Trustworthy Prevention of Code Injection in Linux on Embedded Devices

Hind Chfouka[1], Hamed Nemati[2], Roberto Guanciale[2], Mads Dam[2], and Patrik Ekdahl[3]

[1] University of Pisa, Italy
chfouka@di.unipi.it
[2] KTH Royal Institute of Technology, Stockholm, Sweden
{hnnemati,robertog,mfd}@kth.se
[3] Ericsson AB, Lund, Sweden
patrik.ekdahl@ericsson.com

Abstract. We present MProsper, a trustworthy system to prevent code injection in Linux on embedded devices. MProsper is a formally verified run-time monitor, which forces an untrusted Linux to obey the executable space protection policy; a memory area can be either executable or writable, but cannot be both. The executable space protection allows the MProsper's monitor to intercept every change to the executable code performed by a user application or by the Linux kernel. On top of this infrastructure, we use standard code signing to prevent code injection. MProsper is deployed on top of the Prosper hypervisor and is implemented as an isolated guest. Thus MProsper inherits the security property verified for the hypervisor: (i) Its code and data cannot be tampered by the untrusted Linux guest and (ii) all changes to the memory layout is intercepted, thus enabling MProsper to completely mediate every operation that can violate the desired security property. The verification of the monitor has been performed using the HOL4 theorem prover and by extending the existing formal model of the hypervisor with the formal specification of the high level model of the monitor.

1 Introduction

Even if security is a critical issue of IT systems, commodity OSs are not designed with security in mind. Short time to market, support of legacy features, and adoption of binary blobs are only few of the reasons that inhibit the development of secure commodity OSs. Moreover, given the size and complexity of modern OSs, the vision of comprehensive and formal verification of them is as distant as ever. At the same time the necessity of adopting commodity OSs can not be avoided; modern IT systems require complex network stacks, application frameworks etc.

The development of verified low-level execution platforms for system partitioning (hypervisors [12,16], separation kernels [18,6], or microkernels [10]) has enabled an efficient strategy to develop systems with provable security properties without having to verifying the entire software. The idea is to partition the

system into small and trustworthy components with limited functionality running alongside large commodity software components that provide little or no assurance. For such large commodity software it is not realistic to restrict the adversary model. For this reason, the goal is to show, preferably using formal verification, that the architecture satisfies the desired security properties, even if the commodity software is completely compromised.

An interesting usage of this methodology is when the trustworthy components are used as an aid for the application OS to restrict its own attack surface, by proving the impossibility of certain malicious behaviors. In this paper, we show that this approach can be used to implement an embedded device that hosts a Linux system provably free of binary code injection. Our goal is to formally prove that the target system prevents all forms of binary code injection even if the adversary has full control of the hosted Linux and no analysis of Linux itself is performed. This is necessary to make the verification feasible, since Linux consists of million of lines of code and even a high level model of its architecture is subject to frequent changes.

Technically, we use Virtual Machine Introspection (VMI). VMI is a virtualized architecture, where an untrusted guest is monitored by an external observer. VMI has been proposed as a solution to the shortcomings of network-based and host-based intrusion detection systems. Differently from network-based threat detection, VMI monitors the internal state of the guest. Thus, the VMI does not depend on information obtained from monitoring network packets which may not be accurate or sufficient. Moreover, differently from host-based threat detection, VMIs place the monitoring component outside of the guest, thus making the monitoring itself tamper proof. A further benefit of VMI monitors is that they can rely on trusted information received directly from the underlying hardware, which is, as we show, out of the attackers reach.

Our system, MProsper, is implemented as a run-time monitor. The monitor forces an untrusted Linux system to obey the executable space protection policy (usually represented as $W \oplus X$); a memory area can be either executable or writable, but cannot be both. The protection of executable space allows MProsper to intercept all changes to the executable code performed by a user application or by the Linux kernel itself. On top of this infrastructure, we use standard code signing to prevent code injection.

Two distinguishing features of MProsper are its execution on top of a formally verified hypervisor (thus guaranteeing integrity) and the verification of its high level model (thus demonstrating that the security objective is attained). To the best of our knowledge this is the first time the absence of binary code injection has been verified for a commodity OS. The verification of the monitor has been performed using the HOL4 theorem prover and by extending the existing formal model of the hypervisor [16] with the formal specification of the monitor's runtime checks.

The paper is organized as follows: Section 2 introduces the target CPU architecture (ARMv7A), the architecture of the existing hypervisor and its interactions with the hosted Linux kernel, the threat model and the existing formal

models; Section 3 describes the MProsper architecture and design, it also elaborates on the additional software required to host Linux; Section 4 describes the formal model of the monitor and formally states the top level goal: absence of code injection; Section 5 presents the verification strategy, by summarizing the proofs that have been implemented in HOL4; Section 6 demonstrates the overhead of MProsper through standard microbenchmarks, it also presents measures of the code and proof bases; Finally, Sections 7 and 8 present the related work and the concluding remarks.

2 Background

2.1 The Prosper Hypervisor

The Prosper hypervisor supports the execution of an untrusted Linux guest [16] along with several trusted components. The hosted Linux is paravirtualized ; both applications and kernel are executed unprivileged (in user mode) while privileged operations are delegated to the hypervisor, which is invoked via hypercalls. The physical memory region allocated to each component is statically defined. The hypervisor guarantees spatial isolation of the hosted components; a component can not directly affect (or be affected by) the content of the memory regions allocated to other components. Thus, the interactions among the hosted components are possible only via controlled communication channels, which are supervised by the hypervisor.

