1-2013

vTRUST: A formal modeling and verification framework for virtualization systems

Jianan HAO

Yang LIU

Wentong CAI

Guangdong BAI

Jun SUN
Singapore Management University, junsun@smu.edu.sg

vTRUST: A Formal Modeling and Verification Framework for Virtualization Systems

Jianan Hao[1], Yang Liu[1], Wentong Cai[1], Guangdong Bai[2], and Jun Sun[3]

[1] School of Computer Engineering, Nanyang Technological University
[2] NUS Graduate School for Integrative Sciences and Engineering
[3] ISTD, Singapore University of Technology and Design

Abstract. Virtualization is widely used for critical services like Cloud computing. It is desirable to formally verify virtualization systems. However, the complexity of the virtualization system makes the formal analysis a difficult task, e.g., sophisticated programs to manipulate low-level technologies, paged memory management, memory mapped I/O and trusted computing. In this paper, we propose a formal framework, vTRUST, to formally describe virtualization systems with a carefully designed abstraction. vTRUST includes a library to model configurable hardware components and technologies commonly used in virtualization. The system designer can thus verify virtualization systems on critical properties (e.g., confidentiality, verifiability, isolation and PCR consistency) with respect to certain adversary models. We demonstrate the effectiveness of vTRUST by automatically verifying a real-world Cloud implementation with critical bugs identified.

1 Introduction

Over the last few years, virtualization is widely used in many areas especially in Cloud computing. Enterprise users can save money on establishment and upgrading of fundamental computing resources by outsourcing their business logics on the virtualization server which provides instant-ready, pay-by-use and elastic computing services.

Technically, a virtualization server, especially Infrastructure-as-a-Service (IaaS), employs a middleware called hypervisor to multiplex limited hardware resources to multiple virtual machines (VMs). Each VM should provide an illusion of virtualized hardware whose configurations include processor count, memory size, storage space and communication capabilities. For a user, a typical virtualization service involves calculation of user-provided computation and feedback of the result. Therefore, it is critical for virtualization systems to guarantee critical properties such as secrecy of user's information and verifiability of computed results.

However, research [15] shows that virtualization systems are vulnerable due to a larger attack surface and immature implementations. Software bugs in critical components (e.g., hypervisor), improper usage of secure technique (e.g., Trusted Computing) and flawed security assumptions can all lead to vulnerabilities. To investigate whether vital properties of a virtualization system can be guaranteed, it is particularly necessary for the system to be formally verified before the deployment.

L. Groves and J. Sun (Eds.): ICFEM 2013, LNCS 8144, pp. 329–346, 2013.

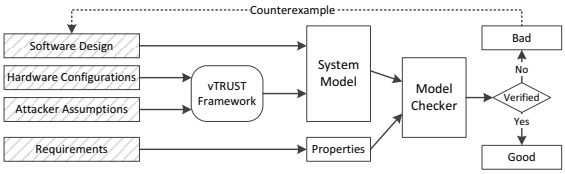


Fig. 1. Workflow

Unfortunately, virtualization systems are rarely analyzed by rigorous techniques like formal methods. So far, seL4 [8] is the only hypervisor that has been formally verified for functionalities at source code level using a theorem proving approach. However, the verification of seL4 on ARM platform took 20 person*year in total for complete proof based on Isabelle/HOL [10]. Since common hypervisors employed in Cloud systems are much bigger than seL4, it is arguably infeasible to verify them using a similar approach. Especially, because theorem proving is highly dependent on expert knowledge and manually created verification scripts can be error-prone. In this case, it is desirable to investigate an automatic approach to formally analyze virtualization systems.

In this paper, we propose a formal framework, namely vTRUST, to model and analyze virtualization systems using model checking tools with minimal manual effort. Fig. 1 presents the essential workflow to formally analyze the implementation of a virtualization system with the help of vTRUST. For a general virtualization system, it can be decomposed into 4 parts: software design, hardware configurations, attacker assumptions and service requirements are shown as slashed blocks for inputs. Based on software descriptions, the designer can implement executable programs on vTRUST architecture. Compared to real architecture (e.g., x86 and ARM), the vTRUST architecture focuses on the most critical (low-level) operations such as handling of virtualization traps, manipulation of memory protection and interaction with Trusted Computing, which are technically sophisticated and error-prone for software implementation. It is arguably safe to convert verified executable programs to native code on real architecture without the risk of introducing critical bugs. The designer can also model malicious programs according to various attacker assumptions. Based on hardware configurations and malicious programs, vTRUST framework can generate hardware and adversary models respectively. Additionally, critical properties will be specified in the system requirements. With the system model (including programs, hardware and adversary models) and the properties as inputs, a model checker can analyze the virtualization system. Especially, when a certain property is invalid, a counterexample will be given to direct implementation revision.

For page limitation, this paper only illustrates the most essential part of vTRUST. For more details, one may refer to [2] which includes full source code and testing results. The contributions of this paper can be summarized as follows.

– We propose a formal framework vTRUST, which is capable of modeling and analyzing virtualization systems with minimal manual efforts. It can cover common low-level details of virtualization systems and automatically explore design vulnerabilities. Moreover, the framework is extensible for more features.

- High-level properties of virtualization systems, e.g., confidentiality, verifiability, isolation and PCR consistency are formally specified. A verifier can test whether a property is satisfied with tolerance to a specific attacker model.
- As a case study, a Cloud implementation is modeled and analyzed by the vTRUST framework with PAT [1] as the model checker. A critical bug is found regarding unexpected relocation of hypervisor in a protected memory region, which is difficult to reveal manually.

