Verification of XMHF HPT Protection Setup

Sarah J. Andrabi

University of North Carolina, *Chapel Hill*

***Abstract*—Over the years virtualization and hypervisors have gained immense popularity and at the same time there has been an increased concern about their security and many exploits have been demonstrated. Verification of hypervisors is challenging and one of the reasons is a large TCB. XMHF is a hypervisor with a small TCB of only 6018 LoC. In this paper we verify the page table setup of the XMHF initialization, using model checking and proving an invariant about the model. Thus ensuring that the hypervisor is in fact successful in providing a reduced memory map to the guest, in order to protect the XMHF Core from the guest.**

***Index Terms*—Hypervisor, Verification, XMHF, DRIVE, Memory Integrity, Page Tables, DMA, VT-d, CBMC, UCLID, Model Checking.**

I. INTRODUCTION

Virtualization has been around for many years but has gained more popularity for enterprise systems in the past few decades and can be considered as one of the most important issues in IT, starting a top to bottom overhaul of the computing industry [12]. As a technology, virtualization seems to represent one of those revolutionary paradigms that change the way we think and approach computing—the advent of Cloud Computing, Mobile Virtualization, Data Center Virtualization have ensured that virtualization affects our day-to-day lives and not just the big machines in the big rooms. Many analysts have predicted that, looking forward from 2014, in the majority of companies, more than 50% of all workloads will be virtualized and some companies will be approaching 100% virtualization [14]. This shift towards virtualization has a major impact on how enterprises work and at the same time raises the question on the security of these virtualized environments. One way in which VMMs provide security is by providing isolation between the host and guest executions as well as between the various guests the VMM is running. This can provide isolation of execution constructs like OS, memory, applications running on the host, etc. and prevent an attacker from gaining access to the host system or other guests. This is particularly useful in enterprises where a user has to access security-sensitive information from a company's intranet and also access the internet, which is like opening Pandora's Box and making not only the user's system but also the company's intranet vulnerable to security threats.

Hypervisors have been known to be susceptible to vulnerabilities some which are common to other systems and some that are unique to the virtualized environment, like information leaks between the host and the VMM that a malicious guest can see, information leaks between multiple guests, VM monitoring from host, VM monitoring from another VM, bugs that compromise isolation [21], side channel attacks that can reveal confidential information [22], etc. Thus highlighting the need to verify hypervisors and prove security properties about them.

Verification of hypervisors presents its own challenges due to the complex implementation and large data structures that hypervisors use. There are many examples of hypervisors that have been developed and verified like Xen [15], SecVisor [16], KVM [17], QEMU etc. This paper focuses on the verification of the initialization of the page tables of the XMHF hypervisor in order to protect the hypervisor core [1] from the guest. XMHF supports a single guest and provides core functionality through the XMHF core and extensible functional and security properties through hypervisor based solutions known as *hypapps*. This ensures that the XMHF core is small and also has a TCB of 6018 SLoc. XMHF claims to have comparable performance to popular high-performance general-purpose hypervisors [1], which makes verifying its properties even more interesting.

The single guest architecture of XMHF, allows the guest direct access to all performance critical system devices and device interrupts. The verification of the XMHF as carried out by [1] ensures memory integrity except during the initialization of the page tables; it is ensured at the end of initialization but not during initialization. The attacker model in [1] considers that an attacker can attempt to access memory during hypervisor (H) initialization, however, the hypervisor property required by their DRIVE methodology, as described later, provides access control only after H's initialization. Also their invariants imply H's memory integrity after initialization and during the execution of the guest. The invariant of interest is the memory invariant, φ_{M}, which requires that M is set as read-only in the memory access control property and an intercept *i* jumps to the starting address of the intercept handler for that event. This invariant holds after initialization is complete, not during initialization. In this paper we extend the invariant, φ_{M} so that it is true during the initialization of the page tables. In this paper bounded model checking is used to verify the correct setup of the protection for page table initialization of the CPU cores assigned to the guest, in order to provide a reduced memory map to the guest.

I also verify that after the DMA VT-d page table setup, our invariant holds. The model of the system is an over-approximated abstraction of the original system and the invariant is proved for this. Thus verifying the memory

isolation provided by the hardware page table initialization of XMHF. The verification carried out is only for the Intel x86 architecture support. The attacker model considered in this paper is the same as presented in [1].

The remainder of this paper is organized as follows. The XMHF hypervisor is summarized in Section II, including the XMHF design and implementation and the existing invariants that are have been verified for XMHF. A brief survey of CBMC and UCLID is presented in section III. Section IV presents the invariants and the verification methodology. Section V presents future work, and then the conclusion.

II. XMHF

eXtensible and Modular Hypervisor Framework (XMHF) was developed as a platform for performing security-oriented hypervisor research and development [1]. XMHF is designed to provide the necessary core functionality, required by all hypervisors and at the same time allows added functionality through the addition of extensions, which the authors of [1] call 'hypapps' and have been designed to reuse the XMHF core. The intuition behind this design is a small TCB for the core functionality that can then be easily verified. XMHF, however, supports only a single guest, in order to achieve its design goals. The driving force behind the XMHF hypervisor is the development of hypapps and simultaneous security verification.

One of the main design goals for the security of XMHF is to ensure memory integrity, in order to prevent guests from accessing each other's memory and the hypervisor core and parts of hypapps. The design, development and verification of XMHF is based on what has been called the *DRIVE* "Designing hypervisors for Rigorous Integrity VErification" methodology [1] comprising of properties that describe desired hypervisor behavior and other system invariants which together imply memory integrity. Some of the properties and invariants are guaranteed by the hardware and the system architecture, while others are satisfied via automated verification [1].