The Prosper hypervisor and the MProsper monitor target the ARMv7-A architecture, which is the most widely adopted instruction set architecture in mobile computing. In ARMv7-A, the virtual memory is configured via page tables that reside in physical memory. The architecture provides two levels of page tables, in the following called L1s and L2s. These tables represent the configuration of the Memory Management Unit (MMU) and define the access permissions to the virtual memory. As is common among modern architectures, the entries of ARMv7 page tables support the NX (No eXecute) attribute : an instruction can be executed only if it is fetched from a virtual memory area whose NX bit is not set. Therefore, the system executable code is a subset of the content of the physical blocks that have at least an executable virtual mapping.

To isolate the components, the hypervisor takes control of the MMU and configures the pagetables so that no illicit access is possible. This MMU configuration can not be static; the hosted Linux must be able to reconfigure the layout of its own memory (and the memory of the user programs). For this reason the hypervisor virtualizes the memory subsystem. This virtualization consists of a set of APIs that enable Linux to request the creation/deletion/modification of a page table and to switch the one currently used by the MMU.

Similarly to Xen [4], the virtualization of the memory subsystem is accomplished by direct paging. Direct paging allows the guest to allocate the page tables inside its own memory and to directly manipulate them while the tables are not in active use by the MMU. Once the page tables are activated, the hyper-

request r	**DMMU behavior**
$switch(bl)$	makes block bl the active page table
$free_{L1}(bl)$ and $free_{L2}(bl)$	frees block bl, by setting its type to D
$unmap_{L1}(bl, idx)$, $unmap_{L2}(bl, idx)$	unmaps entry idx of the page table stored in block bl
$link_{L1}(bl, idx, bl')$	maps entry idx of block bl to point the L2 stored in bl'
$map_{L2}(bl, idx, bl', ex, wt, rd)$ and $map_{L1}(bl, idx, bl', ex, wt, rd)$	map entry idx of block bl to point to block bl' and granting rights ex, wt, rd to user mode
$create_{L2}(bl)$ and $create_{L1}(bl)$	makes block bl a potential L2/L1, by setting its type to $L2/L1$

Table 1: DMMU API

visor must guarantee that further updates are possible only via the virtualization API.

The physical memory is fragmented into blocks of 4 KB. Thus, a 32-bit architecture has 2^{20} physical blocks. We assign a type to each physical block, that can be: *data*: the block can be written by the guest, *L1*: contains part of an L1 page table and should not be writable by the guest, *L2*: contains four L2 page tables and should not be writable by the guest. We call the *L1* and *L2* blocks "potential" page tables, since the hypervisor allows to select only these memory areas to be used as page tables by the MMU.

Table 1 summarizes the APIs that manipulate the page tables. The set of these functions is called DMMU. Each function validates the page type, guaranteeing that page tables are write-protected. A naive run-time check of the page-type policy is not efficient, since it requires to re-validate the L1 page table whenever the *switch* hypercall is invoked. To efficiently enforce that only blocks typed D can be written by the guest the hypervisor maintains a reference counter, which tracks for each block the sum of descriptors providing access in user mode to the block. The intuition is that a hypercall can change the type of a physical block (e.g. allocate or free a page table) only if the corresponding reference counter is zero.

A high level view of the hypervisor architecture is depicted in Fig. 1. The hypervisor is the only component that is executed in privileged mode. It logically consists of three layers: (i) an interface layer (e.g. the exception handlers) that is independent from the hosted software, (ii) a Linux specific layer and (iii) a critical core (i.e. the DMMU), which is the only component that manipulates the sensible resources (i.e. the page tables). Fig. 1 demonstrates the behavior of the system when a user process in the Linux guest spawns a new process.

This design has two main benefits: (i) the critical part of the hypervisor is small and does not depend on the hosted software and (ii) the Linux-specific layer enriches the expressiveness of the hypercalls, thus reducing the number of context switches between the hypervisor and the Linux kernel. From a verification point

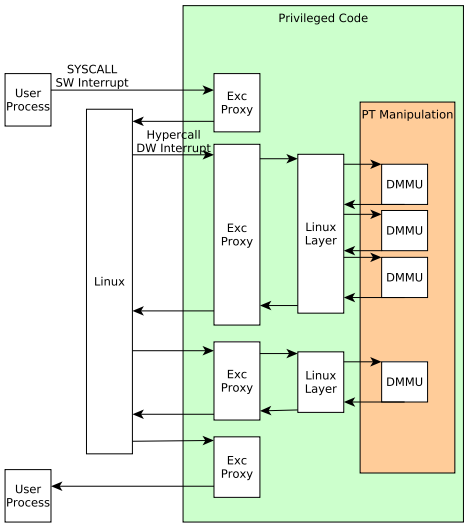

Fig. 1: Hypervisor Architecture

of view, to guarantee security of the complete system it is not necessary to verify functional correctness of the Linux layer; it suffices to verify that this layer never changes directly the sensitive resources and that its execution does not depend on the sensitive data. These tasks can be accomplished using sandboxing techniques [20] or tools for information flow analysis [3].

2.2 The Attack Model

The Linux guest is not trusted, thus we take into account an attacker that has complete control of the partition that hosts Linux. The attacker can force user programs and the Linux kernel to follow arbitrary flows and to use arbitrary data. The attacker can invoke the hypervisor, including the DMMU API, through software interrupts and exceptions. Other transitions into privileged memory are prevented by the hypervisor. The goal of the attacker is "code injection", for example using a buffer overflow to inject malicious executable code. This attack is normally performed by a malicious software that is able to write code into a data storage area of another process, and then cause this code to be executed.