Related Works. Formal verification of virtualization systems has been receiving more and more academic interest. One of the most famous works is the formal verification of seL4 [8], making it the only hypervisor verified at source code level so far. Although seL4 has only 8,700 and 600 lines of C and assembly code in ARM platform, the verification took 20 person*year to complete the proof on Isabelle/HOL, which makes the method infeasible to more complex systems. VCC [5] is another work to analyze the correctness of C programs by annotating the code with contracts in C preprocessor macros. Annotated programs are translated to logical formulas which will be passed to SMT solver Z3. Especially, Microsoft Hyper-V has been verified by VCC [9]. Other work focuses on verifying the integrity of hypervisor. Datta *et al.* [6] proposed a logic system to formally prove integrity of programs in the system using Trusted Computing. Vasudevan *et al.* [14] summarized the requirements for hypervisor based on hardware-assisted virtualization technology. Moreover, Soren Bleikertz *et al.* [4] proposed an automated verification approach for virtualized infrastructures.

2 Preliminary

2.1 CSP# Language

CSP# [12] is a modeling language which extends Hoare's CSP with new language features. CSP# integrates high-level modeling operators (e.g., parallel composition, choice, interrupt, channel communication, etc.) with shared variable and low-level procedural codes, for the purpose of efficient mechanical system verification. Part of the syntax of CSP# is given in the following, which will be used in the later content.

$$
\begin{array}{lll}
P ::= [b]P & & \text{– state guard} \\
\quad | \quad e \rightarrow P & & \text{– event prefixing} \\
\quad | \quad c?m \rightarrow P(m) \mid c!m \rightarrow P & & \text{– channel input/output} \\
\quad | \quad P;\ Q & & \text{– sequential composition} \\
\quad | \quad e\{program\} \rightarrow P & & \text{– data operation prefixing} \\
\quad | \quad P \,\Box\, Q & & \text{– external choices} \\
\quad | \quad P \,|||\, Q & & \text{– interleaving} \\
\quad | \quad (|||\, i : \{x..y\} \bullet P(i)) & & \text{– indexed interleaving} \\
\quad | \quad P \,\triangle\, (e \rightarrow Q); & & \text{– interrupt}
\end{array}
$$

where P and Q are processes, e is an event, b is a Boolean expression and c is a channel. In $e\{program\} \rightarrow P$, $program$ is executed atomically with the event e. Channel input and output events can be synchronous or asynchronous (with bounded channel buffer). $P \,|||\, Q$ allows processes P and Q to execute independently except they communicate

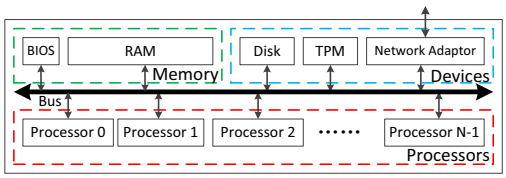


Fig. 2. Machine Model

with shared variables or synchronous channels. Especially, $(|||\ i : \{x..y\} \bullet P(i))$ is an indexed interleaving composition, which is equal to $P(x) \ ||| \ P(x+1) \ ||| \ ... \ ||| \ P(y)$ where x and y are integers (assuming $x \leq y$). Similar syntax go for indexed external choices. $P \triangle (e \rightarrow Q)$ behaves as P until event e is engaged and then behaves as Q.

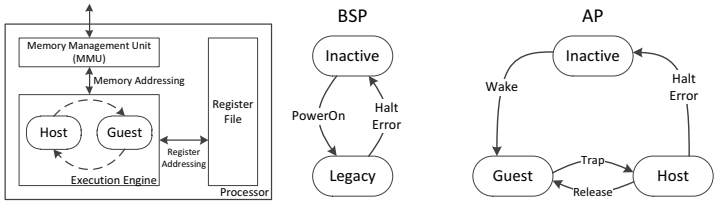
2.2 Hardware Technologies Related to Virtualization Systems

Virtualization systems rely on featured hardware technologies. First of all, processors should support classical virtualization, or trap-and-emulate virtualization [11], where the processor operates in dual modes, i.e., host and guest. Virtual Machine (VM) execution in the guest mode will be monitored for specific events. Upon an event, the execution in the guest mode will be suspended. A system software called hypervisor will take over to handle the event in the host mode. Especially, the guest program can actively invoke 'hypercall' for requesting services offered by the hypervisor. To provide memory resource among VMs, paged memory management are supported by Memory Management Unit. Memory-Mapped I/O is employed to access peripheral devices.

Virtualization systems can leverage on Trusted Computing for security enhancement. Trusted Computing is a technology aiming to enhance security of modern computers. Rather than confining what software can be carried out, Trusted Computing measures critical software stack, which is usually called Trusted Computing Base (TCB), as evidence. A secure chip, namely Trusted Platform Module (TPM) [13], must be installed to achieve Trusted Computing. TPM internally protects resources such as Platform Configuration Registers (PCRs), Endorsement Key (EK) and Storage Root Key (SRK). A PCR stores SHA-1 results from measuring memory data. Its value can be extended by inputting new data to it. Particularly, a method called Dynamic Root of Trust Measurement (DRTM) can be used to measure a piece of code as initial TCB with measurement result stored in PCR, and execute the code transactionally [3]. DRTM can only be performed by invoking 'late launch' instruction. The EK proves this TPM conforms to specification and SRK protects keys generated in the TPM. In addition, TPM can attest PCR value to a remote verifier. By comparing it to a known hash value of the software, one can realize what is running on the system.

2.3 Attacker Assumptions

For a virtualization system, attackers usually target the server where users' information is stored. In this work, we assume that attacks are performed by malicious software. The most important capability of the attacker is to construct malicious programs, and let the malicious programs be invoked on the server, which depends on the attacker's access



(a) Framework (b) States

Fig. 3. Processor Model

right. Hence, we define type 1 attacker as the malicious virtualization user who can only interact with the server remotely. Therefore, the attacker will compromise uploaded computational program which will be executed on the server. We also define type 2 attacker as one who has physical access to the server. Such an attacker can overwrite programs stored on the server's harddisk.

3 vTRUST: A Formal Framework for Virtualization Systems

This section is devoted to the proposed framework vTRUST, which includes formal descriptions for hardware modeling, software modeling and adversary modeling. One may refer to [2] for the complete model.