Currently XMHF supports both Intel and AMD x86 architectures and Windows and Linux OSes and has a TCB of 6018 LoC of hypervisor Core [1]. XMHF has been verified using CBMC and manual auditing—where CBMC doesn't work. The design of the XMHF core and its implementation is described next. We limit the description only to the XMHF core, as that is the focus of the verification for this paper.

A. XMHF Core Design and Implementation

XMHF is a type-1 hypervisor that is, it runs directly on top of the host's hardware to control the hardware and manage the guest OS. XMHF supports only a single guest. Also XMHF allows the guest to directly access the host's hardware. The XMHF Core has three main components i.e. the XMHF BootLoader (BL), the SecureLoader (SL) and the Runtime. The *hypapps* run alongside with the XMHF runtime. When the hypervisor starts it first loads the BL, then it transfers control to the SL, which finally lets the runtime take over and allow the *hypapps* to run.

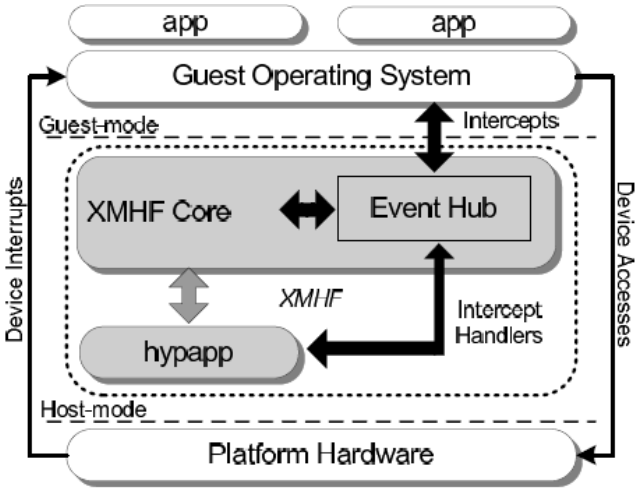

Fig 1. XMHF Single-Guest Execution Model

The runtime module calls the modules that initialize the memory and DMA protection, using the memprot and dmaprot modules. XMHF uses Extended Page Tables (EPT) for the Intel Architecture, and for the guest it uses PAE formatted page tables. It establishes a Dynamic Root of Trust (DRT). For Intel that is the Trusted Execution technology (SINIT module). This feature is implemented partially by a signed software module.

The 3 XMHF components, BL, SL and Runtime are part of the DRTM. The initialization module is untrusted and it has to dynamically launch the SL. The BL sets up the necessary hash values, sets up the page tables (this does not include the setting up of the protection for the page tables, which is done in the initialization of the runtime module), the heap, and then transfers control to the SL. The Bootloader enforces the integrity check strategy, which is formed based on the reverse build order. The Bootloader INIT gets the expected hash values for the Runtime, and the low 64K of the secure loader. The Runtime's hash gets embedded into the first 64K of the SL. The Bootloader INIT module assigns the highest physical memory address where the hypervisor binary is relocated to. It identifies the physical cores in the system, identifies and initializes the virtual buffers for all the cores and sets up the virtual buffers for the cores. The Bootloader finally transfers control to the SecureLoader.

In memory only the first 64K are trusted. SecureLoader (SL) starts with three empty pages, except for the first 4 bytes. The entry point, points beyond these 3 pages to the true entry point i.e. the fourth page. On the Intel system, these three pages will

be overwritten with the Measured Launched Environment (MLE) page tables. The MLE header will be written into the MLE at the beginning of the 4th page. The SL hashes the entire XMHF Runtime Memory image and compares it with a stored 'golden' hash value, which is saved within the SL at build time (remember the reverse build order i.e. how the hash for the runtime is taken and then stored in SL). After the SecureLoader finishes it transfers control to the *runtime_entry* module, which leads to the Runtime Startup module. This is where the protection for the EPT and VT-d's is set in the *memprot* and *dmaprot* modules, respectively. Apart from setting up the memory and DMA protection, the Runtime Startup module also calls the Baseplatform initialization, SMP Guest initialization, the debugging module initialization and the exception handling module. In the *memprot* module the page tables are assigned the Memory type range register (MTRR) types, which for the XMHF include Uncachable (UC), Write-Combined (WC), Write-Through(WT), Write-Protected(WP), Write-Back(WB) and Reserved(RESV); the default memory-type is Uncachable(UC). Another function is provided which is used to return the Memory type for a physical page. Also the page table protection is setup in the *memprot* module.

On the x86 platforms, only one CPU called the Bootstrap processor is enabled when the system starts. The other CPUs remain in halted state until activated by software running on the BSP. During its initialization, XMHF activates the remaining CPUs (including the BSP) to host mode. Next, XMHF sets up the HTPs on all cores and switches the BSP to guest-mode to start the guest; the remaining CPUs remain idle in host mode within XMHF. Finally the XMHF Core SMPGuest component uses a combination of HPTs and intercept handling to ensure that the remaining cores are switched to guest-mode before they execute guest code. This ensures that HPT access control is always enabled for all CPU cores.

The page table setup module in *memprot* assigns values to the page tables according to certain bounds and accordingly marks them as present or not-present, as described in section IV. The various invariants for ensuring memory integrity are described next.

B. Verification and Invariants

The modular design of XMHF allows verifying properties about the hypervisor and the hypapp implementation easily. The DRIVE methodology [1] clearly ascribes the following six properties for the design and verification of the hypervisor: Modularity of design, Atomicity of initialization and intercept handling, Memory Access Control for guest and devices, correct initialization, Proper mediation by the memory access control and finally, Safe State Updates. [1] Describes two system invariants that are used to prove the memory integrity of the hypervisor. The system is defined as a tuple $V = (H, G, D, M)$, where H is the hypervisor, G represents the guest, D represents the devices and M is the hypervisor memory containing both hypervisor code and data. In [1], *V* preserves an invariant φ, such that when the initialization finishes, φ holds; and throughout the execution of *V*—for which both H and G need to preserve the invariants, φ holds. The two invariants as defined in [1] are:

> $\varphi_M = M$ is designated as read-only as per the access-control policy and an intercept i jumps to the starting address of the i^{th} intercept handler, $ih_i()$.
>
> φ_{Med}= The Memory access control for the devices is always active.