In this paper we exemplify our monitor infrastructure using code signing. Signing the system code is a widely used approach to confirm the software author and guarantee (computationally speaking) that the code has not been altered or corrupted, by use of a cryptographic hash. Many existing code signing systems rely on a public key infrastructure (PKI) to provide both code authenticity and integrity. Here we use code signing to define integrity of the system code: integrity of an executable physical block stands for the block having a valid signature. Similarly, the integrity of the system code depends on the integrity of all executable physical blocks. The valid signatures are assumed to be known by

the runtime monitor. We refer to this information as the "golden image" (GI) and it is held by the monitor.

In order to make injected code detectable, we also assume that the attacker is computationally bound; it can not modify the injected code to make its signature compliant with the golden image. We stress that our goal is not to demonstrate the security properties of a specific signature scheme. In fact the monitor can be equipped with an arbitrary signature mechanism and the signature mechanism itself is just one of the possible approaches that can be used to check integrity of the system code. For this reason we do not elaborate further on the computational power of the attacker.

2.3 Formal Model of the Hypervisor

Our formal model is built on top of the existing HOL4 model for ARMv7 [7]. This has been extended with a detailed formalization of the ARMv7 MMU, so that every memory access uses virtual addresses and respects the constraints imposed by the page tables.

An ARMv7 state is a record $\sigma = \langle regs, coregs, mem \rangle \in \Sigma$, where *regs*, *coregs* and *mem*, respectively, represent the registers, coprocessors and memory. In the state σ, the function $mode(\sigma)$ determines the current privilege execution mode, which can be either $PL0$ (user mode, used by Linux and the monitor) or $PL1$ (privileged mode, used by the hypervisor).

The system behavior is modeled by a state transition relation $\xrightarrow{l \in \{PL0, PL1\}} \subseteq \Sigma \times \Sigma$, representing the complete execution of a single ARM instruction. Non-privileged transitions ($\sigma \xrightarrow{PL0} \sigma'$) start and end in $PL0$ states. All the other transitions ($\sigma \xrightarrow{PL1} \sigma'$) involve at least one state in privileged level. A transition from $PL0$ to $PL1$ is done by raising an exception, that can be caused by software interrupts, illegitimate memory accesses, and hardware interrupts.

The transition relation queries the MMU to translate the virtual addresses and to check the access permissions. The MMU is represented by the function $mmu(\sigma, PL, va, accreq) \rightarrow pa \cup \{\bot\}$: it takes the state σ, a privilege level PL, a virtual address $va \in 2^{32}$ and the requested access right $accreq \in \{rd, wt, ex\}$, for *readable*, *writable*, and *executable* in non-privileged respectively, and returns either the corresponding physical address $pa \in 2^{32}$ (if the access is granted) or a fault ($\bot$).

In [16] we show that a system hosting the hypervisor resembles the following abstract model. A system state is modeled by a tuple $\langle \sigma, h \rangle$, consisting of an ARMv7 state σ and an abstract hypervisor state h, of the form $\langle \tau, \rho_{ex}, \rho_{wt} \rangle$. Let $bl \in 2^{20}$ be the index of a physical block and $t \in \{D, L1, L2\}$, $\tau \vdash bl : t$ tracks the type of the block and $\rho_{ex}(bl), \rho_{wt}(bl) \in 2^{30}$ track the reference counters: the number of page tables entries (i.e. entries of physical blocks typed either $L1$ or $L2$) that map to the physical block bl and are executable or writable respectively.

The transition relation for this model is $\langle \sigma, h \rangle \xrightarrow{\alpha} \langle \sigma', h' \rangle$, where $\alpha \in \{0, 1\}$, and is defined by the following inference rules:

- if $\sigma \xrightarrow{PL0} \sigma'$ then $\langle \sigma, h \rangle \xrightarrow{0} \langle \sigma', h \rangle$; instructions executed in non-privileged mode that do not raise exceptions behave equivalently to the standard ARMv7 semantics and do not affect the abstract hypervisor state.
- if $\sigma \xrightarrow{PL1} \sigma'$ then $\langle \sigma, h \rangle \xrightarrow{1} H_r(\langle \sigma', h \rangle)$, where $r = req(\sigma')$; whenever an exception is raised, the hypervisor is invoked through a hypercall, and the reached state is resulting from the execution of the handler H_r

Here, *req* is a function that models the hypercall calling conventions; the target hypercall is identified by the first register of σ, and the other registers provide the hypercall's arguments. The handlers H_r formally model the behavior of the memory virtualization APIs of the hypervisor (see Table 1).

Intuitively, guaranteeing spatial isolation means confining the guest to manage a part of the physical memory available for the guest uses. In our setting, this part is determined statically and identified by the predicate $G_m(bl)$, which holds if the physical block *bl* is part of the physical memory assigned to the guest partition. Clearly, no security property can be guaranteed if the system starts from a non-consistent state; for example the guest can not be allowed to change the MMU behavior by directly writing the page tables. For this reason we introduce a system invariant $I_H(\langle \sigma, h \rangle)$ that is used to constrain the set of consistent initial states. Then the hypervisor guarantees that the invariant is preserved by every transition:

Proposition 1 *Let* $I_H(\langle \sigma, h \rangle)$. *If* $\langle \sigma, h \rangle \xrightarrow{i} \langle \sigma', h' \rangle$ *then* $I_H(\langle \sigma', h' \rangle)$.

We use the function $content : \Sigma \times 2^{20} \rightarrow 2^{4096*8}$ that returns the content of a physical block in a system state as a value of 4 KB. Proposition 2 summarizes some of the security properties verified in [16]: the untrusted guest can not directly change (1) the memory allocated to the other components, (2) physical blocks that contain potential page tables, (3) physical blocks whose writable reference counter is zero and (4) the behavior of the MMU.

