a new mixed-trust task model and construct a detailed schedulability analysis. We also present the design and implementation of a coordination protocol between the legacy guest OS scheduler and the micro-hypervisor based scheduler (called hyper-scheduler) to preserve the synchronization between üobject executions and untrusted components while preventing dependencies that can compromise üobject executions.

Our current hyper-scheduler implementation consists of a non-premptive scheduler üobject which operates in conjunction with the open-source ZSRMV[6] Linux OS scheduler, boot-strapped by the überXMHF prime üobject collection on an ARMv8 platform [36].

7 Present Activities

We are presently finishing up the **überSpark** language framework that enforces the üobject abstraction within any existing legacy C and Assembly code-base, towards assume-guarantee (compositional) reasoning on the CHIC stack. This will enable us to automate all the invariants that were manually constructed during the verification effort of the x86 implementation of our micro-hypervisor prime üobject collection (§6). We will refactor the x86 micro-hypervisor prime üobject collection implementation using the **überSpark** language framework as part of evaluating the language efficacy.

More broadly, the **überSpark** language framework, will enable automated verification of foundational properties such as memory safety, memory integrity and control-flow integrity for any existing C and Assembly CHIC stack codebase, thereby supporting development compatible, incremental and composable verification that can keep pace with the codebase evolution.

[6]https://github.com/cps-sei/zsrmv

Work is also underway to verify the ARMv8 implementation of our micro-hypervisor prime üobject collection (§6). We will be leveraging the **überSpark** language framework, in addition to adding relevant ARMv8 hardware modeling, to verify additional properties that can be formulated as invariants (e.g., information flow) as well as supporting functional correctness properties.

We are further investigating meshing of different flavors of properties, such as timing and logical properties, in the context of our hyper-scheduler implementation (§6). Reasoning based on timed automata is proving to be a very useful building piece in this context.

Last but not least, we are exploring other existing, ubiquitous, legacy code bases such as the Linux OS kernel[7], the PX4 open-source autopilot[8], and Open vSwitch (OVS)[9] to provide verified properties such as dependable interrupts, threading, secure inter-process communications and network level packet-flow attribution (e.g., via trustworthy IoT security gateways [89]).

We anticipate the **überSpark** architecture, language, and development tool-chain, in conjunction with our verified micro-hypervisor prime üobject collections, to facilitate incremental and composable verification of such existing legacy codebases, in a development compatible manner.

8 Related Work

Micro-kernels [1, 56, 64, 81, 82, 108, 115, 130], separation kernels [66, 101], MILS [5], isolation kernels [129], exo-kernels [37, 62, 70], small-TCB hypervisors [23, 29, 35, 49, 107, 111, 118, 123], lightweight process contexts [83], and type-safe containerization kernels [14, 39], attempt to minimize bugs via privilege disaggregation. However, they do not provide any formal guarantees or privileged code disaggregation; they remain vulnerable to the attacks against small kernels [47, 48].

Approaches verifying a privileged OS kernel both in a monolithic [28, 42, 65, 73, 74, 77, 88, 93, 105, 123, 127, 132] and compositional manner [52, 53, 124] primarily focus on the verification methodology that best applies to a specific subsystem (e.g., kernel or hypervisor). However, it is unclear if such methodologies can individually be applicable to every component of a CHIC stack, e.g., specifying interactions among verified and unverified components. Furthermore, they often rely on deep refinement proofs that are likely to be prohibitive for a rapidly evolving CHIC stack.

Recent full-system stack-verification approaches impressively verify the entire OS, application stack, and in some cases the hardware platform [3, 60, 131]. However, changes in system configuration entail lengthy, costly re-verification (sometimes, measuring many person years). Further, they are

[7]https://kernel.org
[8]https://px4.io
[9]https://openvswitch.org

bound to a specific platform or programming paradigm and lack support for co-existence with unverified components.

9 Conclusions

Taking stock of today's computing ecosystem, and advances in verification technologies we asked ourselves: *How do we achieve practical, provable end-to-end guarantees on today's complex commodity heterogeneous interconnected edge-computing (CHIC) platforms?* Our inability to find a satisfying answer motivated the genesis of **überSpark**. Elements of **überSpark** act in synergy to offer required capabilities for achieving provable end-to-end guarantees on the CHIC stack. While we expect a myriad of implementation hurdles in our quest, we are encouraged by our early results and on-going research in this direction. We anticipate **überSpark** to enable the combination of otherwise incompatible hardware, software, and tools towards offering strong guarantees for tomorrow's user on today's evolving CHIC platforms.