For the verification of the DRIVE properties, a sequential execution model of *V* is used, and CBMC, which supports sequential execution, is used. This however, limits the ability to reason about entire page tables.

Thus by showing that the invariants hold for *V*, it is proved that the six DRIVE properties hold for *V*. This is the basis of the theorem that has been proved in [1]. The design of XMHF entails from the design for verification philosophy, and thus the system is designed in order to ensure that those properties hold, for example, to ensure memory protection XMHF uses two-level address translation through hardware page tables and ensures DMA protections are in place and that the hypervisor memory is marked as not-present for the guest, in order to prevent any form of access to the hypervisor core memory and data. For verification purposes of memory protection, it is claimed that the static allocation of the HPT and the DMA protection data structures ensures the property is satisfied. Similarly, the design of XMHF incorporates the other properties and then verifies them as presented in [1].

The next section gives a brief introduction to the tool that was used in [1], CBMC, to verify the properties and invariants of the DRIVE methodology and UCLID the tool that was used for verification in this paper.

III. A BRIEF SURVEY OF TOOLS

For the purpose of verification of XMHF [1], the authors have used CBMC, a bounded model checker for ANSI C and C++ programs. Though CBMC has its advantages, it also has certain limitations as is described below. For the verification carried out in this paper we use UCLID, a tool for analyzing the correctness of models of hardware and software systems. This section describes both CBMC and UCLID, explaining how both the tools work and a comment on their usability.

A. CBMC

CBMC is a Bounded Model Checker for ANSI-C and C++ programs, allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions [3]. To begin with, CBMC acts like a compiler taking the filename of the file as the command line argument and then, like a linker,

it translates the program and merges the function definitions from various files, producing a symbolic simulation on the program. CBMC then prints the list of properties that it has to check, and it determines this on its own and based on the assertions that are used to specify the program properties. CBMC transforms the equation, obtained from the linker into a clausal normal form (CNF) and passes it to a SAT solver. It then proves that the equation is either valid, or produces a counterexample.

The basic idea of CBMC is to model the computation of the programs up to a particular depth, amounting to unwinding of loops, in fact all loops have to have a finite upper run-time bound in order to guarantee that all bugs are found. CBMC actually checks that enough unwinding is performed [3], in fact CBMC is a sound tool only if enough unwinding of loops, function calls are done [4]. In many cases, CBMC is able to automatically determine an upper bound on the number of loop iterations but it may fail when the loops are highly data-dependent [3]. This is in fact one of the reasons that CBMC is not well suited to do complete iterations over entire page tables in XMHF, and hence the authors resort to manual auditing. The loop-based unwinding bound is not always appropriate, often it can become difficult to control the size of the generated formula when using the --unwind option [3]. To avoid some of these problems, CBMC provides the option of using the number of instructions for unwinding bounds, irrespective of the number of loop iterations [3].

One of the main problems with this approach is that if the formula is verified but cannot prove that sufficient unwinding has been performed the claim fails verification [4]. Also CBMC is resource demanding [4], it used 2GB of RAM for the verification of 5208 lines of code of XMHF [1].

B. UCLID

UCLID is a tool for term-level modeling and verification of infinite-state systems expressible in the logic of counter arithmetic with lambda expressions and uninterpreted functions (CLU) [5]. The CLU logic of UCLID [6] is a decidable fragment of first-order logic with restricted lambda expressions, uninterpreted functions and equality, counter arithmetic (i.e. addition by constants) and ordering (<) [5]. The tool has been implemented in Moscow ML, which is an ML dialect and interfaces to a SAT solver.

UCLID is used as a formal verification tool for infinite-state systems and has a degree of automation as compared to that of model checking tools for finite-state systems [7]. The UCLID modeling language describes systems where the state variables are Booleans, integers, bit-vectors, and functions mapping integers to integers or Booleans, or bit-vectors to bit-vectors [8]. The safety properties that UCLID proves rely on a decision procedure that translates a quantifier-free formula into an equi-satisfiable Boolean formula and then applies a SAT Solver [7]. UCLID has been used to verify a number of hardware designs and protocols [9, 10, and 11].

1) UCLID Decision Procedure

Operation—the decision procedure performs a series of transformations to reduce a first-order formula to a Boolean formula. The resulting CLU formula is translated to an equi-satisfiable Boolean formula using the following sequence of steps [5]: (i) First, lambda expressions are removed using Beta-reduction; (ii) Second, function applications are replaced with symbolic constants using optimizations like exploiting positive equality; (iii) Finally, integer-valued symbolic constants are either instantiated over a finite domain or atomic predicates over these symbolic constants are encoded using fresh Boolean variables and transitivity constraints are imposed. The generated formula is checked using a SAT solver.

Counterexample generation—if the formula is found to be invalid, UCLID generates a counterexample trace, specifying the values of the Boolean and integer variables, for which the formula is invalid. First, assignments for the integer variables are constructed, and then for each function application, the arguments and the result of the application are evaluated from the integer variables that represent them [5].

UCLID, thus allows the use of restricted Lambda's and overcomes the limitations of some other methods and it allows to reason about entire page tables. For the verification purposes of this work, I used UCLID, along with MiniSAT [20] as the SAT solver to reason about the page table initialization of the XMHF model using a short world's model. UCLID was used to simulate the model for a 1000 steps, and it was able to do so in 1450.911s, working with over 411347 variables and 1232815 clauses for the formula to be verified. Thus to avoid the limitations that CBMC has with respect to verifying page tables using our model, UCLID is a good alternative.