Proposition 2 *Let* $I_H(\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}) \rangle)$. *If* $\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}) \rangle \xrightarrow{0} \langle \sigma', h' \rangle$ *then:*

2.1 For every bl such that $\neg G_m(bl)$ *then* $content(bl, \sigma) = content(bl, \sigma')$
2.2 For every bl such that $\tau(bl) \neq D$ *then* $content(bl, \sigma) = content(bl, \sigma')$
2.3 For every bl if $content(bl, \sigma) = content(bl, \sigma')$ *then* $\rho_{wt}(bl) > 0$
2.4 For every va, PL, acc *we have* $mmu(\sigma, va, PL, acc) = mmu(\sigma', va, PL, acc)$

3 Design

We configured the hypervisor to support the interaction protocol of Figure 2; the monitor mediates accesses to the DMMU layer. Since the hypervisor supervises the changes of the page tables the monitor is able to intercept all modifications to the memory layout. This makes the monitor able to know if a physical block is writable: This is the case if there exists at least one virtual mapping pointing to the block with a guest writable access permission . Similarly it is possible

to know if a physical block is executable. Note that the identification of the executable code (also called "working set") does not rely on any information provided by the untrusted guest. Instead, the monitor only depends on HW information, which can not be tampered by an attacker.

1. For each DMMU hypercall invoked by a guest, the hypervisor forwards the hypercall's request to the monitor.
2. The monitor validates the request based on its validation mechanism.
3. The monitor reports to the hypervisor the result of the hypercall validation.

Fig. 2: The interaction protocol between the Prosper hypervisor and the monitor

The first policy enforced by the monitor is code signature: Whenever Linux requests to change a page table (i.e. causing to change the domain of the working set) the monitor (i) identifies the physical blocks that can be made executable by the request, (ii) computes the block signature and (iii) compares the result with the content of the golden image. This policy is sufficient to prevent code injection that are caused by changes of the memory layout setting, due to the hypervisor forwarding to the monitor all requests to change the page tables.

However, this policy is not sufficient to guarantee integrity of the working set. In fact, operations that modify the content of a physical block that is executable can violate the integrity of the executable code. These operations cannot be intercepted by the monitor, since they are not supposed to raise any hypercall. In fact, a simple write operation in a block typed D does not require the hypervisor intermediation since no modification of the memory layout is introduced. To prevent code injections performed by writing malicious code in an executable area of the memory, the monitor enforces the executable space protection policy $W \oplus X$, preventing physical blocks from being simultaneously writable and executable. As for the hypervisor, a naive run-time check of the executable space protection is not efficient. Instead, we reuse the hypervisor reference counters: we accept a hypercall that makes a block executable (writable) only if the writable (executable) reference counter of the block is zero.

An additional complication comes from the Linux architecture. An unmodified Linux kernel will not survive the policies enforced by the monitor, thus its execution will inevitably fail. For example, when a user process is running there are at least two virtual memory regions that are mapped to the same physical memory where the process executable resides: (i) the user "text segment" and (ii) the "kernel space" (which is an injective map to the whole physical memory). When the process is created, Linux requests to set the text segment as executable and non writable. However, Linux does not revoke its right to write inside this memory area using its kernel space. This setting is not accepted by the monitor, since it violates $X \oplus W$, thus making it impossible to execute a user process.

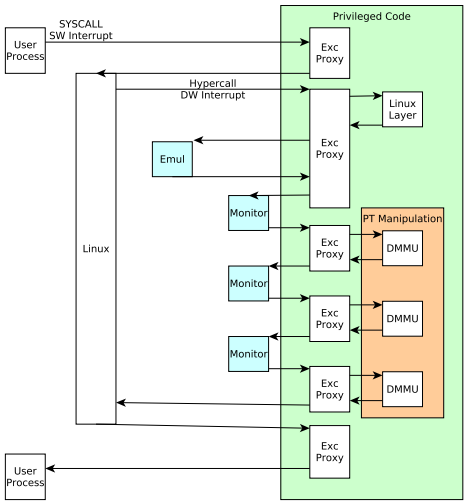

Fig. 3: MProsper's Architecture

Instead of adapting a specific Linux kernel we decided to implement a small emulation layer that has two functionalities:

- It proxies all requests from the Linux layer to the monitor. If the emulator receives a request that can be rejected by the monitor (e.g. a request setting as writable a memory region that is currently executable) then the emulator (i) downgrades the access rights of the request (e.g. setting them as non writable) and (ii) stores the information about the suspended right in a private table.
- It proxies all data and prefetch aborts. The monitor looks up in the private table to identify if the abort is due to an access right that has been previously downgraded by the emulator. In this case the monitor attempts (i) to downgrade the existing mapping that conflicts with the suspended access right and (ii) to re-enable the suspended access right.

Note that a malfunction of the emulation layer does not affect the security of the monitor. Namely, we do not care if the emulation layer is functionally correct, but only that it does not access sensible resources directly.

Fig. 3 depicts the architecture of MProsper. Both the runtime monitor and the emulator are deployed as two guests of the Prosper hypervisor. The Linux layer prepares a list of requests in a buffer shared with the emulation guest. After the Linux layer returns, the hypervisor activates the emulation guest, which manipulates the requests (or adds new ones) as discussed before. Then the hypervisor iteratively asks the monitor to validate one of the pending requests and upon success it commits the request by invoking the corresponding DMMU function.