Availability

Active open-source development of **überSpark** continues at:

https://uberspark.org

Acknowledgements

We thank Sagar Chaki, Anupam Datta, and Limin Jia for their deep insights and feedback during the early stages of **überSpark**. We also thank Martin Abadi, Dionisio de Niz, Grace Lewis, Matt Loring, and Andrew Ferraiuolo for their reviews and feedback. Finally, we thank various anonymous reviewers for their detailed comments and feedback throughout different stages of research and development connected to the **überSpark** architecture and components.

This work was funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002[10]

[10]Copyright 2020 ACM.

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

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. DM20-0448

References

[1] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer*. USENIX Association, 93–113.

[2] AliveCor. 2020. KardiaBand: Take an EKG anytime, anywhere. https://www.alivecor.com/. Accessed: May 28, 2020.

[3] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. 2009. Balancing the Load: Leveraging Semantics Stack for Systems Verification. 42, Numbers 2-4 (2009), 389–454.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR (Poster)*. OpenReview.net.

[5] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. 2006. The MILS architecture for high-assurance embedded systems. *IJES* 2, 3/4 (2006), 239–247.

[6] Michael Ameri and Carlo A. Furia. 2016. Why Just Boogie? - Translating Between Intermediate Verification Languages. In *IFM (Lecture Notes in Computer Science)*, Vol. 9681. Springer, 79–95.

[7] ARM. 2010. Virtualization Extensions Architecture Specification. http://infocenter.arm.com. Accessed: May 28, 2020.

[8] ARM. 2020. ARM Architecture Reference Manuals. https://developer.arm.com/docs/ddi0487/fb. Accessed: May 28, 2020.

[9] Astah Inc. 2016. Astah UML2C. https://astah.net/product-plugins/uml2c-export/. Accessed: May 28, 2020.

[10] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (Lecture Notes in Computer Science)*, Vol. 4111. Springer, 364–387.

[11] Aditi Barthwal and Michael Norrish. 2009. Verified, Executable Parsing. In *ESOP (Lecture Notes in Computer Science)*, Vol. 5502. Springer, 160–174.

[12] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*. ACM, 29–44.

[13] Cinzia Bernardeschi, Nicoletta De Francesco, Giuseppe Lettieri, Luca Martini, and Paolo Masci. 2008. Decomposing bytecode verification by abstract interpretation. *ACM Trans. Program. Lang. Syst.* 31, 1 (2008), 3:1–3:63.

[14] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *SOSP*. ACM, 267–284.

[15] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Found. Trends Priv. Secur.* 1, 1-2 (2016), 1–135.

[16] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM (Lecture Notes in Computer Science)*, Vol. 4085. Springer, 460–475.

[17] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2013. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 107–115.

[18] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security Symposium*. USENIX Association, 917–934.

[19] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *CAV (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 463–469.

[20] Caretaker Medical. 2018. Wireless CNIBP and Vital Signs. http://www.caretakermedical.net/. Accessed: May 28, 2020.

[21] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. 2006. A protocol for property-based attestation. In *STC*. ACM, 7–16.

[22] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *ASE*. ACM, 826–831.

[23] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*. ACM, 2–13.

[24] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*. ACM, 143–156.

[25] Adam Chlipala. 2010. A verified compiler for an impure functional language. In *POPL*. ACM, 93–106.

[26] Adam Chlipala. 2013. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP*. ACM, 391–402.

[27] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS (Lecture Notes in Computer Science)*, Vol. 2988. Springer, 168–176.

[28] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs (Lecture Notes in Computer Science)*, Vol. 5674. Springer, 23–42.

[29] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *SOSP*. ACM, 189–202.

[30] Rob Coombs. 2015. Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online. https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-01-06-27/TrustZone-and-FIDO-white-paper-final.pdf. Accessed: May 28, 2020.

[31] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. 2015. Linux Kernel Development How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. https://www.linuxfoundation.org/events/2015/02/linux-kernel-development-how-fast-it-is-going-who-is-doing-it-what-they-are-doing-and-who-is-sponsoring-it-2015/. *The Linux Foundation* (2015). Accessed: May 28, 2020.

[32] Intel Corporation. 2016. Intel Architecture Software Developer Manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html. Accessed: May 28, 2020.