IV. METHODOLOGY

In the XHMF hypervisor, the functionality of the hypervisor is extended by hypapps, and the core functionality of the hypervisor is provided by the XMHF core. As a guest OS is run on top of the hypervisor, it is allowed direct access to the platform hardware and as such protection is enforced by presenting a reduced memory map to the guest—the hypervisor memory is marked as not present.

A. Problem Definition

The Intel x86 architecture support provided by XMHF uses a four-level PAE enabled page tables. With PAE paging [18], a set of 4 PDPTE registers is maintained, which are loaded from an address in CR3. Linear addresses are translated using 4 hierarchies of in-memory paging structures, each located using one of the PDPTE registers. In XMHF, the PAE paging maps

linear addresses to 4-KByte pages. The PDPTE registers contain the address for the Page Directory Tables, each PDE contains the addresses of the corresponding page tables, and each PTE points to the appropriate 4-KB Page. Figure 2 shows the address translation of the Intel IA-32 Architecture.

We verify that during the initial setup of the page table protection, the protection is setup correctly i.e. all the page table entries for the hypervisor core memory are marked as not present. Verification is done for this Intel x86 architectural support provided by XMHF.

For DMA access, XMHF allows the guest direct access to the platform hardware. It uses a direct assignment model, where the unmodified guest OS driver controls the device it is assigned. Intel VT-d enables system software to create multiple DMA protection domains [19]. With each protection domain containing an isolated portion of the physical memory. The VT-d architecture thus assigns I/O devices to protection domains, as shown in Figure 3.

DMA isolation is thus achieved by restricting access to a protection domain's physical memory from I/O devices not assigned to it by using address-translation tables [19]. XMHF uses a similar technique to ensure memory integrity for DMA access as well.

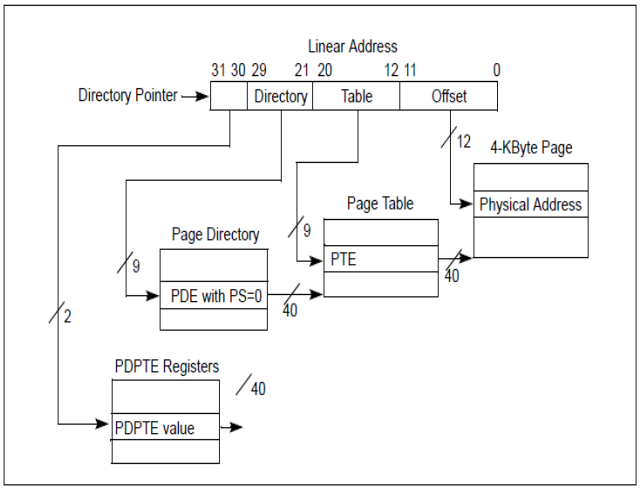

Fig 2. Linear Address Translation to 4KB pages using PAE paging

I verify that the setup of the DMA VT-d tables doesn't change the memory protection of the hypervisor memory.

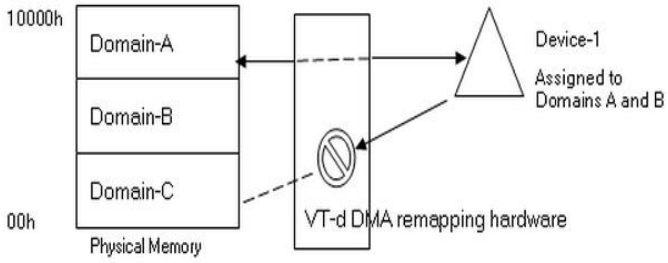

Fig 3. VT-d DMA Remapping—Device 1 doesn't have access to Domain C, thus when it tries to access it, it is blocked by the VT-d DMA remapping hardware

B. Verification

To verify the page table memory protection setup, bounded model checking is used for an over-approximated abstraction of the system where all the variables that do not affect our properties are abstracted away.

Two properties that describe the behavior of the system are used, and prove an invariant based on these properties. For model checking the first property provides the bounds, while the second property describes the intended behavior of the system.

Property 1: If M is the hypervisor memory then $|M| \in [l, h]$, where l and h are the bounds on the memory provided by the Runtime Physical Base Address and the Runtime Physical Size.

It simply states that the hypervisor memory lies within a certain range, which is known based on the system parameters. This provides us with the bounds that are required for the model.

Property 2: The hypervisor memory M, is marked as not-present.

This property simply states what is desired to provide a reduced memory-map.

The invariant that is proved is based on these two properties. For the hypervisor memory, M, the invariant is

$$\varphi'_M = For\ all\ |M| \in [l, h],\ M.pbit = notpresent,$$

Where, $M.pbit$ represents the presence bit for every page table entry belonging to the hypervisor memory.

Informally, ensuring that this invariant holds during the page table setup ensures that the guest will not have access to the hypervisor memory, at least based on the initialization. The

guest could still somehow, through the interaction with the API, modify the page table entries and gain access to the hypervisor memory.

Remember φ'_M, is an extension to the φ_{M} invariant proved in [1]. Together they ensure the memory integrity of the system, during initialization and after initialization.

To prove φ'_M the code of the XMHF hypervisor is modelled in UCLID, which allows us to reason about the behavior of entire loops over the page tables. Our page table model for XMHF memory is as follows: Since there is a unity mapping between the runtime virtual address and the system-physical address in XMHF, there is one page table structure, system. It has the structure as shown in Figure 2: a page directory pointer table (PDPtable), a page directory table (PDtable) and a page table (Ptable). Entries in the Ptable have three fields indicating: present/not-present (pbit), memory-type and address (addr). The memory-type refers to the Memory type range register (MTRR) types, as per the Intel x86 architecture. They can be Uncachable, Write Combining, Write Through, Write Back and Write Protected. They don't affect the pbit, which is the focus for providing a reduced memory map to the guest.

