Abstract

How do we formally verify security properties in today’s malleable and evolving Commodity System Software (COSS) ecosystem? Recent advances in applying formal methods to systems software, e.g., IronClad [16] and seL4 [19], promise that this vision is not a fool’s errand after all. In this position paper we explore the challenges involved in this problem, what research questions the state of the art leaves still open, and our proposal for the next step towards realizing this vision.

1 Problem Statement

Today’s commodity system software (COSS) stack comprises chiefly the BIOS, hypervisor (e.g., cloud), and the OS. These components are complex since they deal with low-level hardware features, especially at early stages of initialization. The complexity increases considerably with extensions required to enable additional platform functionality. For example, BIOS extensions such as option ROMs allow device-specific initialization, and EFI BIOS extensions enable filesystem and network access. Similarly, although hypervisors and VMMs started off as monolithic software aimed at managing virtual machines, they evolved to a convenient point of observation and mediation of useful (security) services that are realized via hypervisor extensions [11, 14, 15, 21, 23, 25, 29–31, 33, 35, 36, 38, 39, 42, 43]. Finally, OS kernels have long been extensible via device drivers and custom modules.

Adding an extra dimension of complexity, there is vertical interaction in this stack: the OS relies on the hypervisor through paravirtualization interfaces, which in turn depends on the BIOS for some system services.

These low-level components and their extensions are developed by different entities and change rapidly; UEFI [6], OpenBIOS [5], Xen [10], KVM [4], Linux [3] are prime examples of this diversity and flux. Such changes not only modify existing extensions, but also add new ones to incorporate additional functionality. Yet, these low-level components form the basis of security in the systems we use today, since they set up and enforce fundamental system services that applications rely on to provide protection, trusted execution, reference monitoring, privacy, and a host of other critical security properties. How do we formally verify security properties in this malleable and evolving COSS ecosystem?

2 A Motivating Example

Imagine a COSS stack consisting of a BIOS, a hypervisor and a guest OS. Now imagine a BIOS extension for a network card that handles initialization and provides a runtime interface to the network card firmware. Assume a hypervisor on top of the BIOS managing virtualization of guest above, but also supporting additional security extensions. For example, consider an extension approveexec enabling only explicitly approved code to run in the guest OS kernel mode. This extension is used to run a guest OS network card driver that has exclusive control of the network card – together these extensions provide (P1) trusted network logging.

Suppose that every component of the stack, the BIOS (with extensions), hypervisor (with extensions) and the guest OS driver are verified for P1. This implies proving memory integrity and confidentiality of the core BIOS and the hypervisor, network card option ROM correctness to initialize and keep the card in the required operating state, proving the approved code execution property of approveexec and proving network card I/O channel isolation for the protected guest OS driver. Further, we prove P1 on the composition of all the components.

Consider now adding new BIOS option ROM extensions for a USB keyboard and display that handle initialization and provide runtime interface to the keyboard and a graphics card. Another hypervisor extension hyperdep which prevents guest OS code from marking memory pages as both writable and executable, and a OS driver that takes on-demand exclusive control of the keyboard and display hardware for (P2) trusted path [43]. Now, we must ensure that these new extensions guarantee P2, but also that they don’t violate P1.

This scenario described is not science fiction (except for the formal verification bit). It is a typical example of how features are added at various layers, at various times, and with varying configurations, to provide a plethora of (intended but unverified) security properties; in practice, the number of options at every level is larger.

3 Building Verified Systems

Getting from an extensible, evolvable system to verified properties typically requires leveraging one or more of a small number of formal-verification techniques:

Software Model Checkers execute the software symbolically and exhaustively, deriving formulas that represent possible states (variable values) at every execution
<table>
<thead>
<tr>
<th>1. REPORT DATE</th>
<th>2. REPORT TYPE</th>
<th>3. DATES COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 JAN 2015</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE AND SUBTITLE</th>
<th>5a. CONTRACT NUMBER</th>
<th>5b. GRANT NUMBER</th>
<th>5c. PROGRAM ELEMENT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEEC All the Way Down</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6. AUTHOR(S)</th>
<th>5d. PROJECT NUMBER</th>
<th>5e. TASK NUMBER</th>
<th>5f. WORK UNIT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datta /Sagar Chaki Amit Vasudevan Limin Jia Petros Maniatis Anupam</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</th>
<th>8. PERFORMING ORGANIZATION REPORT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</th>
<th>10. SPONSOR/MONITOR’S ACRONYM(S)</th>
<th>11. SPONSOR/MONITOR’S REPORT NUMBER(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12. DISTRIBUTION/AVAILABILITY STATEMENT</th>
<th>13. SUPPLEMENTARY NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approved for public release, distribution unlimited.</td>
<td>The original document contains color images.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15. SUBJECT TERMS</th>
<th>16. SECURITY CLASSIFICATION OF:</th>
<th>17. LIMITATION OF ABSTRACT</th>
<th>18. NUMBER OF PAGES</th>
<th>19a. NAME OF RESPONSIBLE PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a. REPORT</td>
<td>SAR</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unclassified</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. ABSTRACT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unclassified</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. THIS PAGE</td>
<td>unclassified</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The original document contains color images.
point, and checks via a solver that certain boolean predi-
cates over variables (e.g., “all executable pages have
been approved” in the case of approveexec) hold at spe-
cific execution points. To use a software model checker,
the program must be annotated with assertions specifying
properties to be verified. Software model checking
is automated, and has been shown to handle large pro-
grams in specific domains [9]. However, complex loops
and data structures require special handling.

Interactive theorem-provers can be used to verify com-
plex properties of programs. However, proving end-to-
end properties of low-level software requires consider-
able manual effort to define properties to be verified and
construct proofs that the design and implementation of
the software satisfy those properties. Often, properties
are first verified on an abstract model of the software. Re-
finement between the implementation and model is sep-
arately proved. Refinement proofs are labor intensive.

Type-safe languages can be used to build high-
assurance software systems. The type-safety properties
guaranteed by existing languages ensure that objects are
used with the correct types and, in the case of Java and
functional languages, memory safety. However, type-
safety is not enough to prove many important security
properties (e.g., those violated by logical errors).

4 Challenges, Goals and Non-goals

Verification Challenges. Although verification tech-
niques have matured considerably, scalability, composi-
tability, and evolvability are still key challenges for
COSS.

Program Size. With increasing functionality, COSS
stack grows in size and extensions. Verification complex-
ity increases as the size of the target software becomes
larger. For example, the complexity of model-checking
grows exponentially with the number of state variables.

Configuration. COSS has a large number of con-
figurations; each layer of COSS can be configured to
in(ex)clude features and extensions, and properties must
be verified on all possible configurations. Thus, verifica-
tion complexity grows with the configuration space.
Verifying each configuration independently leads to ex-
ponential blowup in the number of extensions.

Evolvability. COSS stack consists of rapidly evolving
pieces largely written in low-level languages such C and
assembly. The verification process must accommodate
incremental development in these low-level languages.

Untrusted Extensions. COSS stack often contain ex-
tensions that target special use cases and originate from
third-party vendors (e.g., profilers, sensor applications,
remote management services etc.). Verifying such exten-
sions might be too expensive. Although the core com-
ponents of the software are verified, they have to exe-
cute with these untrusted extensions. A buggy untrusted
and unverified extension [9] may overwrite critical data
structures, jump to core component routines in the wrong
order, or otherwise invalidate the verified properties of
other system components.

Interference. Multiple extensions often access the
same system resource. For example, both hyperdep and
approveexec manipulate the memory protections of guest
OS and operate on the same micro-hypervisor data struc-
tures (the guest’s Extended Page Tables). Reasoning
about the security of the two extensions is challenging
when both access their shared memory arbitrarily.

Concurrency. Extensions can be multi-threaded. The
large number of interleavings of threads leads to a huge
statespace, and very complex invariants, making it very
challenging for model checking and theorem proving.

Goals. Our overarching goal is to develop design princi-
pies and verification methodologies that can verify secu-
ritv properties in today’s COSS ecosystem while being
performant. More specifically our goals are: (i) Com-
positional verification: addition of new components or
changes to existing component does not require rever-
ifying the entire software stack; (ii) Extensibility: one
should be able to compose verification results both hor-
izontally (within a layer of the stack) and vertically
(across layers); (iii) Customizable configurations: the
verification process needs to be able to handle any com-
bination of components; (iv) Evolvable and automated
verification: verification needs to support low-level lan-
guages and re-verification of fast changing components
needs to be efficient; and (v) Legacy compatibility: un-
trusted, unverified (legacy) components can run with ver-
ified components without violating verified properties.