[33] Intel Corporation. 2016. Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. Accessed: May 28, 2020.

[34] Intel Corporation. 2019. Intel Trusted Execution Technology – Software Development Guide. http://cqcontent.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf Document number 315168-016. Accessed: May 28, 2020.

[35] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram S. Adve. 2007. Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP*. ACM, 351–366.

[36] Dionisio de Niz, Björn Andersson, Mark H. Klein, John P. Lehoczky, A. Vasudevan, Hyoseung Kim, and Gabriel A. Moreno. 2019. Mixed-Trust Computing for Real-Time Systems. In *RTCSA*. IEEE, 1–11.

[37] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*. ACM, 251–266.

[38] ETH Zurich and Microsoft Research. 2018. Barrelfish OS. http://www.barrelfish.org/. Accessed: May 28, 2020.

[39] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. 2006. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys*. ACM, 177–190.

[40] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *PLDI*. ACM, 420–435.

[41] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *POPL*. ACM, 599–612.

[42] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*. ACM, 287–305.

[43] Jean-Christophe Filliâtre and Claude Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *CAV (Lecture Notes in Computer Science)*, Vol. 4590. Springer, 173–177.

[44] US Food and Drug Administration. 2011. Philips IntelliVue GuardianSoftware Rev A.00. https://www.accessdata.fda.gov/cdrh_docs/pdf11/K111905.pdf. Accessed: May 28, 2020.

[45] US Food and Drug Administration. 2018. Philips IntelliVue GuardianSoftware Rev D.0. https://www.accessdata.fda.gov/cdrh_docs/pdf18/K180534.pdf. Accessed: May 28, 2020.

[46] Frama-C Team. 2015. ACSL: ANSI/ISO C Specification Language v1.9. http://www.frama-c.com. Accessed: May 28, 2020.

[47] Jason Franklin, Sagar Chaki, Anupam Datta, and Arvind Seshadri. 2010. Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worrying about Data Structure Size. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 365–379.

[48] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. 2008. *Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor*. Technical Report CMU-CyLab-08-008. CMU CyLab.

[49] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: a virtual machine-based platform for trusted computing. In *SOSP*. ACM, 193–206.

[50] Deepak Garg, Jason Franklin, Dilsun Kirli Kaynar, and Anupam Datta. 2010. Compositional System Security with Interface-Confined Adversaries. In *MFPS (Electronic Notes in Theoretical Computer Science)*, Vol. 265. Elsevier, 49–71.

[51] Trusted Computing Group. 2003. Trusted platform module main specification, Version 1.2, Revision 103. https://trustedcomputinggroup.org/resource/tpm-main-specification/. Accessed: May 28, 2020.

[52] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM, 595–608.

[53] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*. USENIX Association, 653–669.

[54] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *IJCAI*. ijcai.org, 3675–3681.

[55] Christian Hammer and Gregor Snelting. 2004. An improved slicer for Java. In *PASTE*. ACM, 17–22.

[56] Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. *Commun. ACM* 13, 4 (1970), 238–241.

[57] William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *PLDI*. ACM, 317–328.

[58] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv.* 44, 3 (2012), 16:1–16:58.

[59] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2017.

IronFleet: proving safety and liveness of practical distributed systems. *Commun. ACM* 60, 7 (2017), 83–92.

[60] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *OSDI*. USENIX Association, 165–181.

[61] Andrew Henderson, Aravind Prakash, Lok-Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *ISSTA*. ACM, 248–258.

[62] Jason Hennessey, Sahil Tikale, Ata Turk, Emine Ugur Kaynar, Chris Hill, Peter Desnoyers, and Orran Krieger. 2016. HIL: Designing an Exokernel for the Data Center. In *SoCC*. ACM, 155–168.

[63] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. In *HASP@ISCA*. ACM, 11.

[64] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. 2004. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *ACM SIGOPS European Workshop*. ACM, 22.

[65] Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. *Operating Systems Review* 41, 2 (2007), 37–49.

[66] Information Assurance Directorate. 2007. US Government Protection Profile for Separation Kernels in Environments Requiring High Robustness. https://www.niap-ccevs.org/Profile/Info.cfm?PPID=65&id=65. (2007). Accessed: May 28, 2020.

[67] Limin Jia, Shayak Sen, Deepak Garg, and Anupam Datta. 2015. A Logic of Programs with Interface-Confined Code. In *CSF*. IEEE Computer Society, 512–525.