3.1 Hardware Modeling

A virtualization system requires featured hardware for its functionalities. For example, Fig. 2 shows a typical hardware model, which consists of processors, memory and devices. An internal network will connect them for communication.

To efficiently model low-level details of these hardware components, we make necessary abstractions to facilitate automatic verification like model checking. However, the most critical behaviors of hardware features are preserved and kept similar to real hardware. We will explain each subsystem in the following paragraphs. Especially, we employ a uniform model for memory and device subsystems. Lastly, various models will be composed together as a complete system model.

Execution Model. Fig. 3(a) illustrates the internals of a processor. For processor i, it can be active or inactive, which is represented by Boolean state $active[i]$. When the processor is active, it can operate in different modes. Especially, the first active processor, or bootstrap processor (BSP), always works in the legacy mode where virtualization is disabled. Application processors (APs) can be woken up by invoking the instruction $WAKE$ in BSP or other active APs. When the AP is active, it operates in the guest mode initially and can be trapped into the host mode. After the hypervisor has done its job, it 'releases' the control back to the guest mode. The operational mode is modeled by state variable $mode[i]$ and transitions are illustrated by Fig. 3(b).

The multiprocessor subsystem can thus be composed by parallel composition as follows where $Proc(i)$ represents processor i. Process $Proc(i)$ can be activated by setting $active[i]$ to true and $mode[i]$ to legacy or guest for BSP or APs respectively. After event $wakeup.i$ is engaged, $LegacyOrGuest(i)$ models the execution in the legacy or guest mode as follows. The execution is essentially a loop of fetching and executing programs. The loop modeled by the $FELoop(i)$ can be interrupted when channel $prochalt[i]$ receives a message to halt current processor. In this case, $active[i]$ will be set to $false$ and the processor will wait for the next wake-up.

$$Processors() = (||| \ i : \{0..(N-1)\} \bullet Proc(i));$$
$$Proc(i) = [active[i]]wakeup.i \rightarrow LegacyOrGuest(i);$$
$$LegacyOrGuest(i) = FELoop(i) \triangle (prochalt[i]?0\{active[i] = false; \ \} \rightarrow Proc(i));$$

According to trap-and-emulate virtualization, the execution in the guest mode will be trapped into the host mode upon specific events such as access to privileged resource, executing illegal instructions or intentionally invoking hypercall. Process $Trap$ is employed to model the trap as follows.

$$Trap(i, context) = g2h\{mode[i] = HOST; \ saveContext(context); \ \} \rightarrow Host(i);$$
$$Host(i) = FELoop(i) \triangle (release[i]?0\{mode[i] = GUEST; \ \} \rightarrow Skip);$$

where parameter $context$ denotes the essential information (e.g., source, reason, affected instruction and operators, etc.) of the trap. Event $g2h$ models the transition from the guest mode to the host mode by changing operational mode and saving context. Process $Host(i)$ models the execution in the host mode. It is similar to $LegacyOrGuest(i)$ except that $FELoop(i)$ can be interrupted by receiving event of synchronous channel $release[i]$ which 'releases' the control back to the guest mode.

Specifically, every program in vTRUST is modeled as a CSP# process, e.g., $BIOS(i)$ or $Bootloader(i)$ where i indicates its execution environment on processor i. A program is a container of instructions and assigned with an identifier such as $Prog\_BIOS$ or $Prog\_Bootloader$. Processor i will fetch and execute a program each time repeatedly as modeled by $FELoop(i)$.

$$FELoop(i) = fetch.i\{prog[i] = fetchProgram(pnp[i]); \ pnp[i]++; \ \} \rightarrow Execute(i); \\ FELoop(i);$$
$$Execute(i) = [prog[i] == Prog\_BIOS]BIOS(i) \ \square \\ [prog[i] == Prog\_Bootloader]Bootloader(i) \ \square \ ...$$

Event $fetch.i$ loads the current program from the memory address $pnp[i]$ which stands for the pointer to the next program. The $prog[i]$ will internally cache fetched programs during its execution. The pointer to the next program will be increased. After that, process $Execute(i)$ models the execution of program $prog[i]$ by dispatching according to program identifier.

Table 1. Primitive Instructions

Mnemonic	Legacy mode	Host mode	Guest mode
$MOVE\ dst, src$	$dst \leftarrow eval(src)$		
$JUMP\ adr$	$pnp \leftarrow eval(adr)$		
$WAKE\ hv, ptp, pnp$	Activate another processor. hv = hypervisor (physical address) ptp = paging table pointer (physical address) pnp = guest entry (logical address)		Trap
$HALT$	Inactivate current processor		Trap
$RELS$	Illegal	Release	Trap
$HYPC\ id$	Illegal	Illegal	Trap (Hypercall id)
$LL\ start, len$	Late launch	Illegal	Trap

Instruction Set. vTRUST defines only 7 primitive instructions whose semantics are summarized in Table 1. They are the only architectural interfaces defined to modify system states such as operational mode, register/memory values and device status. Each instruction is modeled as a CSP# process and its parameters are used to represent instruction operands. Note that an instruction may behave differently in different modes. For example, instruction $RELS$ can be modeled as follows[1].

$$\begin{aligned} RELS(i) = {} & [mode[i] == LEGACY]Error(UNDEFINED\_INSTRUCTION)\ \Box \\ & [mode[i] == HOST]release[i]!0 \rightarrow Stop\ \Box \\ & [mode[i] == GUEST]\,Trap(i, INVOKE\_INSTR\_RELS) \end{aligned}$$

Particularly, executing $RELS$ in legacy mode throws an error of undefined instruction. In the host mode, the instruction will terminate hypervisor's execution and release the control back to the guest program which triggers the trap. Lastly, when it is executed in the guest mode, it results in a trap with 'invoke $RELS$' as the context.