Let EPT = $(I, S, Init, X)$ be the page table model with

- $I = \{i, j, k\} : i, j\ and\ k$, index into PDPtable, PDtable and Ptable, respectively.
- $S = \{PDPtable, PDtable,\ Ptable,\ RuntimePhysLow,\ RuntimePhysHigh\ \}$; This is a set of state variable where PDPtable, PDtable and Ptable are modeled as functions that map indices to bit vectors. PDPtable, PDtable and Ptable return 64-bit vectors: (1-bit present, 32-bit address and the remaining most significant bits as zeros), RuntimePhysLow and RuntimePhysHigh are constant 64-bit vectors, which represent the memory limits.
- $Init = \{PDPtable_0, PDtable_0, Ptable_0\}$, Where each of them are initialized to arbitrary values.
- $X = \{G\}$, where G is a 128 bit vector constant, which represents the virtual address pointer to the page tables. It is a set of assignments to variables in S. Assignments define how state variables are updated, and thus define the transition relation of the system.

For each simulation of execution, PDPtable, PDtable, Ptable are updated with values based on those provided by G, in the C implementation these are values assigned to each CPU core, in our model these are allowed to take on arbitrary values.

It is verified that for each entry of the Ptable, the pbit is set to not-present. The invariant that is proved in UCLID is as follows:

$\varphi'_M = \ \forall k\ .\big((paddr \geq RuntimePhysLow)\ \wedge$
$(paddr < RuntimePhysHigh) \wedge$
$(Ptable[k].addr = paddr)\big) \longrightarrow Ptable[k].pbit = notpresent$, where paddr is any legal address.

It is proved that for a transition, defined by the model, from any arbitrary state to another state within the bounds of the model, φ'_M holds, for the model. Since the model is an over-approximated abstraction of the actual system, then the invariant holds for the actual system. The model is simulated for a single step, and the simulation is done for a single entry k in PT corresponding to a single entry j in PDT, which corresponds to a single entry i in PDPT. For each PT, PDT and PDPT fresh symbolic constants are used to choose any arbitrary entry from each of these. The formula is valid and no spurious counter-examples are generated.

After verifying that φ'_M holds for the EPT setup, the DMA VT-d protection module is modeled in a similar way. There is one page table structure, system with the following structure: a page directory pointer table (PDPT), a page directory table (PDT) and a page table (PT). Entries in the Ptable have two fields indicating: address (addr) and permissions (read/write).

DPT = $(I^D, S^D, Init^D, X)$ is defined as the DMA VT-d page table model with

- $I^D = \{l, m, n\} : l, m\ and\ n$, index into PDPtable, PDtable and Ptable, respectively.
- $S^D = \{PDPT, PDT,\ PT\}$; PDPT, PDT and PT are modeled as functions that map indices to bit vectors. PDPT, PDT and PT return 64-bit vectors: (1-bit read/write, 32-bit address and the remaining most significant bits as zeros).
- $Init^D = \{PDPT_0, PDT_0, PT_0\}$, Where each of them are initialized to arbitrary values.
- $X = \{G\}$, where G is a bit vector constant, which represents the virtual address pointer to the page tables. It is used to setup the page tables.

It is finally verified that after the execution of the DMA protection module our invariant φ'_M still holds, i.e., the DMA module does not modify the protection set in place for the hypervisor EPTs and again no spurious counter-examples are generated. Thus proving that during the page table initialization, the hypervisor memory is not visible to the guest, successfully providing a reduced memory map to the guest and preserving memory integrity of the hypervisor core, which is one of the primary design goals of the XMHF design.

V. FUTURE WORK AND CONCLUSION

As an extension to the current work, complete verification of the correct setup of the DMA protection module can be carried out. Also an interesting thing to verify using model checking with short world model is to verify that during the interaction of the guest with the hypervisor, the memory protections do not change for both the hypervisor core and DMA.

Reasoning about a system and coming up with a model to correctly reflect the system is a challenging and one can always question, if the model is correct. In this paper, the model is built by reasoning about the implementation of the system to reflect the system as close as possible and model checking is used to prove an invariant about an abstraction of the system. Our invariant holds for the page table setup of the XMHF hypervisor, thus formally proving the correctness of the protection setup for memory integrity, which is provided by providing a reduced memory map. It is also proved that the setup of the VT-d tables for DMA protections does not affect the reduced memory map and our invariant still holds. No spurious counter-examples are generated and our formula holds.

REFERENCES

[1] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome and Anupam Datta, "*Designing, Implementation and Verification of an eXtensible and Modular Hypervisor Framework"*, in Proc. IEEE Symposium on Security and Privacy 2013.

[2] Intel: "*Advantages of using virtualization technology in the enterprise"*, http://software.intel.com/en-us/articles/the-advantages-of-using-virtualization-technology-in-the-enterprise

[3] CBMC, http://www.cprover.org/cbmc/

[4] Kostyantyn Vorobyov, Padmanabhan Krishnan. "*Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches"*, In Proceedings of SSV, 2010.

[5] Shuvendu K. Lahiri and Sanjit A. Seshia, "*The UCLID Decision Procedure"*, In Computer-Aided Verification (CAV '04), LNCS 3114. pp 475–478, 2004.

[6] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "*Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions"*. In Computer-Aided Verification (CAV '02), LNCS 2404, pages 78-92, 2002.

[7] Randal E. Bryant, "*Formal Verification of Infinite State Systems Using Boolean Methods*", Lecture Notes in Computer Science Volume 4098, pp 1-3, 2006.

[8] User's Guide to UCLID Version 3.0, 2008

[9] A.M. Gharehbaghi and M. Fujita, *"Formal verification guided automatic design error diagnosis and correction of complex processors"*, In Proc. Int. High Level Design Validation and Test Workshop (HLDVT 11), IEEE Press, pp. 121-127, Nov. 2011.