[68] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *POPL*. ACM, 247–259.

[69] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *ESOP (Lecture Notes in Computer Science)*, Vol. 7211. Springer, 397–416.

[70] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application Performance and Flexibility on Exokernel Systems. In *SOSP*. ACM, 52–65.

[71] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. 2013. Coq: the world's best macro assembler?. In *PPDP*. ACM, 13–24.

[72] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609.

[73] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2:1–2:70.

[74] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*. ACM, 207–220.

[75] Ulrich Kühn, Marcel Selhorst, and Christian Stüble. 2007. Realizing property-based attestation and sealing with commonly available hard- and software. In *STC*. ACM, 50–57.

[76] Leslie Lamport. 2019. The TLA+ home page. https://lamport.azurewebsites.net/tla/tla.html. Accessed: May 28, 2020.

[77] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM (Lecture Notes in Computer Science)*, Vol. 5850. Springer, 806–809.

[78] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54.

[79] Chao Li, Liang Dou, and Zongyuan Yang. 2014. A metamodeling level transformation from UML sequence diagrams to Coq. In *ICTCS (CEUR Workshop Proceedings)*, Vol. 1231. CEUR-WS.org, 147–157.

[80] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. 2011. VIPER: verifying the integrity of PERipherals' firmware. In *ACM Conference on Computer and Communications Security*. ACM, 3–16.

[81] Jochen Liedtke. 1993. Improving IPC by Kernel Design. In *SOSP*. ACM, 175–188.

[82] Jochen Liedtke. 1996. Toward Real Microkernels: The inefficient, inflexible first generation inspired development of the vastly improved second generation, which may yet support a variety of operating systems. *Commun. ACM* 39, 9 (1996), 70–77.

[83] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *OSDI*. USENIX Association, 49–64.

[84] Macworld. 2018. Apple Watch's heart rate sensor can detect diabetes, Cardiogram study finds. https://www.macworld.com/article/3253341/cardiogram-diabetes-apple-watch.html. Accessed: May 28, 2020.

[85] Claude Marché, Guillaume Melquiond, Andrei Paskevich, and et al. 2013. The why3 platform. http://why3.lri.fr/. Accessed: May 28, 2020.

[86] Victor J. Marin and Carlos R. Rivero. 2018. Towards a framework for generating program dependence graphs from source code. In *SWAN@ESEC/SIGSOFT FSE*. ACM, 30–36.

[87] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: hi-fi tests for lo-fi emulators. In *ASPLOS*. ACM, 337–348.

[88] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *USENIX Security Symposium*. USENIX Association.

[89] Matt McCormack, Amit Vasudevan, Guyue Liu, Sebastián Echeverría, Kyle O'Meara, Grace Lewis, and Vyas Sekar. 2020. Towards an Architecture for Trusted Edge IoT Security Gateways. In *HotEdge*. USENIX Association.

[90] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 143–158.

[91] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP@ISCA*. ACM, 10.

[92] Mike King. 2018. Smart Technology Takes Hold in Retirement Communities. https://www.nextavenue.org/smart-technology-retirement-communities/. Accessed: May 28,2020.

[93] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *PLDI*. ACM, 395–404.

[94] Jan Nordholz. 2020. Design of a symbolically executable embedded hypervisor. In *EuroSys*. ACM, 6:1–6:16.

[95] Colin O'Flynn. 2017. Breaking Electronic Door Locks like you're on CSI: CYBER. https://www.blackhat.com/docs/us-17/wednesday/us-17-OFlynn-Breaking-Electronic-Locks.pdf. *Black Hat USA* (2017). Accessed: May 28, 2020.

[96] Sangho Park and Henry A Kautz. 2008. Privacy-Preserving Recognition of Activities in Daily Living from Multi-view Silhouettes and RFID-based Training. In *Proc. AAAI Fall Symposium: AI in Eldercare: New Solutions to Old Problems*. AAAI, 70–77.