Primitive instructions used in every mode are $MOVE$ and $JUMP$. The former is used for data movement and the latter can intentionally modify the pointer to the next program. Note that function $eval(x)$ evaluates x based on the current execution environment and x stands for one of or a combination of register, memory and immediate number addressing modes. For example, $MOVE(Mem(0), Reg(x))$ copies data from register x to memory address 0. Moreover, the operand can also leverage CSP# syntax for arithmetic operations. For instance, $MOVE(Mem(0), Mem(0) + Mem(0))$ will double the value stored in $Mem(0)$.

As shown in Fig. 3(a), memory addressing is handled by MMU which is a featured component for virtualization systems to allocate memory resources among VMs. In the guest mode, every memory addressing will be processed by MMU for address translation and access right checking. The memory address referred to in the operand is called the logical address which is defined in the context of each processor; and the translated address is called the physical address which is globally indexed. Both logical and physical memory spaces are continuously grouped into pages as the granularity of memory management. The page size is configurable in vTRUST. A system structure called paging table describes how a logical address can be translated and what access rights are granted. Paging tables are stored in memory and a control register is employed to select

[1] In fact, CSP# does not allow a process to interrupt itself. The real implementation relies on another independent process as a proxy.

one of them as the active one in the guest mode. A paging table consists of a set of entries where entry $e(v, p, ac)$ indicates the vth page in logical memory space should be translated to the pth page in physical memory space. The ac stands for access control which describes the access rights granted for this page as a combination of $READ$, $WRITE$ and $EXECUTE$. However, MMU works differently in the legacy or host mode. The logical address is identically translated to the physical address and access right checking is omitted. This mechanism facilitates hypervisor to manage memory resources.

$$
\begin{aligned}
MOVE(i, dst, src) = iMOVE.i\{executeMOVE(i, dst, src)\} \rightarrow \\
if(lastState(i) == SUCCESS)\{/ * DoNothing * / \\
\}elseif(lastState(i) == FAILED)\{ \\
if(mode[i] == LEGACY \ || \ mode[i] == HOST)\{Error()\} \\
else\{Trap(i, Context(INSTR\_MOVE, dst, src))\}\}
\end{aligned}
$$

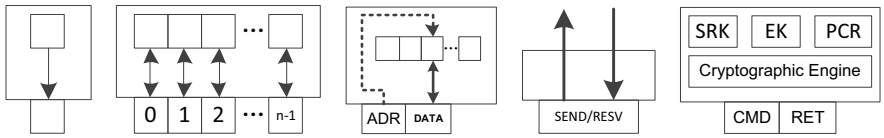
MMU's behavior is modeled by data operations of each instruction. For example, above statements model the execution of instruction $MOVE$ in processor i. The program attached with event $iMOVE.i$ invokes function $executeMOVE$ to calculate state changes after execution of the instruction, which includes translation and access right checking in MMU. When it is successful, the updated state will be committed atomically at the occurrence of event $iMOVE.i$; otherwise, the state is unchanged and the processor will trigger an error or a trap according to current operational mode.

Additionally, $WAKE(hv, ptp, pnp)$ is the instruction to activate another application processor. Especially, the new activated processor will start execution in the guest mode where hv is the hypervisor entry, ptp is the paging table pointer and pnp is the logical memory address of the first guest program. On the contrary, $HALT$ is the instruction to inactivate the current processor.

Memory and Device Model. Unlike a processor that actively executes programs, memory is passive. Therefore, memory access is modeled as a part of data operations in instruction execution, e.g., in the function $executeMOVE$ for instruction $MOVE$. To access devices, Memory-Mapped I/O (MMIO) technology is employed. That is, from the view of a processor, when it performs memory addressing at particular physical memory address, the system bus can route the request to a specific device rather than memory. In other words, a physical address can be mapped to a memory unit or a functional port of a device; otherwise, it is reserved and thus illegal to be accessed.

As memory and devices work in a similar way by atomically responding to requests from the system bus, we can summarize their common behaviors using an unified base class in C#. Each type of device can thus be modeled as a derived class by implementing the interfaces such as configuration upon installation, response of reading from or writing to particular functional port, and initialization upon machine reset.

Currently, vTRUST provides 5 common devices, i.e., RAM, ROM, disk, network adapter and TPM as shown in Fig. 4. ROM (Fig. 4(a)) is the simplest device. It defines one readonly functional port. Reading on it retrieves internal content which represents the flashed data upon installation and will be persistent across power cycles. RAM



(a) ROM (b) RAM (c) Harddisk (d) Net Adapter (e) TPM

Fig. 4. Memory and Device Models

(Fig. 4(b)) is mostly used in the system. It defines n functional ports as memory units and each is linked to a corresponding internal data for reading and writing. Especially, initialization will zero all units to model its volatile property. Disk (Fig. 4(c)) provides permanent storage across power cycles. Different from memory, storage units in disk cannot be accessed directly. To visit a specific cell, one must write the offset of that cell to the selector port. After that, one can read/write the data port to access selected cell. Network adapter (Fig. 4(d)) offers capabilities to communicate with remote users. One can read from 'receive' port for retrieving data or write to 'send' port for transmitting data. Internally, they are modeled by receving/sending event of CSP# synchronous channel. Therefore, the execution of the instruction that involves receiving/sending will be blocked until connected a remote party is ready to send/receive data. TPM (Fig. 4(e)) plays a vital role in Trusted Computing. A port is defined to receive encapsulated command to the TPM; and the other port can be read to retrieve command result. Essential functionalities mentioned in Sec. 2.2 are modeled.

These 5 common devices are expected to cover popular virtualization hardware; if not, one may further model new devices by implementing additional subclasses.

3.2 Software Modeling

In the vTRUST framework, software is implemented as functional programs. Each program can be modeled as a CSP# process that mainly consists of primitive instructions. For example, a BIOS firmware program can be modeled by process $BIOS(i)$ where i represents the execution environment of processor i and its identifier will be $Prog\_BIOS$.

$$\begin{aligned} BIOS(i) &= DISKLOAD(i, Mem(1), 0);\ JUMP(i, 1); \\ DISKLOAD(i, dst, offset) &= MOVE(i, Mem(DISK\_SEL), offset); \\ &\quad MOVE(i, dst, Mem(DISK\_DATA)); \end{aligned}$$

where $DISKLOAD$ defines a macro for loading a unit from harddisk to memory by calling two $MOVE$ instructions on selector and data ports of the harddisk respectively. $BIOS$ loads the first storage unit to $Mem(1)$ and transfers the control to it.