[10] Bijan Alizadeh, Amir Masoud Gharehbaghi, and Masahiro Fujit, *"Pipelined Microprocessors Optimization and Debugging"*, In ARC 2010, LNCS 5992, pp. 435–444, 2010.

[11] R. Sinha, C. Sturton, P. Maniatis, S. A. Seshia, and D. Wagner. *"Verification with Small and Short Worlds"*. In FMCAD, 2012.

[12] Mike Dahlin, Ryan Johnson, Robert B. Krug, Michael McCoyd, William Young, "*Toward the Verification of a simple hypervisor"*, EPTCS 70, pp 28-45. 2011.

[13] William Hau, Rudolph Araujo, MacAfee, "*Virtualization and Risk – Key Security Considerations for your Enterprise Architecture"*, http://www.mcafee.com/us/resources/white-papers/foundstone/wp-virtualization-and-risk.pdf

[14] Doug Hazelman, Enterprise Systems, "*The Impact of Virtualization's Rise in 2013"*, http://esj.com/Articles/2012/12/12/Virtualization-Rise-2013.aspx

[15] 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".* In SOSP '03: Proceedings of the 19th ACM symposium on Operating systems principles, 2003.

[16] A. Seshadri, M. Luk, N. Qu, and A. Perrig. *"SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes"*. In 16th SOSP, pages 335–350, Oct 2007.

[17] A. Kivity. *"KVM: the Linux virtual machine monitor*". In OLS '07: The 2007 Ottawa Linux Symposium, pages 225-230, July 2007.

[18] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, May 2011.

[19] TW Burger, Intel® Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices, May 2012.

[20] MiniSAT, http://minisat.se

[21] T. Garfinkel and M. Rosenblum. *"When Virtual is Harder than Real: Security Challenges in Virtual Machine Bases computing Environments*". In Proc. of the 10th Workshop on Hot Topics in Operating Systems, Jun. 2005.

[22] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "*Cross-VM side channels and their use to extract private keys*". In G. Danezis and V. Gligor, editors, Proceedings of CCS 2012. ACM Press, Oct. 2012.

APPENDIX A

XMHF PAGE TABLE SETUP C Code

```
//---setup EPT for VMX-------------------------------------------------------

static void _vmx_setupEPT(VCPU *vcpu){

        //step-1: tie in EPT PML4 structures

        //note: the default memory type (usually WB) should be determined using

        //IA32_MTRR_DEF_TYPE_MSR. If MTRR's are not enabled (really?)

        //then all default memory is type UC (uncacheable)

        u64 *pml4_table, *pdp_table, *pd_table, *p_table;

        u32 i, j, k, paddr=0;

        pml4_table = (u64 *)vcpu->vmx_vaddr_ept_pml4_table;

        pml4_table[0] = (u64) (hva2spa((void*)vcpu->vmx_vaddr_ept_pdp_table) | 0x7);

        pdp_table = (u64 *)vcpu->vmx_vaddr_ept_pdp_table;

        for(i=0; i < PAE_PTRS_PER_PDPT; i++){

                pdp_table[i] = (u64) ( hva2spa((void*)vcpu->vmx_vaddr_ept_pd_tables + (PAGE_SIZE_4K * i)) | 0x7 );

                pd_table = (u64 *)  ((u32)vcpu->vmx_vaddr_ept_pd_tables + (PAGE_SIZE_4K * i)) ;

                for(j=0; j < PAE_PTRS_PER_PDT; j++){

                        pd_table[j] = (u64) ( hva2spa((void*)vcpu->vmx_vaddr_ept_p_tables + (PAGE_SIZE_4K * ((i*PAE_PTRS_PER_PDT)+j))) | 0x7 );

                        p_table = (u64 *)  ((u32)vcpu->vmx_vaddr_ept_p_tables + (PAGE_SIZE_4K * ((i*PAE_PTRS_PER_PDT)+j))) ;

                        for(k=0; k < PAE_PTRS_PER_PT; k++){

                                u32 memorytype = _vmx_getmemorytypeforphysicalpage(vcpu, (u64)paddr);

        //the XMHF memory region includes the secure loader +
        //the runtime (core + app). this runs from
        //(rpb->XtVmmRuntimePhysBase - PAGE_SIZE_2M) //with a size

        //of (rpb->XtVmmRuntimeSize+PAGE_SIZE_2M)

                //make XMHF physical pages inaccessible

                if( (paddr >= (rpb->XtVmmRuntimePhysBase - PAGE_SIZE_2M)) && (paddr < (rpb->XtVmmRuntimePhysBase + rpb->XtVmmRuntimeSize)) ){

p_table[k] = (u64) (paddr)  | ((u64)memorytype << 3) | (u64)0x0 ;
        //not-present

                }

                else{

                        if(memorytype == 0)

                                p_table[k] = (u64) (paddr)  | ((u64)memorytype << 3) |  (u64)0x7 ;    //present, UC

                        else

                                p_table[k] = (u64) (paddr)  | ((u64)6 << 3) | (u64)0x7 ;          //present, WB, track host MTRR

                }

                paddr += PAGE_SIZE_4K;

                }

                }

        }

}
```