Non-goals. We only aim for specific security properties
(e.g., hypervisor code is not modified by guest OS) and
not full functional correctness (i.e., the implementation
behaves exactly as specified in a high-level design). This
lowers verification complexity while still enabling us to
prove a large class of security properties. For instance, if
a hypervisor implementation always ignores page faults
caused by the guest OS, it may not functionally correct,
but does not violate the security property that the hyper-
visors’s code region is not modified by the guest OS.

5 State-of-the-Art

Based on the design principles and verification granular-
ity, we classify state-of-the-art approaches as follows.

Monolithic Extension. Monolithic approaches add
features to an existing (privileged) code base to enforce
desired security properties. The extensions run at the
same privilege level as the core. SELinux [26], AppAr-
nor [1] and FBAC [27] are some examples of OS kernel
modifications that support various access control poli-
cies. Such an approach suffers from the lack of sepa-
Isolation (SFI) is a software primitive interfaces for accessing key system state. Software Fault to XMHF are assumed to be confined to a set of narrow fication is for a single-threaded execution and extensions automated with minimal annotations but only focuses on a micro-hypervisor design. The verification is largely au-
tomated with invariants and assertions to be verified, and a modular approach iso-
atates system components into privileged (trusted) and deprivileged (untrusted) components. For instance, Xen/Xoar [13] partitions the Xen hypervisor and runs the partitions in deprivileged mode. Unfortunately, the core hypervisor is still monolithic and runs in privileged mode. NOVA [32] deprivileged everything, including the virtualization modules or VMM, except for a small privilege micro-kernel. With such an approach, unprivileged components cannot affect the privileged code. Buggy privileged code (even if it is smaller) can still both other privileged components and all untrusted components. Further, the above mentioned systems are not verified and therefore, the modular design provides merely code disaggregation, but no formal guarantees on its own.

Verified Sandboxing. Taking advantage of modular designs, privileged components are getting smaller and amenable to formal verification. Verified sandboxing approaches attempt to only verify the privileged portion that provides required system properties while running all other (untrusted) code as deprivileged entities. For instance, the C implementation of the micro-kernel seL4 [19] is fully verified for functional correctness and it runs with other deprivileged services. However, the verification process used interactive theorem proving and took around 22 person-years [18]. Further, the verification focuses on single-threaded execution with concurrency still remaining a challenge. Singularity used software mechanisms to isolate untrusted processes [17]. The privileged portion (the nucleus) supports a small subset of hardware and is verified for correctness using an automated theorem prover. The nucleus is manually annotated for verification. The verification was for single-threaded execution. The Hyper-V/VCC project [12, 20] verifies the Hyper-V hypervisor with drivers and untrusted guest OSes running within virtual machines. The C implementation was manually annotated with invariants and assertions to be verified, and an automated theorem prover was used to discharge the proof obligations. The verification is modular and assumes multi-threaded execution. The latest report suggests only 20% of the hypervisor code-base was verified with developer inserted annotations, indicating that full verification remains a challenge [20]. XMHF [34] adopts a micro-hypervisor design. The verification is largely automated with minimal annotations but only focuses on the memory integrity of the micro-hypervisor. The verification is for a single-threaded execution and extensions to XMHF are assumed to be confined to a set of narrow interfaces for accessing key system state. Software Fault Isolation (SFI) [22, 24, 28, 37, 41] is a software primitive aimed at application isolation. The focus is on verified address space separation of the (deprivileged) domains. The above mentioned verified sandboxes cannot be easily extended to provide additional security guarantees. If the extension is added as an untrusted component, the verification process does not provide ways to verify deprivileged components and integrate the verification results with those of the verified components, since verified sandboxing treats the deprivileged code as code without properties. Direct addition of code to the privileged portion requires full reverification of the privileged portion, which is non-trivial for theorem proving-based approaches. In our COSS example, adding the trusted network logging extensions to seL4 will require new specifications of the extensions which are compatible with existing specifications, and new refinement proofs—all of which is complex and time-consuming.

Evolvability is an issue where annotations are needed. For example, adding COSS trusted path extensions to Hyper-V (or Singularity) requires code contracts and annotations which can mesh with already existing annotations and verification conditions. This requires several iterations of trial-and-error before correct annotations can be found. This process becomes very complicated and expensive when changes are made to a large portion of the hypervisor or kernel with several extensions.