Using a dedicated guest on top of the hypervisor permits to decouple the enforcement of the security policies from the other hypervisor functionalities,

thus keeping the trusted computing base minimal. Moreover, having the security policy wrapped inside a guest supports both the tamper-resistance and the trustworthiness of the monitor. In fact, the monitor can take advantage from the isolation properties provided by the hypervisor. This avoids malicious interferences coming from the other guests (for example from a process of an OS running on a different partition of the same machine). Finally, decoupling the run-time security policy from the other functionalities of the hypervisor makes the formal specification and verification of the monitor more affordable.

4 Formal model of MProsper

The formal model of the system (i.e. consisting of the hypervisor, the monitor and the untrusted Linux) is built on top of the models presented in Section 2.3. Here we leave unspecified the algorithm used to sign and check signatures, so that our results can be used for different intrusion detection mechanisms. The golden image GI is a finite set of signatures $\{s_1, \ldots, s_n\}$, where the signatures are selected from a domain S. We assume the existence of a function $sig : 2^{4096*8} \rightarrow S$ that computes the signature of the content of a block.

The system behavior is modeled by the following rules:

$$\frac{\langle\sigma,h\rangle \xrightarrow{0} \langle\sigma',h'\rangle}{\langle\sigma,h,GI\rangle \xrightarrow{0} \langle\sigma',h',GI\rangle} \qquad \frac{\langle\sigma,h\rangle \xrightarrow{1} \langle\sigma',h'\rangle \;\; validate(req(\langle\sigma,h\rangle),\langle\sigma,h,GI\rangle)}{\langle\sigma,h,GI\rangle \xrightarrow{1} \langle\sigma',h',GI\rangle}$$

$$\frac{\langle\sigma,h\rangle \xrightarrow{1} \langle\sigma',h'\rangle \;\; \neg validate(req(\langle\sigma,h\rangle),\langle\sigma,h,GI\rangle)}{\langle\sigma,h,GI\rangle \xrightarrow{1} \epsilon(\langle\sigma,h,GI\rangle)}$$

User mode transitions (e.g. Linux activities) require neither hypervisor nor monitor intermediation. Proposition 2.1 justifies the fact that, by construction, the transitions executed by the untrusted component can not affect the monitor state; (i) the golden image is constant and (ii) the monitor code can be statically identified and abstractly modeled. Executions in privileged mode require monitor intermediation. If the monitor validates the request, then the standard behavior of the hypervisor is executed. Otherwise the hypervisor performs a special operation to reject the request, by reaching the state that is returned by a function ϵ. Hereafter, the function ϵ is assumed to be the identity. Alternatively, ϵ can transform the state so that the requestor is informed about the rejected operation, by updating the user registers according to the desired calling convention.

The function $validate(req(\langle\sigma, h\rangle), \langle\sigma, h, GI\rangle)$ represents the validation mechanism of the monitor, which checks at run-time possible violations of the security policies. In Table 2 we briefly summarize the policies for the different access requests. Here, PT is a function that yields the list of mappings granted by a page table, where each mapping is a tuple (vb, pb, wt, ex) containing the virtual block mapped (vb), the pointed physical block (pb) and the unpriviledged rights to execute (ex) and write (wt). The rules in Table 2 are deliberately more abstract that the ones modeled in HOL4 and are used to intuitively present the behavior of the monitor. For example, the function PT is part of the hardware model and

request r	$validate(r, \langle \sigma, (\tau, \rho_{wt}, \rho_{ex}), GI \rangle)$ **holds iff**
$switch(bl)$	always
$free_{L1}(bl)$ and $free_{L2}(bl)$	always
$unmap_{L1}(bl, idx)$, $unmap_{L2}(bl, idx)$ and $link_{L1}(bl, idx, bl')$	$\rho_{ex}(bl) = 0$
$map_{L2}(bl, idx, bl', ex, wt, rd)$ and $map_{L1}(bl, idx, bl', ex, wt, rd)$	$sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, bl') \wedge$ $sound_S(ex, bl', \sigma, GI) \wedge \rho_{ex}(bl) = 0$
$create_{L2}(bl)$ and $create_{L1}(bl)$	$\forall (vb, pb, wt, ex) \in PT(content(bl, \sigma))$. $sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, pb) \wedge$ $sound_S(ex, pb, \sigma, GI)$ $\forall (vb', pb', wt', ex') \in PT(content(bl, \sigma))$. $no\text{-}conflict(vb, pb, wt, ex)(vb', pb', wt', ex')$

where

$sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, bl) = (ex \Rightarrow \neg wt \wedge \rho_{wt}(bl) = 0) \wedge (wt \Rightarrow \neg ex \wedge \rho_{ex}(bl) = 0)$

$sound_S(ex, bl, \sigma, GI) = (ex \Rightarrow integrity(GI, bl, content(bl, \sigma)))$

$$no\text{-}conflict(vb, pb, wt, ex)(vb', pb', wt', ex') = \begin{pmatrix} (vb \neq vb' \wedge pb = pb') \Rightarrow \\ (ex \Rightarrow \neg wt' \wedge wt \Rightarrow \neg ex') \end{pmatrix}$$

Table 2: Security policies for the available access requests

is not explicitly used by the monitor code, that is instead more similar to an iterative program. This makes our verification more difficult, but it also makes the monitor model as near as possible to the actual implementation, enabling further verification efforts that can establish correctness of the implementation.

Note that the monitor always checks that a mapping is not writable and executable simultaneously. Furthermore, if a mapping grants a writable access then the executable reference counter of the pointed physical block must be zero, guaranteeing that this mapping does not conflict (according with the executable space protection policy) with any other allocated page table. Similarly, if a mapping grants an executable access, then the writable reference counter of the pointed block must be zero.

To formalize the top goal of our verification we introduce some auxiliary notations. The working set identifies the physical blocks that host executable binaries and their corresponding content.