For a program that involves arithmetics and flow control, one can leverage CSP# syntax to achieve that. However, since the program executes in a single processor, syntax such as concurrency composition is not allowed.

3.3 Adversary Modeling

In vTRUST, an attack is performed by executing malicious software. First, malicious programs can be manually modeled as a vTRUST program. The program can additionally invoke $knowledge.add(x)$ to add a new entry x to the attacker's knowledge. Especially, a process $Eavesdrop$ is defined to add all accessible data (i.e., via memory and register addressing) to model eavesdropping. Furthermore, we use non-deterministic choice to enumerate all possible attack behaviors. In particular, vTRUST provides a library to model a piece of code that can execute arbitrary instructions.

$$
\begin{aligned}
ArbCode(i, n, opt) \quad &= Eavesdrop(); \\
&\quad if(n > 0)\{ArbInstruction(i, opt);\ ArbCode(i, n-1, opt)\} \\
ArbInstruction(i, opt) &= Skip \ \Box \\
&\quad [opt.allowMOVE]ArbMOVE(i, opt.move) \ \Box \\
&\quad [opt.allowJUMP]ArbJUMP(i, opt.jump) \ \Box \\
&\quad [opt.allowLL]ArbLL(i, opt.ll) \ \Box \ ...
\end{aligned}
$$

Here, process $ArbCode(i, n, opt)$ models a piece of code that contains up to n instructions where each is modeled by process $ArbInstruction(i, opt)$. An eavesdropping behavior is inserted between every two arbitrary instructions. Internal choices are employed to model non-determinism including doing nothing ($Skip$) or executing a particular instruction with undetermined operands. Especially, opt describes options for constructing these instructions. The option contains Boolean switches for executing each instruction with constraints of the operands. For example, an instruction $JUMP(i, adr)$ with specific range as adr can be modeled as follows.

$$ArbJUMP(i, range) = (\Box\ adr : \{range_{min}..range_{max}\} \bullet JUMP(i, adr));$$

Constructing malicious code with non-determinism is powerful to discover vulnerabilities. However, due to large choices of instructions and their operands, modeling a malicious program tends to reach state explosion in model checking. As a tradeoff, a compromised program can be modeled as a revision to its original program. The designer may rely on his domain knowledge to insert pieces of malicious code to proper positions of the original program, or skip certain original instructions.

For type 1 attacker, only uploaded program can be compromised aiming to exploit bugs on server software. For type 2 attacker, he is able to compromise all software on server's harddisk including the bootloader and even the hypervisor.

3.4 Compose a Complete Model

To complete the system, models of hardware, software and adversary must be composed together. Before power on the server, its hardware must be configured, which can be modeled by $ConfigServer$ as follows.

$$
\begin{aligned}
ConfigServer() &= ConfigProcessors(4,4);\ InstallDev(0, ROM(Prog\_BIOS));\\
&\quad InstallDev(1, RAM(23));\ InstallDev(24, Disk(Prog\_Bootloader));\\
&\quad InstallDev(28, TPM(AKey(ek), SKey(srk));\\
&\quad InstallDev(32, Net(chnout, chnin));\\
OnlineServer() &= ResetAll();\ (Processors() \triangle (sysreset?0 \rightarrow PowerOn()));\\
VirtSys() &= ConfigServer();\ (OnlineServer() \ |||\ Users());
\end{aligned}
$$

where $ConfigProcessors(n, ps)$ plugs n processors with ps units as a memory page; $InstallDev(adr, dev)$ installs a device (memory is considered as a special device) dev on the system and maps its functional ports starting from physical memory address adr. Each device is initialized by a constructor in C#. $ROM(p)$ models ROM flashed by p as its content; $RAM(n)$ provides RAM with length of n; $Disk(x_0, x_1, ..., x_{n-1})$ models a harddisk whose storage units are initialized by array x; $TPM(ek, srk)$ configures a TPM chip with ek and srk as endorsement storage keys; and $Net(out, in)$ models a network adaptor binding its outbound and inbound ports to channels out and in.

Process $OnlineServer$ models server's running behaviors. $ResetAll$ resets all components. The server hardware will start to work as modeled by process $Processors$. The system can be interrupted only after reset, which is modeled by the synchronous channel $sysreset$. In this case, the server will be restarted by invoking $PowerOn$ again. The complete behaviors of the system can thus be modeled by process $VirtSys$ as a sequence of server configuration and interaction between online server and users.

4 Properties of Virtualization Systems

For a virtualization system, it is desirable to provide guarantees for its software implementation. This section discusses some important properties for virtualization systems.

Confidentiality. Confidentiality guarantees that the system can prevent user's program, data and computed result from disclosure. It is a general requirement for virtualization systems to convince their users that sensitive information outsourced to a server are safe. Confidentiality is vital because when it is unsatisfied, the user's information which may contain intellectual properties can be disclosed to an untitled entity, which leads to loss of commerce and reputation.

In vTRUST, an attacker's knowledge has been formally modeled by $knowledge$, which makes the specification straightforward. First, $NonConfidentiality$ defines the insecure state that the user's secret has been obtained by the attacker, which breaks the confidentiality requirement. In other words, $secret$ can be recovered from the attacker's $knowledge$ where $secret$ represents sensitive information which generally includes user-provided data, program and computational result. Therefore, the model checker can test the system's reachability to such insecure state, which is specified by $reaches$ syntax. If the result is valid, the model checker will generate an execution trace to break the confidentiality as a counterexample; otherwise, it indicates state $NonConfidentiality$ is unreachable, i.e., confidentiality is satisfied.