APPENDIX B

UCLID SPECIFICATION

```
MODEL setupEPT

CONST

        rpb:BITVEC[128];

        vcpu:BITVEC[160];

        pdp_table_init:BITVECFUNC[64];

        pd_table_init:BITVECFUNC[64];

        p_table_init:BITVECFUNC[64];

        paddr_init:BITVEC[32];

        pdpt_table_init:BITVECFUNC[64];

        pdt_table_init:BITVECFUNC[64];

        pt_table_init:BITVECFUNC[64];


        (*//DMA VT-d consts*)

        vtd_pdpt_paddr:BITVEC[32];

        vtd_pdpt_vaddr:BITVEC[32];

        vtd_pdts_paddr:BITVEC[32];
```

```
        vtd_pdts_vaddr:BITVEC[32];
        vtd_pts_paddr:BITVEC[32];
        vtd_pts_vaddr:BITVEC[32];
        (*//*****************DMA VT-d Setup, marking everything as read/write***************)

MODULE VTd

INPUT

VAR
        physaddr:BITVEC[32];
        pdpt:BITVECFUNC[64];
        pdt:BITVECFUNC[64];
        pt:BITVECFUNC[64];

CONST
        l:BITVEC[32];
        m:BITVEC[32];
        n:BITVEC[32];
DEFINE
        zero_bit:=(0#[0:0]);
        one_bit:=(1#[0:0]);
        u64_zero:=(0x0000000000000000#[63:0]);
        u32_zero:=(0x00000000);

        PAGE_SIZE_4K:=(0x00800000#[31:0]);(*1<<12*)
        PAGE_SIZE_2M:=(0x00000200#[31:0]);(*1<<21*)
        PAE_PTRS_PER_PT:=(0x00200000#[31:0]); (*512*)
        PAE_PTRS_PER_PDT:=(0x00200000#[31:0]); (*512*)
        u32_512:=(0x00200000#[31:0]); (*512*)
        PAE_PTRS_PER_PDPT:=(0x40000000#[31:0]); (*4*)
        PCI_BUS_MAX:=(0x00100000#[31:0]); (*256*)
        VTD_READ:=(0x10000000#[31:0]); (*0x1*)      (*//Vt-d page-table bits*)
        VTD_WRITE:=(0x20000000#[31:0]); (*0x2*)
        pdptphysaddr:=vtd_pdpt_paddr#[31:0];
        pdtphysaddr:= pdptphysaddr +_32 PAGE_SIZE_4K;
        ptphysaddr:=pdtphysaddr +_32 (PAGE_SIZE_4K *_32 PAE_PTRS_PER_PDPT);

        pdpt_addr:=pdpt(l)#[31:0];
        pdt_addr:=pdt(m)#[31:0];
        physaddr:=n *_32 PAGE_SIZE_4K;

ASSIGN
        init[pdpt]:= Lambda(l).pdpt_table_init(l);
        next[pdpt]:=Lambda(l).
                case
                  (l> u32_zero & l< PAE_PTRS_PER_PDPT ):(u32_zero@((pdtphysaddr +_32 (l*_32 PAGE_SIZE_4K))||VTD_READ||VTD_WRITE));
                default:
                        pdpt(l);
                esac;

        (*//set pdt*)
        init[pdt]:=Lambda(m).pdt_table_init(m);
        next[pdt]:=Lambda(m).
                case
                  (m>= u32_zero & m< PAE_PTRS_PER_PDT & pdpt_addr = (u32_zero@((pdtphysaddr +_32 (l*_32 PAGE_SIZE_4K))||VTD_READ||VTD_WRITE)) ): u32_zero@((ptphysaddr +_32 (l *_32 PAGE_SIZE_4K *_32 u32_512)+_32 (m*_32 PAGE_SIZE_4K))||VTD_READ || VTD_WRITE);
                default:
                        pdt(m);
                esac;

        (*//set pt*)
        init[pt]:=Lambda(n).pt_table_init(n);
        next[pt]:=Lambda(n).
                case
```

```
                (n>= u32_zero & n< PAE_PTRS_PER_PT &
pdt_addr =(u32_zero@((ptphysaddr +_32 (l *_32 PAGE_SIZE_4K
*_32 u32_512)+_32 (m*_32 PAGE_SIZE_4K))‖VTD_READ ||
VTD_WRITE))):u32_zero@(physaddr || VTD_READ ||
VTD_WRITE);
            default:
                pt(n);
            esac;
(*//*******************************EPT setup
Module**************)
MODULE EPT
INPUT
VAR
    pdp_table:BITVECFUNC[64];          (*these
represent the entries of the respective tables*)
    pd_table:BITVECFUNC[64];
    p_table:BITVECFUNC[64];
    paddr:BITVEC[32];

    memory_type:BITVEC[8];(*//MTRR_TYPE_UC 0x0
,MTRR_TYPE_WC 0x1,MTRR_TYPE_WT
0x4,MTRR_TYPE_WP 0x5,MTRR_TYPE_WB
0x6,MTRR_TYPE_RESV 0x7*)

CONST

    i:BITVEC[32];
    j:BITVEC[32];
    k:BITVEC[32];
    m:BITVEC[32];

DEFINE
    XtVmmRuntimePhysBase:=rpb#[63:0];
    XtVmmRuntimeSize:=rpb#[127:64];
    (*//VCPU sturcture*)
    vmx_vaddr_ept_pml4_table:=vcpu#[31:0];
    vmx_vaddr_ept_pdp_table :=vcpu#[63:32];
    vmx_vaddr_ept_pd_tables:= vcpu#[95:64];
    vmx_vaddr_ept_p_tables:= vcpu#[127:96];
    vmx_ept_memorytypes:=vcpu#[159:128];

    (*//bit definitions*)
    zero_bit:=(0#[0:0]);
    one_bit:=(1#[0:0]);
    u64_zero:=(0x0000000000000000#[63:0]);
    u32_zero:=(0x00000000#[31:0]);
    u56_zero:=(0x00000000000000#[55:0]);