Definition 1. *Let σ be a machine state. The working set of σ is defined as*

$$WS(\sigma) = \{\langle bl, content(bl, \sigma) \rangle \mid \exists pa, va. mmu(\sigma, PL0, va, ex) = pa \wedge pa \in bl\}$$

By using a code signing approach, we say that the integrity of a physical block is satisfied if the signature of the block's content belongs to the golden image.

Definition 2. *Let $cnt \in 2^{4096*8}$ be the 4KB content of a physical block bl and GI be the golden image. Then $integrity(GI, bl, cnt)$ if, and only if, $sig(bl, cnt) \in GI$*

Notice that our security property can be refined to fit different anti-intrusion mechanisms. For example, $integrity(GI, bl, cnt)$ can be instantiated with the execution of an anti-virus scanner.

The system state is free of malicious code injection if the signature check is satisfied for the whole executable code. That is:

Definition 3. *Let σ be a machine state, bl be a physical block and GI be the golden image. Then integrity(GI, σ) if, only only if, for all $\langle bl, cnt\rangle \in WS(\sigma)$, integrity$(GI, bl, cnt)$*

Finally, we present our top level proof goal: No code injection can succeed.

Proposition 3 *If $\langle \sigma, h, GI\rangle$ is a state reachable from the initial state of the system $\langle \sigma_0, h_0, GI\rangle$ then integrity(GI, σ)*

5 Verification strategy

Our verification strategy consists of introducing a state invariant $I(s)$ that is preserved by any possible transition and demonstrating that the invariant guarantees the desired security properties.

Definition 4. *$I(\sigma, (\tau, \rho_{wt}, \rho_{ex}), GI)$ holds if*

$$
\begin{array}{l}
I_H(\sigma, (\tau, \rho_{wt}, \rho_{ex})) \wedge \\
\forall bl \ . \ (\neg(\tau(bl) = D)) \Rightarrow \forall (vb, pb, wt, ex) \in PT(content(bl, \sigma)). \\
\qquad sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, pb) \wedge sound_S(ex, pb, \sigma, GI)
\end{array}
$$

Clearly, the soundness of the monitor depends on the soundness of the hypervisor, thus I requires that the hypervisor's invariant I_H holds. Notice that the invariant constrains not only the page tables currently in use, but it constrains all potential page tables, which are all the blocks that have type different from D. This allows to speed up the context switch, since the guest simply re-activates a page table that has been previously validated. Technically, the invariant guarantees protection of the memory that can be potentially executable and the correctness of the corresponding signatures.

We verified independently that the invariant is preserved by unprivileged transitions (Theorem 1) and by privileged transitions (Theorem 2). Moreover, Lemma 1 demonstrates that the monitor invariant guarantees that there is no malicious content in the executable memory.

Lemma 1. *If $I(\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}), GI\rangle)$ then integrity(GI, σ).*

Proof. The proof is straightforward, following from $sound_S$ of every block that can be executable according with an arbitrary potential page table. □

Theorem 1 demonstrates that the invariant is preserved by instructions executed by the untrusted Linux. This depends on Lemma 2, which shows that the invariant forbids user transitions to change the content of the memory that is executable.

Lemma 2. *Let* $\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}), GI\rangle \xrightarrow{0} \langle \sigma', h', GI'\rangle$ *and* $I(\langle \sigma, h, GI\rangle)$ *then*

$$\forall bl \ . \ (\neg(\tau(bl) = D)) \Rightarrow \begin{pmatrix} PT(content(bl, \sigma')) = PT(content(bl, \sigma)) \wedge \\ \forall (vb, pb, wt, ex) \in PT(content(bl, \sigma')) \ . \\ \qquad (ex \Rightarrow content(pb, \sigma) = content(pb, \sigma')) \end{pmatrix}$$

Theorem 1. *If* $\langle \sigma, h, GI\rangle \xrightarrow{0} \langle \sigma', h', GI'\rangle$ *and* $I(\langle \sigma, h, GI\rangle)$ *then* $I(\langle \sigma', h', GI'\rangle)$.

Proof. From the inference rules we know that $h' = h$, $GI' = GI$ and that the system without the monitor behaves as $\langle \sigma, h\rangle \xrightarrow{0} \langle \sigma', h\rangle$. Thus, Proposition 1 can be used to guarantee that the hypervisor invariant is preserved ($I_H(\sigma', h')$).

If the second part of the invariant is violated then there must exist a mapping in one (hereafter bl) of the allocated page tables that is compliant with the executable space protection policy in σ and violates the policy in σ'. Namely, $content(bl, \sigma')$ must be different from $content(bl, \sigma)$. This contradicts Proposition 2.2, since the type of the changed block is not data ($\tau(bl) \neq D$).

Finally we must demonstrate that every potentially executable block contains a sound binary. Lemma 2 guarantees that the blocks that are potentially executable are the same in σ and σ' and that the content of these blocks is unchanged. Thus is sufficient to use the invariant $I(\sigma, h, GI)$, to demonstrate that the signatures of all executable blocks are correct. □

To demonstrate the functional correctness of the monitor (Theorem 2 i.e. that the invariant is preserved by privileged transitions) we introduce two auxiliary lemmas: Lemma 3 shows that the monitor correctly checks the signature of pages that are made executable. Lemma 4 expresses that executable space protection is preserved for all hypervisor data changes, as long as a block whose reference counter (e.g. writable; ρ'_{wt}) becomes non zero has the other reference counter (e.g. executable; ρ_{ex}) zero.