Scalability is an issue with direct additions in cases of model checking. For example, adding the COSS trusted network logging extensions to XMHF will likely double the code size which can cause the model checker to blow up. Composition is an issue with approaches based on unverified rewriting, e.g., SFI. For example, even though we can isolate the COSS trusted path and trusted network logging extensions from each other, we cannot prove that their security guarantees still hold since their behavior is not guaranteed to be preserved by the rewriting.

Full Verification Full software verification verifies the entire platform (hardware and/or software). Verve [40] is a simplified OS design where the OS and applications are verified for type and memory safety. A Hoare-style verification condition generator along with automated theorem prover is used to verify the correctness of the privileged portion of the OS (the nucleus) while a typed assembly language checker verifies safety of the kernel and applications. The verification is for a single-threaded execution and automated with support for low-level assembly language instructions. Ironclad [16] extends Verve with support for higher-level application properties. High-level specifications are translated to corresponding code with verification conditions that are then discharged via an automated theorem prover. The verification took 3 person-years. Verisoft [7] tightly integrates hardware and software, building on top of a custom processor. The verification was >20 years effort on a simple
OS with demand paging and disk driver.

Full software verification approaches provide verification of properties down to the instruction-level. However, they sacrifice current COSS ecosystem compatibility. For example, if we consider the COSS trusted path extensions, one needs to implement them entirely in a type-safe language (in Verve) or in high-level specifications (in Ironclad) or on a specific simplified processor architecture. Further, these approaches lack support for co-existence with unsafe (untrusted) programs, a typical case on commodity platforms.

6 Composable Verification for Commodity System Software

To achieve the verification goals listed in Section 4, we borrow ideas from state-of-the-art approaches [19,32,34] and augment the modular system design with hardware-based interface confinement primitive. The main idea is that in addition to partitioning the system into several modules, the majority of which runs in deprivileged mode, accesses to privileged system resources are restricted to a set of privileged interfaces, leveraging hardware support. That is, a buggy or malicious module is limited only to interfaces it is allowed to invoke to access resources or to invoke other modules. Unauthorized accesses will be stopped by hardware. Such interface confinement facilitates compositional verification. For instance, interface confinement enables local-only verification of a module, since the behavior of the module can be (conservatively) overapproximated by invariants on the interfaces that the module calls. We refer to modules to denote both components of the system software stack’s core components, and extensions, since piecemeal verification is important for either type of component.

Unlike prior state of the art that used language-based interface confinement leveraging the hardware allows us to tackle the challenge of evolvability and untrusted extensions while achieving our goals of extensibility and legacy compatibility with reasonable performant.

6.1 Granular Extensible interface Confinement (GEEC)

The basic idea of GEEC is that resources are placed behind a set of narrowly-defined interfaces so that the effect of accessing the resources can be over-approximated by examining these interfaces. GEEC leverages both hardware support and trusted and verified software components to provide interface confinement to resources at various levels of granularity such as a function, object or module. A slab is a logical unit of execution within a GEEC-based system (Figure 1) and enforces desired access control policies for code executing on the CPU. In our COSS example the core BIOS, option ROMs, micro-hypervisor core and extensions can be individual slabs. In contrast, device-processes are code and associated data which execute on a system device (e.g. GPU) which can generate interrupts on the CPU and directly access system resources (e.g., via DMA).

GEEC Resource access control. With GEEC all accesses to low-level machine resources such as as physical memory, control registers, privileged instructions, and devices are restricted to a set of interfaces (Figure 1). Access is allowed by the hardware according to access-control polices stored in the machine state. For example, the Memory Management Unit (MMU) authorizes memory accesses according to the active page table; when an access is rejected, a page-fault occurs and a handler is executed. During system runtime, we can extend the low-level resource confinement provided by GEEC to realize higher-level properties. For instance, within the micro-hypervisor layer in our example, we may wish to allow the hyperdep and approvexec extensions to modify the memory resource access-control state to provide finer-grained protections to guest OS applications. However, modifying access-control state is a sensitive operation. Therefore, we ensure all access-control modifications take effect via narrowly and precisely defined interfaces, denoted μAPI that are implemented in software and verified. We leverage hardware to enforce that all μAPI accesses must first pass through a μAPI sandbox, which consults a μAPI access policy before allowing access. An example policy could be where a restricted set of slabs are allowed access to the memory-protection μAPI; untrusted and unverified slabs are not allowed access to any μAPI. Another policy could give the unverified slab implicit trust to invoke a particular μAPI as long as we are able to preserve required μAPI invariants. This offers a layer of flexibility compared to prior approaches.