$\#define\ NonConfidentiality\ (knowledge.know(secret))$;
$\#assert\ VirtSys()\ reaches\ NonConfidentiality$;

Verifiability. For the user who outsources his computation to a virtualization server, it is desirable to have a guarantee that the computational result is faithfully calculated from the input program and data. Verifiability defines a such property that users are capable of detecting if the result is forged. Similar to modeling confidentiality, we can model non-verifiability first and test its reachability, as shown below. The state of non-verifiability is reached when 'the user believes the result is good' but 'it is actually forged'. The definitions of two conditions are problem-specific.

$\#define\ NonVerifiability\ (BelieveGoodResult\ \&\&\ ForgedResult)$;
$\#assert\ VirtSys()\ reaches\ NonVerifiability$;

Isolation. For a cloud service with multiple users, it is desirable to guarantee the isolation among VMs. That is, any two VMs should not share same physical memory page unless it is expected. Especially, we specify 'strong isolation' and 'weak isolation' properties respectively. The former does not allow any overlapped mapping whereas the latter relaxes it by allowing shared memory page if $WRITE$ right is not granted.

Given any a processor i in the guest mode and another processor j in any mode, their active paging tables are denoted by pt_i and pt_j. The negative proposition of strong isolation can thus be defined as $\exists\, e_a \in pt_i, e_b \in pt_j \longrightarrow (e_a.p = e_b.p)$, where e_a and e_b refer to paging entries as mentioned in 3.1, and $e.p$ denotes its physical page number. Similarly, the negative proposition of weak isolation is defined as $\exists\, e_a \in pt_i, e_b \in pt_j \longrightarrow (e_a.p = e_b.p) \wedge (e_a.ac \& WRITE \neq 0)$. With the definitions of non-strong-isolation and non-weak-isolation, we model theses conditions in C# and test their reachability in the same way as confidentiality and verifiability.

PCR Consistency. For a system leveraging on Trusted Computing, when its PCR indicates a good TCB, the expected hypervisor should be always loaded for handling the traps, defined as PCR consistency. In other words, if a compromised hypervisor is activated to handle the traps but the PCR fails to reflect that the TCB is bad, PCR value is inconsistent with system status.

Suppose $goodpcr$ is the condition to indicate whether the PCR is a good value and program $Hypervisor\_Bad$ refers to the malicious hypervisor that should never be executed in the host mode when the good PCR value is present. The inconsistent state can be modeled as follows.

$\#define\ InconsistentPCR\ (goodpcr\ \&\&\ hv(Hypervisor\_Bad))$;
$\#assert\ VirtSys()\ reaches\ InconsistentPCR$;

where $hv(x)$ tests if program x is being executed in the host mode. Again, testing state reachability will verify the property.

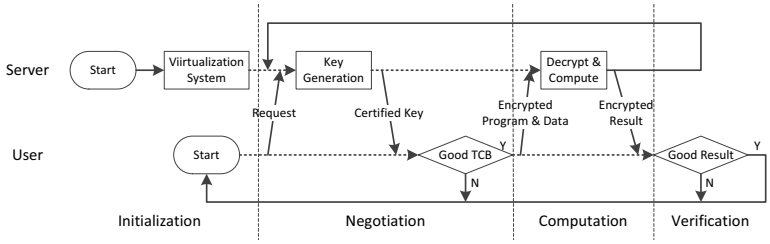


Fig. 5. Interactive Protocol

5 Case Study: Formal Analysis of Trusted Block as a Service

Trusted Block as a Service (TBaaS) [7] is a Cloud computing implementation. The system allows individual users to outsource their programs and data to the Cloud and retrieve computed results. We choose it as an example for its generality.

TBaaS involves interactions between a Cloud server and Cloud users. As shown in Fig. 5, the interactions can be divided into 4 stages: initialization, negotiation, computation and verification. In the initialization stage, the server is expected to setup a secure virtualization environment ready for providing services. In the negotiation stage, the user requests the service and a server will reply with evidence that the server system is securely built up. If the evidence is accepted, the user will enter the next stage; otherwise, the protocol aborts. The computation stage starts with uploading user's data and program to the server via an established secure channel. The server will create an isolated Trusted Block as a sandbox for the computation. The computed result and its integrity proof will be securely sent back to the user once the computation is done. In the verification stage, the user will verify the integrity of the result and finish the session.

Particularly, the initialization of the server involves considerable low-level operations with hardware. For example, it relies on bootloader to load the late launch entry (the first program after late launch) and the hypervisor into memory for DRTM. To minimize TCB, TBaaS moves potentially untrusted code such as network driver to the management VM. The isolation is achieved by paged memory management. As involved in DRTM, the late launch entry and the hypervisor must be bug-free.

5.1 System Modeling

In this section, we configure hardware and implement software from its design. Furthermore, we model malicious software according to attack assumptions.

Hardware Configuration. TBaaS server equips with 4 processors and groups 4 memory units as a page. First 24 physical memory units are allocated for ROM and RAM. Harddisk, TPM and network adaptor are installed in separate memory pages. The configuration is the same as mentioned in Sec. 3.4 except that harddisk stores programs $Prog\_Bootloader$, $Prog\_Driver$, $Prog\_LLEntry$ and $Prog\_Hypervisor$ in sequence.