    u64_seven:=(0x7000000000000000#[63:0]);
    u64_six:=(0x6000000000000000#[63:0]);
    hex_3:=(zero_bit @ zero_bit @ one_bit @ one_bit);
    PAGE_SIZE_4K:=(0x00010000#[31:0]);(*1<<12*)
    PAGE_SIZE_2M:=(0x00000200#[31:0]);(*1<<21*)
    PAE_PTRS_PER_PT:=(0x00200000#[31:0]); (*512*)
    PAE_PTRS_PER_PDT:=(0x00200000#[31:0]); (*512*)
    PAE_PTRS_PER_PDPT:=(0x40000000#[31:0]); (*4*)

    (*pdt_addr:=pdp_table(i)#[31:0];*)
    (*pt_addr:=pd_table(j)#[31:0];*)
    paddr:=k *_32 PAGE_SIZE_4K;
    memory_type:=vmx_ept_memorytypes#[31:0];

ASSIGN
    (*//setup pdp_tables*)
    init[pdp_table]:=Lambda(i).pdp_table_init(i);
    next[pdp_table]:=Lambda(i).
        case
            (i >= u32_zero & i< PAE_PTRS_PER_PDPT):
u32_zero@(vmx_vaddr_ept_pd_tables +_32 (PAGE_SIZE_4K
*_32 i))‖u64_seven;
        default: pdp_table(i);
        esac;

    (*//setup the pd_table*)
    init[pd_table]:=Lambda(j).pd_table_init(j);
    next[pd_table]:=Lambda(j).
```

```
                case

                 (j>= u32_zero & j< PAE_PTRS_PER_PDT &
(pdp_table(i) = vmx_vaddr_ept_pd_tables +_32 (PAGE_SIZE_4K
*_32 i))):u32_zero @ (vmx_vaddr_ept_p_tables +_32
(PAGE_SIZE_4K *_32(i *_32 PAE_PTRS_PER_PDT)+_32
j))||u64_seven;

                default: pd_table(j);

                esac;


        (*//initiaize p_table*)

        init[p_table]:=Lambda(k).p_table_init(k);

        next[p_table]:=Lambda(k).

                case

                (k>= u32_zero & k< PAE_PTRS_PER_PT &
(pd_table(j)= u32_zero@(vmx_vaddr_ept_p_tables +_32
(PAGE_SIZE_4K *_32((i *_32 PAE_PTRS_PER_PDT)+_32 j))))
& (paddr>= (XtVmmRuntimePhysBase -_32 PAGE_SIZE_2M)) |
(paddr< (XtVmmRuntimePhysBase +_32 XtVmmRuntimeSize))):

                        ((u32_zero @ paddr) || (u56_zero @
(memory_type <<_8 hex_3) )|| u64_zero);

                (*//second case*)

                ((memory_type = u32_zero) & (pd_table(j)=
u32_zero@(vmx_vaddr_ept_p_tables +_32 (PAGE_SIZE_4K
*_32((i *_32 PAE_PTRS_PER_PDT)+_32 j))))):((u32_zero @
paddr) ||(u56_zero @ (memory_type <<_8 hex_3 ))|| u64_seven);

                default: p_table(k); (*//This is the problem*)

                esac;

(*//---------CONTROL MODULE----------*)

CONTROL


EXTVAR


STOREVAR

        i1:BITVEC[32];

        j1:BITVEC[32];

        k1:BITVEC[32];

        pdp_table1:BITVEC[64];

        pd_table1:BITVEC[64];


        rpb1:BITVEC[128];

        vcpu1:BITVEC[160];


        c1:BITVEC[64];

        c2:BITVEC[64];

        c3:BITVEC[64];


        dmapdpt:BITVEC[64];

        dmapdt:BITVEC[64];

        dmapt:BITVEC[64];

        dmaphysaddr:BITVEC[32];



VAR

        p_table1:BITVEC[64];

        paddr1:BITVEC[32];


CONST



DEFINE

        zero_bit:=(0x0#[0:0]);

        one_bit:=(0x1#[0:0]);

        u32_zero:=(0x00000000#[31:0]);


        PAGE_SIZE_2M:=(0x00000200#[31:0]);(*1<<21*)

        presenceCondition:=((EPT.paddr >=
(EPT.XtVmmRuntimePhysBase -_32 PAGE_SIZE_2M)
)&(EPT.paddr < (EPT.XtVmmRuntimePhysBase +_32
EPT.XtVmmRuntimeSize)) &
(EPT.p_table(EPT.k)#[31:8]=EPT.paddr#[31:8]))=>EPT.p_table(E
PT.k)#[0:0]=zero_bit;                (*//To check that the p_table
entry is not-present*)


EXEC


simulate(1);


i1:=EPT.i#[31:0];
```

```
j1:=EPT.j#[31:0];

k1:=EPT.k#[31:0];

rpb1:=rpb#[127:0];

vcpu1:=vcpu#[159:0];

pdp_table1:=EPT.pdp_table(EPT.i)#[63:0];

pd_table1:=EPT.pd_table(EPT.j)#[63:0];

p_table1:=EPT.p_table(EPT.k)#[63:0];

paddr1:=EPT.paddr#[31:0];

c1:=(EPT.XtVmmRuntimePhysBase -_32
PAGE_SIZE_2M)#[31:0];

c2:=(EPT.XtVmmRuntimePhysBase +_32
EPT.XtVmmRuntimeSize)#[31:0];

dmapdpt:=VTd.pdpt(VTd.l)#[63:0];

dmapdt:=VTd.pdt(VTd.m)#[63:0];

dmapt:=VTd.pt(VTd.n)#[63:0];

dmaphysaddr:=VTd.physaddr#[31:0];


print(paddr1);


decide(presenceCondition);

printexpr(i1);

printexpr(j1);

printexpr(k1);

printexpr(rpb1);

printexpr(vcpu1);

printexpr(pdp_table1);

printexpr(pd_table1);

printexpr(p_table1);

printexpr(paddr1);

printexpr(c1);

printexpr(c2);

printexpr(dmapdpt);

printexpr(dmapdt);

printexpr(dmapt);

printexpr(dmaphysaddr);
```