---

1 As with all the prior approaches, we assume the hardware to be functionally correct. Given the simplicity in hardware design and rigorous development practices, we believe this assumption is justified.
that requires same level of rigor, same (type-safe) language everywhere.

**GEEC Access control state switch.** To achieve mutual separation, each slab is assigned a resource access control policy stored in a dedicated access control state (ACS). GEEC employs a trusted entity (either implemented in software and verified or implemented in the hardware), called the *trampoline*. The trampoline is invoked via a hardware-enforced entry point and is in charge of switching the runtime ACS as control flow transfers between slabs (Figure 2). A slab’s resource ACS is also setup in such a way that any access to data or executes to memory location not within its own memory region causes the hardware to transfer control to the trampoline for error handling. Thus, the trampoline implements a “call–return–signal” semantics. This allows us to model control-flow transfer between slabs as a function call during verification greatly simplifying analysis. Since the trampoline is trusted, in addition to switching the resource access control context at slab call/return points, it can also be tasked to enforce more expressive *slab call policies* (i.e., which slab can call whom).

### 6.2 Verification

If implemented correctly, GEEC provides separation: each slab can only modify its own state and can only modify ACS via μAPIs. Such properties enable compositional verification of slabs. We obtain the following two sound composition principles: (1) if the property provided by slab A relies only on A itself (i.e., does not use any μAPIs), then A can be composed with any other slab to achieve the same property and A will not interfere with the other slabs’ properties; and (2) if the property provided by a slab A relies on only a set of μAPIs: μAPI\(_i\), then A can be composed with any set of slab that are not allowed to access μAPI\(_i\), and any slabs that accesses μAPI\(_i\) but operates on a disjoint resource state. These principles generalize naturally to a set of slabs.

The high-level proof of GEEC, which we call *proof-of-composability* is a non-trivial inductive proof that is derived from the model of GEEC and reasoning about its properties. From the high-level proofs, we extract concrete local properties to be verified on the GEEC implementation. A key observation is that GEEC is a foundational primitive and therefore once implemented correctly is unlikely to change further for a given architecture. Thus, the proof-of-composability once completed remains valid as long as the GEEC design is unchanged. The local properties required of the GEEC implementation also remain the same.

With the proof-of-composability in place, each slab can be verified for individual properties based on the μAPI invariants. Based on the slab properties and the μAPI used, a given set of slabs can be verified separately and composed if they are mutually non-conflicting. Untrusted, unverified slabs are denied access such μAPIs over which invariants are defined.

### 6.3 Practicality and Performance

Commodity hardware-virtualized x86 [2] and recent ARM platforms [8] can support GEEC, making this primitive practical today. For example, on the x86 the I/O MMU (e.g., VT-d) can be employed to efficiently restrict device process memory accesses. GEEC trampoline can be implemented leveraging a host of mechanisms such as hardware virtual machines, segmentation and page-tables. Recent improvements in hardware performance (e.g., tagged page-tables [2, 8]) is already laying the foundation to making GEEC performant at the granularity of task/modules. However, smaller granularity such as functions or object level slabs will require changes and/or addition of new hardware mechanisms. GEEC μAPI can be implemented via fast hardware system calls such as SYSENTER or SYSCALL which once again have a very low overhead and compare favorably to existing OS kernel calls [3, 18].

### 7 Conclusion and Open Problems

Application of GEEC enables infusing verified security properties that is both compositional and performant without sacrificing the rapidly evolving nature and legacy compatibility of today’s commodity system software stack – more compelling than is the case at present. The GEEC philosophy advocates the use of platform hardware in an attempt to get at the sweet-spot between verification complexity and performance. In this context, development of new (commodity) hardware extensions that can ease verification burden presents a new and exciting research area to explore. A few immediate and important open problems in this space include hardware support for verification at very early stages of the boot phase such as the BIOS where system devices are unitialized and CPU support is minimal and hardware support for high-performance fine-grained (e.g., byte, function or object) interface confinement.
References

[1] Novell apparmor and selinux comparison.


CMU/SEI Copyright

Copyright 2015 Carnegie Mellon University. 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. This material has been approved for public release and unlimited distribution.

DM-0002063