Lemma 3. *If* $\langle \sigma, h, GI\rangle \xrightarrow{1} \langle \sigma', (\tau', \rho'_{wt}, \rho'_{ex}), GI'\rangle$ *and* $I(\langle \sigma, h, GI\rangle)$ *then for all* bl, $\tau'(bl) \neq D \Rightarrow \forall (vb', pb', wt, ex) \in PT(content(bl, \sigma')).sound_S(ex, pb', \sigma', GI)$

Lemma 4. *Assume (i)* $I(\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}), GI\rangle)$, *(ii)* $\forall bl.(\rho_{ex}(bl) = 0 \wedge \rho'_{ex}(bl) > 0) \Rightarrow (\rho_{wt}(bl) = 0)$, *and (iii)* $\forall bl.(\rho_{wt}(bl) = 0 \wedge \rho'_{wt}(bl) > 0) \Rightarrow (\rho_{ex}(bl) = 0)$. *For all blocks* bl, *if* $sound_{W\oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, bl)$ *then* $sound_{W\oplus X}(wt, ex, \rho'_{wt}, \rho'_{ex}, bl)$

Theorem 2. *If* $\langle \sigma, h, GI\rangle \xrightarrow{1} \langle \sigma', h', GI'\rangle$ *and* $I(\langle \sigma, h, GI\rangle)$ *then* $I(\langle \sigma', h', GI'\rangle)$.

Proof. When the request is not validated ($\neg validate$) than the proof is trivial, since ϵ is the identity function.

If the request is validated by the monitor and committed by the hypervisor, then the inference rules guarantee that $GI' = GI$ and that the system without the monitor behaves as $\langle \sigma, h\rangle \xrightarrow{0} \langle \sigma', h\rangle$. Thus, Proposition 1 can be used to guarantee that the hypervisor invariant is preserved ($I_H(\sigma', h')$). Moreover, Lemma 3 demonstrates that the $sound_S$ part of the invariant holds.

The proof of the second part (the executable space protection) of the invariant is the most challenging task of this formal verification. This basically establishes the functional correctness of the monitor and that its run-time policies are strong enough to preserve the invariant (i.e. they enforce protection of the potentially executable space). Practically speaking, the proof consists of several cases: one for each possible request. The structure of the proof for each case is similar. For example, for $r = map_{L2}(bl, idx, bl', ex, wt, rd)$, we (i) prove that the hypervisor (modeled by the function H_r) only changes entry idx of the page table stored in block bl (that is, all other blocks that are not typed D are unchanged), (ii) we show that only the counters of physical block bl' are changed, and (iii) we establish the hypothesis of Lemma 4. This enables us to infer $sound_{W \oplus X}$ for the unchanged blocks and to reduce the proof to only check the correctness of the entry idx of the page table in the block bl. □

Finally, Theorem 3 composes our results, demonstrating that no code injection can succeed.

Theorem 3. *Let $\langle \sigma, h, GI \rangle$ be a state reachable from the initial state of the system $\langle \sigma_0, h_0, GI_0 \rangle$ and $I(\langle \sigma_0, h_0, GI_0 \rangle)$, then integrity$(GI, \sigma)$ holds*

Proof. Theorems 1 and 2 directly show that the invariant is preserved for an arbitrary trace. Then, Lemma 1 demonstrates that every reachable state is free of malicious code injection. □

6 Evaluation

The verification has been performed using the HOL4 interactive theorem prover. The specification of the high level model of the monitor adds 710 lines of HOL4 to the existing model of the hypervisor. This specification is intentionally low level and does not depend on any high level theory of HOL4. This increased the difficulty of the proof (e.g., it musts handle finite arithmetic overflows), that consists of 4400 lines of HOL4. However, the low level of abstraction allowed us to directly transfer the model to a practical implementation and to identify several bugs of the original design. For example, the original policy for $link_{L1}(bl, idx, bl')$ did not contain the condition $\rho_{ex}(bl) = 0$, allowing to violate the integrity of the working set if a block is used to store an L1 page table and is itself executable.

The monitor code consists of 720 lines of C and the emulator consists of additional 950 lines of code. Finally, 100 lines have been added to the hypervisor to support the needed interactions among the hosted components.

We used LMBench to measure the overhead introduced on user processes hosted by Linux. We focused on the benchmarks “fork”, “exec” and “shell”, since they require the creation of new processes and thus represent the monitors worst case scenario. As macro-benchmark, we measured in-memory compression of two data streams. The benchmarks have been executed using Qemu to emulate a Beagleboard-Mx. Since we are not interested in evaluating a specific signature scheme, we computed the signature of each physical block as the xor of the contained words, allowing us to focus on the overhead introduced by the monitor's

Benchmark	fork	exec	shell	tar -czvf 1.2 KB	tar -czvf 2.8 MB
No monitor	10240 μs	10174 μs	42889 μs	0.05 s	20.95 s
P Emu	11792 μs	11944 μs	48473 μs	0.09 s	21.05 s
P Emu + P Mon	13965 μs	13837 μs	54303 μs	0.10 s	21.02 s
P Emu + U Mon	17512 μs	17154 μs	67273 μs	0.11 s	20.98 s

Table 3: Qemu benchmarks

infrastructure. Table 3 reports the benchmarks for different prototypes of the monitor, thus enabling to compare the overhead introduced by different design choices. "No monitor" is the base configuration, where neither the monitor or the emulation layer are enabled. In "P Emu" the emulation layer is enabled and deployed as component of the hypervisor. This benchmark is used to measure the overhead introduced by this layer, which can be potentially removed at the cost of modifying the Linux kernel. In "P Emu + P Mon" both the monitor and the emulation layer are deployed as privileged software inside the hypervisor. Finally, in "P Emu + U Mon" the monitor is executed as unprivileged guest.