Software Implementation. The server starts with the BIOS program which loads bootloader and transfers the control to it.

$$\begin{array}{l} \#define\ PADR\_LLENTRY\ 4; \quad \#define\ PADR\_HV\ 5; \\ Bootloader(i) = \\ \quad DISKLOAD(i, Mem(PADR\_LLENTRY), 2);\ //Mem(4) <= Prog(LLEntry) \\ \quad DISKLOAD(i, Mem(PADR\_HV), 3); \qquad //Mem(5) <= Prog(Hypervisor) \\ \quad LL(i, PADR\_LLENTRY, 2); \end{array}$$

Essentially, the bootloader loads late launch entry and hypervisor from harddisk to adjacent memory addresses defined by $PADR\_LLENTRY$ and $PADR\_HV$. Late launch is invoked with measurement of these memory units. If successful, late launch entry will take the control. Its task is to setup virtualization environment. Firstly, it prepares the paging table for the management VM (stored in $Mem(PADR\_MVMPT)$). The first page is mapped to the first physical page with all access rights for executing the driver program and accessing buffers; The second page is mapped to the functional port of the network adaptor (defined by $PADR\_NET$) with read/write access to allow the driver program to operate the adaptor. $PADR\_DRIVER$ and $LADR\_DRIVER$ respectively defines the driver's physical address and logical address based on newly prepared paging table. A $WAKE$ instruction is invoked to create the management VM by providing its hypervisor, paging table and entry point.

$$\begin{array}{l} \#define\ PADR\_DRIVER\ 2; \quad \#define\ LADR\_DRIVER\ 2; \\ \#define\ PADR\_MVMPT\ 6; \quad \#define\ PADR\_NET\ 32; \\ LLEntry(i) = \\ \quad DISKLOAD(i, Mem(PADR\_DRIVER), 1); \qquad //Mem(2) <= Prog(Driver) \\ \quad MOVE(i, Mem(PADR\_MVMPT), PageTable([0, RWX], [PADR\_NET, RW])); \\ \quad WAKE(i, PADR\_HV, PADR\_MVMPT, LADR\_DRIVER);\ HALT(i); \end{array}$$

The hypervisor is the most vital component in the system. TBaaS hypervisor only accepts two traps: the hypercall from the management VM for data arrival and the hypercall from a Trusted Block for the notification of completed computation. We summarize its structure in the following *pseudo-code* below due to space limitation.

$$\begin{array}{l} Hypervisor(i) = \\ \quad if\,(\textit{hypercall from Management VM}\ \&\&\ FuncID == DATAARRIVAL)\{ \\ \quad\quad \textit{Response network message according to network protocol;}\ RELS(i); \\ \quad \}else\ if\,(\textit{hypercall from TB}\ \&\&\ FuncID == JOBDONE)\{ \\ \quad\quad \textit{Encapsulate the result of TB;}\ Halt(i); \\ \quad \}else\{\textit{Error Handler}\} \end{array}$$

The virtualization users interact with the server by network communication which is modeled by operating channels bound to the server's network adaptor. Especially, the computational program uploaded to the server must follow this template:

$$\begin{array}{l} UserProg(i) = \textit{Read input data, calculate the result and store it to a specified location} \\ \qquad\qquad HYPC(i, JOBDONE); \end{array}$$

Table 2. Model Checking Results

Attacker	Malicious Software			Properties	Results				
	Bootloader	Hypervisor	UserProg		Valid?	$\mid S \mid$	$\mid T \mid$	Time(s)	Mem(MB)
Type 1	-	-	5 instructions	NonConfidentiality	NO	762K	1182K	249	74
				NonVerifiability	NO	762K	1182K	247	93
				NonStrongIsolation	NO	762K	1182K	249	88
				NonWeakIsolation	NO	762K	1182K	248	123
				InconsistentPCR	NO	762K	1182K	253	143
Type 2	4 instructions	3 instructions	-	NonConfidentiality	YES	163K	251K	42	102
				NonVerifiability	YES	163K	251K	42	85
				NonStrongIsolation	YES	163K	251K	42	81
				NonWeakIsolation	YES	163K	251K	42	104
				InconsistentPCR	YES	163K	250K	42	106
Type 2 (Bug fixed)	4 instructions	3 instructions	-	NonConfidentiality	NO	2481K	6551K	705	104
				NonVerifiability	NO	2481K	6551K	713	95
				NonStrongIsolation	NO	2481K	6551K	713	141
				NonWeakIsolation	NO	2481K	6551K	710	109
				InconsistentPCR	NO	2481K	6551K	717	132

Malicious Software. For the type 1 attacker model, we assume one of the users, e.g., Alice, uploads a malicious program which can be modeled as follows.

$$UserProg\_Bad(i) = ArbCode(i, l, opt);$$

where l indicates how complex the malicious program will be constructed, and opt controls the options of modeled code. They should be adjusted according to time and space constraints in the verification, i.e., more capable attackers can have more ways to execute the code, but this leads to longer verification time.

Type 2 attacker can compromise any program on the harddisk. We model the malicious programs by inserting pieces of malicious code into the original programs at the proper positions. For example, compromised bootloader can be modeled by inserting malicious code before original late launch instruction. Compromised late launch entry, driver and hypervisor can be modeled similarly. To load these compromised programs, the original harddisk will be replaced by $Disk(Prog\_Bootloader\_Bad, Prog\_Driver, Prog\_LLEntry, Prog\_Hypervisor, Prog\_Driver\_Bad, Prog\_LLEntry\_Bad, Prog\_Hypervisor\_Bad)$.

5.2 Verification and Evaluation

The TBaaS system should satisfy the properties as mentioned in Sec. 4. Specifically, confidentiality is specified by defining the secret as good users' uploaded programs, data and computed results. For verifiability, we model user's judgement of the result's integrity as a variable $UserJudge$ which can be $undefined$, $good$ and $forged$. The user should keep it as $undefined$ before the verification stage where it will be updated to $good$ or $forged$ according to the interaction protocol. Therefore, condition $BelieveGoodResult$ in Sec. 4 can be defined as '$UserJudge == good$'. For the condition $ForgedResult$, it can be evaluated by comparing user-received result with the genuine result pre-computed by the designer. Strong isolation and weak isolation must be achieved to isolate user's information among TBs. As Trusted Computing is used in TBaaS, PCR consistency must be satisfied as well.

Based on the system model generated by vTRUST and specified properties, we analyze the system against certain adversary models. Table 2 summarizes the experimental results where the testbed is a workstation with Intel Xeon E3-1245 CPU and 8GB RAM. Depth-first search is used to verify the properties.