[97] Philips. 2019. IntelliVue Guardian EWS: Automated Early-warning scoring system. https://www.usa.philips.com/healthcare/clinical-solutions/early-warning-scoring/intellivue-guardian-ews. Accessed:

May 28, 2020.
[98] Philips. 2019. Intellivue Guardian Software Solution: Keep Watch and Intervene Early. http://images.philips.com/is/content/PhilipsConsumer/Campaigns/HC20140401_DG/Documents/en_US/20190225-intellivue-guardian-software-igs-solution-brochure.pdf. Accessed: May 28, 2020.
[99] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Löser, Dennis Mattoon, Magnus Nyström, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. 2016. fTPM: A Software-Only Implementation of a TPM Chip. In *USENIX Security Symposium*. USENIX Association, 841–856.
[100] Alvin Rajkomar, Eyal Oren, Kai Chen, Andrew M Dai, Nissan Hajaj, Michaela Hardt, Peter J Liu, Xiaobing Liu, Jake Marcus, Mimi Sun, et al. 2018. Scalable and accurate deep learning for electronic health records. *CoRR* abs/1801.07860 (2018).
[101] John M. Rushby. 1981. Design and Verification of Secure Systems. In *SOSP*. ACM, 12–21.
[102] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram S. Adve. 2013. Using likely invariants for automated software fault localization. In *ASPLOS*. ACM, 139–152.
[103] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *POPL*. ACM, 379–391.
[104] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ASPLOS*. ACM, 305–316.
[105] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*. USENIX Association, 1–12.
[106] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doom, and Pradeep K. Khosla. 2007. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Malware Detection*. Springer, 253–289.
[107] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*. ACM, 335–350.
[108] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *SOSP*. ACM, 170–185.
[109] Helgi Sigurbjarnarson, James Bornholt, Nicolas Christin, and Lorrie Faith Cranor. 2017. Push-Button Verification of File Systems via Crash Refinement. In *USENIX Annual Technical Conference*. USENIX Association.
[110] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *OSDI*. USENIX Association, 287–305.
[111] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. 2006. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys*. ACM, 161–174.
[112] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2016. A design and verification methodology for secure isolated regions. In *PLDI*. ACM, 665–681.
[113] Rohit Sinha, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2015. Moat: Verifying Confidentiality of Enclave Programs. In *ACM Conference on Computer and Communications Security*. ACM, 1169–1184.
[114] Colin F. Snook and Michael J. Butler. 2006. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.* 15, 1 (2006), 92–122.
[115] Udo Steinberg and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. In *EuroSys*. ACM, 209–222.
[116] Martin Strecker. 2002. Formal Verification of a Java Compiler in Isabelle. In *CADE (Lecture Notes in Computer Science)*, Vol. 2392. Springer, 63–77.
[117] G. Edward Suh and Srinivas Devadas. 2007. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *DAC*. IEEE, 9–14.
[118] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *OSDI*. USENIX Association, 279–292.
[119] Andrew S. Tannenbaum and Maarteen van Steen. 2016. *Distributed Systems: Principles and Paradigms*. Createspace Independent Publishing Platform.
[120] The Home Depot. 2019. Schlage Smart Door Locks. https://www.homedepot.com/b/Smart-Home-Smart-Home-Security-Smart-Locks/Schlage/N-5yc1vZc7byZ1c3. Accessed: May 28, 2020.
[121] UML Designer. 2018. UML to Java generator and reverse. http://www.umldesigner.org/ref-doc/umlgen.html. Accessed: May 28, 2020.
[122] Amit Vasudevan and Sagar Chaki. 2018. Have Your PI and Eat it Too: Practical Security on a Low-Cost Ubiquitous Computing Platform. In *EuroS&P*. IEEE, 183–198.
[123] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan M. McCune, James Newsome, and Anupam Datta. 2013. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 430–444.
[124] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. 2016. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *USENIX Security Symposium*. USENIX Association, 87–104.
[125] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. 2009. *Lockdown: A Safe and Practical Environment for Security Applications*. Technical Report CMU-CyLab-09-011. CMU CyLab.
[126] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. 2012. Lockdown: Towards a Safe and Practical Architecture for Security Applications on Commodity Platforms. In *TRUST (Lecture Notes in Computer Science)*, Vol. 7344. Springer, 34–54.
[127] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *SOSP*. ACM, 203–216.
[128] Mark Weiser. 1981. Program Slicing. In *ICSE*. IEEE Computer Society, 439–449.
[129] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and Performance in the Denali Isolation Kernel. In *OSDI*. USENIX Association.
[130] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. 1974. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. ACM* 17, 6 (1974), 337–345.
[131] Jean Yang and Chris Hawblitzel. 2011. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM* 54, 12 (2011), 123–131.
[132] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 79–93.
[133] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *SOSP*. ACM, 275–290.