7 Related work

Since a comprehensive verification of commodity SW is not possible, it is necessary to architect systems so that the trusted computing base for the desired properties is small enough to be verified, and that the untrusted code cannot affect the security properties. Specialized HW (e.g. TrustZone and TPM) has been proposed to support this approach and has been used to implement secure storage and attestation. The availability of platforms like hypervisors and microkernels extended the adoption of this approach to use cases that go beyond the ones that can be handled using static HW based solutions.

For example, in [9] the authors use the seL4 microkernel to implement a secure access controller (SAC) with the purpose of connecting one front-end terminal to either of two back-end networks one at a time. The authors delegate the complex (and non security critical) functionalities (e.g. IP/TCP routing, WEB front-end) to untrusted Linuxes, which are isolated by the microkernel from a small and trusted router manager. The authors describe how the system's information flow properties can be verified disregarding the behavior of the untrusted Linuxes.

Here, we used the trustworthy components to help the insecure Linux to restrict its own attack surface; i.e. to prevent binary code injection. Practically, our proposal uses Virtual Machine Introspection (VMI), which has been first introduced by Garfinkel *et al.* [2] and Chen *et al.* [5]. Similarly to MProsper, other proposals (including Livewire [2], VMWatcher [8] and Patagonix [14]) use VMI, code signing and executable space protection to prevent binary code injection in commodity OSs. However, all existing proposals rely on untrusted hypervisors and their designs have not been subject of formal verification.

Among others non trustworthy VMIs, hytux [11], SecVisor [21] and NICKLE [19] focus on protecting integrity of the sole guest kernel. SecVisor establishes a trusted channel with the user, which must manually confirm all changes to the kernel. NICKLE uses a *shadow memory* to keep copy of authenticated modules and guarantees that any instruction fetch by the kernel is routed to this memory.

OpenBSD 3.3 has been one of the first OS enforcing executable space protection ($W \oplus X$). Similarly, Linux (with the PaX and Exec Shield patches), NetBSD and Microsoft's OSs (using Data Execution Prevention (DEP)) enforce the same policy. However, we argue that due to the size of the modern kernels, trustworthy executable space protection can not be achieved without the external support of a trusted computing base. In fact, an attacker targeting the kernel can circumvent the protection mechanism, for example using *return-oriented programming* [23]. The importance of enforcing executable space protection from a privileged point of view (i.e. by VMI) is also exemplified by [13]. Here, the authors used model checking techniques to identify several misbehaviors of the Linux kernel that violate the desired property.

8 Concluding remarks

We presented a trustworthy code injection prevention system for Linux on embedded devices. The monitor's trustworthiness is based on two main principles (i) the trustworthy hypervisor guarantees the monitor's tamper resistance and that all memory operations that modify the memory layout are mediated, (ii) the formal verification of design demonstrates that the top security goal is guaranteed by the run-time checks executed by the monitor. These are distinguishing features of MProsper, since it is the first time that absence of binary code injection has been verified for a commodity OS.

Even if the MProsper's formal model is not yet at the level of the actual binary code executed on the machine, this verification effort is important to validate the monitor design; in fact we were able to spot security issues that were not dependent on the specific implementation of the monitor. The high level model of the monitor is actually a state transition model of the implemented code, operating on the actual ARMv7 machine state. Thus the verified properties can be transferred to the actual implementation by using standard refinement techniques (e.g. [22]).

Our ongoing work include the development of a end-to-end secure infrastructure, where an administrator can remotely update the software of an embedded device. Moreover, we are experimenting with other run-time binary analysis techniques that go beyond code signature checking: for example an anti-virus scanner can be integrated with the monitor, enabling to intercept and stop self-decrypting malwares.

Acknowledgments Work supported by framework grant "IT 2010" from the Swedish Foundation for Strategic Research, and a project grant from Ericsson AB through the KTH ACCESS Linnaeus Excellence Centre.

References

1. Introduction to code signing. `http://msdn.microsoft.com/en-us/library/ie/ms537361(v=vs.85).aspx`.
2. F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, and D. Kaeli. Virtual machine monitor-based lightweight intrusion detection. *SIGOPS Oper. Syst. Rev.*, 45(2):38–53, July 2011.
3. M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1080–1091. ACM, 2014.
4. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
5. P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.
6. M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 223–234. ACM, 2013.
7. A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *proc. ITP'10*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.
8. X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM.
9. G. Klein. From a verified kernel towards verified systems. In *Programming Languages and Systems*, pages 21–33. Springer, 2010.
10. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220. ACM, 2009.
11. . Lacombe, V. Nicomette, and Y. Deswarte. Enforcing kernel constraints by hardware-assisted virtualization. *Journal in Computer Virology*, 7(1):1–21, 2011.
12. D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. FM'09*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer Berlin Heidelberg, 2009.
13. S. Liakh, M. Grace, and X. Jiang. Analyzing and improving linux kernel memory protection: a model checking approach. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 271–280. ACM, 2010.
14. L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, pages 243–258, 2008.
15. Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2. `http://support.microsoft.com/kb/875352`, 2008.
16. H. Nemati, R. Guanciale, and M. Dam. Trustworthy virtualization of the armv7 memory subsystem. In *SOFSEM 2015: Theory and Practice of Computer Science*, pages 578–589. Springer, 2015.
17. PaX-Team. Documentation for the pax project overall description. `http://pax.grsecurity.net/docs/pax.txt`, 2008.

18. R. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer US, 2010.
19. R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.
20. D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security Symposium*, pages 1–12, 2010.
21. A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.
22. T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. *ACM SIGPLAN Notices*, 48(6):471–482, 2013.
23. H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.