For type 1 attacker (Alice), uploaded program contains 5 arbitrary instructions (including macros such as $DISKLOAD$) with all options enabled. The result shows expected behavior that all properties are satisfied. For type 2 attacker, he compromises software on the server side: bootloader is compromised by adding 4 arbitrary instructions; a malicious hypervisor is modeled by introducing 2 arbitrary instructions before creating the Trusted Block and another arbitrary instruction before encryption of the computed result.

The experimental result shows that a critical bug is found to break all properties by compromising bootloader and hypervisor when type 2 attacker is present. Based on the counterexample generated by PAT, we reconstruct the malicious bootloader as follows.

$$
\begin{array}{ll}
Bootloader\_Bad(i) = & \\
\quad DISKLOAD(i, Mem(4), 2); & //Mem(4) <= Prog(LLEntry) \\
\quad DISKLOAD(i, Mem(5), 3); & //Mem(5) <= Prog(Hypervisor) \\
++\, MOVE(i, Mem(1), Mem(4)); & //Mem(1) <= Mem(4) \\
++\, MOVE(i, Mem(2), Mem(5)); & //Mem(2) <= Mem(5) \\
++\, DISKLOAD(i, Mem(5), 6); & //Mem(5) <= Prog(Hypervisor\_Bad) \\
++\, LL(i, 1, 2); & //\textit{Late launch } \{Mem(1) \mid Mem(2)\} \\
\quad LL(i, 4, 2); &
\end{array}
$$

The original code firstly loads good late launch entry and hypervisor to $Mem(4)$ and $Mem(5)$ where the malicious code relocates them to $Mem(1)$ and $Mem(2)$ and further overwrites $Mem(5)$ by the malicious hypervisor. Late launch with memory range $Mem(1)$ to $Mem(2)$ will be invoked. As a result, the same PCR value will be obtained as the original bootloader would, which makes Cloud user infeasible to distinguish the difference. In original late launch entry, instruction $WAKE$ assigns $Mem(5)$ as the hypervisor for the management VM, which grants the malicious hypervisor the right to execute in the host mode, making all properties unsatisfied. To amend it, the late launch entry must perform a sanity check of where it is being loaded at the very beginning. Verification shows that amended system can defend against type 2 attacker.

6 Discussion and Conclusion

The vTRUST framework is a tradeoff between details and efficiency. For complex virtualization systems, vTRUST models the most critical low-level features such as trap-and-emulate execution, paged memory management, Memory Mapped I/O and Trusted Computing. With these features, one can model complex software logic based on simplified instruction set which is the fundamental for automatic exploration of vulnerabilities. Abstraction techniques such as explicit definition of programs help to achieve the goal without losing capabilities of modeling high-level properties.

Specifically, CSP# allows to define user-customized data types and their determined behavior in C#, which facilitates vTRUST to model internal states and their transitions

of hardware components. Most importantly, Object-Oriented Programming is utilized to model memory and devices for reusability. By implementing determined operations in C#, CSP# code can focus on non-determinism modeling and thereby improve readability. On the other side, we leverage CSP# syntax for flow-control and arithmetic operations and therefore avoid enlarging the instruction set.

The critical bug found by vTRUST is subtle. For the implementation regarding sophisticated technologies such as Trusted Computing and virtualization, it is common to overlook something in coding. For example, although the fact 'DRTM is irrelevant to memory location' can be derived by PCR extending operation, this issue is not highlighted in popular documents such as [3]. For a designer of virtualization systems, it is highly possible to overlook this deep-level information. Even worse, we argue similar bugs can hardly be revealed by traditional software testing or manual inspection. We remark that the properties that can be verified are not limited to the four types as shown in Sec. 4. Other functional properties like correctness can also be verified, which is mainly constrained by the model checker used.

In this paper, we proposed a formal framework vTRUST to analyze implementations of virtualization systems. Based on the vTRUST architecture, the system designer can implement software with low-level details as executable programs. The framework covers most common critical features in virtualization systems and thus can be used as a general modeling and analysis tool. Especially, we employed vTRUST to analyze a Cloud prototype, called TBaaS, and found a critical bug regarding a relocation issue which is arguably hard to discover by software testing or manual inspection. In the future, we will extend this work for more hardware features such as hardware interrupts and master devices. Hardware-based attacks will also be covered. Moreover, we will investigate optimization of modeling malicious code with more case studies.

Acknowledgement. We would like to thank project 'IDD11100102' from Singapore University of Technology and Design which supports this work.

References

1. Process Analysis Toolkit, `http://www.comp.nus.edu.sg/`
2. vTRUST Website, `http://www.comp.nus.edu.sg/%7Epat/vtrust`
3. AMD. Secure Virtual Machine Architecture Reference Manual
4. Bleikertz, S., Groß, T., Mödersheim, S.: Automated Verification of Virtualized Infrastructures. In: CCSW, pp. 47–58 (2011)
5. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
6. Datta, A., Franklin, J., Garg, D., Kaynar, D.: A Logic of Secure Systems and its Application to Trusted Computing. In: SP, pp. 221–236 (2009)
7. Hao, J., Cai, W.: Trusted Block as a Service: Towards Sensitive Applications on the Cloud. In: TrustCom, pp. 73–82 (2011)
8. Klein, G., et al.: seL4: Formal Verification of an OS Kernel. In: SOSP, pp. 207–220 (2009)

9. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
10. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
11. Popek, G.J., Goldberg, R.P.: Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM 17, 412–421 (1974)
12. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating Specification and Programs for System Modeling and Verification. In: TASE, pp. 127–135 (2009)
13. Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 116 (2011)
14. Vasudevan, A., McCune, J.M., Qu, N., van Doorn, L., Perrig, A.: Requirements for an Integrity-protected Hypervisor on the x86 Hardware Virtualized Architecture. In: Acquisti, A., Smith, S.W., Sadeghi, A.-R. (eds.) TRUST 2010. LNCS, vol. 6101, pp. 141–165. Springer, Heidelberg (2010)
15. Williams, B., Cross, T.: Virtualization System Security. In: IBM (2010)